STUDIES OF INSPECTION ALGORITHMS AND ASSOCIATED MICROPROGRAMMABLE HARDWARE IMPLEMENTATIONS

A Thesis submitted for the degree of

Doctor of Philosophy

of the University of London

by

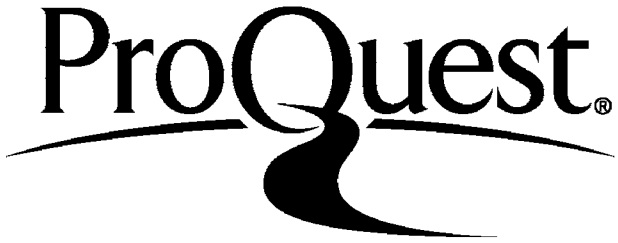
John Mark Edmonds

Machine Vision Group, Department of Physics

Royal Holloway and Bedford New College

University of London

February 1988



Abstract

This work is concerned with the design and development of real-time algorithms for industrial inspection applications. Rather than implement algorithms in dedicated hardware, microprogrammable machines were considered essential in order to maintain flexibility.

After a survey of image pattern recognition where algorithms applicable to real-time use are cited, this thesis presents industrial inspection algorithms that locate and scrutinise actual manufactured products. These are fast and robust - a necessary requirement in industrial environments.

The National Physical Laboratory have developed a Linear Array Processor (LAP) specifically designed for industrial recognition work. As with most array processors, the LAP has a greater performance than conventional processors, yet is strictly limited to parallel algorithms for optimum performance. It was therefore necessary to incorporate sequentialism into the design of a multiprocessor system. A microcoded bit-slice Sequential Image Processor (SIP) has been designed and built at RHBNC in conjunction with the NPL. This was primarily intended as a post-processor for the LAP based on the VMEbus but in fact has proved its usefulness as a stand-alone processor. This is described along with an assembler written for SIP which translates assembly language mnemonics to microcode.

This work, which includes a review of current architectures, leads to the specification of a hybrid (SIMD/MIMD) architecture consisting of multiple autonomous sequential processors. This involves an analysis of various configurations and entails an investigation of the source of bottlenecks within each design. Such systems require a significant amount of interprocessor communication: methods for achieving this are discussed, some of which have only become practical with the decrease in

cost of electronic components. This eventually leads to a system for which algorithm execution speed increases approximately linearly with the number of processors. The algorithms described in earlier chapters are examined on the system and the practicalities of such a design are analysed in detail.

Overall, this thesis has arrived at designs of programmable real-time inspection systems, and has obtained guidelines which will help with the implementation of future inspection systems.

CONTENTS

CHAPTER 1

INTRODUCTION

"Where there is no vision the people will perish:..."

Proverbs 29:18

1.1 INTRODUCTION

Thirty years ago computers were bulky contraptions that were applied to problems where large amounts of repetitive number crunching was involved. The concept of trying to make computers "see" was developed in the fifties when applications such as fingerprint recognition [36],[60], character recognition [68],[69],[81],[91], robot guidance and assembly [17],[18],[2], and industrial inspection [2],[14],[16],[9] were cited as the main areas for vision machines. Numerous "vision algorithms" suitable for computer representation were developed but it quickly came to light that the current generation of computers were too slow for practical use. Special computer architectures were developed in an attempt to reduce the execution time of these algorithms, many of which are currently still employed. However, these tend to be only suited for specific classes of algorithms. A concentrated effort has been applied to solving this problem but the solutions generally tend to be ad hoc. Probably more important is that, while algorithmic development is at a fairly advanced stage, architecture development has lagged behind the technology.

This thesis is concerned with the problems associated with machine vision with particular emphasis on the problems of industrial inspection. Electronic components are cheaper and faster than several years ago. This means that one can take advantage of some of the remarkable technological developments that have emerged in the last few years to produce affordable automated inspection systems. First, let us discuss some of the problems associated with machine vision in general.

1.2 THE PROBLEM WITH ROBOT VISION

"Humans are not logical". This familiar Vulcan proverb illuminates one of the issues frustrating computer scientists. While computers out-perform the human brain in solving certain classes of mathematical and logical problems, they appear inadequate for other tasks that humans can do instantly such as pattern recognition. Even supercomputers with subnanosecond gate delays get bogged down on true real-time vision tasks.

The reason for this lies in the enormous amount of computation involved. The human eye has the equivalent resolution of a computer image of about $10^4 x 10^4$ picture elements (pixels) [53]. Humans can recognise a variety of objects in a room in a matter of milliseconds. For a computer to achieve the same capabilities, a single processor working at over 10^{15}MHz would be required based on these figures. Today's processors run at speeds in the region of 25MHz meaning that several million orders of magnitude speed up is required to achieve true real-time pattern recognition on the same tasks.

A study into how the brain works reveals that the response time of the neuron is in the millisecond range [127]. It would therefore appear that faster processors are not the solution if they are to achieve the same tasks that humans can do. Making computers faster would only mean

that they would excel at the tasks on which today's computers already perform well, and it would only come a small fraction closer to defeating the problems associated with real-time pattern recognition. So how can we achieve it? The answer is vital for our future as an industrial nation - not least in the area of automated inspection to be covered in later chapters of this thesis.

1.3 ACHIEVING REAL-TIME PATTERN RECOGNITION

The answer almost certainly lies in the neural-net model based on the interconnection pattern of neurons observed in the brain. The hallmark of the neural-net is massive parallelism and high interconnectivity between a large number of relatively slow and simple processors (neurons) [127]. The general consensus that is emerging is that the information stored in a neural processor is distributed among the various nodes and their connections rather than being at discrete spatial memory locations as in the computer representation of an image. When a neuron receives a message, it immediately transmits it to the other neurons connected to it.

The obvious approach would therefore be to design a computer architecture with a large number of autonomous processors and high connectivity. However, such designs are generally expensive and often introduce data bottlenecks because of processor autonomy and their connectivity arrangements. As with any multiprocessor system, efficient use of processing resources is important. The computational load in the human brain is evenly distributed between communication and decision making so, at any given time, a substantial fraction of the decision making units are performing meaningful computation [127]. This is not always the case with autonomous machines and tends to be program specific. Architectures designed to eliminate data bottlenecks and obtain 100% utilisation of the available processing resources have

nonautonomous processors and only have local connectivity between neighbouring processors (see Chapter 7). They thus only contain information local to a specific part of the image and are hence inadequate for certain classes of pattern recognition algorithms that would otherwise be beneficial to the solution of a problem.

1.3.1 Programming Problems

The problems with both of the above approaches is the programming problem. Autonomous processors can invoke data bottlenecks if processors frequently need to access shared memory. Where one processor takes longer to run, others may lie idle making inefficient use of processing power and resources. Both of these problems occur through inefficient partitioning of an algorithm. For efficient use of processing power and resources, an algorithm must be partitionable into N independent processes for N processors. In general, this is not possible. Nonautonomous processors share the same amount of processing and programming is made easier; however, sequential algorithms cannot be executed efficiently as they require processor autonomy and access to global information. This has meant that ad hoc implementations of algorithms have been typical in past work.

1.4 THE PROBLEMS WITH COMPUTERS

Computers lend themselves to solving problems that by nature are structured in such a way that they use algorithms having many short steps. Humans do not recognise scenes by sequential steps but rather by a process of global associations [127], processing all the received data simultaneously. To expect a computer to be able to approximate this process in step-by-step algorithms seems unrealistic. Recognising objects in an image is thus inherently difficult for a computer. For instance, take the example of locating a blue mini in a car park with a

given registration number, bearing in mind that a computer has to approach this problem in a sequential manner.

Complications initially arise for the recognition of the registration number - for instance, broken or occluded letters present a problem. A much wider problem exists when it is required to recognise the car. The most obvious approach is to store a picture of the car in the computer's memory then, for every car encountered, match it with the stored picture on a point-by-point basis. However, various factors need to be considered. Many pictures would have to be stored in order to take into account all possible different lighting conditions and the different orientations of the car. In fact, the number of pictures to be stored would be exceptionally large. Also, since the car cannot be guaranteed to be in the same position in the image, each stored picture would have to be applied to every point in the image until a match was found. This would clearly take such an enormous amount of time that this approach has limited practical applications. However, the same sort of problems arise for relatively "simple" tasks such as those associated with industrial inspection where an object has to be undergo detailed scrutiny in typically one-tenth of a second.

1.5 SO WHERE DO WE GO FROM HERE?

It is postulated that the human brain has about 10^{10} neurons and each neuron may be connected to up to 10,000 other neurons [127]. Such a system is beyond the scope of this thesis; however, by applying the same principles as the neural-net architecture on a smaller scale in that the information is fairly evenly distributed among the processors and each processor has access contention free access to the same information when required, this would solve many of the problems associated with today's architectures, i.e. efficient use of processing resources and execution of parallel and sequential algorithms.

Current vision systems are usually employed as inspection machines for recognising and inspecting one particular product. More complex systems involved in the mechanical assembly of machinery involves recognition of several, well defined objects. However, because of the cost involved and the amount of processing time permitted by industrial requirements, these machines have not progressed greatly and are limited to simple tasks involving two dimensional data. If such machines could be made to execute complex pattern recognition tasks in real-time while remaining cost-effective and without the burden of having to match the algorithm to the architecture, one could consider developing practical inspection systems that could perform more complex intelligent vision tasks such as gauging three-dimensional products in real-time. The recognition stage alone for 3-D inspection currently takes several minutes on today's processors. Such systems could in principle be made much more flexible and robust.

1.6 THE AIM OF THIS THESIS

The need to achieve real-time recognition is evident from the last two sections. Achieving greater processing power would mean that computers could perform more complex decisions on the basis of visual inspection. This work attempts to develop the interprocessor communication problem with the aim of eliminating data bottlenecks traditionally associated with multiprocessor systems while maintaining processor autonomy. However, the cost of such a system must also be considered. A bit-slice processor called SIP (Chapter 5) is described which offers a cost effective solution for high speed processing. This processor is used to investigate the communication problem and architectures based on the neural-net model, i.e. the information is evenly distributed among the processors and each processor has contention free access to the same information when required. This

represents a radical departure from traditional digital computer architectures.

Such systems are not beneficial unless they are accompanied by efficient, robust and reliable algorithms. This means that they should be tolerant to effects such as noise and incomplete data. Algorithms suitable for these purposes are cited, discussed and it is shown how they may be implemented with minimum programming effort on the SIP system. This thesis emphasises the industrial vision area as this is currently a practically useful application for vision systems.

1.7 OVERVIEW OF FOLLOWING CHAPTERS

The work is divided into three sections. Part 1 (Chapters 2-4) is concerned with the concepts of designing efficient and robust algorithms suitable for industrial applications. Part 2 (Chapters 5-6) is concerned with the execution of such algorithms in real-time and the design and development of a high speed, low cost bit-slice processor is described. The concept of combining a parallel processor with a sequential processor is proposed and an investigation into its feasibility is undertaken. Part 3 (Chapters 7-8) is concerned with an investigation into architectures that have been developed for image processing. This work, building on the results from Part 1 and Part 2 arrives at an architecture that reduces processor bottlenecks traditionally associated with interprocessor communication: its study entails an investigation of the origins of data bottlenecks.

Chapter 2 - Reviews current and past work in the fields of image processing, image analysis and pattern recognition. An attempt to highlight the topics relevant to real-time work and robustness is undertaken and shows how they can be applied to real-world problems.

Chapter 3 - Describes two industrial algorithms. These are "typical" algorithms in that they both contain sequential and parallel processes. This is emphasised as the results from these algorithms are used later in the thesis.

Chapter 4 - Describes the sequential implementation of the industrial algorithms given in Chapter 3. These results are compared and a discussion on the usefulness of sequential and parallel processors in an industrial environment is undertaken. This attempts to highlight the areas where these processors would be suited, bearing in mind the applications and the cost-effectiveness in each case.

Chapter 5 - Introduces the idea of microcode and microcoded processors A low cost, high-speed processor is described along with an accompanying assembler.

Chapter 6 - Describes the Linear Array Processor (LAP) and is used to investigate a parallel/sequential processor architecture in conjunction with SIP (Chapter 5). This arrangement is necessary in order to execute general purpose algorithms (algorithms consisting of parallel and sequential tasks) efficiently. Machine performance is also discussed particularly in relation to SIP and the LAP.

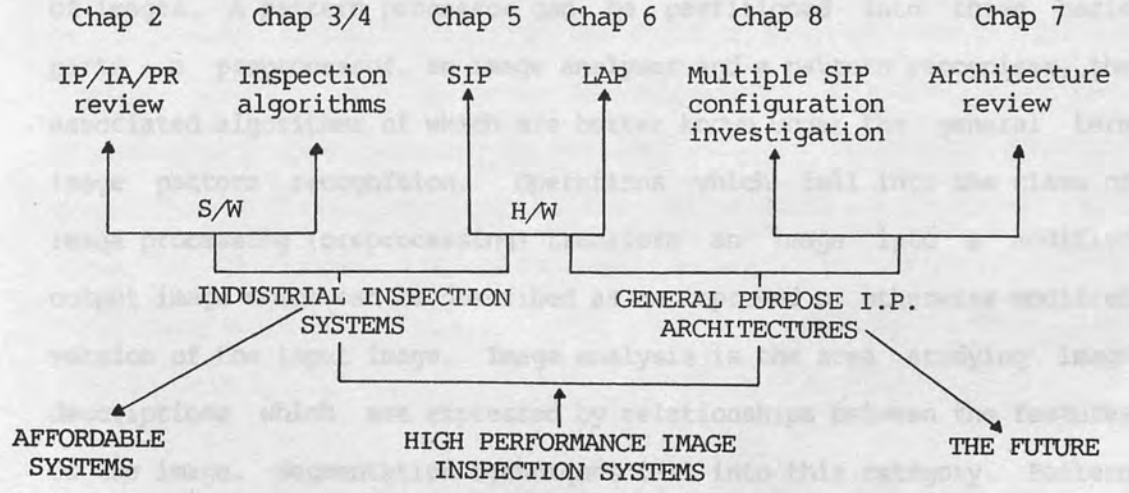
Chapter 7 - Reviews current architectures that have been used for image pattern recognition. Methods that have been used to achieve a higher instruction throughput in the past are discussed. Current trends towards architectures for image processing are highlighted: included is a discussion of the usefulness for general purpose applications. Tradeoffs are discussed

with regard to speed and cost.

Chapter 8 - Discusses the design of a novel architecture that attempts to reduce data bottlenecks commonly associated with multiple autonomous processor architectures. The methodology for reducing the bottlenecks is discussed and proposes a system which increases performance approximately linearly as N increases. This is based on the neural-net model described above.

Chapter 9 - Concludes the thesis. Conclusions and suggestions for future work are proposed.

With this wide scope ranging over the whole of vision and real-time cost-effective architectures, a fair amount of review material is required: this is provided in Chapters 2 and 7. The remaining chapters (Chapters 3,4,5,6 and 8) all describe new work. Below is a map showing how the chapters are "linked" together. This provides an overall view of this thesis.



CHAPTER 2

THE REPRESENTATION OF REAL-TIME ALGORITHMS

"Experience is the name everyone gives to their mistakes"

Lady Windermere's Fan

2.1 INTRODUCTION

Image recognition has several applications in the real-world including document processing (reading of printed or hand written characters), industrial automation (inspection of products and robotic assembly), medicine and biology (blood cell counting, tumour detection) and remote sensing (environmental monitoring, metrology) [103].

The general goal of a pattern processor is to generate descriptions of images and relate those descriptions to models characterising classes of images. A pattern processor can be partitioned into three basic parts - a preprocessor, an image analyser and a pattern recogniser, the associated algorithms of which are better known under the general term image pattern recognition. Operations which fall into the class of image processing (preprocessing) transform an image into a modified output image which can be described as an improved or otherwise modified version of the input image. Image analysis is the area studying image descriptions which are expressed by relationships between the features of the image. Segmentation techniques fall into this category. Pattern

recognition is primarily concerned with the description and classification of measurements taken from the image analysis section.

In any application area, the stages in the analysis process are similar. An image is initially preprocessed, for example, to standardize the grey-level, to remove noise or to deblur it. Segmentation often follows to partition the image into regions; the result is then passed to the pattern processor to analyse these regions and determine the properties of the image. If we have a model that describes these properties then we can recognise the image as belonging to a certain class - hence recognition is performed.

This chapter reviews some of the areas that have been developed and used successfully in image processing, image analysis and pattern recognition. However, there also exists a class of operations that locates and recognises specific features of an image, e.g. the centres of circles. Strictly speaking, this comes under the heading of image analysis; however, it deserves special attention as it is particularly relevant to industrial applications and will be discussed under the separate heading of "shape analysis and feature extraction". Throughout this chapter, those areas particularly relevant to industrial applications are emphasised and will be discussed in detail. Clearly for space, not all topics can be covered. However, an attempt is made to highlight those topics from the field of image pattern recognition that are particularly relevant for real-world problems, and short discussions follow many of the sections in order to cite the practical usefulness of these operations. First, let us consider the three basic types of operations that constitute the entire range of picture processing functions.

2.2 SEQUENTIAL AND PARALLEL FUNCTIONS

Picture processing operations fall into three categories:

1. Single point operations - every point in the resultant image plane P' is a function of its equivalent point in P. These are restricted to the most primitive class of operations such as thresholding and grey-scale modification as they are not dependent on the semantic content of the image.

2. Neighbourhood operations - every point in P' is a function of its equivalent point in P and its neighbours. This is probably the most common type of operation as it is context dependent. Examples are edge detectors, smoothing algorithms and thinning operations.

3. Distant neighbour operations - every point in P' is the result of its equivalent point in P and any or all of the points in P. An example is the Fourier transform. Because of the amount of computation frequently required by these types of operations, they are not suited to real-time image analysis tasks.

The second and third classes of operation are usually classed as parallel functions, i.e. a point in the original image and its neighbours are used to yield a result for the equivalent point in a new image. Because the order in which all the points in the image are accessed is immaterial, they can be considered as being operated on in parallel. In other words, the same operation can be applied to all pixels simultaneously. Conversely, sequential functions depend at each stage on the result from previous operations. Therefore, as a result is evaluated, it is written back into the current location in the original image. In this case the order of evaluation is important.

2.3 GREY LEVEL VS BINARY PROCESSING

All operations described above can be applied to both grey-level and binary pictures. Grey-level pictures allow any point in an image to be one of (usually) 256 different levels (zero representing black and 255 representing white) while binary images only consist of two values, which may be represented as one and zero. The choice of whether grey-level or binary images are used is application dependent. For instance, textural information processing is not usually carried out for binary images while chain coding and thinning are generally restricted to binary images (except see [87] and Chapter 4). Environmental conditions must also be considered, for instance, thresholding an image to extract edges where edges are not clearly defined from the background is nontrivial; however, when lighting and other conditions can be controlled, processing can be often kept to a minimum by thresholding.

A traditional pattern processing machine consists of a preprocessor, an image analyser and a pattern recogniser. A pattern processing operation will usually fall into one of the above three categories. However, image processing and pattern recognition are fields that developed rather separately [74] and therefore there are some areas that cover both topics. For instance, image processing includes not only coding, filtering and enhancement but also analysis and recognition of images. On the other hand, pattern recognition consists not only of feature extraction and classification but also preprocessing of patterns. For example, consider optical character recognition (OCR). The characters generally have to be preprocessed before analysis can proceed; however, preprocessing consists of segmenting and thinning the characters - operations that come under the heading of image analysis.

The algorithms related to the fields of image processing, image analysis, feature extraction and pattern recognition will now be discussed under their respective headings. Particular emphasis will be placed on the use of each topic in industrial inspection, where the performance of an algorithm is critical.

2.4 IMAGE PROCESSING

An image can be seriously degraded by effects such as illumination (shadows), reflectivity (glints), noise (general camera interference) and blur. Image processing (or preprocessing) is frequently concerned with the transformation of images such that the output image is an improved or otherwise modified version of the input image. This is important if we want to classify pixels based on their grey-level values. In this section we will discuss several preprocessing operations that can be applied in order to improve the quality of an image from the four most frequently occurring picture degradation effects: nonuniform lighting, low contrast, blurred, and noisy images.

2.4.1 Grey-level Correction

The brightness of a point in a scene (f) is effected by several factors including the illumination (i) and the reflectivity (r). However, it is often the case that i varies slowly across an image producing a nonuniformly lit scene. We can write the effects of this in the form $f(x,y)=i(x,y)*r(x,y)$. By rewriting this in the additive form $\log(f)=\log(i)+\log(r)$, using high emphasis spatial frequency filtering and then taking the antilog, the effects of i can be reduced.

2.4.2 Grey-scale Transformations

A low contrast image is often the result of low lighting

conditions. We can increase the contrast by adjusting the grey-levels by an appropriate transform which can be expressed in the form $z'=h(z)$ where z' and z are the new and old grey-levels respectively. If we consider that a range of pixel intensities that occur frequently lies in a range R while the rest lie outside R (which is the case in low contrast images), then we can stretch R and compress the rest of the scale such that the full bandwidth of the grey-scale range is used.

2.4.3 Sharpening

Blurring can often arise from an incorrectly focussed camera or through the effects of motion. The outcome of this is that high spatial frequencies are weakened more than the low ones. Sharpening is the process of emphasising the high spatial frequencies by filtering; however, this cannot be done indiscriminately since noise is usually stronger than the image signal at high frequencies. A simple way of sharpening the image (to a first approximation) is to apply the Laplacian operator:

$$\nabla^2 f = (\partial^2 f/\partial x^2) + (\partial^2 f/\partial y^2)$$

and subtract a multiple of this from the blurred image. This can be explained in the following way. The Laplacian is proportional to f-f' where f is the original image and f' is the blurred version. Now in f', high spatial frequencies have been weakened more than low ones. Hence, when we subtract f' from f, the low frequencies in f are more or less cancelled out while the high ones remain relatively intact. Thus, when we add a multiple of f-f' to f, we are boosting the high frequencies while leaving the low ones relatively unaffected.

2.4.4 Smoothing

Perhaps the most frequently occurring source of image degradation is the presence of noise. If the noise can be distinguished from the signal then it becomes easy to remove. An example of this is isolated dots in a binary image, commonly referred to as "salt-and-pepper" noise. An ideal image will not, by definition, contain noise, so application of a simple operation, e.g. the elimination of a white point if it is surrounded by black points (and vice-versa) will often suffice. Grey-level smoothing can be carried out by replacing each pixel with the average of its neighbours. The generalised cases of both operations are given below.

$$P' = \frac{1}{n} \sum_{j=-nn}^{nn} \sum_{i=-nn}^{nn} f(x+i,y+j) \qquad \text{Grey level case}$$

$$P' = \begin{cases} 0 & \text{if } \sum_{j=-nn}^{nn} \sum_{i=-nn}^{nn} f(x+i,y+j) < T \\ 1 & \text{otherwise} \end{cases} \qquad \text{Binary case}$$

where x and y are the (x,y) coordinates at that point in the image, n is the number of pixels in the NxN window, nn is (N-1)/2 and T is an arbitrary threshold, depending on the level of smoothing required. However, this strategy for smoothing grey-level images is well-known to have the effect of blurring the edges. This is undesirable as it masks much of the information at the edges - contrary to the reason for applying the preprocessing operations in the first place. To avoid this, several schemes have been considered, two of which have been proposed by Rosenfeld [102].

1. Average each pixel only with those neighbours whose grey-levels are closest to that of the pixel.

2. Perform edge detection[1] at every point in the image. If an edge is present then average only in the direction along the edge or only with those neighbours on the same side of the edge as the current point.

These methods are known under the heading of selective averaging. We give weights to the neighbours, chosen in such a way that neighbours belonging to the same region as the given pixel have high weights. Both of these schemes can be iterated if desired to increase the degree of smoothing. The disadvantage with these techniques is that they are computationally expensive. However, in a later paper, Davis and Rosenfeld [30] produced a technique called the K-nearest neighbour method which produced a trade-off in computation and enhancement power with both schemes mentioned above.

Another alternative for noise removal is to apply a median filter, i.e. replace each point with the median intensity value of its neighbours. This has the advantage that the edges are not blurred although again, this is computationally expensive (but see [20] for a slightly improved version).

2.4.5 Advantages and Disadvantages of Image Processing

Application of such steps as the median filter and selective averaging are, in general, computationally expensive, making them less appealing for real-time work. Ultimately, the need to preprocess an image depends on the amount of control over the lighting of the scene. For instance, inspection of industrial objects is often carried out

---

[1] As edge detection falls into the class of image analysis topics, this is a case where an image analysis topic falls into the class of image processing.

under controllable lighting conditions. Thus, in the majority of cases, the effects described above can be avoided by suitably adjusting the lighting and hence eliminating the need to preprocess the image. This is highly advantageous as the overall execution time of the algorithm will be reduced.

Preprocessing an image has other advantages that are not immediately obvious. For instance, when thresholding an image, edges of objects can become broken because of an incorrectly chosen threshold. Thus, operations such as tracking algorithms will fail to track around the complete edge. Smoothing an image can often join up the breaks. However, the disadvantage with this is that if the object in question is to be analysed for defects, these breaks may actually be real defects and could go unnoticed.

2.4.6 Brief Summary

This section has discussed several image processing algorithms. These transform images into other images such that the output image is a modified version of the input. The result of this stage is usually passed to the image analysis stage where the pixels can then be classified. Because the majority of image analysis operations classify pixels on their grey-level values, one must be careful when applying preprocessing algorithms as they change the grey-level values.

2.5 IMAGE ANALYSIS

Image analysis is the area studying image descriptions which are expressed in the form of relationships between, and properties of, image parts. This section discusses some of the types of operations that fall into the class of image analysis operations.

2.5.1 Segmentation of Images

Probably one of the most important classes of techniques used in image analysis is that of segmentation. This can be described as the division of an image into regions of homogeneity based on the properties associated with each pixel, such as the local texture or the gradient. Segmentation is a critical component in the pattern processing stage because errors here might propagate to the feature extraction and classification stages. It is important to note that if an image can be classified, segmentation is the first stage at which a description or classification can be attempted.

Segmentation techniques can be categorised into three classes: (1) edge detection, (2) thresholding and clustering, and (3) region extraction. Two basic points arise that are common to all three classes:

1. Every point must be in a region - this means that the segmentation algorithm must process all points in an image.

2. Regions must be contiguous.

Segmentation algorithms have historically been somewhat ad hoc [51]. There are no general algorithms for all images because a two-dimensional image can represent an effectively infinite number of possibilities. To build a general image understanding system, the computer would require a vast amount of knowledge. For this reason, a priori contextual knowledge is usually incorporated into segmentation algorithms. For example, take the famous image of a dalmatian dog. Without a priori knowledge this appears to be noise to many human observers. However, given the knowledge that a dalmation dog is present in the picture, the dog becomes instantly recognisable.

Most segmentation algorithms are based either on the concepts of discontinuity (e.g. edge detection), or similarity (e.g. thresholding). The next few sections look at the case of discontinuity. The case of similarity follows in Section 2.5.3.

2.5.2 Edge Detectors

Edge detection techniques are important because, in many cases, most of the useful information in an image lies at the boundaries between different regions. (This would therefore appear an appropriate subject to cover with regard to industrial inspection.) Since high spatial frequencies are associated with sharp changes in intensity, one can extract the edges by performing high-pass filtering, i.e. take the Fourier transform of the image, multiply this by a linear spatial frequency filter and take the inverse transform. The main disadvantage with this approach is that much of the localised edge information is lost.

The class of edge detectors that extract local information from the edge points fall into two categories - parallel and sequential edge detectors. Parallel edge detectors are those whose operations can, in principle, be applied to every point in the image simultaneously while sequential edge detectors depend on the results of previous operations. Both of these will be covered. We will first describe the two types of edge detector that exist for the parallel case: the differential gradient and the template match edge detectors.

2.5.2.1 Differential Gradient Edge Detectors -

Most of the edge detection techniques examine the grey-level intensity changes within a local neighbourhood. The differential gradient edge detectors determine the magnitude and direction of the

intensity gradient at each point by calculating the x and y derivatives. A high magnitude results where there is an abrupt change in grey-level (an edge) and a low magnitude where there is little change in grey-level (no edge). (A threshold is usually applied to the magnitude to suppress the effects of noise.) This method has the particular advantage that, since the directions of the edges are readily available, only edges with a preferential direction, e.g. horizontal, may be detected. The simplest case of an edge detector is

$$\frac{\partial f}{\partial x} \sim f(x+1,y) - f(x,y) = \Delta_x f(x,y)$$

$$\frac{\partial f}{\partial y} \sim f(x,y+1) - f(x,y) = \Delta_y f(x,y)$$

thus, the magnitude of the gradient is approximately

$$|\nabla f(x,y)| \sim \sqrt{([\Delta_x f(x,y)]^2 + [\Delta_y f(x,y)]^2)}$$

and the direction is given by

$$\theta = \tan^{-1}[\Delta_y f(x,y)/\Delta_x f(x,y)]$$

where $f(x,y)$ is the intensity value of the pixel at the current position (x,y). The information derived from these values can then be analysed to give some insight into the properties of the image; for example, if we accumulated all edge points with their corresponding angles, large accumulations would correspond to the most prominent angles in the scene [34].

Another well known edge detector is the Robert's cross operator [100]. This estimates the derivatives diagonally and is equivalent to a linear fit over a 2x2 neighbourhood, i.e.

$$\Delta_1 = f(x,y) - f(x+1,y+1)$$

$$\Delta_2 = f(x+1,y) - f(x,y+1)$$

again, the magnitude is given by

$$M = \sqrt{[\Delta_1^2 + \Delta_2^2]}$$

Note that the calculation of the magnitude (as with many others) tends to be computationally expensive, involving two multiplications and a square root calculation. A simpler approximation is

$$M = |\Delta_1| + |\Delta_2|$$

This is not as accurate but is adequate for many purposes. An alternative to the gradient estimates within a 2x2 window is the maximum difference operator [53]. This finds the maximum and minimum values within the four-pixel group and subtracts the minimum from the maximum. However, this tends to be very sensitive to noise and could give misleading results.

The above approaches all use 2x2 windows. These are the least computationally expensive but because the window is relatively small, they require that there be distinct changes in intensity between two adjacent points. Thus, only very sharp edges with high contrast between the surfaces which form the edges will be detected while ill-defined edges (edges formed by a gradual change in intensity across the edge) will not be detected. The result is therefore quite susceptible to noise. An alternative is to use a 3x3 neighbourhood. Examples of such edge detectors are Prewitt [96], Frei-Chen [50] and the Sobel [33], which are known as the "fast edge detection operators". These masks are given in Figures 2.1a to 2.1c, the differences being the weights assigned to the elements in the 3x3 window. The Prewitt mask smooths the gradient (average of both sides of the window) while the Frei-Chen mask is based on "isotropic weighting functions". This is intuitively better as the weightings reflect that the corner neighbours within the

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

Figure 2.1a Prewitt Masks

$$\begin{pmatrix} -1 & 0 & 1 \\ -\sqrt{2} & 0 & \sqrt{2} \\ -1 & 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & \sqrt{2} & 1 \\ 0 & 0 & 0 \\ -1 & -\sqrt{2} & -1 \end{pmatrix}$$

Figure 2.1b Frei-Chen Masks

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

Figure 2.1c Sobel Masks

3x3 window are $\sqrt{2}$ further from the centre pixel than the other neighbours. The Sobel operator depicted in Figure 2.1c organises its neighbourhood pixel weightings such that it reflects the proportion of a circle which is present within the neighbour [26].

More complex edge detectors such as the Marr-Hildreth [75] "difference of gaussian" and that developed by Hueckel [58],[59] use larger masks. Hueckel's edge detector used a 52 element mask which is arranged such that it approximates a disk-like shape as shown in Figure 2.2. This has the advantage that it removes most of the local noise; however, the disadvantage with both of these is that they are computationally expensive and therefore less suited for industrial inspection requirements.

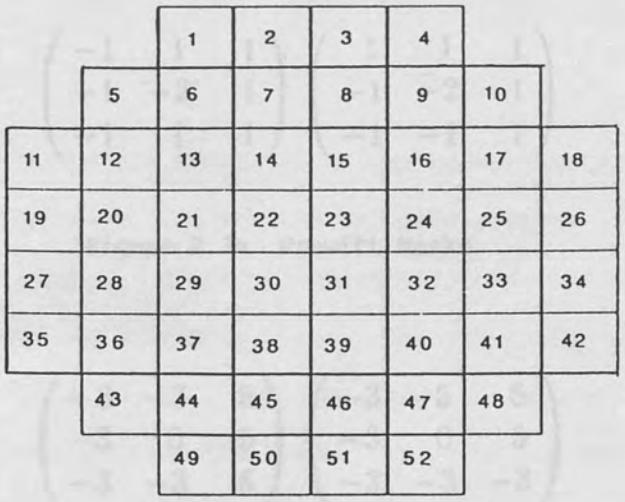


Figure 2.2 Heuckel's mask

2.5.2.2 Template Match Edge Detectors -

Another approach to edge detection employs template matching masks. The edges are found by applying a series of masks (each of which represents an ideal edge) to every point in the image. (In general, eight masks are used, one for each of the eight main compass directions.) The angle is determined from the template which gives the maximum response. Examples of template masks are: Prewitt [96], Kirsch [65], and the "3-level" and "5-level" Robinson masks [28]. Two of the eight possible compass directions for these are given in Figures 2.3a to 2.3d.

The main disadvantage with the template matching approach is that it is not as accurate as the differential edge approach, either for estimating magnitudes or for determining orientations. Also, it is computationally quite expensive since all eight masks have to be applied to every point in the image.

$$\begin{pmatrix} -1 & 1 & 1 \\ -1 & -2 & 1 \\ -1 & 1 & 1 \end{pmatrix} \begin{pmatrix} 1 & 1 & 1 \\ -1 & -2 & 1 \\ -1 & -1 & 1 \end{pmatrix}$$

Figure 2.3a Prewitt Masks

$$\begin{pmatrix} -3 & -3 & 5 \\ -3 & 0 & 5 \\ -3 & -3 & 5 \end{pmatrix} \begin{pmatrix} -3 & 5 & 5 \\ -3 & 0 & 5 \\ -3 & -3 & -3 \end{pmatrix}$$

Figure 2.3b Kirsch Masks

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

Figure 2.3c 3-level Robinson masks

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

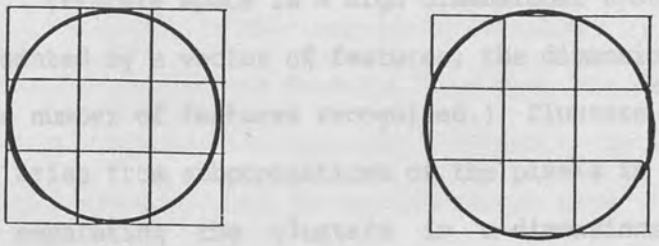
Figure 2.3d 5-level Robinson masks

The masks for many differential gradient and template matching operators appeared to be ad-hoc and Davies attempted to overcome this problem in each case [26],[28].

2.5.2.3 Analysis of Parallel Edge Detectors -

Abdou and Pratt [1] investigated the probabilities of "true edge" and "false edge" detection for the 2x2 differential gradient operators, the 3x3 differential gradient operators and the template match operators. Not surprisingly, they showed that the 3x3 operators detected more "true edges" than the 2x2 operators. However, their results also revealed that the Prewitt operator was better for detecting vertical edges than the Sobel, but the Sobel was better for detecting diagonal edges. With regard to the template operators, they showed that the 3-level and 5-level Robinson operators exhibited almost identical performance while being superior to the Kirsch operator. Finally, they also showed that the Sobel and Prewitt operators performed slightly better than the 3-level and 5-level template masks.

Lee [71] developed a method for improving the execution time of the Sobel operator at the expense of storage. By retaining the results from the previous line, a factor two speedup could be achieved. In a discussion on the detailed implementation of circular operators, Davies [26] showed that the Sobel operator is near optimal. It is known [26],[53] that a circle (rather than a 3x3 digitised approximation) will give the maximum response to any image processing algorithm. However, whether the circle should be drawn within the 3x3 region as depicted in Figure 2.4a or outside as in Figure 2.4b is a relevant question. Davies investigated how the angular accuracy of circular differential gradient operators depends on the radius of the circular neighbourhood. The problem with Figure 2.4b is that the pixels outside the 3x3 neighbourhood must also be included. This is obviously undesirable as it adds to the computation time and makes life very awkward in general. Davies went on to show that the most accurate version, lies somewhere between the two cases but lies much closer to the former case (Figure 2.4a) than the latter (Figure 2.4b).



(a) (b)

Figure 2.4 3x3 windows for the circular operators

2.5.2.4 Sequential Edge Detectors -

All the above algorithms are classed as parallel edge detectors. Thus, edge classification is based only on the grey level values of a points' neighbours. For sequential edge detectors, the result at a point is contingent upon the results of previous operations [76]. This class of edge detectors are of less interest to us as they tend to be computationally expensive and will not be discussed any further.

So far, we have discussed the segmentation of a scene by classifying pixels as edge/non-edge. Quite often, it is necessary to differentiate between several regions, for instance, when distinguishing between regions in a satellite image, we may want to classify pixels according to their texture. In this case, we need to cluster the pixels, classify them and assign them values in the original picture. This is discussed next.

2.5.3 Thresholding and Clustering

Each pixel in an image generally has one or more features associated with it. The "first-order" features are grey level and spatial coordinates while its "higher order" features includes such items as gradient and texture. These features can be mapped to a point

in feature space. (Feature space is a high dimensional space where each point is represented by a vector of features, the dimensionality being equivalent to the number of features recognised.) Clusters in feature space therefore arise from subpopulations of the pixels in the original image space. By separating the clusters in n-dimensions such that feature space is partitioned into a number of mutually exclusive and contiguous regions, the points in feature space can be mapped back to the original spatial domain to produce a segmented picture. We now give some examples of this.

The most widely used and simplest technique of clustering is that of thresholding. The features in this case are based only on the grey-level intensity of the pixels. Clusters are formed which represent dark (generally objects) and light (generally background) regions. By classifying the clusters as object and background respectively and assigning them the appropriate label in the original image, objects become easily distinguished from the background in high contrast images. There are a number of schemes for separating the clusters in this case, most of which are based on the grey-level histogram for determining the threshold. A dark object on a white background will produce a well defined bimodal histogram (Figure 2.5b). The threshold is chosen at the valley of the two peaks such that the clusters are partitioned. The results are then mapped back to the original image, to produce the segmented image. This has many disadvantages, the main one being that high contrast images are needed between the "object" and the "background" and noise should be low - neither condition being accurately achievable in practice, particularly in an industrial environment. Low contrast images have the effect of merging the clusters together, hence making separation difficult.

Mapping an image into several different regions is basically a multidimensional extension to the concept of thresholding, i.e. several clusters are formed from a single feature. However, as the number of regions grows, it becomes increasingly difficult to partition the clusters using just one feature. We may therefore have to take into account multiple features (as in the case of multispectral images such as LANDSAT) in order to resolve the problem. As an example, consider the case of a dark object on a light background, i.e. a high contrast image. If we chose our feature as being based on the mean grey-level then the mean grey levels of the object will fall into the category S_0 and those of the background will fall into the category S_1. A straight threshold in this case will often suffice to segment the image as before.

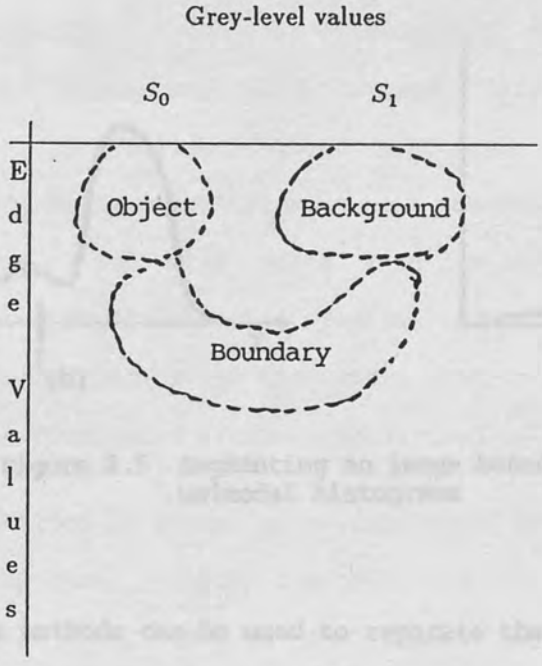


Figure 2.5a Clustering of pixels based on edge and grey-level values

Now consider the case that the edges of the objects are blurred. The grey-level values of the pixels at the edges will typically fall between S_0 and S_1, hence leaving them undefined and separation of the

clusters difficult. Panda and Rosenfeld [89] considered a two-dimensional feature space based on the values of the grey-level pixels and the magnitude of the gradients to solve this problem. Although the mean grey levels of the pixels at the edges will be undefined, the magnitudes of the gradient at these points will be high. Thus, by use of the two-dimensional feature space previously described, a map will be produced as in Figure 2.5a where the boundary points have a high edge value and fall between the regions S_0 and S_1 and the areas away from the edges fall distinctly into either region. This is a sort of trimodal histogram in two dimensions. In cases where the edge elements also become undefined, it may be necessary to adopt three or more features.

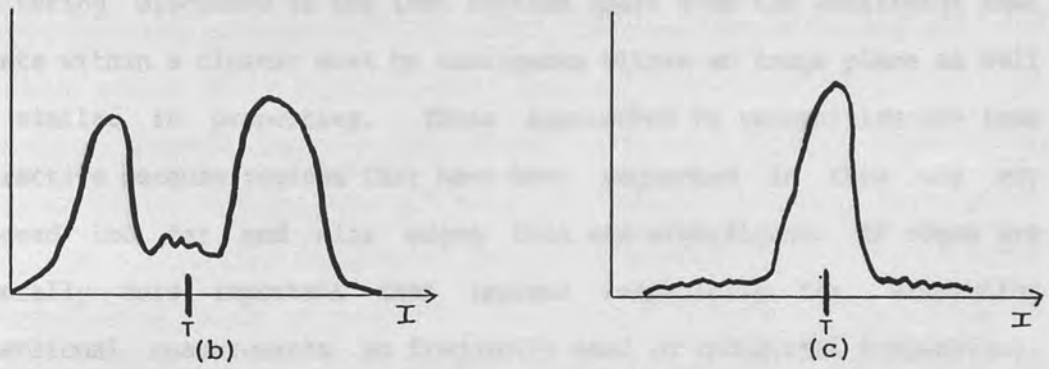


Figure 2.5 Segmenting an image based on bimodal and unimodal histograms

Various methods can be used to separate the clusters in this case. One such method is to choose the original bimodal histogram based on the grey-level values and choose a threshold as depicted in Figure 2.5b. Alternatively, one could consider a unimodal histogram based on the edge magnitudes and base the threshold on the mode as shown in Figure 2.5c - this corresponds to those points on the borders between object and background. The separation of clusters will be discussed in

more detail later in Section 2.7.1. First, we shall look at another form of segmentation based on similarity but which is different from the above method for reasons that will be explained.

2.5.4 Region Extraction

Two methods formally exist for extracting regions - region growing and region dividing. Region growing is effectively a bottom-up process where the region starts from a single point and "grows" by grouping all neighbouring points that possess a similar property, e.g. its grey-level value. On the other hand, region dividing (top-down process) initially treats the whole image as one region and decomposes it into regions that are again similar in nature. These two methods are similar to clustering discussed in the last section apart from the constraint that points within a cluster must be contiguous within an image plane as well as similar in properties. These approaches to recognition are less attractive because regions that have been segmented in this way may proceed too far and miss edges that are significant. As edges are generally more important than regions (especially for extracting dimensional measurements so frequently used in industrial inspection), edge detection techniques are more often used.

Edge detection is often a prerequisite for shape analysis and feature recognition. These are both used for the recognition and the inspection of objects. This is particularly important in industrial applications where it may be required to locate an object, recognise it and then scrutinise it for defects. The next section discusses several shape analysis and feature recognition techniques that have been developed which are advantageous for object recognition.

2.6 SHAPE ANALYSIS AND FEATURE RECOGNITION

Shape analysis is important for object recognition. After the edges of an object have been found (whether by thresholding or edge detection) it is usually necessary to examine the edges in order to get some sort of shape description, following which it can be recognised.

Techniques for shape analysis can be classed into information preserving and non-preserving techniques. Such methods are the transform schemes, spatial techniques and global shape analysis techniques. The two problems that frequently occur are dependency on scale and orientation. Common spatial techniques, i.e. working in the image space domain, are the boundary encoding schemes, the most popular of which is that developed by Freeman [45] known as the chain code. A shape analysis scheme using the chain code has been developed [49] and has been successfully employed in an industrial environment [19]. The transform schemes include the Hough transform - this is discussed in Section 2.6.4. Another approach uses the Fourier boundary descriptors [99]. This is easily implemented but lacks the ability to extract local information which is crucial for industrial inspection where small defects may need to be located. However, they are useful in object recognition as described in Chapter 4.

So far, we have only been concerned with the boundary of an object. An alternative approach is to examine its medial axis by thinning. This can often have distinct advantages over boundary algorithms which will be discussed in the next section.

2.6.1 Binary Thinning Algorithms

Other terms commonly associated with thinning are "skeletonisation" and "symmetric axis transformation". In general, thinning is restricted to binary images where the boundary of the shape to be thinned is

clearly defined; however, there have been attempts to thin grey-scale images (Section 2.6.3). For the moment, we will assume that the images are in a binary format. The purpose of thinning is to thin an object many pixels wide down to just a single pixel wide. There are several advantages to be gained from doing this:

1. The large reduction of volume of data resulting from thinning produces a significant improvement in the storage efficiency. This is particularly important for real-time applications as a significant amount of redundant information is removed. The processing time involved in analysis is (in principle) thus decreased.

2. It plays a particularly important role in optical character recognition (OCR) where, to some extent, thinning provides a unification of character shapes by reducing the effects of various types of fonts. It also helps in extracting the fundamental features of letters.

3. Thinning is a prerequisite for many shape analysis routines such as the chain code or Fourier descriptors as discussed in Chapter 4.

As all the critical information is contained within the skeleton for some industrial uses, thinning results in the elimination of a lot of redundant information. This makes it appealing for industrial use, particularly in OCR and post-edge detection techniques. Various techniques of thinning have been developed, all of which rely on the steady erosion of the boundaries while maintaining connectivity of the shape. The basic idea is to iteratively delete edge points while ensuring that (1) end points are not removed, (2) connectedness is maintained and (3) excessive erosion leading to eventual skeleton bias is not caused. One of the general problems that occurs in thinning is the need for a consistent definition of connectedness when a rectangular

tessellation is used. For instance, consider the box below:

```
1 0
0 1
```

Here, the 1's represent an object and the 0's represent the background. The natural interpretation of this is that the 1's represent a diagonal line of a skeleton bisecting two background areas. However, there is no reason why this should not be interpreted as the background bisecting the object and hence producing a discontinuity in the object. In other words, connectivity is not defined rigorously and consistently. This is known as the crossing paradox and has been discussed in detail by Rosenfeld [102]. There are two main types of connectedness that exist for rectangular tessellations - 4-connectivity and 8-connectivity. When an object is 4-connected, only the horizontal and vertical edge elements are allowed to be connected, while 8-connectivity allows the diagonals to also be connected. Thus, allowing the background to be 4-connected and the object 8-connected solves the paradox as the background diagonals cannot now be connected.

Thinning algorithms differ in the way they conduct their tests to meet the criteria, although there is usually a partial commonality between the approaches. As Davies and Plummer [23] pointed out, most authors give no standards for skeleton precision, and leave the definition of a skeleton undefined. For example,

1. Montanari [79] propagates wavefronts from the inside of the edge of the figure. The skeleton in this case is defined as the locus of the intersections of wavefronts from opposite sides.

2. Rosenfeld [92] defines a skeleton as being the shape formed from the centres of maximal discs placed within the object. From the skeleton, the original object can be reconstructed by retaining the radii of the maximal discs and simply re-drawing them.

A recent approach is that described by Zhang and Suen [130]. This consisted of only two subiterations - the first subiteration deleted the centre point from a 3x3 window if

(a) $3 \leq B(P_0) \leq 6$

(b) $A(P_0)=1$

(c) $P_3*P_1*P_7=0$

(d) $P_1*P_7*P_5=0$

where $A(P_0)$ is the number of 01 patterns in the ordered set $P_1,P_2,P_3,P_4...P_8$ and $B(P_0)$ is the number of non-zero neighbours in the set $P_1,P_2,P_3,P_4...P_8$ as shown in Figure 2.6. The second iteration is the same except for

(c) $P_3*P_1*P_5=0$

(d) $P_3*P_7*P_5=0$

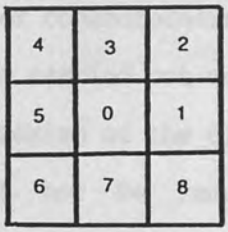


Figure 2.6 3x3 thinning window

The first sub-iteration removes the south-east boundary points while the second removes the north-west boundary points that do not contribute to the ideal skeleton. Condition (a) ensures that the end points of the skeleton are preserved and diagonal elements with a thickness of two will not disappear while condition (b) prevents the deletion of those points that lie between the end points of a skeleton line. Both of these conditions are shown in Figure 2.7. These steps are repeated until there is no further change. Naccache and Shinghal [82] reviewed 14 different thinning algorithms, the majority of which were similar to that just described. They found that the disadvantage with most of them

was the lack of reconstructability from the skeleton. They went on to derive a fast method of thinning that achieved reconstructability.

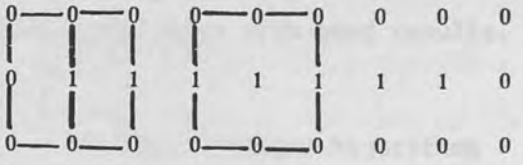


Figure 2.7 Preventing the deletion of end points

The problems with the majority of thinning algorithms is that no two skeletons are always identical for every possible shape. In attempt to set standards for the precision with which they could be performed, Davies and Plummer approached the subject systematically. Their method initially propagated the distance function throughout the shape. (The 8-connected definition of connectedness was chosen as this allowed the propagation function to be carried out more rapidly.) The second step was to mark all local maxima of the distance function points with the constraint that these must not be removed in the next step. The original figure was then "slimmed", i.e. thinned but with the constraint that those points marked in the last section were to remain. The steps were repeated until there was no change.

Although this may appear to be a computational burden on the algorithm, Davies and Plummer noted that the slim algorithm used could be a simple one as little care needs to be taken over the end-points, this being the main source of complexity in conventional thinning algorithms. Their definition of a skeleton is useful because strict connectivity is maintained, unlike the definition by Rosenfeld. An optional step proposed by Davies was the "purge" step that eliminated noise spurs. A decision on whether a point is noise is highly subjective and is inherently data and problem dependent. However, all

the relevant information is retained in the final skeleton so one can interpret those points with a distance function of (say) unity as noise. Stentiford and Mortimer [119] applied heuristics to avoid such effects as spurious tails and distortions with good results.

2.6.2 Disadvantages of Binary Thinning Algorithms

The binary thinning approach has some severe disadvantages. First, a simple threshold to achieve a binary image where edges are clearly distinguished from the background is by no means realistic. As Smith pointed out [117], thinning is frequently used in OCR but variations in paper colour and shade, print quality, contrast and lighting introduce severe difficulties in thresholding black and white images. This is particularly true for paper that contains lines, text and pictures, all with different levels of contrast, e.g. text printed in shaded boxes. When such cases arise, it rapidly becomes impossible to threshold effectively.

Unfortunately, even fast thinning algorithms run too slowly to be of any practical use without the use of parallel hardware. As shown by Naccache and Shinghal, a CDC 170-825 computer required 15-30 minutes to thin a page consisting of 60 lines with 80 characters per line and an average character size of 23x17 pixels. A visual inspection task that employs both thinning and template edge detection is described by Kaufmann, Medioni and Nevatia [62]. Their algorithm inspected PCB's designed for watches. The object of the algorithm was to detect broken PCB's and missing components. The steps consisted of edge detection, edge thinning, thresholding and approximation of edges by line segments. Although no times were given, this demonstrates a practical example of three of the techniques discussed so far (although one suspects that from previous attempts of thinning, this algorithm would probably take a relatively long time to execute).

2.6.3 Grey-scale Thinning Algorithms

As mentioned above, achieving a suitable threshold is often impossible. The aim of thinning a grey-scale image is to retain the connectivity of the original image while producing a result which is not determined wholly by the original outline of the image but which is sensitive to grey-level values and lies along darker ridges in the image. Thus, well-defined edges are no longer mandatory. One could suggest applying an edge detector and thresholding the magnitudes to produce a binary image, i.e. thresholding the edges. However, Paler and Kittler [87] stated that this had several disadvantages, namely that thresholding removes information concerning the position of the maximum of the edge magnitude and the methods are very sensitive to noise. Instead, they chose a method whereby, for each edge pixel found, two neighbouring pixels with an angle nearest to the perpendicular to the edge direction of the edge pixel were found. These angles were compared with the edge pixel and if they were both within a predefined angular tolerance, the edge magnitudes of the edge pixel and both neighbours were compared. If the magnitude of the edge pixel was less than either of the two neighbours then it was set to zero. The process was continued until all edge pixels had been processed.

Another technique was that described by Hilditch [54]. Here, a binary image is thinned in the usual way, i.e. by any of the methods described in the Section 2.6.1, except that the deletion of a point was also governed by the value of the equivalent point in the grey-level image. This method was generalised to the grey-scale picture; however, this required a definition of connectedness. The most common definition is that two pixels in a grey-level image are connected if there is a path joining them that contains no pixels lighter than both of them. A pixel is set equal to the lightest of its neighbours if it does not disconnect any of its neighbours. This condition proved to have been

too "strong" and a definition of the "strength" of connectedness had to be derived.

Another alternative to the above method is as follows [54]. Suppose that in the binary case $\{N_1\}$ is the set of neighbours (4- or 8-connected) which must have a value one and $\{N_0\}$ is the set of neighbours that must have a value zero. The grey-level equivalent is that $\{N_1\}$ is the set of neighbours which must have a value greater than or equal to that of the pixel under consideration and $\{N_0\}$ is the set of neighbours which must have a lower value. If these requirements are both satisfied then the pixel is thinned by setting its value to the darkest of the set of neighbours $\{N_0\}$.

As an epilogue to the thinning section, a novel application for thinning is that of palm reading [85]. Palms are in fact a better means of identification than fingerprints but the method is less practical. However, under controlled conditions it would be possible to detect the presence of the life, head, heart and fate line (as these are the most prominent lines on the palm), thin them to a single pixel width, then chain code the thinned lines for further analysis.

2.6.4 The Hough Transform

We shall now discuss the Hough transform. As we will see, this is a very useful transform as it can still work without a complete set of data. This is particularly useful where robust algorithms such as those employed in industrial recognition systems are required.

The classic Hough transform technique was originally developed by P.V.C Hough [57]. It constitutes a class of procedure for extracting analytically defined shapes. For example, consider the task of detecting collinear points, i.e. straight lines, in a grey-level image. The equation of a straight line is given by

$$y = ax + b$$

which has slope of a and intercept b. This can be transformed into Hough space by rearranging the equation into a form

$$b = -xa + y$$

Hough space being represented by the parameters a and b. This means that a point in x-y space is mapped into a line in Hough space with a slope of -x and an intercept y. The Hough transform accumulates lines in an accumulator array (conveniently represented by a two dimensional array) with slopes and intercepts corresponding to the (x,y) coordinates of the points. Thus, collinear points in image space correspond to lines in Hough space that intersect at exactly one location and peak locations in Hough space give the position and orientation of the line in image space.

Generally, Hough space can have any one of a number of parameterisations, the parameters being chosen on the information required from the image. Points in x-y space are accumulated in Hough space and the information required is derived either from the relationships between the accumulated points or the values of the peak heights. The advantage of this is that global information about the image can be read directly from Hough space when it would be difficult to obtain from the image in other ways. Another advantage is that it is reversible, so by applying the inverse function to the points in Hough space, the original image can in principle be reproduced. The two main advantages of the transform that make it suitable for industrial use are:

1. It is insensitive to noise. A suitable threshold for detecting high count cells (representing the geometric properties of the object) will eliminate cells with low counts arising from noise.

2. The transform will work even when the boundary is disconnected because of noise or occlusions. This is generally not true for other strategies which track edge elements, and makes it particularly useful for industrial analysis applications where objects may be overlapped or occluded.

In effect, each point "votes" on where a line exists and the accumulator "adds up the evidence". Hence, very little deterioration in performance need occur if some points on the boundary of an object are missing. This explains why the Hough transform is particularly robust.

Kimme et. al. [63] applied the transform for finding circles by using three parameters: two for the circle and one for the radius. This has been extended for finding parabolas (four parameters) and ellipses (five parameters) [116]. In a later paper, Sklansky applied these techniques for detecting the rib cage in chest radiographs [125]. When a rib is viewed on from an angle, the dorsal and ventral portions of the ribs appear parabolic. The Hough transform in this case was useful because, even if parts of the rib were occluded because of an overlapping tumour or a damaged rib, the model outline could still be determined. Further analysis revealed whether the rib was normal. Ballard [5] generalised the Hough transform for detecting arbitrary (analytic and nonanalytic) shapes.

The Hough transform has been applied to various practical applications such as optical character recognition (OCR) to recognise printed Hebrew characters [68]. All the characters in the Hebrew alphabet are composed of straight lines allowing the number of classes of characters to be significantly reduced. Other applications include 3-dimensional object recognition [115], data compression [106], determining the orientation of rectangular objects (Section 3.4), decomposition of polyhedral scenes [123] and locating straight-line edge

segments in outdoor scenes [34]. Probably one of the more interesting applications of the Hough transform is to correct for linear variation of background illumination in images [84]. The linear brightness distribution may be characterised by

$$f(x,y) = m_x x = m_y y + c$$

Application of the Hough transform means choosing an appropriate parameter space for m_x, m_y and c. An accumulator array contains the count of those points matching the variation defined by that particular m_x, m_y and c. Searching through the image space for the maximum value gives us the most likely brightness variation. Thus, determination of these areas makes feasible their suppression thereby restoring the image.

2.6.5 Template Matching

Another approach that is frequently used in industrial inspection is template matching. Here, a template of the pattern to be matched is compared with the current pattern in the image on a pixel-by-pixel basis. The percentage of right/wrong matches gives the degree of fit. If a large pattern exists then naturally, using a large template is computationally expensive. Alternatively, a small template can be used but several applications of different templates may be needed in order to match the feature. The disadvantage with template matching is that it is sensitive to variations in the lighting, reflectivity of the material, size and orientation of the object. In this case, a cross-correlation scheme is often employed in order to maximise some measure of the degree of match between the pattern and picture.

The template matching scheme has found its uses where these effects are generally not important, e.g. in industrial applications where the conditions can be controlled. They are often employed in binary images; for instance, a series of templates recognises removable points in binary thinning algorithms. A classic example is in the case of PCB track inspection [16] where the image can be suitably thresholded. Here, four templates are used (one for each 90° orientation) that represent "nicks" along the edges. These are depicted in Figure 2.8. Another example of locating PCB defects involves the use of a series of 5x5 templates [61] that describes the normal track of a perfect PCB. If this did not match then a fault locator algorithm was executed. A similar approach has also been applied to detect the defective portions of an IC mask [16]. Other applications include cosmetic inspection of jars, bottles, cans and correct labeling of drugs etc. [2]. Because of its triviality and easy implementation, template matching has been widely used in many applications, although these are very specific uses and always restricted to a 3x3 or at most a 5x5 neighbourhood for real-time work.

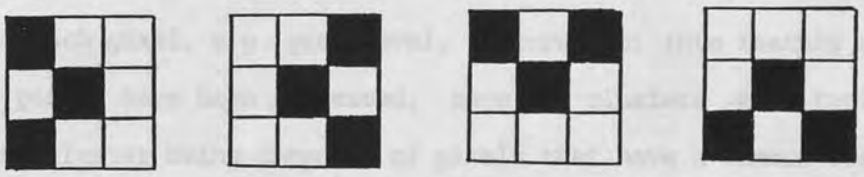


Figure 2.8 Templates for detecting 'nicks' on a PCB

2.7 PATTERN RECOGNITION

Image and feature analysis algorithms describe the features of the image as a parsable string of numbers. For example, this could be the edge coordinates, the chain code of the boundary of an object, or the coordinates of mapped pixels in a cluster. This string of numbers is input to an appropriate pattern recogniser which classifies the input

pattern - this is known as pattern recognition.

We can view the process of pattern recognition as the classification or the parsing process of the patterns that have so far been created. It can be divided into two main areas - statistical (decision-making) and syntactic (linguistic). However, a third category exists which is a hybrid between both methods. Hybrid methods are the first step towards an attempt to create a unifying theory in pattern recognition. Because of the size of the subject, each of these areas will only be briefly described. See [74] and [103] for further information.

2.7.1 Statistical Pattern Recognition

In statistical pattern recognition, the measurements taken from N features can be thought to represent an N dimensional vector space, whose coordinates correspond to the measurements taken. Two major decision making processes exist - clustering analysis and fuzzy set reasoning. As mentioned above, clusters are formed by mapping the feature of each pixel, e.g. grey-level, texture etc. into feature space. After all pixels have been processed, several clusters may typically exist, each cluster being composed of pixels that have a common feature. The concept of cluster analysis is to partition the clusters such that each pixel in the cluster can be assigned an appropriate code which is unique to that cluster and then re-mapped back into image space, hence producing a segmented image. However, problems can occur. For instance, consider Figure 2.9. The partitioning of these clusters ranges from "easy" to "difficult". In general, Figure 2.9a is the exception rather than the rule.

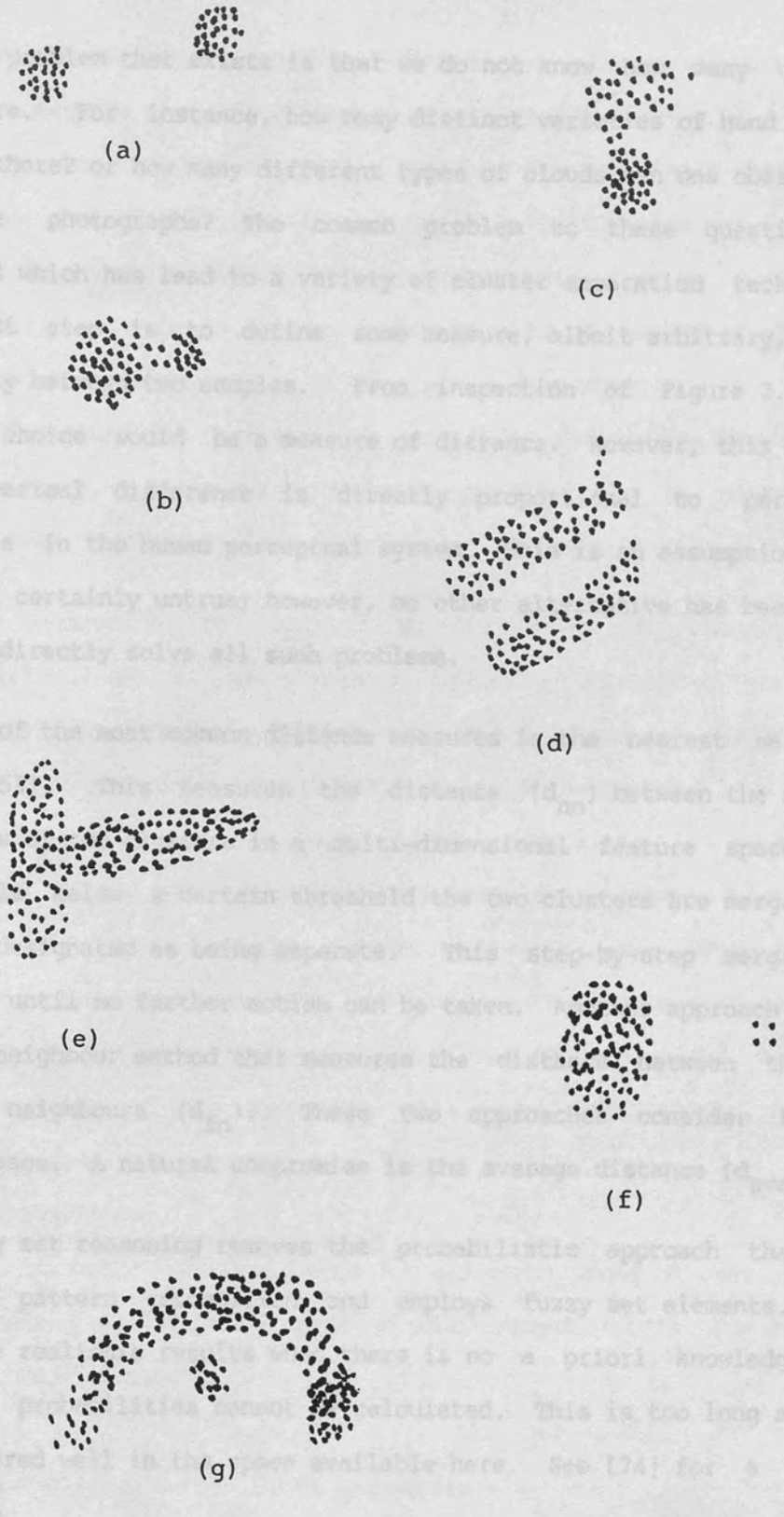


Figure 2.9 Examples of clusters

The problem that exists is that we do not know how many clusters there are. For instance, how many distinct varieties of hand printed 2's are there? or how many different types of clouds can one observe in satellite photographs? The common problem to these questions is vagueness which has lead to a variety of cluster separation techniques. The first step is to define some measure, albeit arbitrary, of the similarity between two samples. From inspection of Figure 2.9a, an initial choice would be a measure of distance. However, this assumes that numerical difference is directly proportional to perceptual difference in the human perceptual system. This is an assumption which is almost certainly untrue; however, no other alternative has been found that can directly solve all such problems.

One of the most common distance measures is the nearest neighbour method [53]. This measures the distance (d_{nn}) between the nearest neighbours of two clusters in a multi-dimensional feature space. If this falls below a certain threshold the two clusters are merged else they are designated as being separate. This step-by-step merging is continued until no further action can be taken. Another approach is the farthest neighbour method that measures the distance between the two farthest neighbours (d_{fn}). These two approaches consider the two extreme cases. A natural compromise is the average distance (d_{ave}).

Fuzzy set reasoning removes the probabilistic approach that has dominated pattern recognition and employs fuzzy set elements. This gives more realistic results when there is no a priori knowledge and therefore probabilities cannot be calculated. This is too long a topic to be covered well in the space available here. See [74] for a fuller discussion.

2.7.2 Syntactic Pattern Recognition

Syntactic pattern recognition (often termed structural recognition) aims to describe an image by a recursive description of a complex pattern in terms of simpler patterns based on the "physical" shaping of the scene. For example, consider a cow in a field with a car - this is described as a complex scene. However, the image can be partitioned into three simpler parts: a cow, a car and a field. Information can be included in the description, e.g. the car is near the cow. This in turn can be described in terms of its primitives such as the cow has four legs and a tail, the car has four wheels and the field is green. The scene is thus partitioned into much simpler elements.

This approach has been applied to the industrial inspection of PCB's [16]. Images are not always conveniently represented by strings; however, certain pictorial patterns such as PCB images can be made to fit the string model. Local features can be decomposed into a small number of unique primitives, e.g. corners and lines. A structural description of the primitives and the relationships between them can be determined to form a string grammar. Given a set of primitives describing common defects, one can use an automatic test procedure to locate them by searching the string describing the PCB under test for all occurrences of the defective primitives. For example, consider Figure 2.10 where the pattern is

abadadadabca

To detect the fault, one could look for all occurrences of the pattern bca. Cheng [14] has described a VLSI based architecture to match patterns and strings.

2.7.3 Hybrid Methods of Pattern Recognition

The above approaches basically rely on images which are free of noise and distortions. In reality, the presence of noise will change the input pattern and hence the grammar. The final result is that the

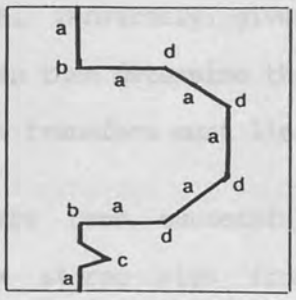


Figure 2.10 Illustration of a string grammar to detect faults in a PCB

input pattern is generated by more than one pattern grammar and ambiguity occurs. In this case, stochastic languages are employed for pattern recognition and Bayes' decision rule can be used for recognition.

2.8 EXTENSION TO THREE DIMENSIONS

The ability to extend to three dimensions is an attractive proposition for industrial automation purposes. For instance, a biscuit or a disc is relatively straightforward to recognise because the required data can easily be extracted from a two-dimensional view. However, a 3-D knowledge of objects is often necessary, for instance, in the automated assembly of components in a car plant. Understanding depth from a 2-D image is crucial to the problem of image understanding.

Information in the x and y directions is trivial to obtain but the need to extract information from the third dimension (z) is required in order to form a model of the object. The relationship between 3-D points in the world coordinate system and the corresponding 2-D points in the image plane is essentially a perspective transformation. When this transform is known, given the x,y and z coordinates in the world coordinate system, we can find the corresponding 2-D coordinates of x'

and z′ in the image plane. Conversely, given the x′ and z′ coordinates in the image plane, one can then determine the corresponding ray which all points satisfying this transform must lie.

Several approaches have been successfully used. Wu, Wang and Bajcsy [129] employed a stereo view from two cameras to obtain 3-D spatial data of a real object on a turntable. Spatial information was obtained by first calibrating the camera and then taking four pairs of images of the object by rotating the turntable. For each step, stereo matching and the determination of the 3-D coordinates of the point was carried out by use of an appropriate transform. Depth measurement was found to be better than 1% at a distance of 1.5m.

Another approach was to use a single camera and project patterns onto the objects. Various patterns have been tried including spots, parallel lines, grids, concentric circles and spirals. By incorporating a priori knowledge about the nature of the projected pattern, one can measure the deviations of the pattern and fit this to a polynomial equation. From this, the x,y and z coordinates can be derived. Oshima and Shirai [86] used this 3-D information for the general description of a scene. Spots were projected onto the scene which were then grouped into planes and the planes were then merged. The regions were then classed into plane, curved or undefined. Curved regions were then merged to other curved or undefined regions in an attempt to fit a quadratic surface to them. The scene was finally described in terms of the properties of regions and relations between regions.

Shading has been successfully used for extracting 3-dimensional information from a scene [104]. Shading can be described as the variation of grey-level across a region. If the region of an object represents a uniformly reflective surface, this variation must be due to the changes in slope of the reflective surface to the source of

illumination and the viewer. Therefore, grey-level variations impose constraints on the 3-dimensional shape of the surface. In order to avoid ambiguous information, several pictures of the same scene under different conditions of illumination are generally used.

Texture has also been used to derive 3-dimensional information [83]. Smooth variations in texture (texture gradients) can give clues to local surface shape. For instance, consider looking along the side of a brick wall. The bricks near to the eye appear large while the bricks at the end of the wall appear small and more closely spaced. This gives a strong sense of the orientation of the wall which can be thought of as a change in texture. There are three main causes of texture gradients (1) variations in distance, (2) variations in surface orientation and (3) by variations in the physical texture itself; however, normally the physical texture is assumed to be constant. An example of a typical 3-D problem is the automatic inspection and assembly of light bulb filaments [73]. This requires that the (x,y,z) coordinates of the two spikes of the filament holder to be returned in order for a mechanical arm to thread the filament.

2.9 OPTICAL IMAGE PROCESSING

A potentially powerful approach to image processing is optical processing. It is appealing because of the enormous data rate it can hope to achieve, all points in a scene (typically the equivalent of over one million pixels [41]) being processed simultaneously. However, optical processing does have its disadvantages. It is often not as accurate as digital processing and the purely optical processor is restricted in the generality of algorithms that it can perform. The advocates of optical processing have reacted to this in two different ways:

1. Using it for problems with immense input data, e.g. analysing kilometers of 16mm-motion picture film [41].

2. Broadening the generality by combining optics with electronics, i.e. a hybrid arrangement.

One problem with optical processing is that the output from an optical processor is usually related to its input by a linear processing function. However, many functions desirable from a recognition or object location point of view are necessarily non-linear. Non-linearities can be introduced by combining optics and electronics (see point 2 above) [41].

A purely optical application is template matching [70]. Here, the input image is first processed by a simple lens. At a plane one focal length behind the lens, all light emanating from the image with the same spatial frequency appears a set distance from the system's optical axis. This distance is directly proportional to the spatial frequency; the lens thus performs a spatial Fourier transform on the image. These frequency components are matched with those of the reference pattern stored in a filter at that plane, the matching process in the focal plane being one of multiplication. The filtered light is then retransformed by a second lens to give the correlation output in the output plane. (It is well known [33] that multiplication in the Fourier domain is equivalent to a convolution with the transform filter in the spatial domain.)

Although conceptually elegant, it is a somewhat inflexible scheme and difficult in practice as it is susceptible to differences in scale, rotation and out of plane orientation between the reference and input images. Such systems also require a spatial light modulator which currently costs around £10,000 [70]; however, a market is emerging which will hopefully reduce the current high cost and provide solutions to

many problems that require such enormous data rates.

Though successful in terms of computation, the type of optical processing described above embody a highly inflexible algorithm and therefore its use is liable to be restricted. For example, it is difficult to see how to make a system perform median filtering or other useful operations. For these reasons we shall ignore optical systems in the remainder of this thesis, though this form of computation will undoubtedly become increasingly widespread in the future.

2.10 CONCLUSIONS

This chapter has looked at a varied set of methods and techniques that are commonly associated with image processing, image analysis and pattern recognition. In real-world problems where objects need to be recognised, and faults need to be detected, procedures for examining local information are important. Apart from being computationally efficient they must also be robust. Methods such as the Fourier transform do not provide the best means of extracting local information; however, they do find use where rotation and comparison on a size independent basis is required, but the fact that floating point arithmetic is necessary limits its practical uses.

Real-world problems generally involve all three image pattern recognition topics in one form or another. However, the majority of this work seems to have been concentrated on the image analysis and feature recognition techniques. This includes topics such as edge detection, thinning, and template matching. A particularly useful tool is the Hough transform as it is an efficient method for the recognition of general shapes. Being resistant to noise and discontinuities in the edges of objects, it is in many ways an optimal choice. Thinning is computationally expensive and template matching tends to be practical

only with small neighbourhoods and binary images. Classification schemes such as clustering tend to be of less interest as a priori knowledge can usually be incorporated into a scene without the need for complex pattern classification methods. However, classification is clearly important at the final stage of inspection where defects are being analysed and categorised. Although this can make the algorithm less general, speed is more important than generality for industrial recognition purposes.

Three-dimensional work has increased over the last few years so that depth can be obtained from a 2-D image and a 3-D model constructed, thus facilitating 3-D object recognition. However, recognition is only the first step in an inspection algorithm. The inspection of 3-D objects from depth information is currently an important research topic.

Optical processing has been discussed and is attractive because of the enormous amount of parallelism involved. However, a purely optical approach is not as flexible as the digital approach. To increase the flexibility, some researchers have developed hybrid systems consisting of optics and electronics, but these are still at the research stage though they will undoubtably become popular in the future.

We have seen in this chapter that certain techniques such as thinning are useful although they are too computationally intensive for an industrial environment without parallel hardware. The next chapter considers two industrial algorithms, based on some of the techniques cited in this chapter as being useful for such applications. The execution time of these algorithms is crucial in order to meet industrial recognition speeds. In order to achieve this, a programmable high-speed processor has had to be developed - this is described in Chapter 5. This processor is later used in Chapter 8 to develop a cost-effective multiprocessor architecture, suitable for increased

execution speed of sequential and parallel algorithms. Using this architecture, it should thus be possible to apply computationally expensive algorithms such as thinning and similar processes to industrial recognition tasks, and should therefore not be immediately discarded as being too computationally expensive.

For the moment, we shall concentrate on developing the following two industrial recognition algorithms for a single processor system; however, they will be implemented on the architecture developed in Chapter 8.

CHAPTER 3

THE DEVELOPMENT OF REAL-TIME INSPECTION ALGORITHMS

"The secret of science is to ask the right question, and it is the choice of problem more than anything else that marks the man of genius in the scientific world."

in C.P. Snow A postscript to Science and Government (Oxford: Oxford UP, 1962)

3.1 INTRODUCTION

This chapter describes two real-time industrial recognition algorithms. These extract information from a digitised image of a manufactured product and scrutinises the information in order to detect whether the product is defective. Both algorithms apply the same general technique for extracting information from these two rather different products. The ability to manipulate the data in real-time is emphasised as this leads onto the topic of program optimisation techniques. A breakdown of execution times for both algorithms on a series of images is given. This allows us to locate bottlenecks within the algorithm; methods for eliminating these will then be discussed. Execution times for both algorithms running on SIP (see Chapter 5) are also given. This will show that bit-slice processors are capable of forming the basis of cost-effective, real-time recognition systems.

3.2 THE NEED FOR INDUSTRIAL RECOGNITION SYSTEMS

There is currently much interest in the field of industrial recognition systems. These are systems that analyse and scrutinise moving products on a conveyor belt, the data originating from a video or line scan camera - they are hence geared towards quality control. The possibility of a manufactured product being located, scrutinised for defects and rejected on a pass/fail inspection basis is appealing to a manufacturer as it has several advantages over human inspection:

1. In many areas, quality control is only carried out on a sampling basis. Frequently, 100% inspection is required. The rate of which inspection can take place is usually dictated by the line speed. At typical production rates of about 5-10 products per second [25], and with the possibility that several inspection tasks may need to be carried out per product, it is generally not possible for the human eye to assimilate the amount of information needed in the time available. Small defects thus go undetected. However, a vision system should be able to achieve 100% inspection in the available time.

2. A human cannot 'measure' the features of an object, e.g. the length or the amount of chocolate coating, to the same degree of accuracy as a vision system. With a high accuracy vision system, statistical information can be analysed and fed back to the production equipment in order to ensure product uniformity and efficient use of production tools and materials.

3. It is often necessary to have continuous inspection of products over periods of seven days or more. Because of the tedious nature of product recognition, 30 minutes is probably the limit for reliable human control. A vision system does not tire or suffer from the boredom which is often the cause of human error in this kind of

situation.

4. Machine inspection can be performed in unfavourable environments, e.g. in the presence of excessive noise, heat or a fat-laden atmosphere.

Because quality control is highly repetitive, one would assume that a computer based system of some kind would be highly suited for this task. Vision systems have been successfully employed commercially [2],[16],[29] and have proved their effectiveness.

One of the main problems with industrial recognition is the need to scrutinise the product in real-time, i.e. to complete the task in the time it takes for the product to pass by. Typical product rates range between 5-7 products per second meaning image acquisition and analysis must be carried out in 150-200ms. Sequential processors are usually too slow to process the amount of information in the required time whereas parallel processors are expensive and generally not suited for image analysis because of the high degree of sequentialism often incorporated in inspection algorithms (Sections 3.3 and 3.4).

Because of the large amount of processing that may be required, it is often necessary to develop dedicated hardware or general purpose (programmable) high speed hardware. Dedicated hardware (hardware accelerators) usually executes those parts of an algorithm that constitute the bottleneck. This has the advantage that it is very high speed and can often process the information well within the constraints of the application. On the other hand, it has a fixed purpose and can only be used for the application it was developed for. This may be suitable for the majority of cases where only one product is being manufactured, but for a manufacturer that produces a number of different products, programmable high-speed hardware may be more advantageous. The same hardware can then be employed throughout the factory, it being

only necessary to change the software for the product in question. However, this option is usually more expensive and will obviously require a detailed analysis before investment. It may only be advantageous for a long term investment or if many systems are required.

Cost is often the determining factor for acquiring vision systems for automated quality control. Those currently available with tailored software cost more than £40,000 (e.g. the CRS1000 workstation, MegaVision by Prostab, etc.) which is generally affordable only by large manufacturers. Experience in this research group is that an affordable cost would be rather less than £10,000 (excluding software) [25] - such a system is described in Chapters 5 and 6. In fact, as Davies notes with regard to the food industry, "above this figure, the rather low profit margins characteristic of foodproduct manufacture might be eroded excessively" [25]. In many cases it may not be necessary to invest in special purpose hardware if the execution time of the algorithm can be improved by using readily available resources. For instance, it may only be necessary to invest in a faster processor if the algorithm shows significant improvement after various real-time techniques have been applied. These techniques are discussed in Section 3.5.

Sections 3.3 and 3.4 discuss two real-time analysis algorithms. The first detects and scrutinises O-rings while the second detects and scrutinises a rectangular chocolate covered biscuit. The rationale behind these choices is that the first algorithm represents a small machine part problem while the second represents a rather difficult type of application in the food industry, which is currently a real problem. The operation common to both algorithms is the Hough transform as described in Section 2.6.4. This was chosen for two reasons: (1) it is insensitive to noise - a necessary requirement for industrial environments where conditions are electrically noisy; and (2) it still works with defective shapes or, in this case, defective products. This

allows us to form a model template of the shape hence simplifying inspection of the product without loss in throughput. The presentation of both algorithms is in the form of a top-down approach. First, the product and types of defects that occur will be described. An outline of the procedure for detecting the faults is then given. A more detailed description of the algorithm and reasons for adopting the approaches will then be given followed by run-time results and timings. In reality, these algorithms would run on the high-speed processor SIP (Chapter 5) in order to achieve the necessary speeds - times for this will also be given. The first of the two inspection algorithms will now be described.

3.3 THE O-RING ALGORITHM

An O-ring is a round rubber ring of a known radius. The aim of the O-ring algorithm is to locate the centres of multiple O-rings in an image. By locating the centre we have (a) shown that an O-ring (or other circular-like object) exists in the image and (b) obtained enough information to test for defects, i.e. because we know the location of the centre, we know where to expect the edges of the ring. O-rings are formed by pouring a hot rubber solution into a mould. The most common fault that arises is when the ring is cooled too quickly - brittle sections can occur which makes the ring vulnerable to snapping if used under stressful conditions. These brittle sections of the ring are detected by matt black shadings on the surface of the ring. As the ring itself is normally a semi-glossy black, 8-bit digitisation is not accurate enough to detect the differences. However, by demonstrating that the centres of the O-rings can be found, all the necessary information preparatory to detecting the brittle sections is available. (Oddly, deformed and broken O-rings are extremely rare.) In order to show the robustness of the algorithm, the eight images used contain

random scenes of O-rings. Overlapping rings and additional non-circular objects are included in some images so as to obscure parts of the ring; however, there will usually be several rings in a row (depending on the manufacture's method of producing O-rings), separated from each other on the conveyor belt. Overlapping rings and rings obscured by other objects are therefore detected as defects.

To limit the amount of processing that needs to be carried out in order to achieve real-time recognition, it is often necessary for a priori knowledge of the product to be incorporated into the algorithm (Section 3.5.1). For this reason, it is justifiable to assume (in this case) that the radius of the O-ring is 22 pixels in all cases. (Note that the scale is close to 1 pixel per mm - this was chosen for convenience as it allows us to compare measurements taken on a one-to-one basis.) The Hough transform technique is used in order to locate the centres. This technique is described by Kimme et. al. [63] and was recently applied to biscuit inspection for locating the centre of a circular biscuit [27]. Here, the algorithm has been extended to locate multiple centres in an image. The algorithm strategy follows in the next section.

3.3.1 Algorithm Strategy

The algorithm is partitioned into three sections

1. Find all possible centres.

2. Deduce 'true' centres from the above list.

3. Detect faults.

Finding the centres of the rings by the Hough transform method leaves Hough space with large peaks at the centres of the rings (see below). Figure 3.1 represents Hough space at the centre of one of the rings in test image 3 after application of the Hough transform procedure. Locating the highest peak in each cluster gives us (to a first approximation) the centre of the ring. However, a problem occurs when trying to distinguish between similar values within different clusters, i.e. how do we cope with the situation when there is a value of 18 in ring 2 and a value of 18 in ring 3?. As we can see from Figure 3.2, the peak value of 18 represents the centre of ring 2 which is the required point but the value of 18 in ring 3 is not the centre because there is a higher value point adjoining it (value of 24) which is the required point.

This can be solved by making a list of the values of all possible centre points above a given threshold and sorting the list. The highest peak in the sorted list will be, by definition, the likeliest centre of one of the rings. Working down the list (highest to lowest count), if the next point exceeds a threshold distance from any of the centres previously found (initially the position of the highest value in the list) then it too is considered the likeliest centre of one of the rings because, since the list is sorted, this position will represent the highest peak in one of the rings.

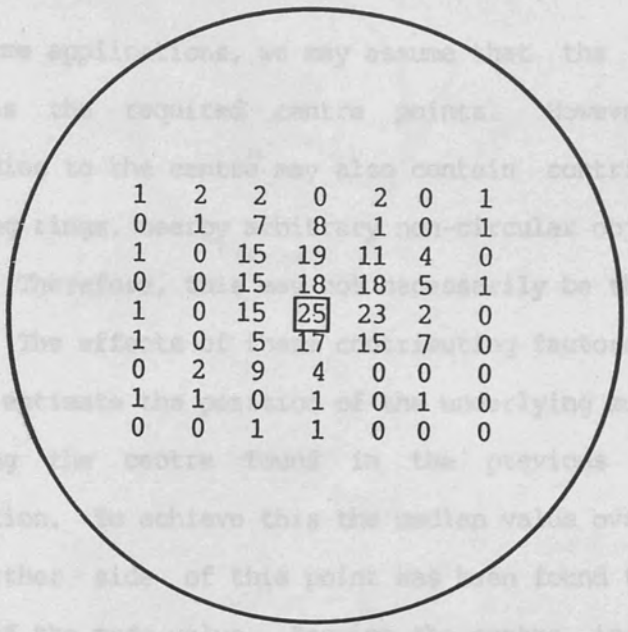
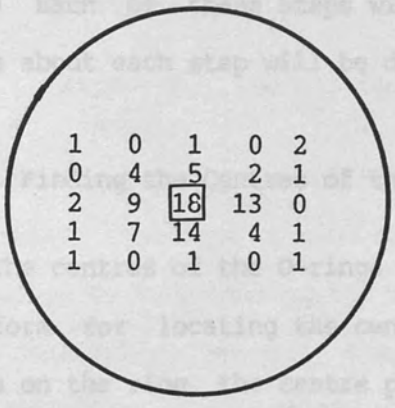


Figure 3.1 Hough space for O-ring 1



(a)

(b)

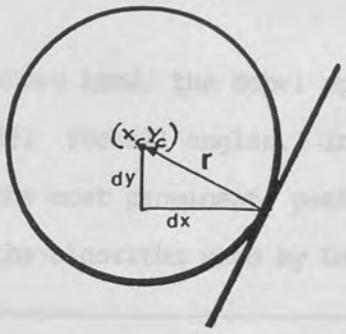
Figure 3.2 Hough spaces for (a) O-ring 2 and (b) O-ring 3

In some applications, we may assume that the centres found will suffice as the required centre points. However, the maximum peak corresponding to the centre may also contain contributions from other overlapping rings, nearby arbitrary non-circular objects and the effects of noise. Therefore, this may not necessarily be the exact centre of the ring. The effects of these contributing factors can be minimised by trying to estimate the position of the underlying mode in each cluster, considering the centre found in the previous step as being a good approximation. To achieve this the median value over a region of a few points either side of this point has been found to be effective as an estimate of the mode value. Knowing the centre, inspection can now take place. (Note that in the cases depicted in Figures 3.1 and 3.2, the highest number in each ring (shown in a box) was calculated as the mode.) Each of these steps will now be described in detail. Specific points about each step will be discussed following the algorithm.

3.3.2 Finding the Centres of the O-rings

The centres of the O-rings are found by application of the Hough transform for locating the centres of circular objects. From all edge points on the ring, the centre point (x_c,y_c) is calculated by the method shown in Figure 3.3a. The point in Hough space corresponding to (x_c,y_c) is then incremented. As a result, large peaks (counts) will exist in Hough space at the positions of the centres of the rings. As can be seen from Figure 3.3b, the calculation of the centre point requires the x and y gradients of the edge. The most common technique for acquiring this information is by the use of a differential edge detector.

An alternative to finding the centre by the above method would be to threshold the image and find the centre of the ring by averaging the x and y values (x and y being the coordinates of those pixels detected as being part of the ring). However, shadows or a defective ring such



(a)

$$dx = radius * \frac{g_x}{grad}$$

$$grad = \sqrt{(dx^2+dy^2)}$$

$$dy = radius * \frac{g_y}{grad}$$

(b)

Figure 3.3 Calculation of the centre on an O-ring

as that depicted in Figure 3.9h(5) would produce an inaccurate centre. A similar technique could instead be applied to the edge points determined by thresholding the magnitude of the gradient produced by an edge detector. However, shadows may affect the situation by making edges slightly thicker on one side than on the other; this would again produce an inaccurate centre.

The main problem now is deciding which edge detector to use, bearing in mind that the it must be economic with regard to computation yet maintain a fair degree of accuracy. The edge detector developed by Hueckel [59] allows the removal of most of the local noise but it is complex and computationally expensive (typically 521 operations per pixel). The Roberts' cross operator requires the minimum number of operations per pixel; however, like the Prewitt operator, it is only accurate to 5-10° [1], depending on the angle, i.e. its accuracy changes

with angle. On the other hand, the Sobel operator has been shown to be accurate to ~½° [64], for all angles. In fact, experimentation found that the Sobel gave the most prominent peaks at the centres of the rings. The form of the algorithm used by Davies[1] is outlined below.

```
1     PROCEDURE find_centres;          { find possible centres of O-rings }
2      BEGIN
3       y:=0;                                          { set image scan }
4       REPEAT x:=0;
5        REPEAT
6         dx := (P8+P1*2+P2) - (P4+P5*2+P6);      { find Sobel - dx }
7         dy := (P6+P7*2+P8) - (P2+P3*2+P4);      { find Sobel - dy }
8         dd := dx*dx+dy*dy                       { Calculate Gradient² }
9
10        IF dd > thresh THEN                     { Thresh is 150x150 }
11         BEGIN                                  { then an edge }
12          dd := SQRT(dd)                        { Calculate Gradient }
13          x_temp := x; y_temp := y;             { save x and y }
14          x := x - radius_of_Oring*dx DIV dd;   { find centre }
15          y := y - radius_of_Oring*dy DIV dd;
16          Q0 := Q0 + 1;                         { inc. Hough space }
17          x := x_temp; y := y_temp;             { restore x and y }
18         END;
19        x := x+1;
20       UNTIL x=128; y := y+1;                   { do x }
21      UNTIL y=128;                              { do y }
22     END;
```

Figure 3.4 Finding the centres of O-rings

Lines 1 to 7 are self-explanatory (note that the accumulator, Q-space is initially set to zero). Line 8 finds the value of the gradient squared at that point in the image (since the actual value of the gradient is not needed here, a less compute intensive approach is to calculate the value of the gradient only when an edge element is found to exist - determined by line 10). As this involves two multiplications for every point in the image and and a square root calculation for every edge point - a relatively time consuming combination - Davies used the approximation

$$\sqrt{(x^2+y^2)} \simeq \text{MAX}(dx,dy,(dx+dy)*7 \text{ DIV } 10)$$

[1] Reproduced with kind permission of Dr. E.R. Davies

where dx and dy are the absolute values of the x and y gradients calculated from the Sobel and '7 DIV 10' is an approximation to $1/\sqrt{2}$, i.e. considering the pixel as an octagon rather than the square digitised approximation. The threshold at line 10 was set at 150x150 - this gave reasonable results for the majority of reasonably well lit scenes. This algorithm is interesting in the fact that there are no floating point or trigonometric calculations. This is important and shall be discussed later in Section 3.5 when we consider program optimsation and efficiency. The next step is now to locate the highest peak in each cluster. Note that the work described above is based on the Davies centre location algorithm. However, this was only designed to locate one product per image.

3.3.3 Locating the True Centres

When the above task is complete, a cluster of accumulated points will exist around the centre of each ring. This will consist of a maximum peak at the centre and smaller, yet similar sized values surrounding the peak. This can be seen in Figure 3.1.

In order to sort all possible centres so the highest peak in each cluster can be determined, the values and positions of all candidate centre points must be found. These can be obtained from the positions and values of those points over a fixed threshold in Hough space. A threshold of six was found adequate to locate the centres of about 15 O-rings. This can of course be adjusted to suit the environment and the requirements.

Once this is achieved, we can sort on the values of the peaks. The sorting algorithm used was the quicksort algorithm developed by Hoare [56]. This is a recursive sorting algorithm that has an average computing time of $O(n\log_2 n)$ and a worst case computing time of $O(n^2)$.

This can be compared favourably with the bubble sort which has a computing time of $O(n^2)$. The algorithm is given in Figure 3.5.

The array 'peaks' is the list of all candidate centre points. The procedure 'swap' in lines 9 and 11 merely interchange the x, y and peak values. The quicksort procedure is called from the main program with the values

```
quick_sort(peaks,1,peak_count);
```

where 'peak_count' is the number of candidate centre points found.

```
1     PROCEDURE quick_sort(VAR peaks :ARRAY_OF_PEAKS; m,n :INTEGER);
2      BEGIN
3       IF m < n THEN
4        BEGIN
5         i := m; j := n+1; k := peaks[m].value;
6         REPEAT
7          REPEAT i := i+1 UNTIL (peaks[i].value>=k) OR (i=n);
8          REPEAT j := j-1 UNTIL (peaks[j].value<=k) OR (j=m);
9          IF i < j THEN swap(i,j);
10        UNTIL i>=j;
11        swap(m,j);
12        quick_sort(peaks,m,j-1);
13        quick_sort(peaks,j+1,n);
14       END;
15     END;
```

Figure 3.5 Quicksort algorithm to sort peak heights

The highest peak (top of the sorted list) is now entered into the 'found' list (a list of coordinates of the highest peak in each cluster) and is used as a reference point in order to locate all other peaks. The position of each point found in 'peaks' is compared with the positions of those in the 'found' list (initially the highest peak from the sorted list). If it exceeds a threshold then it is considered to be the highest value at the centre of one of the rings (the required point) and entered into the 'found' list. In essence, the threshold determines the closeness two centres can be located due to overlapping rings before they are considered a single ring. The distance must be large enough to

overcome the spreading of peak points in Hough space due to noisy images or other objects in an image contributing to the peak value, yet small enough to distinguish between two centres close to each other. A distance of 5 pixels has been found to give good results. The algorithm is given in Figure 3.6. Note that 'peak_count' is the position in the array of the highest peak in the scene deduced using the Quicksort.

```
1     PROCEDURE deduce_centres;
2      VAR
3       search, i, temp :INTEGER;
4       peak_exists :BOOLEAN;
5
6        BEGIN
7         true_peak_count := 1;     { 1st element in list is a centre }
8         found[1].xx := peaks[peak_count].xx;
9         found[1].yy := peaks[peak_count].yy;
10
11        FOR search := peak_count DOWNTO 1 DO
12         BEGIN
13          x := peaks[search].xx; y := peaks[search].yy;
14          peak_exists := TRUE;
15          FOR i := 1 TO true_peak_count DO
16           BEGIN
17            { Pythagoras's theorem }
18
19            temp := (x-found[i].xx)*(x-found[i].xx) +
20                    (y-found[i].yy)*(y-found[i].yy);
21
22            IF temp <= dist_thresh_squared THEN peak_exists := FALSE;
23           END;
24
25          IF peak_exists THEN
26           BEGIN
27            true_peak_count := true_peak_count+1;
28            found[true_peak_count].xx := x;
29            found[true_peak_count].yy := y;
30           END;
31         END;
32       END;
```

Figure 3.6 Algorithm to deduce 'true' centres from peak heights

At the end, 'true_peak_count' is obviously the number of O-rings of radius 22 pixels in the scene. This may appear to be an inefficient way of deciphering between local and distant peaks; however, there are typically only 50-60 peaks in an image (however, this is dependent on

various factors such as the lighting conditions and the number of rings involved) making this section moderately fast (see timings in Section 3.3.5). The location of each peak in the cluster is now known (all points in the 'found' array) and it is only necessary to apply a median at each peak (as previously mentioned) in order to smooth out the effects of noise etc.

3.3.4 Detecting Defects in an O-ring

Now that all the centres of the O-rings have been found, it is necessary to detect faults in each O-ring. As we have previously mentioned, these faults may not be detected (because of the accuracy of data acquisition) or the faults never occur (in the case of broken O-rings). However, by showing that these faults can be detected, the principles used in this section can be applied to circular products where these faults do occur.

The method used for determining faults was the radial histogram approach [27]. This has several advantages, namely that the 'true' radius can be measured accurately and faults are easily detected. The method basically accumulates the intensity of all pixels in a given radius band as shown in Figure 3.7; however, as one might expect, the number of pixels in each band increases as the radius increases. The histogram must therefore be normalised by dividing each accumulation by the number of pixels in each radius band. It has been found [27] that accumulation of pixels with a histogram base of r^2 rather than r allows the radius to be measured more accurately. (Also, r would otherwise have to be derived from r^2 which involves a square root computation. This is undesirable for real-time work.) In this case, the area of the histogram extended four pixels beyond the outer edge of the ring. This allows us to measure accurately both the inner and outer radius and defects such as those in ring 2 in Figure 3.9h.

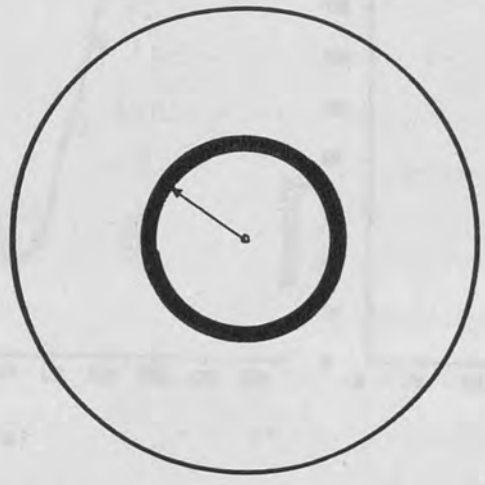


Figure 3.7 Accumulating pixels in a radius band

The radial histograms of the normalised intensity values (y-axis) vs. r^2 (x-axis) for all five rings are given in Figures 3.8a to 3.8e. Figure 3.8a depicts the histogram of a 'perfect' O-ring (ring 1). In order to check for defects, a model of the histogram representing a 'perfect' O-ring is stored, and a correlation is undertaken for each ring's radial histogram encountered.) Figure 3.8b (ring 2) shows that the 'dip' in the curve only extends to half of its true value. This indicates that half of the ring is missing. This is useful as it allows us to measure accurately the proportion of ring that is missing. Figures 3.8c (ring 3) and 3.8d (ring 4) indicates defects at both low and high radii. Further inspection reveals that these fall below the true value indicating a dark patch (note that both graphs are almost identical.)

Unfortunately, this method shows that analysis of overlapping rings is difficult to achieve as this method does not allow us to detect whether the ring is defective or is overlapped with another ring. However, these rings will usually be separated in an industrial environment. Overlapping rings will therefore be detected as faulty.

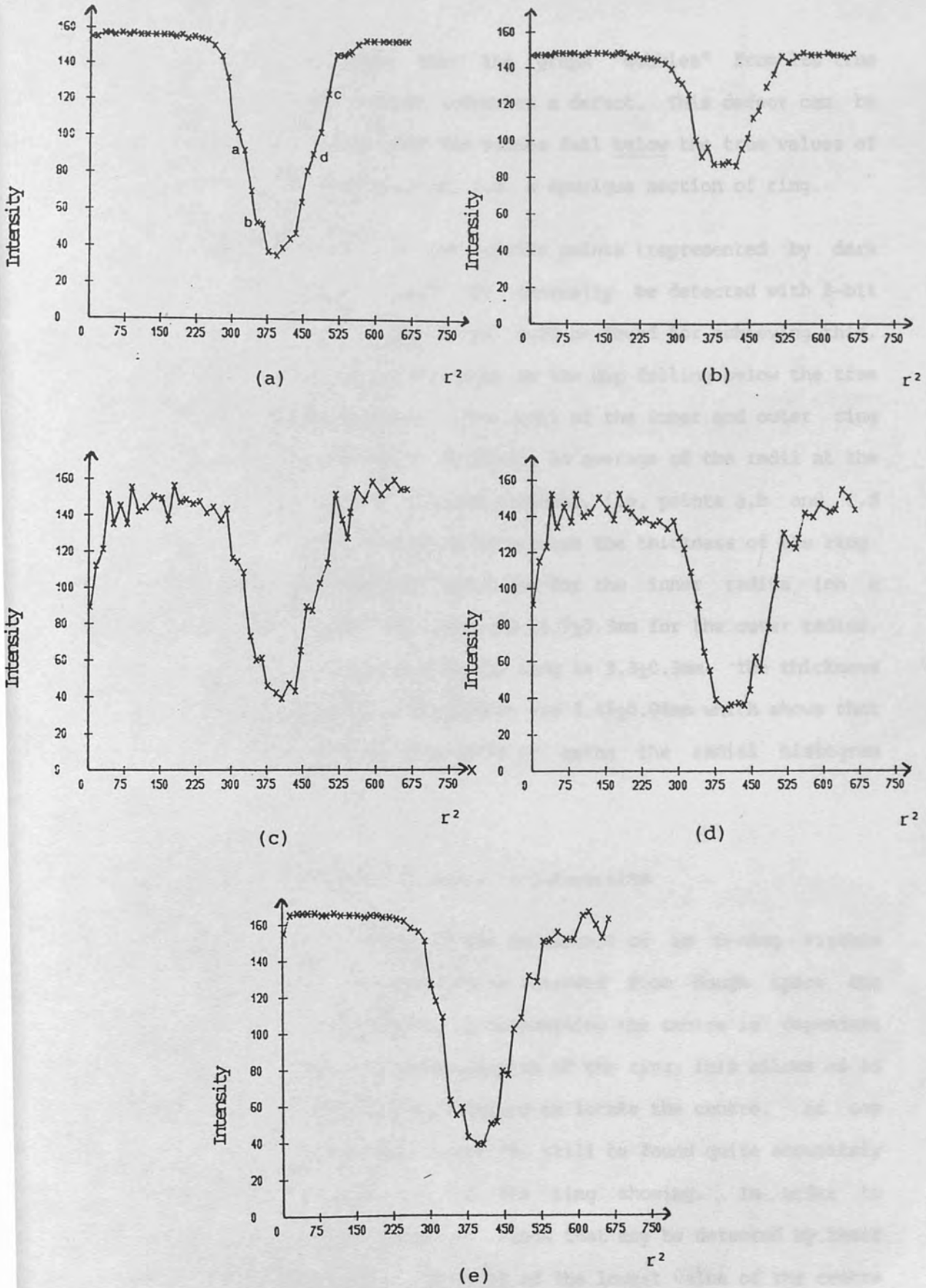


Figure 3.8 Radial histograms of the O-rings depicted in Figure 3.9h

Figure 3.8e (ring 5) shows that the graph "wobbles" from its true position at a high radius which indicates a defect. This defect can be further analysed by noting that the values fall below the true values of ~160 which indicates a dark patch, i.e. a spurious section of ring.

We stated in Section 3.3 that brittle points (represented by dark patches on the O-ring) could not generally be detected with 8-bit digitisation. However, if some means could be found for achieving this, it would be represented on the graph by the dip falling below the true value of 40 (in these examples). The radii of the inner and outer ring can be derived from the graphs by taking an average of the radii at the where the graph has a fairly uniform gradient, i.e. points a,b and c,d in Figure 3.8a. This allows us to measure the thickness of the ring. These radii were calculated at 18.4±0.3mm for the inner radius (on a scale close to 1 pixel per mm) and 21.7±0.3mm for the outer radius. This implies that the thickness of the ring is 3.3±0.3mm. The thickness of the ring as measured by a micrometer was 3.45±0.04mm which shows that measurements can be derived accurately by using the radial histogram approach.

3.3.5 Results and Timings for the O-ring Algorithm

Figure 3.8f gives a graph of the percentage of an O-ring visible vs. the peak value of the centre derived from Hough space for Figure 3.9a. Since the accuracy in determining the centre is dependent on the number of visible edge elements of the ring, this allows us to find the minimum amount of ring required to locate the centre. As one can see from the graph, the centre can still be found quite accurately (within ~1 pixel) for only 25% of the ring showing. In order to determine the maximum number of rings that may be detected by their centres, Figure 3.8g depicts the graph of the lowest value of the centre derived from Hough space vs. the number of O-rings in the image. With a

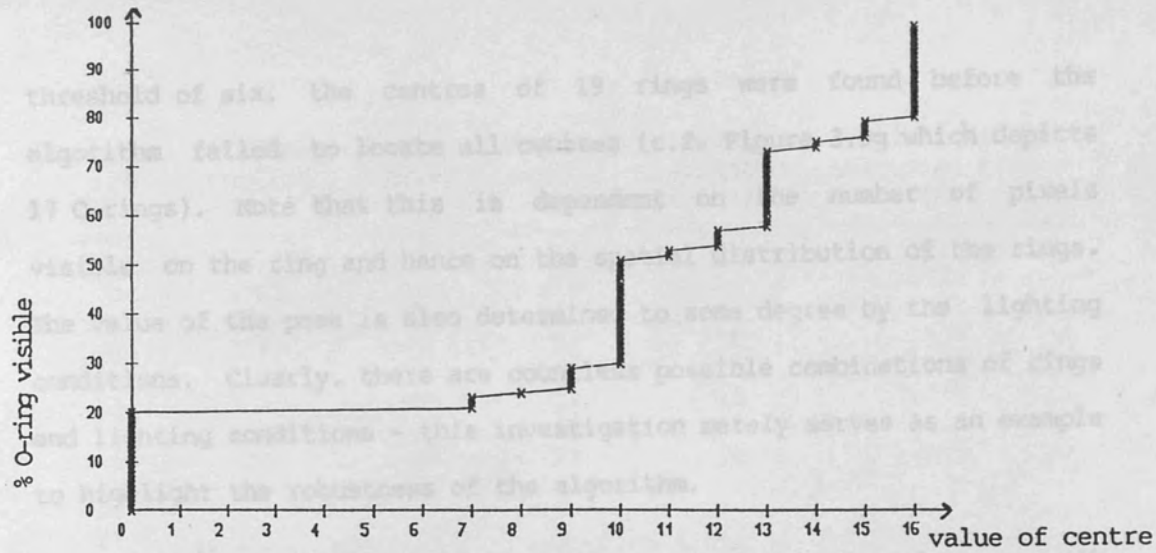


Figure 3.8f Graph of % of O-ring visible vs value of centre in Hough space

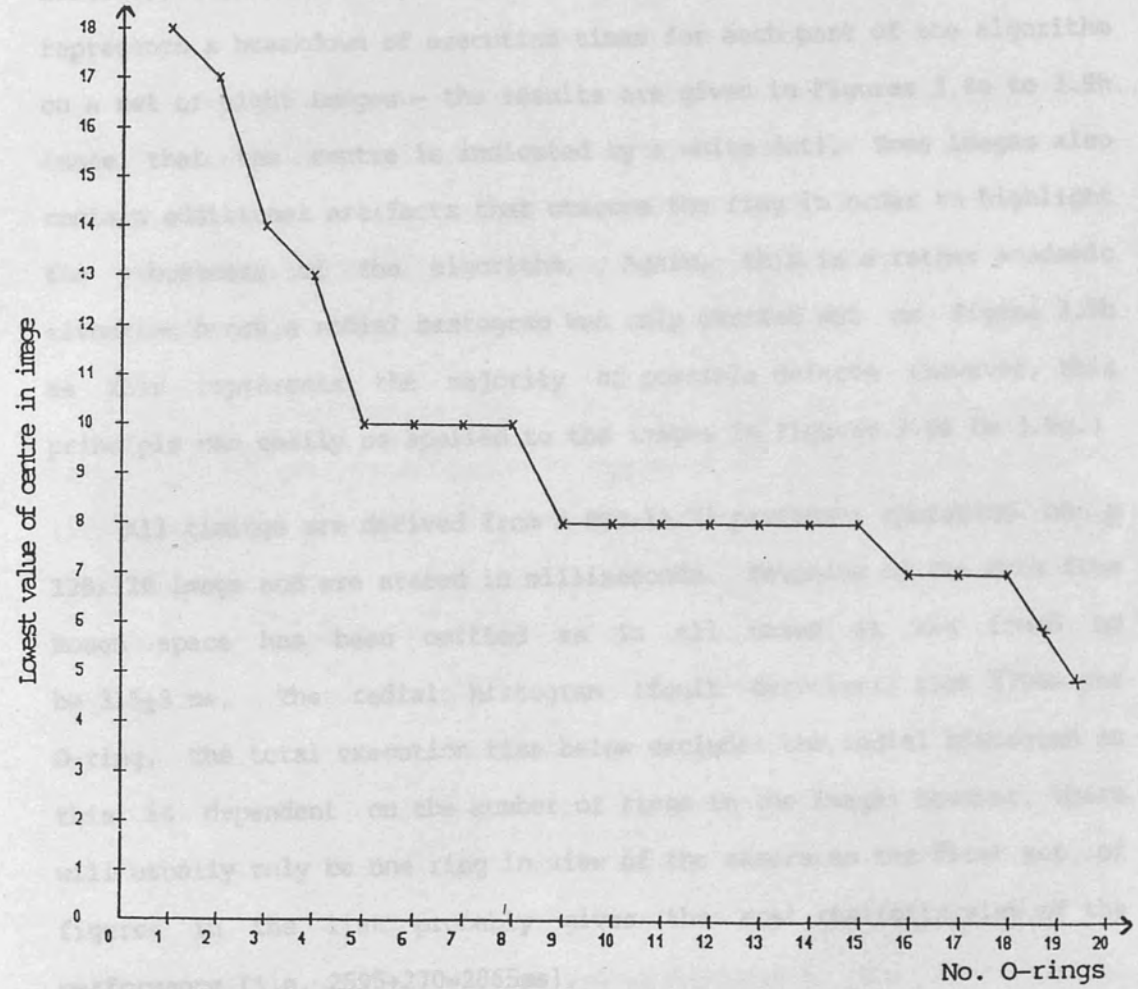


Figure 3.8g Graph of the lowest value of centre vs. number of O-rings in the image

threshold of six, the centres of 19 rings were found before the algorithm failed to locate all centres (c.f. Figure 3.9g which depicts 17 O-rings). Note that this is dependent on the number of pixels visible on the ring and hence on the spatial distribution of the rings. The value of the peak is also determined to some degree by the lighting conditions. Clearly, there are countless possible combinations of rings and lighting conditions - this investigation merely serves as an example to highlight the robustness of the algorithm.

The algorithm has been found to work in the majority of cases with 14 O-rings; however, in practice, these limitations will be rather academic, since most images will only contain one O-ring. Table 3.1 represents a breakdown of execution times for each part of the algorithm on a set of eight images - the results are given in Figures 3.9a to 3.9h (note that the centre is indicated by a white dot). Some images also contain additional artifacts that obscure the ring in order to highlight the robustness of the algorithm. Again, this is a rather academic situation hence a radial histogram was only carried out on Figure 3.9h as this represents the majority of possible defects (However, this principle can easily be applied to the images in Figures 3.9a to 3.9g.)

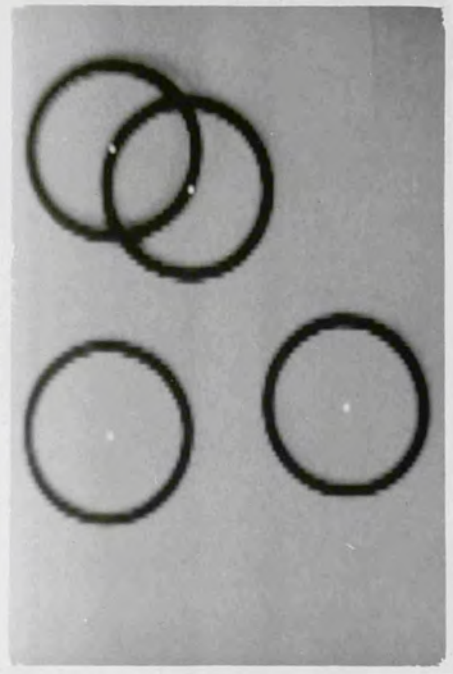
All timings are derived from a PDP-11/73 processor operating on a 128x128 image and are stated in milliseconds. Fetching of the data from Hough space has been omitted as in all cases it was found to be 315$\pm$3 ms. The radial histogram (fault detector) took 270ms per O-ring. The total execution time below excludes the radial histogram as this is dependent on the number of rings in the image; however, there will usually only be one ring in view of the camera so the first set of figures in the list probably gives the most realistic view of the performance (i.e. 2595+270=2865ms).

Figure 3.9	Finding centres (ms)	Sorting centres (ms)	Deducing centres (ms)	Median operation (ms)	Total Algorithm execution time (ms)
a	2274	4	1	2	2595
b	2561	16	17	20	2929
c	2386	24	13	7	2745
d	2478	28	23	18	2862
e	2398	28	19	11	2772
f	2473	32	29	18	2868
g	2650	47	60	31	3106
h	2605	29	24	16	2991

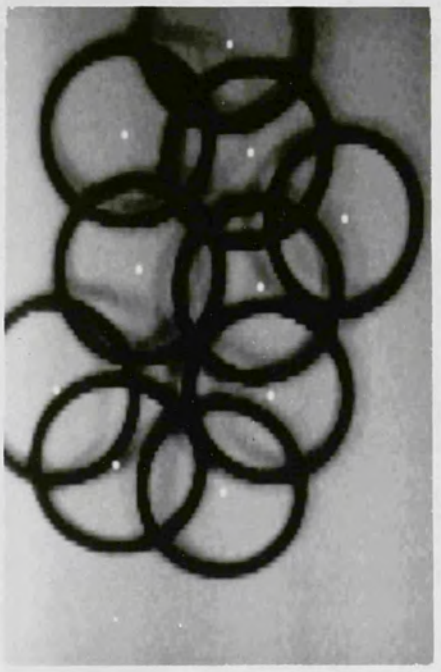
Table 3.1 Breakdown of execution times of the O-ring algorithm on images 3.9a-3.9h

The basic accuracy of the centre determined using the above method is ~$\pm$0.5 pixel [25]. This error must also be combined with ~$\pm$0.5 pixel error arising from the inaccuracy in edge location; however, averaging over all edge points is in principle able to reduce the overall error to well within half a pixel. On the other hand, it should be noted that the accuracy is in practice limited by what is meaningful considering the precision of the product.

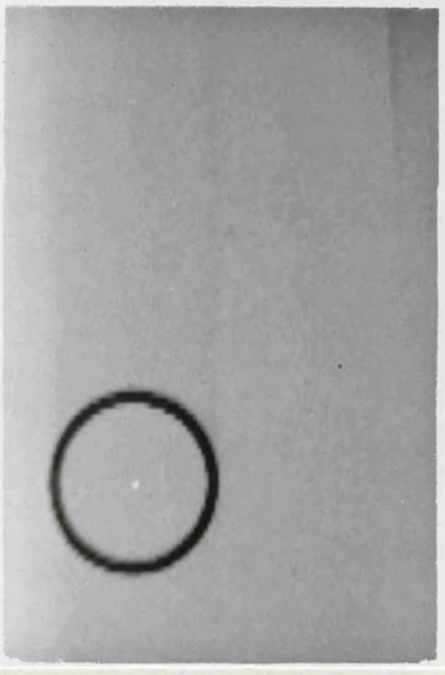
As we can see from the times, the application of the Sobel and the calculation of the centre typically represent 86% of the total execution time. Chapter 5 describes a high-speed sequential processor - this has been shown to give in the region of 25-30 times speed improvement over the PDP-11/73 processor used above. The implementation of the O-ring algorithm on SIP (Chapter 5) shows that the execution time is approximately 140-150ms (including I/O of 50ms) which corresponds to ~7 O-rings/sec (this could still be further optimised by the use of suitable lighting conditions). This is within the time constraints imposed by industrial recognition requirements. The O-ring algorithm written in SIP's native code is given in Appendix B. This is a typical example of a parallel/sequential algorithm. Chapter 8 investigates this further in an attempt to apply it to a multiprocessor system.



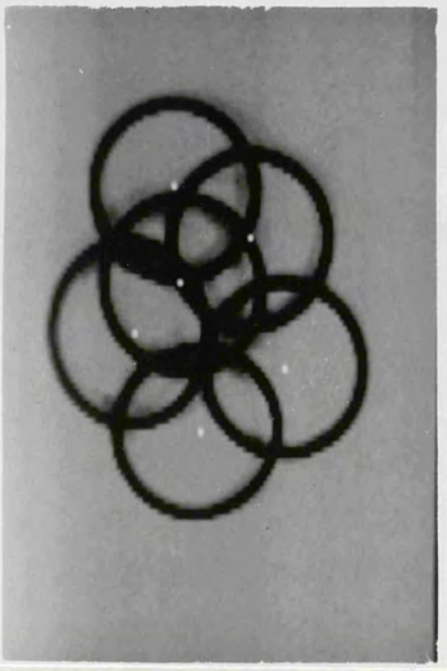
(b)



(d)

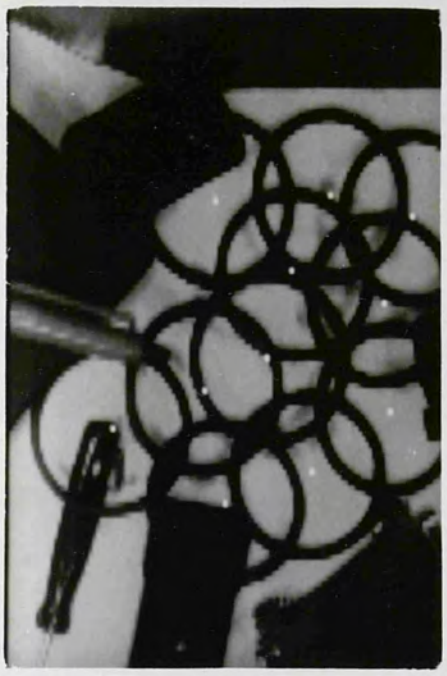


(a)

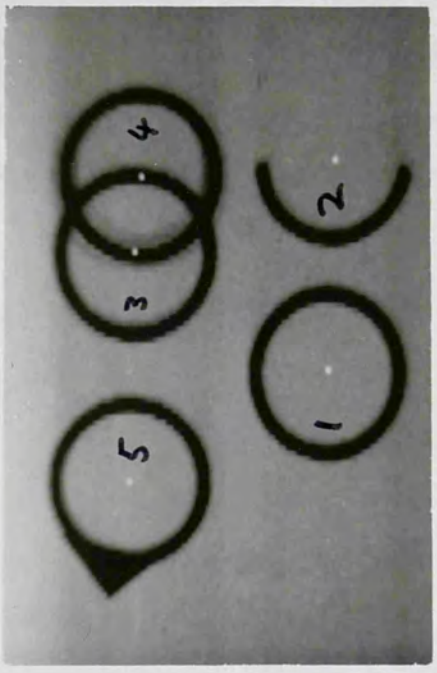


(c)

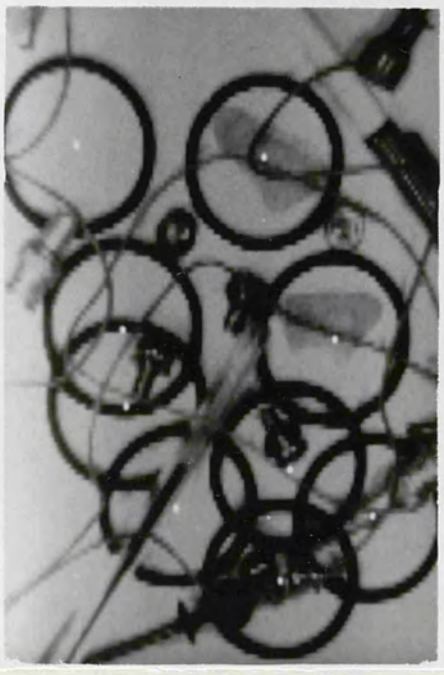
Figure 3.9 Examples of O-rings



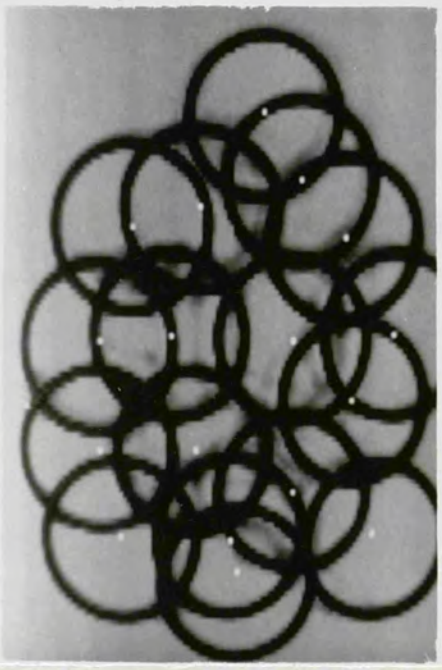
(f)



(h)



(e)



(g)

Figure 3.9 Examples of O-rings

3.4 THE RECTANGULAR BISCUIT ALGORITHM

This section describes an inspection algorithm for the scrutiny of rectangular chocolate covered sandwich biscuits. In general, foodstuffs are difficult to inspect because of the wide range of variations that can occur for a single product: a foodstuff inspection algorithm must therefore be especially robust.

The possible defects that can occur for the biscuit are:

1. Chocolate dripping over the sides.

2. Too little chocolate covering.

3. Incorrect shape.

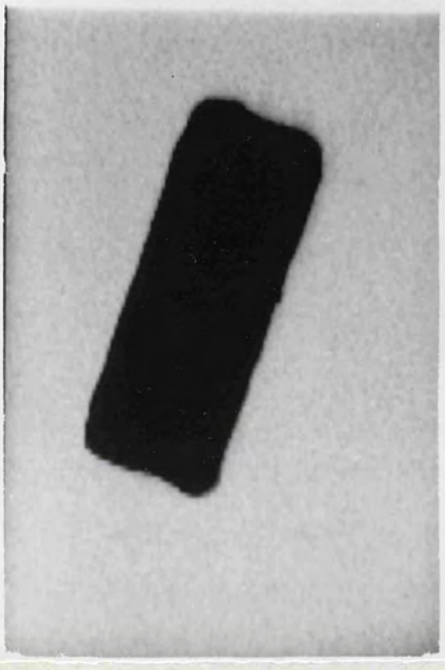
Fault three arises when one of the slabs of biscuit slips (Figure 3.10b) hence producing an incorrectly shaped (or sized) biscuit. (Note that the slabs are of a light colour. This condition will be used later on for the detection of show-through, i.e. lack of chocolate.) Examples of these products are shown in Figures 3.10a to 3.10g. Because an O-ring is circular, its shape (relative to the x and y axis) is independent of orientation whereas a rectangular biscuit is dependent on the orientation. The main problem lies in determining the orientation of the biscuit before analysis can proceed. This is a topic that frequently occurs in the examination of non-circular objects.

3.4.1 Previous Work on Object Orientation

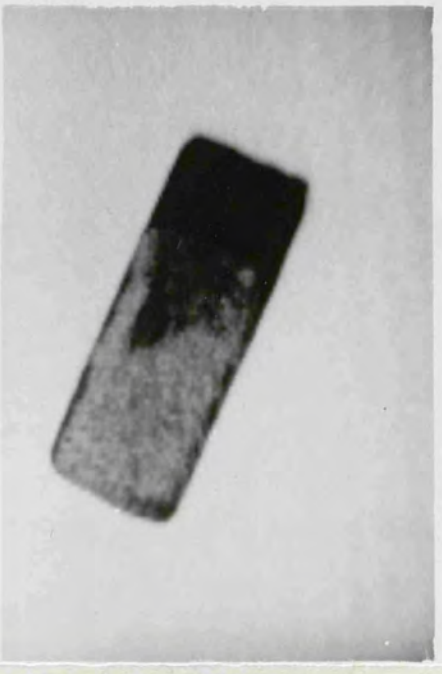
Similar work has been carried out by Brook and Purll [12] for on-line image acquisition and analysis of rectangular objects in a factory environment. Their technique was to determine the orientation of the object by detecting the end-most points in each of the four directions. The corner positions are then used to determine the

(b)

(d)



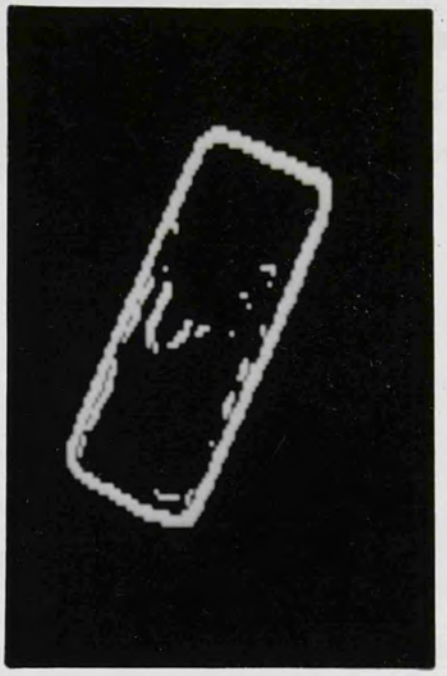
(a)



(c)

Figure 3.10 Examples of Biscuits

(f)



(h)

(e)

(g)

Figure 3.10 Examples of Biscuits

orientation by trigonometry. This is shown in Figure 3.11. Bolles and Cain [11] inspected metal door hinges. These were essentially rectangular in shape with several holes and metal extrusions. The scene was complicated by overlapping hinges. The orientation of each hinge was determined by the positions of the corners, metal extrusions and relative positions of the holes. The whole inspection algorithm took between 8-25 seconds using a combination of a PDP-11/34 and a VAX 11/780 on an image size of 240x240. The large differences in execution times were because the algorithm used the features of the objects to determine the orientation and therefore the time was dependent on the number of features that were visible and not obscured by other hinges.

The first method is unsuitable because, if the edge of the next biscuit on the production line enters the field of view of the camera, the corners would be incorrectly determined giving a false result. The technique also relies on a thresholded image. In a typical factory environment dirt marks on the belt, stray bits of chocolate, an uncovered product (no chocolate) or noise would also lead to incorrect results. The second method is unsuitable because the chocolate biscuit contains no visible features such as holes or extrusions. Corners were detected by moving a jointed pair chords around the boundary and comparing the angle between the chords; however, this method was found unsuitable for rounded corners - inspection of Figures 3.10a to 3.10g shows that these products have rounded corners.

Fan and Tsai investigated the inspection of chinese seal prints [38]. In order to determine the orientation of the print they 'drew' vertical lines through the print and determined the orientation by measuring the angle between each pair of lines and the edge of the print by trigonometry. This was averaged over the whole of the seal edge as shown in Figure 3.12. Several lines had to be drawn as breaks in the print were common. This method again required a thresholded

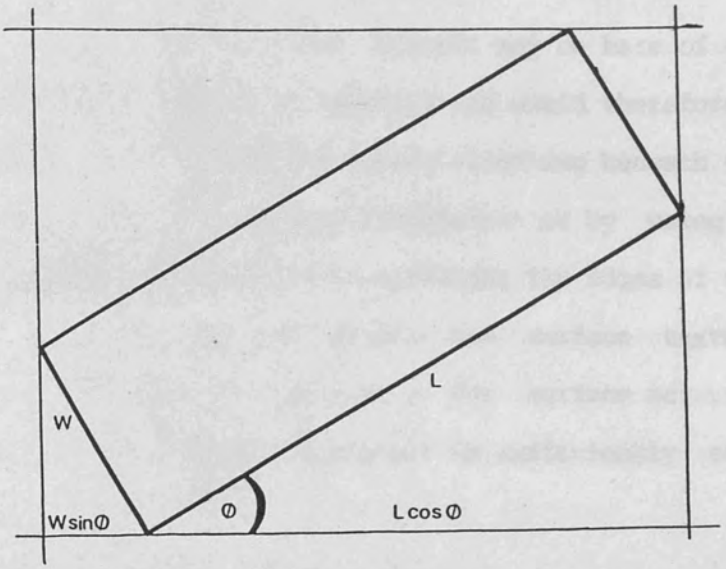


Figure 3.11 Determining the orientation by trigonometry

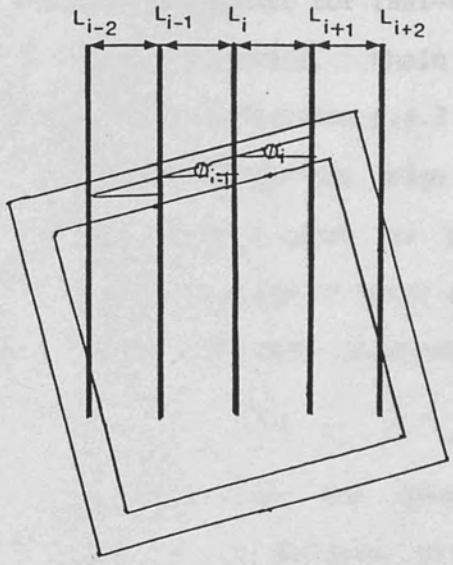


Figure 3.12 Determining the orientation by averaging over the seal edge

image and, bearing in mind the biscuit may be bare of chocolate in places, these areas would not be detected and would therefore fail. One solution would be to use back lighting (lighting beneath the conveyor belt, either by making the conveyor translucent or by using the crack between two adjacent conveyors) to highlight the edges of the biscuit. However, back lighting will not permit the surface texture of the biscuit to be detected - a necessity for surface scrutiny, e.g. to detect (in this case) whether the biscuit is sufficiently covered with chocolate.

The option of finding the corners of the biscuit and determining the orientation by trigonometry sounds appealing but these algorithms operating on grey level images [66],[88] have been found to be computationally expensive (~10s to ~40s on a PDP-11/73 operating on a 128x128 image) and are hence inadequate for real-time operation without the use of special purpose hardware. Chain coding the object and finding the corners as described in Section 4.4.3 could be used but this requires either a thresholded image or edge detection followed by thinning, both of which have disadvantages as previously noted. In general, chain coding is also sensitive to noise and may therefore prove to be unreliable under factory conditions; however, see Chapter 4 for an improved version.

The method we finally adopted was that developed by Dudani and Luk [34] which used the Hough transform method as described in Section 2.6.4. This method has the advantage that dirt marks on the belt and partial sections of other biscuits in the field of view of the camera do not effect the determination of the orientation of the biscuit - a necessary requirement for a factory environment.

Once the object orientation has been found, some workers have normalised the object by rotating it until the base is parallel with the x-axis before further analysis took place. This includes the work by Fan and Tsai on the inspection of square chinese seal prints. This involved normalisation of the square print by rotation before analysis. The rotation scheme used in this case was that developed by Hsieh and Fu which was originally used for the normalisation of IC chips. Rothstein's Code was used by Weiman for the rotation of images by shearing and stretching [126]. This had the advantage that it avoided trigometric calculations and only relied on addition, subtraction and division. This algorithm can also be parallelised and has been implemented on CLIP4 by Clarke and Ip [17]. Both of these methods appear suitable for real-time analysis but practice has shown that distortion of the image occurs for large angles.

Since these methods are either time consuming or inaccurate, it was decided not to normalise the object by orientation. Inspection would therefore have to take place at the angle the object is orientated. This may appear to impose an additional overhead to the run-time execution of the algorithm but this is not the case as shown by the timings in Section 3.4.9.

The algorithm is partitioned into two main sections: (1) determine model template of the biscuit; (2) detect faults using the template as a reference. The template is determined by finding the orientation of the biscuit by application of the Sobel operator and the Hough transform. From this, each side can be individually located since the perpendicular sides can be identified by their orientation and the parallel sides by the sign of their Sobel x-gradients. (One could consider basing the partitioning on only the Sobel sign of the Sobel x and y gradients; however, this will fail if there is more than one product in the scene, hence making the algorithm less robust).

The model template of the biscuit is then deduced by applying the least squares fit method to the edge points on each side of the biscuit and hence deducing the corners. Since the least squares fit method fits the best line to a set of points, spurious chocolate overflow and underflow will have little effect in contributing to the points on the side. The overall algorithm strategy is given in the next section.

3.4.2 Algorithm Strategy

The algorithm is partitioned into seven sections.

1. Locate biscuit and determine orientation.
2. Determine if biscuit is rectangular in shape.
3. Locate each side individually and hence determine corner points. This essentially fits a 'best fit' rectangle to the biscuit.
4. Determine if biscuit is the correct length and width.
5. Check for sufficient chocolate covering on the biscuit (within the rectangle defined in step 3).
6. Check for overflowing chocolate (outside the rectangle defined in step 3).
7. Reject biscuit if necessary.

All fault detection processing is confined to the area of the template. Each step is now discussed in detail.

3.4.3 Biscuit Orientation

As mentioned before, the method used for determining the orientation of the biscuit was that developed by Dudani and Luk. This used the Hough transform by accumulating the edge points with their

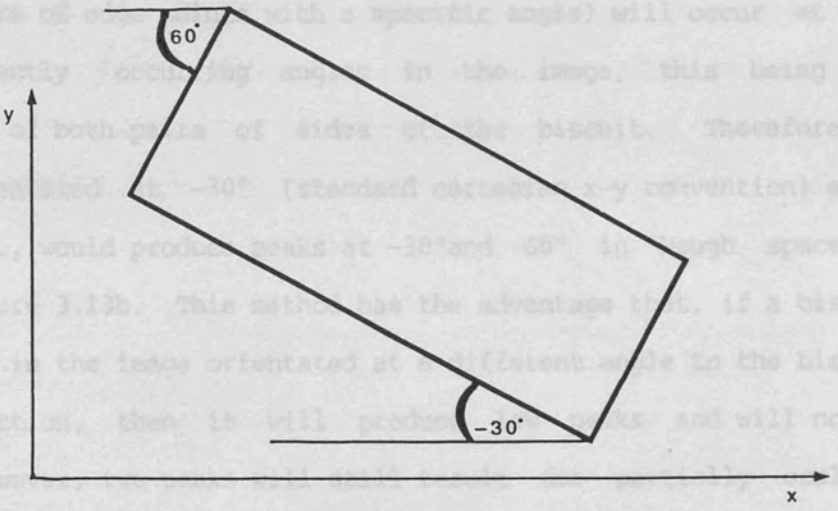


Figure 3.13a Orientation of the biscuit

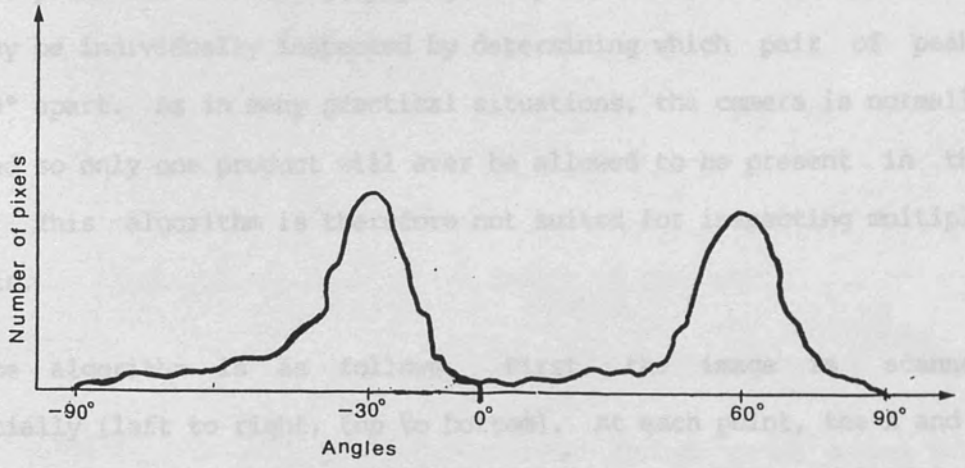


Figure 3.13b Peaks in Hough space reveal the most common angles in the scene

angles (determined from the Sobel operator), the idea being that peaks (large numbers of edge points with a specific angle) will occur at the most frequently occurring angles in the image, this being the orientations of both pairs of sides of the biscuit. Therefore, a biscuit orientated at -30° (standard cartesian x-y convention) as in Figure 3.13a., would produce peaks at -30°and 60° in Hough space as shown in Figure 3.13b. This method has the advantage that, if a biscuit is partially in the image orientated at a different angle to the biscuit under inspection, then it will produce low peaks and will not be detected; however, two peaks will still result for partially occluded biscuits, as long as there are enough edge elements to contribute to the accumulation. If a biscuit is partially in the image and orientated at the same angle as the biscuit under inspection (Figure 3.16b), then this will add to the accumulation - this problem is dealt with later. If multiple peaks are detected (implying the presence of several biscuits), each may be individually inspected by determining which pair of peaks are ~90° apart. As in many practical situations, the camera is normally adjusted so only one product will ever be allowed to be present in the image. This algorithm is therefore not suited for inspecting multiple biscuits.

The algorithm is as follows. First, the image is scanned sequentially (left to right, top to bottom). At each point, the x and y Sobel gradients are calculated. If the estimated gradient magnitude is greater than a fixed threshold, then an edge is taken to exist. At this point the angle of orientation (determined from the x and y gradients) is calculated. This, along with the (x,y) coordinates and the sign of the x-gradient are stored in an array. The algorithm is given in Figure 3.14.

The threshold 'thr1' is chosen such that the value satisfies an edge. A value of 180 was found to give good results with Figures 3.10a to 3.10g. The function 'sobelangle' calculates the angle of the edge from the x and y gradients (dx and dy) in the usual way. The function 'sign' is TRUE if the sign of dx is positive else it is FALSE (this is required later on).

```
1     PROCEDURE find_angles;
2      VAR dx, dy, dxplusdy :INTEGER;
3
4      BEGIN
5       count:=0; y:=0;
6       REPEAT x:=0;
7        REPEAT
8         dx := (P8+P1*2+P2) - (P4+P5*2+P6);  { find Sobel - dx }
9         dy := (P6+P7*2+P8) - (P2+P3*2+P4);  { find Sobel - dy }
10        dxplusdy := ABS(dx) + ABS(dy)
11        IF dxplusdy > thr1 THEN             { an edge element }
12         BEGIN
13          Q0:=255;                          { Set flag in Q-space }
14          edge[count].xx := x;              { save x,y,angle & sign }
15          edge[count].yy := y;
16          edge[count].theta := sobelangle(dx,dy);
17          edge[count].sign := sign(dx);
18          count := count + 1;
19         END; x:=x+1
20       UNTIL x=128; y:=y+1                  { do x }
21      UNTIL y=128;                          { do y }
22     END;
```

Figure 3.14 Finding the angles of the edge points of the biscuit

The next step is to scan the array 'edge' and group all angles such that, for every angle found, a counter is incremented in Hough space corresponding to that angle. The algorithm is given in Figure 3.15. Note that here, Hough space here is one-dimensional (as opposed to two-dimensional for the centre finding algorithm). A simple linear smoothing algorithm was then applied to 'hough_space' in order to smooth the effects of noise.

```
1     PROCEDURE group_angles;
2      VAR i :INTEGER;
3
4       FOR i := 1 TO count DO
5        BEGIN
6         hough_space[edge[i].theta] := hough_space[edge[i].theta] + 1
7        END;
```

Figure 3.15 Hough transform to determine the orientation of the biscuit

3.4.4 Determining the Peak Angles in the Image

The next step is to locate the peaks - this will determine the orientations of the sides of the biscuit. First suggestions might indicate that a fixed threshold would suffice but practice has found that this was inadequate, particularly if another section of biscuit entered the field of view of the camera or the image was noisy. It was therefore necessary to develop an automatic thresholding algorithm. This was accomplished by setting the threshold to a limit that exceeded the normal bounds. A scan along the array 'hough_space' is then carried out to see if two peaks (above the threshold) occurred. (Here, we are assuming that the object is rectangular so only two peaks will occur.) If less than two peaks were found then the threshold is decremented and the process is repeated. If more than two peaks are found at the same threshold or the threshold goes below a minimum, then the object is assumed not to be rectangular. This may appear a time consuming process but in practice, less than three iterations are generally needed (c.f. timings in Section 3.4.9). When two peaks are found, a simple calculation will reveal if they correspond to perpendicular lines before further analysis proceeds. This method has been found to detect angles with an error of $\pm 5°$. This large range (the Sobel from which the angles were derived is accurate to $\sim \frac{1}{2}°$) is because the biscuit has ragged edges. Thus, a fair degree of tolerance must be

taken into account.

The next step is to locate the points in each line individually in order to apply a least squares fit to the points.

3.4.5 Least Squares Fit Matching to the Edge Points

Two main problems occur for trying to fit a series of biscuit edge points to a line:

1. If the chocolate has insufficient chocolate coating, those parts uncovered will be detected as edges whose angles may coincide with those of the true edges (Figure 3.16a).
2. If the next product is partially visible in the scene and orientated at the same angle or a product defect is present that has edge points orientated at the same angle as an edge (Figure 3.16b), this too will give misleading results.

Applying a least squares fit to the points in either case will give a false result. The first problem is essentially to remove those points of the biscuit not associated with a true edge. An example of this is given in Figure 3.10h. From experiment, it was found that these false edges either form a closed loop or they are small. Successive application of the algorithm in Figure 3.17 three times removes all internal points not related to the edge of the biscuit.

The second problem in eliminating the 'false' points with the same angle on the edge of the biscuit (Figure 3.16b) requires more attention. For instance, if we applied a least squares fit to these points, the line will obviously be incorrect because of these false points, i.e. the pairs of points 1 and 2, and 3 and 4 in Figure 3.16b are considered to belong the same line because they have the same orientation and sign of

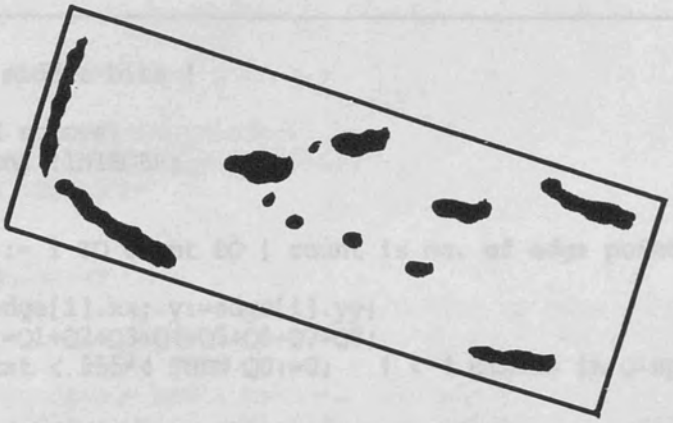


Figure 3.16a Edge points revealed on an uncovered chocolate biscuit. These have the same orientation as the 'true' edge points.

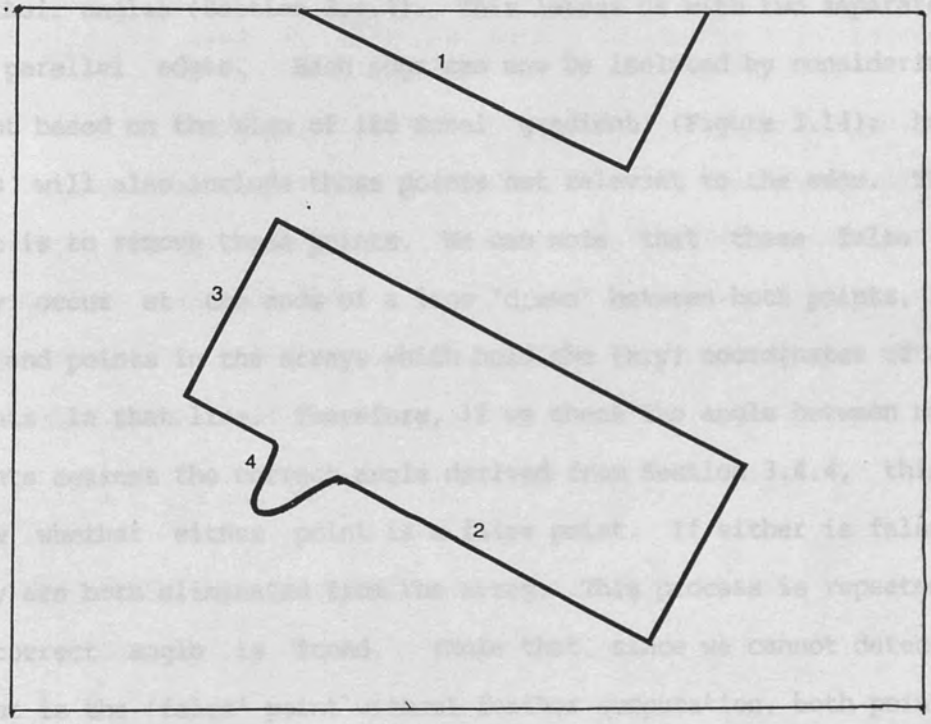


Figure 3.16b Edge points revealed on a biscuit with chocolate overflow. This is complicated by the fact that another biscuit with the same orientation is entering the field of view of the camera

the Sobel gradient. This can be solved as follows.

```
1     { Remove middle bits }
2
3     PROCEDURE remove;
4      VAR i, cnt :INTEGER;
5
6       BEGIN
7        FOR i := 1 TO count DO { count is no. of edge points detected }
8         BEGIN
8          x:=edge[i].xx; y:=edge[i].yy;
10         cnt:=Q1+Q2+Q3+Q4+Q5+Q6+Q7+Q8;
11         IF cnt < 255*4 THEN Q0:=0;   { < 4 pixels in Q-space set? }
12        END;
13     END;
```

Figure 3.17 Removing points not associated with the boundary of the biscuit

The first step is to isolate all four sides of the biscuit. The perpendicular edges are easily separated by partitioning the edges based on their angles (Section 3.4.4). This leaves us with two separate pairs of parallel edges. Each edge can now be isolated by considering each point based on the sign of its Sobel gradient (Figure 3.14); however, this will also include those points not relevant to the edge. The next step is to remove these points. We can note that these false points only occur at the ends of a line 'drawn' between both points, i.e. at the end points in the arrays which hold the (x,y) coordinates of all the points in that line. Therefore, if we check the angle between both end points against the correct angle derived from Section 3.4.4, this will show whether either point is a false point. If either is false, then they are both eliminated from the array. This process is repeated until a correct angle is found. (Note that, since we cannot detect which point is the 'false' point without further computation, both points must be eliminated.) In practice, this has found to produce good results as, despite the removal of 'good' points, there are generally enough points available to produce a good straight line fit to the edges. The

algorithm for this is given in Figure 3.18.

```
1    { Remove incorrect end points }
2
3    PROCEDURE remove_end_points;
4     VAR i, tmp, temp_angle :INTEGER;
5         goon :BOOLEAN;
6
7      BEGIN
8       i:=1; goon:=TRUE;
9       REPEAT                  { point_count = no edge points in line }
10       tmp:=point_count+1-i; { location in array of end of line }
11
12      { temp_angle = angle between both end points }
13      temp_angle:=angle(xarray[tmp],yarray[tmp],xarray[i],yarray[i]);
14      IF NOT inside_range(temp_angle) THEN remove_it(tmp,i) ELSE
15       goon:=FALSE;
16     UNTIL NOT goon;
17    END;
```

Figure 3.18 Removing points not associated with current edge

'remove_it' in line 14 removes the end points from the array. 'inside_range' also in line 14 merely checks the angle between both points.

Having done this, we can now apply the least squares fit method in order to fit the best line to these points. If we consider that d_k is the vertical error distance between the point at (x_k,y_k) and the proposed best-fit line (Figure 3.19), then the least squares fit method fits the line by minimising

$$d_1^2 + d_2^2 + \ldots + d_i^2$$

i.e. the sum of the squares of the vertical errors [101].
Consider the equation of the line

$$y = bx + a$$

If we have a set of points (x_1,y_1), $(x_2,y_2)\ldots(x_i,y_i)$, this can be written in matrix form [101] as

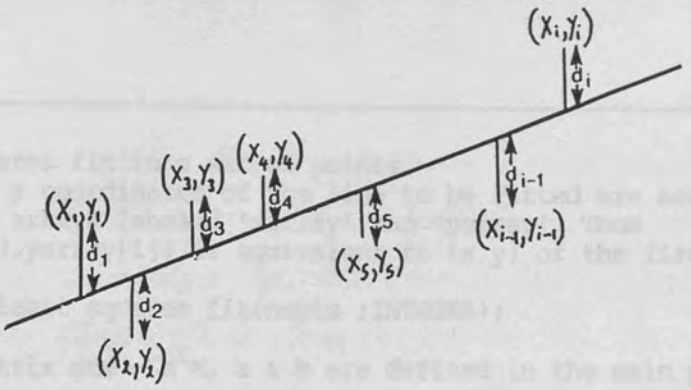


Figure 3.19 Least-squares fit to a set of experimental points. A line is found such that the distance (d) between the line and the points is minimised

$$y = \begin{bmatrix} y_1 \\ y_2 \\ . \\ . \\ . \\ y_i \end{bmatrix} \qquad M = \begin{bmatrix} 1 & x_1 \\ 1 & x_2 \\ . & . \\ . & . \\ . & . \\ 1 & x_i \end{bmatrix} \qquad v = \begin{bmatrix} a \\ b \end{bmatrix}$$

where b and a are the required values, i.e. the gradient and intercept of the line respectively. These values are obtained by the formula

$$v = (M^t M)^{-1} M^t y$$

where M^t is the transpose of M and $(M^t M)^{-1}$ is the inverse of the product of the transpose and the original matrix. In all cases this produces a 2x2 matrix. However, rather than multiply all elements out, further analysis reveals that a significant amount of simplification can be incorporated into the program thus reducing the amount of processing considerably. The algorithm is given below in Figure 3.20. Note that the procedure is called with the number of points to be fitted (nopts).

```
1    { Least squares fit to a set of points.
2      The x and y coordinates of the line to be fitted are assumed to
3      be in two arrays labeled 'xarray' and 'yarray'. Thus
4      (xarray[1],yarray[1]) is equivalent to (x,y) of the first point.
5
6     PROCEDURE least_squares_fit(nopts :INTEGER);
7      BEGIN
8       { 2x2 matrix mtm = M^tM, a & b are defined in the main program }
9
10      a:=0; b:=0; mtm[1,1]:=0; mtm[1,2]:=0; mtm[2,1]:=0; mtm[2,2]:=0;
11      FOR i:= 1 TO nopts DO mtm[2,1]:=mtm[2,1] + xarray[i];
12      FOR i:= 1 TO nopts DO mtm[2,2]:=mtm[2,2] + xarray[i]*xarray[i];
13      mtm[1,1] := nopts;
14      mtm[1,2] := mtm[2,1]
15      determinant := mtm[1,1]*mtm[2,2]-mtm[1,2]*mtm[2,1];
16
17     { 2x2 matrix mtm_inverse = (M^tM)^-1 }
18
19      mtm_inverse[1,1]:=mtm[2,2]/determinant;
20      mtm_inverse[2,2]:=mtm[1,1]/determinant;
21      mtm_inverse[1,2]:=-mtm[2,1]/determinant;
22      mtm_inverse[2,1]:=-mtm[1,2]/determinant;
23
24      FOR i := 1 TO nopts DO
25       a:=a+(mtminverse[1,1]+mtminverse[1,2]*xarray[i])*yarray[i];
26
27      FOR i := 1 TO nopts DO
38       b:=b+(mtminverse[2,1]+mtminverse[2,2]*xarray[i])*yarray[i];
29
30    END;
```

Figure 3.20 Least-squares fit to a set of points

The equation of the line is thus

$$y = bx + a$$

where a and b are determined in lines 24 and 27 respectively.

3.4.6 Finding the Corners and Size of the Biscuit

After the least squares fit to every line we will have four lines represented by the equations:

$$y = b_1x + a_1 \quad (1)$$
$$y = b_2x + a_2 \quad (2)$$
$$y = b_3x + a_3 \quad (3)$$
$$y = b_4x + a_4 \quad (4)$$

where lines (1) and (2) are parallel to each other (as are (3) and (4)),

i.e. b1~b2 and b3~b4 and lines (1),(3), (1),(4), (2),(3) and (2),(4) are perpendicular to each other. The corners of the biscuit are now found by considering that the position of one of the corners is the point where two perpendicular edges of the biscuit meet. For example, consider the two perpendicular lines (1) and (3), the position of the first corner of the biscuit can be found by rearranging the equations for (1) and (3) such that x and y in (1) equals x and y in (3), thus

$$x = (a_3-a_1)/(b_1-b_3)$$

and

$$y = ((a_1/b_1)-(a_3/b_3)) \; / \; (1/b_1-1/b_3)$$

where a_k and b_k, are determined from the least squares fit for the appropriate line. The length and width of the biscuit can then determined by Pythagoras's theorem from the coordinates of the biscuit.

3.4.7 Determining the Amount of Chocolate Coating

In order to determine the amount of chocolate coating on the biscuit, a simple suitably chosen threshold will suffice within the biscuit in order to distinguish between those parts of the biscuit that have been covered by chocolate and those parts that have not. Therefore, by tracking across the biscuit applying the threshold (all areas within the model template from the last step) and accumulating the number of pixels over a threshold, a comparison between this value and the area scanned will produce a percentage of the biscuit not covered with chocolate. This figure is compared with an acceptable figure (1% in this case) to determine if the biscuit should be rejected.

3.4.8 Determining the Amount of Chocolate Overflow

This step is basically the opposite of the last step. Here, a threshold (intensity of the chocolate) is applied to a small area

outside the model template. A percentage of the biscuit outside the template is compared against an acceptable value (0.5% in this case) in order to determine whether there is significant chocolate overflow. The run-time output and execution times follow in the next section.

3.4.9 Run-time Results and Timings

Below are the run-time results for the test images, Figures 3.10a to 3.10g. The resultant images are given in Figures 3.21a to 3.21g. The white outline surrounding the biscuit defines the template from which we will work. Note how extrusions and lack of chocolate are ignored. The white shading signifies both uncovered areas of biscuit (within the template) and excess chocolate (outside the template). Each set of results is accompanied with a PASS/FAIL label.

Following this are the timings for each part of the algorithm in milliseconds (Table 3.2). All times are derived from a PDP-11/73 processor operating on a 128x128 image. The times for SIP (Chapter 5) are also given. It should be noted that the thresholds in the algorithm can be altered for optimum inspection. The thresholds chosen here were determined experimentally and gave good results. Note that determination of the size of the biscuit has been omitted as in all cases the time taken was 1ms. The initialisation procedure took 2ms.

Fig 3.21	Sobel (ms)	Group&Smooth angles (ms)	Determine angles (ms)	Least Squares (ms)	Chocolate coating (ms)	Chocolate overflow (ms)	Total time (ms)
a	1268	26	15	288	146	52	1798
b	1325	31	15	374	158	59	1965
c	1338	32	5	373	212	55	2018
d	1288	28	2	312	147	51	1831
e	1301	29	10	320	156	54	1855
f	1300	28	5	316	151	52	1855
g	1293	28	7	308	145	52	1836

Table 3.2 Breakdown of execution times of the rectangular biscuit algorithm on seven different images

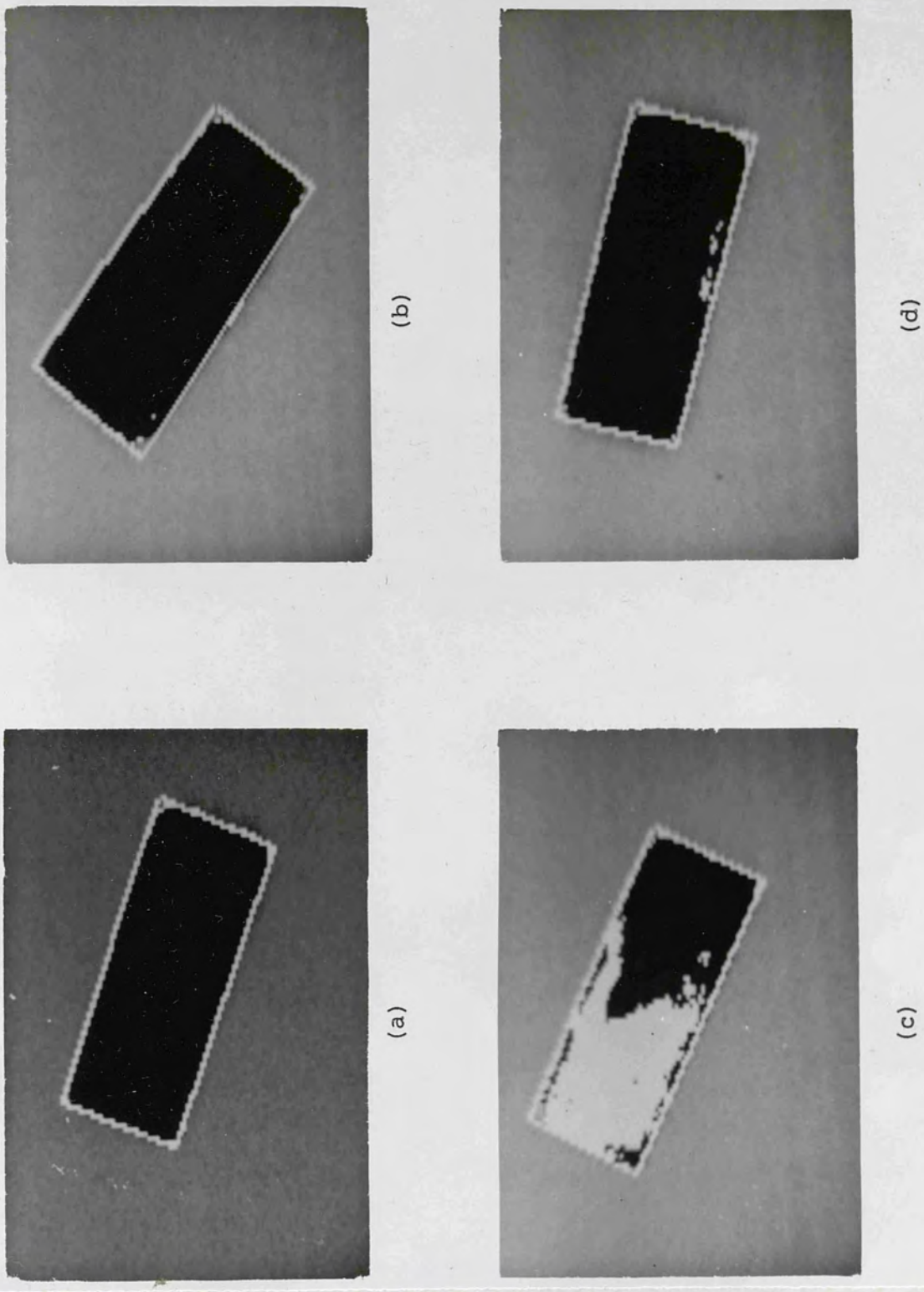
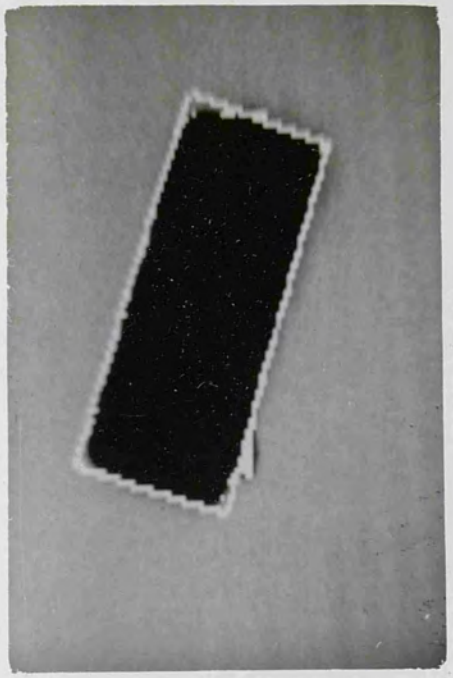
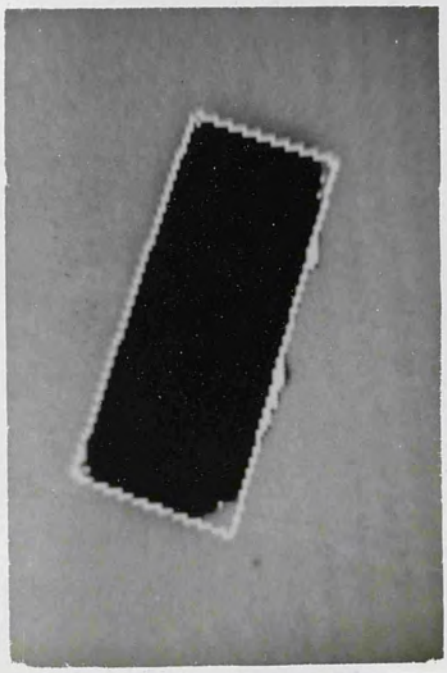


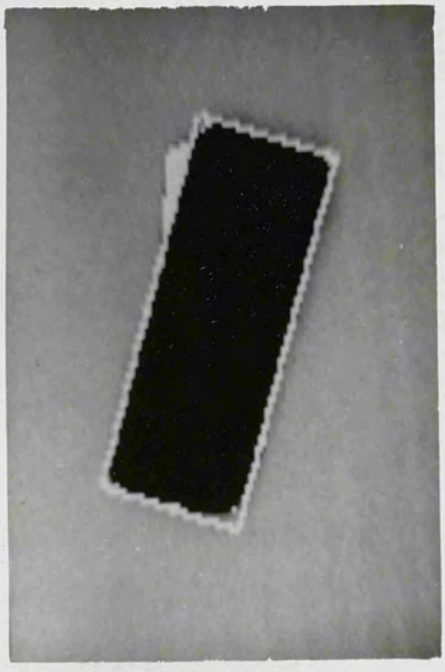
Figure 3.21 Result of the Biscuit Algorithm



(f)



(e)



(g)

Figure 3.21 Result of the Biscuit Algorithm

(a) Orientation of biscuit sides : -27° and 78°
Length of sides : 98 97 39 40 PASS
Biscuit show through : 0.31% PASS
Chocolate overflow : 0.16% PASS
Biscuit PASSED inspection.

(b) Orientation of biscuit sides : -47° and 64°
Length of sides : 109 109 39 39 FAIL
Biscuit show through : 0.47% PASS
Chocolate overflow : 0.11% PASS
Biscuit FAILED inspection.

(c) Orientation of biscuit sides : -35° and 77°
Length of sides : 101 101 40 37 FAIL
Biscuit show through : 34.1% FAIL
Chocolate overflow : 0.0% PASS
Biscuit FAILED inspection.

(d) Orientation of biscuit sides : -23° and 82°
Length of sides : 97 96 39 41 PASS
Biscuit show through : 13.1% FAIL
Chocolate overflow : 0.0% PASS
Biscuit FAILED inspection.

(e) Orientation of biscuit sides : -28° and 77°
Length of sides : 98 98 43 42 FAIL
Biscuit show through : 0.16% PASS
Chocolate overflow : 0.48% FAIL
Biscuit FAILED inspection.

(f) Orientation of biscuit sides : -25° and 81°
Length of sides : 100 97 41 42 PASS
Biscuit show through : 0.64% PASS
Chocolate overflow : 0.50% FAIL
Biscuit FAILED inspection.

(g) Orientation of biscuit sides : -25° and 76°
Length of sides : 97 98 38 38 PASS
Biscuit show through : 0.29% PASS
Chocolate overflow : 1.4% FAIL
Biscuit FAILED inspection.

From Table 3.2, it is interesting to note that in the initial pass over the image, the Sobel procedure constitutes ~70% of the total processing time. This algorithm was implemented on SIP using a lookup table for the determination of the angles. The whole algorithm showed a 30 times speed improvement over the PDP-11/73 timings. This represents a processing rate of 8-9 biscuits per second, including I/O.

3.5 PROGRAM OPTIMISATION

Conventional algorithms may require several passes over an image in order to extract the information required. These algorithms usually involve parallel algorithms such as a Sobel, threshold and filters which are time consuming when 128x128 and a 256x256 images are used. Real-time industrial algorithms extract features and measurements from a known product. This product may typically occupy 30% of the image space. If it is just required to locate the edge elements to determine the necessary information then this may typically represent less than 5% of the image space.

By locating the object and limiting the bounds of the image to only the space occupied by the object, then a significant amount of redundant information can be eliminated hence increasing the performance of the algorithm. Alternatively, if the majority of the information required can be derived from the edge elements (as in the two algorithms described) then the data set can be reduced to only a small set of values (typically less than 1000 for a 128x128 image). The overall effect of this is that the execution time of the algorithm can be significantly decreased.

Below is a list of several points that can help in improving the execution time of an algorithm.

1. Floating point calculations may be present in the algorithm. These should occur only when necessary. Integers should otherwise be used at all times.

2. Trigometric and frequently occurring calculations can often be stored in a lookup table. Thus, the time to do the calculation is in effect the access time of the memory.

3. Limit the area of processing to the area occupied by the object.

4. Only apply an edge detector where necessary - see Section 3.5.1 for a fuller explanation.

5. Incorporate maximum information about the object into the algorithm. For instance, we assumed that the biscuit (Section 3.5) was rectangular hence eliminating the need to determine the features.

6. Writing critical parts of the algorithm in machine code. This is a debatable point as the execution time of an algorithm written in a high level language often depends on the quality of the code generated by the complier.

3.5.1 A Priori Knowledge for Industrial Recognition

In industrial recognition, we have the advantage that we know what the product should 'look' like and the conditions in which it appears, i.e. the reflectance of the product and background illumination etc. We can therefore take advantage of this knowledge in order to improve the performance of an algorithm. For instance, if we know that the majority of edge elements lie between two distinct grey-level values because of the lighting conditions set up, then, rather than calculate a Sobel for every point in the image, it is only necessary to calculate a Sobel for those elements that lie between the two previously mentioned values.

Thus, the number of Sobel calculations can be dramatically reduced, i.e.

```
y:=0;
REPEAT x:=0;
 REPEAT
  IF (P0 > min_thresh) AND (P0 < max_thresh) THEN
   CALCULATE_SOBEL;
   x := x+1;
 UNTIL x=128; y:=y+1
UNTIL y=128;        { Do for a 128x128 image }
```

Table 3.3 shows the percentage decrease in execution time for both the Sobel procedure and the overall execution times applied to the rectangular biscuit algorithm over all seven images.

% decrease in time for Sobel operation (ms)	% decrease in time for overall algorithm (ms)
37	26
54	44
20	10
53	40
47	32
54	27
48	33

Table 3.3 Table of the reduction in execution time for the Sobel operation and the whole algorithm after program optimisation

From the above figures, we can see that a median of 48% reduction in execution time for the Sobel operation can be achieved resulting in a median of a decrease in program execution time of 32%. (Note that the median seems to be a better measure of the "typical" execution time than the average in this case, as the median represents where the bulk of the execution times lie.) Therefore, we can conclude that an important factor in real-time industrial processing is to detect and eliminate as much redundant information as possible by incorporating maximal a priori knowledge of the scene into the algorithm, hence only processing those parts that are necessary to extract the required information.

3.5.2 Eliminating Bottlenecks using Hardware Accelerators

In such algorithms as those described above, there usually exists a bottleneck that predominates over the execution time of the algorithm. If we consider the O-ring algorithm, we can see that the initial pass of applying a Sobel represents ~70% of the total execution time. Analysis of the procedure that contains the Sobel calculation shows that 90-95% of the time is spent calculating the Sobel, i.e. the Hough calculation only accounts for 5-10% of the execution time of the procedure. This would suggest that the Sobel calculation should be carried out in hardware. The sections of an algorithm that should be executed in hardware can be determined by a C*T test (Section 7.5.2), but software optimisation should also be considered. As a typical example, a recent industrial analysis algorithm developed at RHBNC initially took over a minute but was reduced to ~2s (running on a PDP-11/34 with a 128x128 image) after software optimisation [29]. Further increase in performance was gained by upgrading the processor to a PDP-11/73.

3.6 SUMMARY

This chapter has looked at two inspection algorithms that adopt different forms of the Hough transform. These algorithms were different in the sense that one inspected O-rings whose orientation was independent of the x and y axis whereas the other inspected a rectangular product which was orientation dependent. This showed the usefulness of the Hough transform in two different applications. Although these programs have only been applied to one product, they can be applied to similar products, e.g. circular chocolate covered biscuits or rectangular machine parts. Only minor alterations would be necessary to the programs, i.e. changing of thresholds, etc. but the principle methods for detection and scrutiny would remain the same. (With minor alterations to the rectangular biscuit algorithm, e.g. looking for three

peaks in Hough space for triangular products, different straight-lined shaped products can be inspected.)

An investigation of the concepts of real-time algorithms showed that, by extracting only relevant information and hence reducing the data set of the image, this allowed real-time execution times to be accomplished on special hardware (Chapter 5). A priori knowledge improved the performance dramatically and helped in eliminating redundant steps. Analysis of the timings of both algorithms showed that the initial phase of locating the object and applying the Hough transform represented more than 60-80% of the total execution time but was significantly reduced after software optimisation. Overall, important aims in devising inspection algorithms include accuracy, robustness, speed and low cost implementation (Chapter 5). These algorithms fulfil all of these requirements.

Application of a parallel algorithm to pre-process an image may constitute ~70% of the total execution time on a sequential processor. This is an important fact that is typical of many image processing algorithms (c.f. algorithms above, [29]) and will be discussed in detail in Chapters 4 and 8. We will now look at the sequential implementation of both algorithms. This will allow us to investigate the validity of parallel and sequential processors in industrial inspection.

CHAPTER 4

SEQUENTIAL IMPLEMENTATION OF INSPECTION TASKS

"Christ! What are patterns for?"

Patterns in the Complete Poetical Works of Amy Lowell

4.1 INTRODUCTION

Chapter 3 described two inspection algorithms that exhibited both parallel and sequential tasks (Section 2.2). For maximum efficiency, it is important to try to match the task to the architecture of the machine, i.e. sequential tasks should be performed on a sequential machine and parallel tasks on a parallel machine. Also, it is generally easier to implement a sequential algorithm on a sequential machine than it is on a parallel machine and vice-versa.

When relating to industrial inspection other factors such as cost must also be considered. A discussion of these points is undertaken which leads to an investigation into algorithms suitable for sequential implementation. The chain code is cited as being amongst the most useful. The main disadvantage of the chain code is that it is generally restricted to binary images and susceptible to noise degradation. This method is extended to the grey scale case and a novel method for improving the robustness of measurements taken from the chain code in noisy images using the Hough transform is described. This is applied to

the biscuit algorithm described in Section 3.4 and the O-ring algorithm described in Section 3.3. First, let us discuss the possibility of improving the speed of a parallel algorithm by using sequential processes.

4.2 SEQUENTIAL ALGORITHMS IN INDUSTRIAL INSPECTION

A significant amount of information required by an inspection task is contained at the edges of the object, e.g. derivation of the centre points of an O-ring (Section 3.3) and the orientation of the edges of the biscuit (Section 3.4). Edges typically represent ~5% of the total image space. Analysis of the algorithms in Chapter 3 shows that the pre-processing stage entails applying a Sobel edge detector to every point in the image, making ~95% of the processing redundant. A more efficient method would be to apply the Sobel to only the edge points - this suggests tracking around the border of an object. This concept displays certain advantages over the parallel approach - in particular, computation time is reduced and information is localised. Thus, information such as perimeter, area, height etc., can be easily derived - this would be otherwise difficult to achieve on a parallel processor.

There are many other reasons for implementing an inspection algorithm sequentially. Often, if appropriate techniques are used and the algorithm is optimised along the lines given in Section 3.5, a sequential processor can achieve a lower algorithm execution time than a parallel processor. Considering that the cost of a sequential processor is often much less than a parallel processor, it would appear that a sequential processor is a more cost-effective solution for industrial recognition. A comparison with a parallel machine should only be made with an optimally programmed sequential machine. One should not justify the additional cost of hardware for a parallel processor when compared

to a non-optimally programmed sequential machine [107]. However, there are algorithms that are inherently parallel such as thinning (here we take the view that thinning is more appropriate for implementation an a parallel machine that a sequential machine). For this reason, Chapters 5 and 6 investigate a dual-processor architecture.

From this discussion, it appears that sequential processors are more applicable to industrial inspection than parallel processors. It would also appear that localising an object by tracking around its boundary and extracting measurements has many advantages over other techniques. However, the usefulness of these methods depends on the accuracy and the robustness required. This is discussed next.

4.3 BOUNDARY EXTRACTION ALGORITHMS

Various algorithms have been developed for extracting or using the boundary of an object and deriving some measurement (e.g. area), or finding a measure of shape description [49]. One such method is that of the Fourier descriptors. Here, a tracking routine is used to track around the boundary of an object; commonly, rather than representing the edge elements on the standard x-y plane, they are represented on a complex plane as in Figure 4.1. Thus, for each boundary point encountered, a complex number is obtained. At the end of the trace, the discrete Fourier transform (DFT) is calculated from the list of these points which is referred to as the Fourier descriptor (FD). Since the DFT is reversible, no shape information in principle is gained or lost.

Because manipulations occur in the frequency domain, it turns out that dependence on size and orientation can be eliminated. For this reason, Fourier descriptors have found their use in optical character recognition (OCR) [69] and more generally in object recognition such as the identification of three-dimensional aircraft [99]. In OCR, the

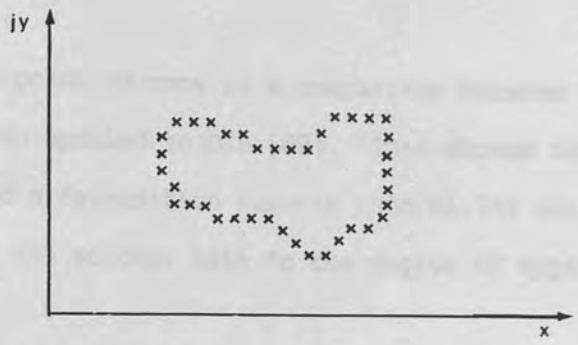


Figure 4.1 Complex plane representation of a boundary

boundary of the character is tracked and its FD is calculated. As Lai, Ching and Suen pointed out, when two closed curves which differ only in position, orientation and size with analagous starting points are transformed, they have identical FD's. However, because the Fourier transform is computationally expensive, this approach is undesirable for real-time industrial inspection.

Probably the most popular boundary extraction routine is the chain code [45] (most others are merely variations on this theme [47]). This has the ability to extract measurements such as the perimeter and area of an object while tracking. This makes it highly attractive for industrial inspection since these features are often required. The chain code has been successfully employed for automatic chocolate decoration [19] using such measurements. Here, the aim was to determine which chocolate was currently in view of the camera and to apply the appropriate chocolate decoration. This involved calculation of the area, perimeter, centre of gravity and location of the corners. As each chocolate was dark and the background was light (and surface information was not required), a simple threshold sufficed for locating the chocolate and determining its shape.

An interesting point to note is a comparison between the FD's and the chain code when applied to OCR [69]. This showed that the Fourier Descriptors achieved a recognition success rate 81.74% while the chain code achieved a 93.74% success rate to the degree of approximation they took.

4.3.1 Problems with Boundary Extraction Algorithms

The main problem with boundary tracking routines is that they require a binary image. In the past, two approaches have been used to achieve this. The first is to threshold the image. This is a common technique to employ as it is probably the least computationally expensive of pre-processing routines. However, this is not always convenient; for instance, consider Figure 3.10c. A straight threshold would mean that those edges not covered with chocolate will go undetected. Another problem is that if an incorrect threshold is applied, this may produce meandering boundaries which do not represent the true boundary. This makes the chain code sensitive to noise.

The second approach is to apply an edge detector e.g. a Sobel, and to threshold the gradient magnitude in order to produce a binary image of the edges. Since this usually produces an edge of 2-3 pixels wide, a common technique is to thin the edges before tracking. However, both of these are time consuming operations without the use of a parallel processor as they are essentially parallel operations. Also, (and probably more important), it is necessary to apply both operations to the whole image which would again mean a significant amount of redundant processing.

It is clear that the chain code has strong possibilities for reducing the execution time of an algorithm relative to those methods just discussed and is investigated in the next section. Subsequent

sections extend these ideas and implement them on both industrial algorithms presented in Chapter 3. Results follow each implementation. First, let us discuss the chain code.

4.4 THE CHAIN CODE

The chain code was originally developed by Freeman [45]. The purpose of the chain code was to be able to represent an arbitrary geometric shape by a set of numbers suitable for computer analysis. Freeman drew up three points that a coding scheme for line structures should achieve:

1. It must faithfully preserve the information of interest.

2. It must permit compact storage and be convenient for display.

3. It must facilitate any required processing.

Representing a boundary by the chain code has many advantages, for instance, the object can be rotated, expanded, shrunk and smoothed by manipulation of the code. Structural features can also be derived such as the area, perimeter and the location of straight lines and corners. The main advantage is that it has a significant speed improvement over other methods both in the derivation and the analysis. In order to clarify these points, let us first discuss how the chain code is derived.

4.4.1 Derivation of the Chain Code

The first step for deriving the chain code is to initially locate the object. If we consider the simple case of chain coding an object in a binary image, finding the object is simply a case of searching for the first pixel (edge point) with a value designated as being part of the object.

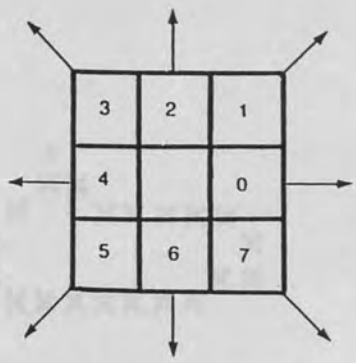
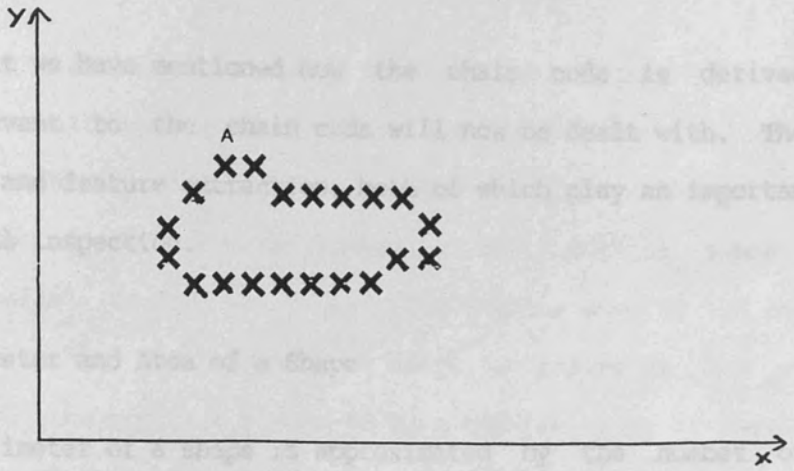


Figure 4.2 One of eight possible directions in a 3x3 window that represents the chain code

On encountering the first edge point of an object, the chain code follows a tracking process. In the case of a quantised binary image, at the first encountered edge point there is only one of eight possible directions that the next position could be as shown in Figure 4.2. The chain code chooses the next position by searching for the new edge element that is in the most counter-clockwise point: the code corresponding to this new edge element is then stored and the position in the image is adjusted to the position of the new edge element. This is repeated until the original position has been reached moving in the same direction (in the case of a closed contour) or no further edge elements exist (in the case of an open contour). Figure 4.3 depicts an example of how the chain code is derived starting at point A. (Note that problems occur at junctions caused by the intersection of two lines. However, we are not concerned with these problems as they do not occur with circular or rectangular shapes.)

The outline of the object can be reconstructed by scanning the code sequentially and adjusting the x,y coordinates as appropriate. A useful feature of the chain code is that, since all positions are relative to each other, only the starting coordinate need be changed in order to shift the shape in any direction. Another advantage is that only three bits (eight possible values) are necessary to store the required information per edge element.



Chain code: 07000076454444443211

Figure 4.3 Boundary of an object and its chain code

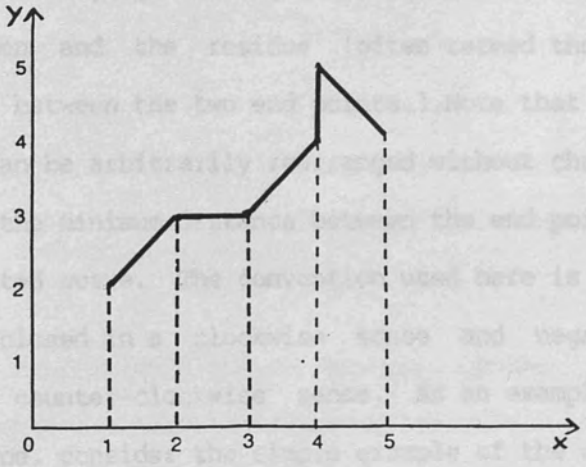


Figure 4.4 Finding the area under a curve

Now that we have mentioned how the chain code is derived, two topics relevant to the chain code will now be dealt with. These are measurement and feature extraction, both of which play an important role in industrial inspection.

4.4.2 Perimeter and Area of a Shape

The perimeter of a shape is approximated by the number of even numbers in the code plus $\sqrt{2}$ times the number of odd digits. This is a typical example of a measurement that would otherwise be difficult to extract any other way (see Section 4.6 for an application of this method).

If a shape is closed and simply connected (which is usually the case), its integral represents the area enclosed by the chain. If the shape is not connected, then the integral represents the area enclosed by the chain with the x axis. (A simple test for closure is to perform a path reduction test [46]. If the curve vanishes then it is closed else it is open and the residue (often termed the closure) is the minimum distance between the two end points.) Note that the digits in the chain code can be arbitrarily rearranged without changing the length of the curve or the minimum distance between the end points in the case of a non-connected curve. The convention used here is that the area is positive when enclosed in a clockwise sense and negative if it is enclosed in a counter-clockwise sense. As an example of finding the area under a shape, consider the simple example of the code

10127

the diagram of which is given in Figure 4.4. The following procedure was suggested by Freeman [45] to determine the area of an arbitrary shape. First, a modifier B is defined to represent the change in distance of the curve from the x-axis. This in effect gives the value of the leading edge of the particular vertical column. Thus, in our

example, the area (A) is initially zero and the modifier B is set equal to two. The initial slope of '1' increases the area by the amount of the first vertical column to 2½ and the modifier to change to three. The next slope of '0' causes no change in the modifier since it is running parallel to the x-axis but increases the area by the amount of the modifier to 5½. The next step of slope '1' increases the area by the value of the modifier plus ½ to nine and the value of the modifier to four. The slope of '2' gives no change in the area but increases the modifier to five. The slope of '7' increases the area by the modifier minus ½ (to 13½) and decreases the modifier to four. This leaves us with a total area of 13½ square units. This procedure can be summed up by developing a set of rules for all eight slopes. These are summed up in Table 4.1.

Slope	Change in Area	Change in modifier (B)
0	+B	+0
1	+B+½	+1
2	+0	+1
3	-B-½	+1
4	-B	+0
5	-B+½	-1
6	+0	-1
7	+B-½	-1

Table 4.1 Table for automatic calculation of the area under a chain coded figure

Thus the area of an arbitrary object, whether closed or open can be determined by simple manipulation of the chain code. Other measurements such as the location of the centre of gravity, the width and height of an object can also be derived quite simply. These will not be discussed here but can be readily found in literature on the subject of chain codes [46],[47]. One advantage with the chain code is that these measurements can be calculated as the code is being derived making it an attractive proposition for real-time calculation of measurements of

objects involving a minimal amount of hardware.

We will now discuss how to extract features from the chain code. This method (in general terms) derives a measurement of curvature and has been applied for locating corners in an object. This topic is covered with a view to implementing the algorithms described in Chapter 3.

4.4.3 Finding Corners in a Shape

One of the more useful possibilities that has emerged from the chain code is the ability to measure curvature of line figures with a view to detecting the presence of corners. The basic technique was developed by Freeman and Davis [48] who observed the behaviour of a line segment of length s being moved around the figure as shown in Figure 4.5. The angular differences d_i between successive segment positions are used as a smoothed measure of local curvature along the chain. Thus, if the curvature only deviated slightly from zero ($d_i \tilde{} 0$) across a sequence of t positions, then this would indicate a straight line of length s+t. However, if the deviation was non-zero but constant across t positions then this would indicate a curve of length s+t of uniform curvature.

Freeman and Davis went on to define a corner as being the concatenation of two straight lines at an angle as in Figure 4.6a or two straight lines joined by a curve as depicted Figure 4.6b. Thus, applying a straight line segment of length s across Figure 4.6a, we will get a d_i of approximately zero along both lines for $p<B$ and $p>B+s$ (where p is the position of the leading point of the line segment) and a nonzero d_i for the sections $p>B$ and $p<B+s$. This nonzero condition lasts for s+1 positions for sharp corners and m+s positions for cases such as those in Figure 4.6b where m is the number of nodes in the curve,

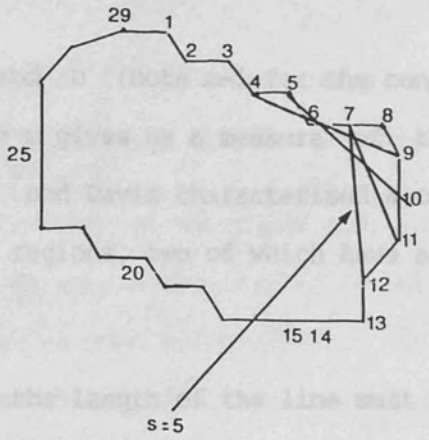
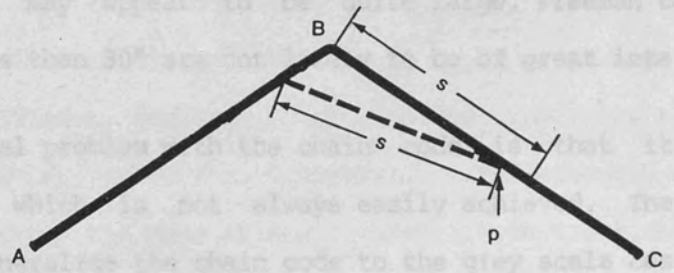
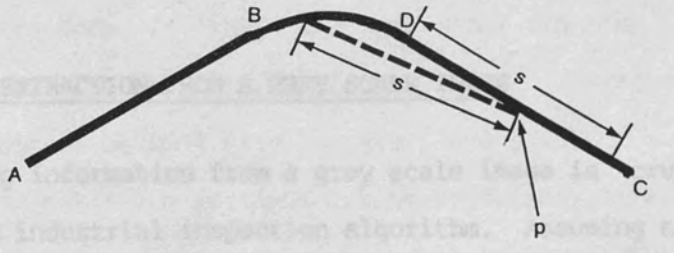


Figure 4.5 Line segment scanned around a chain coded figure



(a)



(b)

Figure 4.6 Illustration of the effect of a line-segment at (a) sharp corners and (b) rounded corners

i.e. between B and D (note m=1 for the concatenation of two straight lines). The value m gives us a measure of the curvature. Thus, in general, Freeman and Davis characterised a corner by the concatenation of three distinct regions, two of which have a d_i of zero and one which has a nonzero d_i.

In practice, the length of the line must be large enough to avoid noise such as that produced by quantisation that causes "wobble" of the line segment yet small enough to detect corners close to each other. A length between 5-13, depending on the closeness of the curves has been found [48] to be suitable for detecting corners of greater than 30°. Although this may appear to be quite large, Freeman commented that corners of less than 30° are not likely to be of great interest anyway.

The general problem with the chain code is that it requires a binary image which is not always easily achieved. The next section attempts to generalise the chain code to the grey scale case. By doing this, we will hopefully show that the chain code can be usefully used in an industrial environment and can gain a significant speed improvement over the methods used in Chapter 3, while retaining the speed and robustness necessary for industrial analysis.

4.5 BOUNDARY EXTRACTION FROM A GREY SCALE IMAGE

Extracting information from a grey scale image is crucial to the success of an industrial inspection algorithm. Assuming a binary image greatly limits the applications because (1) it is not always accurately achieved and (2) information content in the image is reduced, e.g. information on surface texture is removed. It is for these reasons that the chain code has been restricted to relatively simple tasks where binary images suffice. Here, we attempt to show how the chain coded boundary of an object can be extracted from a grey scale image with

little computational effort, and show how it can be applied to more complex inspection tasks such as those in Chapter 3.

From the explanation of the chain code (Section 4.4), each element in the 3x3 window is examined to see if it belongs to the boundary of an object. If instead we took every element in the 3x3 window as the centre of another 3x3 window, and applied an edge detector (the Sobel was used in the actual implementation) to each position, and thresholded the magnitude of the gradient (the threshold defining an edge point), a binary edge image will be produced, only in the region of interest. (The first edge point is found by applying the Sobel and thresholding the magnitude of the gradient.) Since the edge is extracted sequentially, this eliminates the need to apply an edge detector to the whole image. In fact, analysis shows that the time to extract the edges is reduced by a factor ~5. However, simultaneously with the edge extraction process, the edge is chain coded. From Table 4.2, this adds very little time (~20ms) to the total execution time.

One disadvantage with the chain code is that it is susceptible to noise; however, because a thresholded magnitude is applied to the result of the Sobel, the effects of noise are greatly minimised. One advantage with the chain code is that the perimeter and area can be derived simultaneously with the derivation of the code. Therefore, since we expect a product to be of a certain size (~280 pixels for the chocolate biscuit), anything below a perimeter value of 50 and above 400 can be ignored and the scan continued. (Note that these are purely arbitrary values and are dependent on the application.) This produces an effective way of ignoring partial or broken products entering the field of view of the camera.

We shall now consider the implementation of these techniques on the chocolate biscuit algorithm (Section 3.4). The results derived from this are then compared with the original results given in Section 3.4.9.

4.6 SEQUENTIAL IMPLEMENTATION OF THE CHOCOLATE BISCUIT ALGORITHM

Referring back to Section 3.4, the reason for determining the orientation of each edge of the biscuit was to isolate the edge points belonging to each line of the rectangular biscuit. The least squares method was then applied to each group of edge points which determined a template for the biscuit, and the corners were then located. One may note that the isolation of each line is trivial if we just consider the sign of the x and y gradients; however, if more than one (or partial) product exists in the image, this method would no longer work. This is not the case with the tracking method since only one object is being scrutinised at any one time. In this case, a comparison of the sign of x and y gradients will suffice; however, the aim of this exercise is to extract the same information, i.e. we must determine the orientation of the biscuit. (The orientation is in fact useful in some areas of industrial recognition where it is necessary for a robot arm to pick up the object.) This will then provide a more realistic result since both algorithms will only differ in their implementation.

4.6.1 Finding the Corners of the Biscuit

After chain coding the biscuit using the algorithm in Section 4.5, the obvious solution for finding the corners of the biscuit is to apply the corner finding algorithm described in Section 4.4.3. However, this was found unsuitable for rounded corners (such as those in the test images 3.10a to 3.10g); therefore, some other technique had to be developed. The method used was similar to that described in the original algorithm, i.e. finding the most prominent angles in the scene,

only here we are localising the information to the biscuit.

This highlights one of the advantages for using the chain code. Because information is local to the object, one can in principle have a number of products in the scene and scrutinise each one in turn. Thus for each product, an accurate measurement can be made of (in this case) the most prominent angles of the boundary of the product. In the previous algorithm where the most prominent angles in the whole image were taken into consideration, multiple peaks would exist when many products were present. Thus, that method is only suitable when only one complete product is present.

The main problem now is to extract the most prominent angles in the object. Having done this, we can isolate each line, and then apply the least squares fit as before (Section 3.4.5) to the edge points of each line. In order to do this, a novel approach of applying the Hough transform in conjunction with chain code was used. This is described in the next section.

4.6.2 Application of the Hough Transform to the Chain Code

As we have seen in Chapter 3, the Hough transform is a useful method for improving the robustness of an algorithm. In order to find the most prominent angles in the biscuit, we can apply the Hough transform and a technique similar to that described in Section 4.4.3, i.e. to move a line segment along the boundary (Figure 4.5) of the biscuit. However, instead of measuring the deviations between angles as is commonly done to detect corners, we can instead accumulate (in the Hough array) the angles the line segment (ls) makes with the x axis for every position in the code, i.e.

```
hough[angle_of_ls] := hough[angle_of_ls] + 1;
```

Thus, peaks in the Hough array will occur for the most frequently occurring angles in the biscuit, i.e. the sides. It was found that a line segment of length 10 was found satisfactory for both the long and shorter sides, and a threshold of 12 was applied to the Hough array to eliminate spurious angles produced from stray chocolate such as shown in Figures 3.10e, 3.10f and 3.10g.

Having done this, we can now isolate each line as before based on their angles and sign of their Sobel x and y gradient components. This is then followed by the least squares method and scrutiny, all of these being the same as before. The results of this are given in the next section. In effect, these steps replace the first three steps for the biscuit algorithm, i.e. the Sobel and the grouping, smoothing and determination of the angles.

4.6.3 Results

The timings given in Table 4.2 are the result of applying the above algorithm on the same test images as the previous algorithm (Figures 3.10a to 3.10g). All timings are in milliseconds and are derived from the same system, i.e. a PDP-11/73 operating on a 128x128 frame store. Note that the initialisation procedure in all cases took 8ms. Also note that chain coding and the edge detection occur in the same routine; however, the time spent in each routine is given. The times for the whole algorithm (including least squares fit, etc.) are given in the last column. All angles derived were within $\pm 6°$ of the actual orientation as measured manually. This shows that little or no accuracy has been lost. It is worth noting that the Sobel operator is accurate to $\sim\frac{1}{2}°$ [64]. The discrepancy in the results is caused by the irregular sides of the biscuit. Besides, a result of $\pm 6°$ is far more accurate than required in order to inspect the biscuit.

Fig 3.21	Sobel on Edges (ms)	Chain Coding (ms)	Locating Angles in Biscuit (ms)	Total Time (ms)	Whole Algorithm (ms)
a	260	21	209	498	1163
b	259	20	214	501	1328
c	256	19	205	488	1292
d	261	20	209	498	1197
e	268	20	221	517	1244
f	265	21	216	510	1193
g	265	20	217	510	1191

Table 4.2 Breakdown of execution times for determining the orientation of the rectangular biscuit using the chain code

Comparing these results with those in Table 3.2, we can see that the edge detection process has been reduced by a factor ~5 (from ~1300ms). The overall reduction in execution time for the determination of the angles of the biscuit amounts to a factor ~2.7 (from ~1340ms) hence producing an overall reduction in execution time for the whole algorithm by a factor ~1.6 (from ~1900ms). This shows that, by implementing the algorithm sequentially, a significant gain in performance can be achieved with very little loss in accuracy of the results.

4.6.4 Discussion

Chapter 5 describes a high-speed sequential processor specifically designed for algorithms such as this. This has shown to give 25 times increase in execution time over a PDP-11/73 which would reduce the execution time of the algorithm to ~50ms (or ~100ms including I/O) - this is equivalent to ten biscuits per second. Note that this is a similar result after program optimisation has been carried out on the original algorithm in Section 3.5. However, after optimisation of the sequential version by applying similar techniques, an average reduction of 20% in the determination of the angles was achieved. It is interesting to note that the final figure relating the total execution

time of the sequential version of the algorithm after optimisation is less than the edge detect stage of the original algorithm. Since this is well within the limits of industrial inspection constraints, this brings doubt to whether a parallel processor in such an industrial environment would be more cost-effective, especially since it would be extremely difficult (if not impossible) to implement the algorithms given in Chapter 3 on a parallel processor. After all, even if it could achieve a greater throughput, the actual processing is ultimately limited to the product speed which is typically only 5-10 products per second in this particular case or for most lines less than 30 products/sec.

However, there are cases when a parallel processor will have a far greater throughput for those parts of the algorithm that can be executed in parallel rather than implementing them on a sequential processor - for example, algorithms where the whole image (rather than just parts of the image) need to be analysed. Such a case is in the segmentation of satellite images where the whole image is of interest. However, it is rare that an algorithm can be fully parallelised and this generally leads to the parallel implementation of an inherently sequential task. This in turn produces a significant decrease in performance of the parallel processor. It is for this reason that a dual-processor configuration consisting of a parallel processor and a sequential processor may appear to be an optimum solution. This is discussed in the Chapters 5 and 6.

4.7 SEQUENTIAL IMPLEMENTATION OF THE O-RING ALGORITHM

Referring back to Section 3.3 when we discussed the inspection of O-rings - in order to locate the centre of the ring, it was necessary to apply a Sobel to every point in the image and, for every edge point encountered, a candidate centre point was calculated which was recorded

in Hough space. Peaks in Hough space indicated the centres of the rings.

A similar technique can be applied to the O-ring algorithm by adopting the chain code as before. The O-ring is tracked using the same method as the chocolate biscuit, but this time the possible candidate centre points are calculated at every point during the tracking operation described in Section 4.5. The calculation of the centre point requires very little additional effort as the Sobel x and y gradient components are already available. The accuracy of the centre points was found to be ± 1 pixel from the results in the original algorithm on certain pictures (see Section 4.8) and a speed improvement of 4-5 was recorded for the calculation of the centre points. From the timings in Chapter 3, a single O-ring took 2595ms on a PDP-11/73 using a 128x128 frame store. With the sequential version on a single O-ring, this was reduced to 520ms. Thus, including the radial histogram to check for defects (270ms) and, this comes to 790ms, i.e. a reduction of a factor ~3.6. Implementation on SIP (Chapter 5) shows a factor of 25-30 reduction, hence producing a throughput of ~13 O-rings/sec.

4.8 LIMITATIONS OF SEQUENTIAL ALGORITHMS

Although the sequential implementations of the algorithms given in Chapter 3 give a significant reduction in their respective total execution times they do have their limitations. The main disadvantage is that, in the case of the O-rings, the algorithm will fail for more that two crossed O-rings. The reason for this lies in the fact that the tracking routine will begin to track the next overlapping ring at the intersection of the rings and fail to track the whole ring. Thus, only part of the ring is tracked which makes the algorithm susceptible to noise.

This allows us to make a distinction between the case of when one should consider using either a sequential or a parallel algorithm. A sequential task should be considered when objects will not be overlapped or touching, although they may only be partially visible as previously discussed. This should provide a large reduction in execution time of an algorithm over a parallel method and hence a higher inspection product rate. However, if complex scenes are likely to exist such as several overlapping, touching or partially visible objects - a common occurrence in tasks such as sorting - a parallel method should be chosen as this analyses the whole scene.

4.9 SUMMARY

The chain code has previously been restricted to binary images which have been produced by a simple threshold. However, this may not detect all edges (in the case of the inspection of the biscuits) which limits the applications. Alternative methods (e.g. edge detect followed by a thin) are generally computationally expensive and are more suited to a parallel processor. The problem here is that a parallel processor cannot efficiently execute the sequential tasks that are generally present in many algorithms.

In this chapter we have shown how the boundary of an object can be extracted sequentially from a grey scale image and how its chain code can be derived. This was applied to the chocolate biscuit algorithm from Section 3.4 and to the O-ring algorithm from Section 3.3. Those parts implemented by the chain code for the appropriate procedure showed a factor 4-5 reduction in execution time over the original times in Chapter 3. A novel method for maintaining robustness in the measurements by applying the Hough transform was used in the biscuit algorithm which showed little loss in accuracy over the original results. The final execution times of these algorithms when implemented

on SIP brings doubt to whether a parallel processor is a cost-effective solution for industrial inspection.

However, we cited that a parallel processor will achieve, in some cases, a far greater throughput than could ever be achieved by a sequential processor, although it is rare that an algorithm can be fully parallelised without some ad hoc implementation of a sequential algorithm. We also concluded that sequential algorithms were only useful for scenes where only non-overlapping products are likely to occur. Where overlapping or touching objects were present, a parallel algorithm, i.e. one that analyses the whole image, should be used. For these reasons, a dual-processor configuration consisting of a parallel and a sequential processor was proposed. The next chapter (Chapter 5), describes a high-speed sequential processor (SIP) suitable for such a system. Chapter 6 describes the parallel processor (LAP) and the implementation of the system.

CHAPTER 5

A HIGH-SPEED SEQUENTIAL IMAGE PROCESSOR

"I have yet to see a problem, however complicated
which, when you looked at it in the right way, did
not become still more complicated"
New Scientist 25 September 1969

5.1 INTRODUCTION

As we have discussed before (Chapters 3 and 4) many image processing algorithms exhibit a high degree of both sequentialism and parallelism. For this reason, a system composed of a sequential processor and a parallel processor would appear to be an efficient solution for the execution of such algorithms. This chapter describes a high-speed sequential image processor (SIP) along with its accompanying assembler. A full description of the parallel processor and the implementation of the system is described in Chapter 6.

If such a system is to be useful in an industrial environment, each processor must be capable of executing many tasks in real-time while remaining affordable. For this reason, a bit-slice architecture was chosen as the basis for both processors. This typically achieves a higher instruction throughput than conventional microprocessors and allows greater flexibility in the design of an instruction set.

Microcode optimisation is also discussed and how it can be applied to SIP's microcode.

This parallel/sequential processor configuration aims to achieve a high performance image processing system at low cost. First, let us discuss more fully the reasons for such a configuration.

5.2 REASONS FOR A PARALLEL/SEQUENTIAL ARCHITECTURE

Many image processing algorithms often contain both parallel and sequential tasks. For instance, an algorithm might locate the edges of an object in an image, determine a shape description model from the edges and extract measurements such as the perimeter, area, etc. This may involve the operations Sobel, threshold, chain code and manipulation of the chain code. Sobel and threshold are essentially parallel tasks (Section 2.2), while the chain code is a sequential task. It can be shown that sequential tasks executed on a typical parallel machine normally take longer to execute than on a sequential machine and vice-versa (Section 7.12). Therefore, unless an algorithm can be fully parallelised or sequentialised, optimum performance cannot be expected.

For this reason, a parallel/sequential processor configuration would appear to be a suitable solution for executing these types of algorithms - the parallel processor would execute the parallel tasks while the sequential processor would operate on the output of the parallel processor and execute the sequential tasks. Much work has been done on matching an image processing architecture with the task [13], but the concept of a general purpose image processing computer is still not clear. The rest of this chapter is dedicated to a detailed description of the sequential processor and its assembler. The parallel processor is described fully in Chapter 6 along with a discussion of the efficiencies and inefficiencies of the system. First, as an

introduction, let us discuss the basic architectural foundation of both processors - microcode.

5.3 MICROPROGRAMMING

Microcoding was first introduced in the 1950's by Professor Wilkes when he delivered a paper entitled "The Best Way to Design an Automatic Calculating Machine" [78]. A microprogrammed system usually consists of a number of functional elements such as a microprogram memory, a program sequencer, a processor and memory for local data storage such as depicted in Figure 5.1. Each individual bit (or group of bits) output from the program memory controls a functional part of the circuit. Here, two bits select whether the output of the program memory, processor or data RAM will be enabled onto the data bus; another bit controls the read/write line of the memory (such that it will go low on the second half of the clock cycle), and a group of bits designate the desired instruction for the bit-slice processors, etc.

A bit-slice processor is essentially a 'vertical' slice of a microprocessor. Probably the most common bit-slice processor is the AMD 2901 (see system in Figure 5.1). This is a 4-bit cascadable device that consists of an internal register file (of which two registers can be accessed simultaneously), a Q-register (for intermediate results), an ALU capable of executing eight arithmetic and eight boolean operations and a shifter that can optionally shift data entered from the output of the ALU. I/O capability is provided by a data input port (D) and a data output port (Y). Bit-slice processors are often used as building blocks for high performance systems that require speeds greater than that of conventional microprocessors. Because they are basic building blocks with limited functionality, they are cheap, fast and can be configured to form n-bit systems (where n is divisible by 4 for the 2901). Recent advances in bit-slice processors have produced a 16-bit version of the

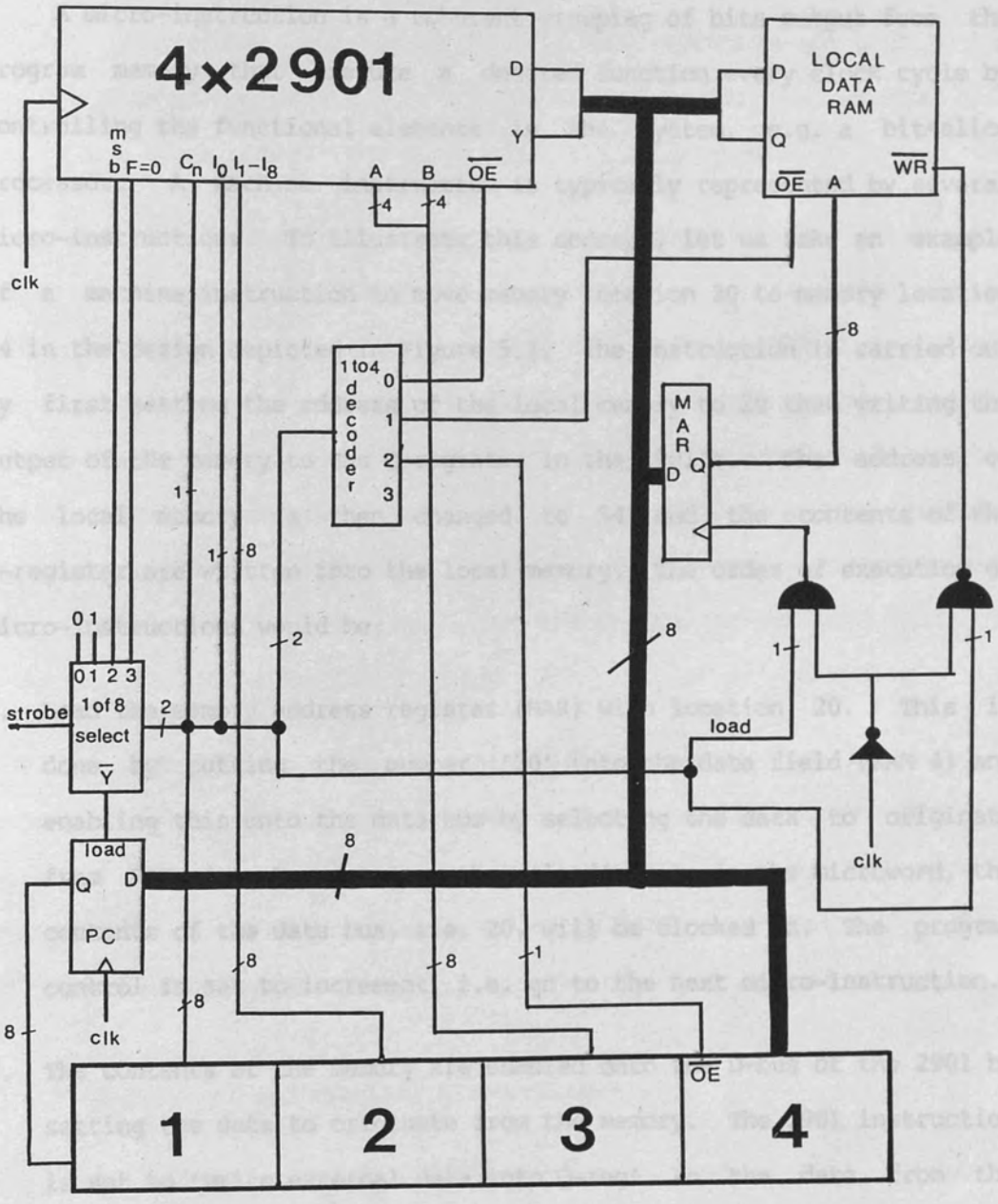


Figure 5.1 NPL prototype bit-slice processor

2901 appropriately called the 29C101 and several 32-bit bit-slices such as IDT's 49C404 which offer greater functionality.

A micro-instruction is a coherent grouping of bits output from the program memory that execute a desired function every clock cycle by controlling the functional elements in the system, e.g. a bit-slice processor. A machine instruction is typically represented by several micro-instructions. To illustrate this concept, let us take an example of a machine instruction to move memory location 20 to memory location 54 in the design depicted in Figure 5.1. The instruction is carried out by first setting the address of the local memory to 20 then writing the output of the memory to the Q-register in the 2901's. The address of the local memory is then changed to 54 and the contents of the Q-register are written into the local memory. The order of execution of micro-instructions would be:

1. Load the memory address register (MAR) with location 20. This is done by putting the number '20' into the data field (RAM 4) and enabling this onto the data bus by selecting the data to originate from RAM 4. By setting the 'load' bit in the microword, the contents of the data bus, i.e. 20, will be clocked in. The program control is set to increment, i.e. go to the next micro-instruction.

2. The contents of the memory are enabled onto the D-bus of the 2901 by setting the data to originate from the memory. The 2901 instruction is set to 'write external data into Q-reg' so the data from the memory (location 20) will now be written into the Q-register. The PC is set to increment.

3. The memory address register is loaded with location 54 by following the procedure as in step 1 except substituting location 54 for 20. PC is set to increment.

4. The contents of the Q-register are written into the memory (now pointing at address 54) by setting the 2901 instruction to output the contents of the Q-register onto the data bus (Y). The microcode bit controlling the write line for the memory is set high so the contents of the data bus is written into memory on the second half of the clock cycle. The PC is again set to increment.

Note that in all cases the PC was set to increment. An alternative would be to select the 'load' pin of the PC to go low (by selecting the '1' from the 1-of-8 select - the output is inverted). The PC would then be loaded with the contents of the data bus. This is mainly used for branch instructions where control of the program is passed to a location in the program memory other than to the next sequential location. Typical machines that are capable of being microprogrammed are: Digital's VAX family, IBM 7950, IBM System/360 ILLIAC IV (Section 7.6.1) and the CDC STAR 100 [55].

On each clock cycle, a micro-instruction (often known as a microword) is produced at the output of the program memory. The term given to this level of programming is microprogramming or microcoding. Two possible forms of microcoding exist: horizontal and vertical. These will be discussed next.

5.3.1 Horizontal and Vertical Microcoding

Two methods exist for microcoding: horizontal and vertical. Horizontal microcoding is when each bit output from the microcode memory is assigned to one particular device. The outcome of this is that maximum parallelism can be achieved (i.e. all devices may be activated simultaneously); however, a wide microword is likely to exist if there are many devices in the system. On the other hand, vertical microcoding allows a bit or several bits to be shared among the devices: additional

instruction bits are therefore required in order to select which device will be enabled. The advantage of this is that the microcode width is reduced (and hence the amount of microcode memory); however, several lines of code may be required in order to activate all devices sequentially. Thus, when optimising a microcoded design, it is important to minimise the width and length of the microcode. The desirable approach is to share fields where parallelism is not required.

5.4 USE OF BIT-SLICE ARCHITECTURES FOR IMAGE PROCESSING

To investigate the possibility of a high-speed, microcoded bit-slice sequential processor for use in image processing, a test module was designed and constructed at the National Physical Laboratory (NPL) as part of the work for this thesis. The architecture in fact is the design depicted in Figure 5.1. This was used to study how image processing algorithms would perform on a bit-slice architecture. The results and ideas obtained were later used in the design and development a more sophisticated sequential image processor called SIP. It was found using the test module that there are several advantages of designing a microcoded machine for use in image processing:

1. They are typically many times faster than conventional microprocessors. This is because conventional microprocessors generally contain a high degree of functionality while bit-slice processors are basic boolean-elements that only perform a small, simple set of instructions. Bit-slice processors are thus suited to image processing where many of the functions traditionally associated with conventional microprocessors (such as the existence of several data types and internal address calculations) are rarely, if never used.

2. Since the programmer has direct control over the individual functional units in the system (processors, memories, etc.), it is possible to operate on any of the units in a single cycle, allowing several instructions to be executed simultaneously. This will be discussed more fully in Section 5.8 when we examine microcode optimisation.

3. Because the programmer has direct functional control of the system, it is possible for a microcoded machine to emulate a range of other machines. It is also possible for the instruction to be customised.

However, microcoding also has its disadvantages:

1. In order to write microcode, the programmer must have a detailed knowledge of the hardware, i.e. bus routes, control signals, etc. In general, this information is not easy to transfer from person to person which makes it notoriously difficult to use.

2. One of the advantages of microcoding is that several instructions may be executed simultaneously as cited above. However, this must also be included amongst the disadvantages as detection of these instructions in a program is often very difficult. Section 5.8 discusses this point in more detail.

Allowing the programmer to define the instruction set is highly advantageous for image processing; for instance, a frequently occurring image processing routine that requires several lines of code in Pascal can be optimised into a single machine-level instruction. An example of such an instruction is APPLY...END to sequentially scan over an image (Section 5.5.2). Customised instructions such as this have been fully exploited in the design of a picture processing language (PPL) (Section 6.2.1).

Machines that have achieved success with bit-slice designs for image processing are: Logica's DIPOD (Section 7.8.1) and its in-house language FIFTH that translates high-level constructs to microcode, WARP (Section 7.9) and the ILLIAC IV (Section 7.6.1). Many of today's microprocessors including Motorola's 68030 are also internally microcoded. The opcode is fetched and decoded into a series of microcode operations. The microcode controls the internal registers and buses, etc. within the microprocessor.

As a consequence of the fetch and decode scheme in a microprocessor, several machine cycles are usually required to execute an instruction. By adopting the Harvard architecture (Section 7.2) such that separate instruction and data buses are maintained and using techniques such as pipelining (Section 7.2.1), it is possible to achieve a rate of one instruction per cycle. Pipelining allows an instruction fetch and an instruction execute to take place simultaneously. The concepts discussed so far have been applied in the design of a high-speed sequential processor for real-time image processing. This will be described next.

5.5 A SEQUENTIAL IMAGE PROCESSOR - SIP

SIP is a 16-bit high-speed microcoded sequential image processor that exhibits a high degree of internal parallelism. The basic functional units are: 4 AMD 29203 bit-slice processors, a high-speed multiplier, 4K microwords of local data RAM, 2 128x128 image planes (designated P and Q), a program memory, a pipeline, a program sequencer and an image processing interface as depicted in Figure 5.2. Both image planes are memory mapped onto the VMEbus for high speed access by other devices. The overall control is from a host PDP-11/73 and a Qbus to VMEbus convertor. The downloading of microcode to the program memory is via eight VME mapped registers as depicted in Figure 5.3. The

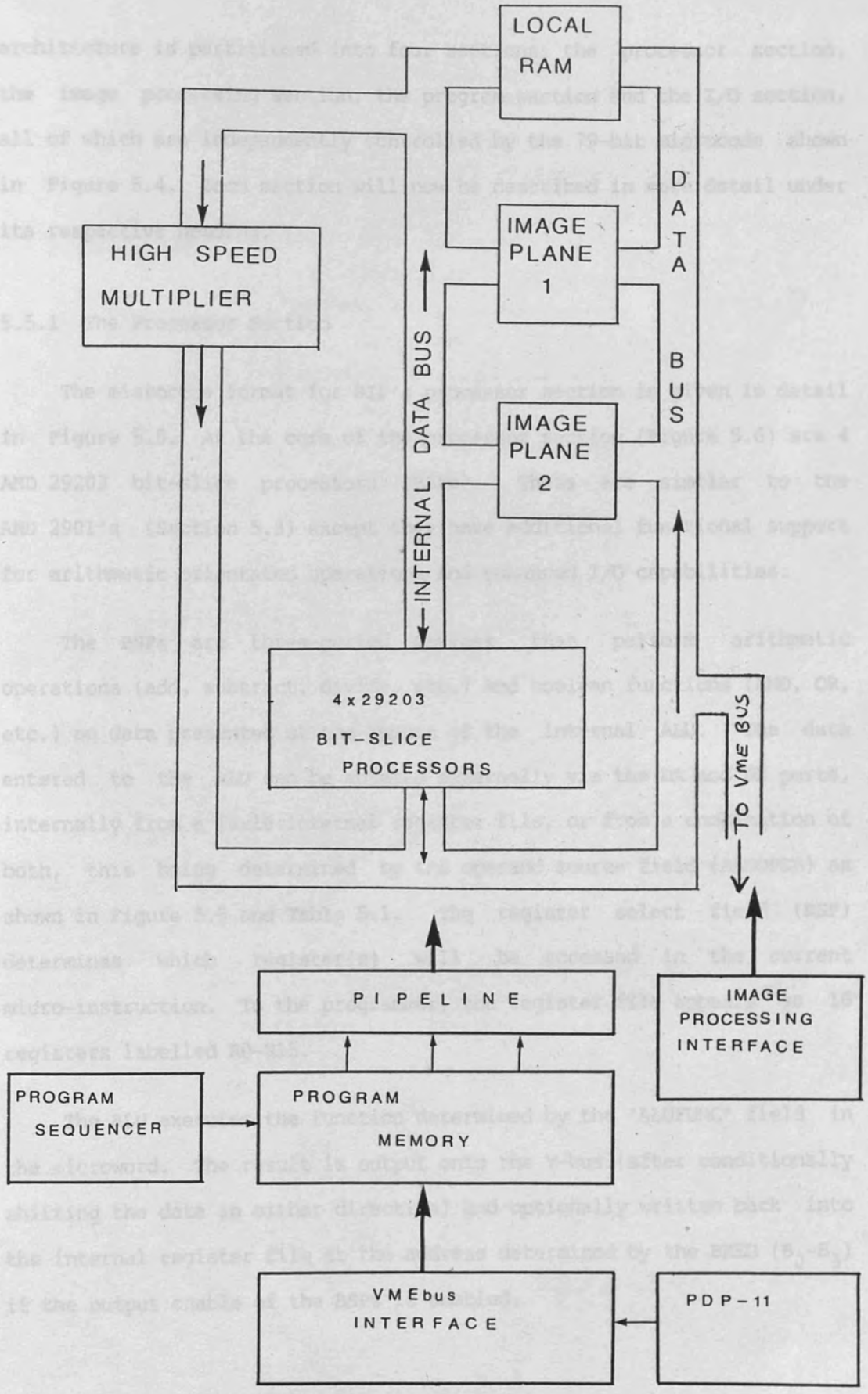


Figure 5.2 Schematic diagram of SIP

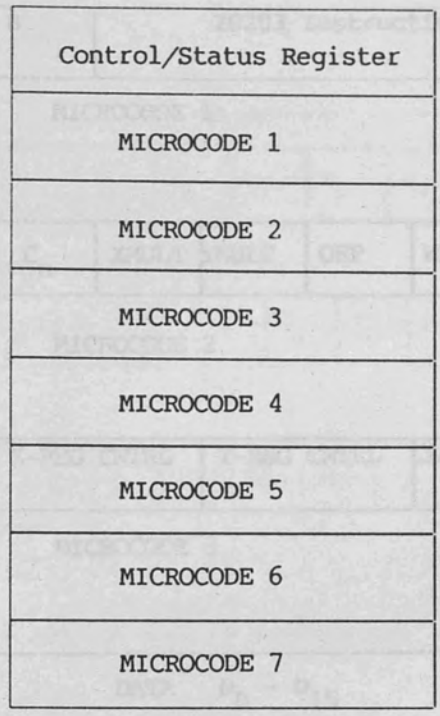
architecture is partitioned into four sections: the processor section, the image processing section, the program section and the I/O section, all of which are independently controlled by the 79-bit microcode shown in Figure 5.4. Each section will now be described in more detail under its respective heading.

5.5.1 The Processor Section

The microcode format for SIP's processor section is given in detail in Figure 5.5. At the core of the processor section (Figure 5.6) are 4 AMD 29203 bit-slice processors (BSPs). These are similar to the AMD 2901's (Section 5.3) except they have additional functional support for arithmetic orientated operations and enhanced I/O capabilities.

The BSPs are three-ported devices that perform arithmetic operations (add, subtract, divide, etc.) and boolean functions (AND, OR, etc.) on data presented at the inputs of the internal ALU. The data entered to the ALU can be entered externally via the DA and DB ports, internally from a 16x16 internal register file, or from a combination of both, this being determined by the operand source field (ALUOPER) as shown in Figure 5.5 and Table 5.1. The register select field (RSF) determines which register(s) will be accessed in the current micro-instruction. To the programmer, the register file appears as 16 registers labelled R0-R15.

The ALU executes the function determined by the 'ALUFUNC' field in the microword. The result is output onto the Y-bus (after conditionally shifting the data in either direction) and optionally written back into the internal register file at the address determined by the BREG (B_0-B_3) if the output enable of the BSPs is enabled.



Bit	7	6	5	4	3	2	1	0
	Strobe /Done	CSRCLK	Select Plane1	Select Plane2	CSR to Pipe	Select PC/CSR	CLK HI/LO	R/H

Figure 5.3 Programmer's model of SIP and the control/status bits in the CSR

Select A reg	Select B reg	29203 Instructions $I_0 - I_7$

MICROCODE 1

I_8	EA	OEB	C_n	XMULT	YMULT	OEP	WRPIC1	WRPIC2	L.RAM

MICROCODE 2

LOCAL RAM CNTRL	X-REG CNTRL	Y-REG CNTRL	CLKOFF REG	COND. CODE SELECT

MICROCODE 3

DATA $D_0 - D_{15}$

MICROCODE 4

PC INSTRUCTION	OE DATA	OE OFF	OE RAM	OE PIC1	OE XREG	OE YREG	OE P2A	OE P2B	Y TO A	Y TO B	OEYM	DONE

MICROCODE 5

REQUEST VMEbus	RELEASE VMEbus	READ/WRITE TO VMEbus	ENABLE STROBES	DS0	DS1	CLOCK VME ADDRESS

MICROCODE 6

X-offset value	Y-offset value

MICROCODE 7

Figure 5.4 Full microword format for SIP

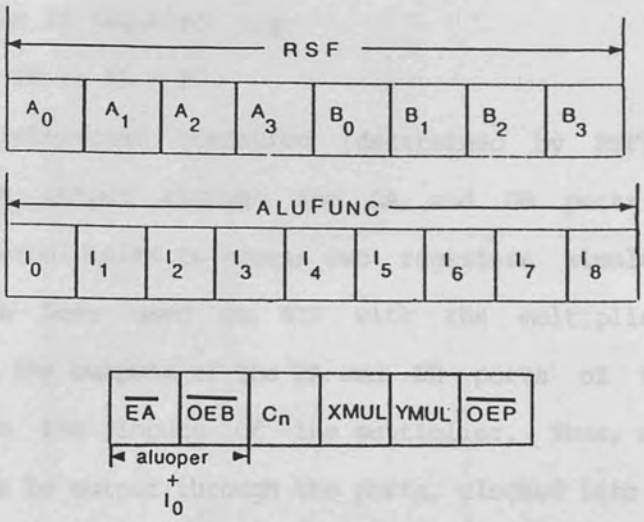


Figure 5.5 Processor microfield

$\overline{EA}$	I_0	$\overline{OEB}$	ALU OPERAND R	ALU OPERAND S
L	L	L	RAM Output A	RAM Output B
L	L	H	RAM Output A	DB_{0-3}
L	H	X	RAM Output A	Q Register
H	L	L	DA_{0-3}	RAM Output B
H	L	H	DA_{0-3}	DB_{0-3}
H	H	X	DA_{0-3}	Q Register

Table 5.1 ALU operand sources for the 29203

Data present on the Y-bus can also be written into the internal register file through the Y port at the address determined by BREG if the output enable is disabled. Thus, for instance, two registers (or any combination of register and external data) can be operated on, output onto the Y-bus and written back to a register in a single cycle, i.e.

```
R0 := R0 + R1
```

The data could also be written into the image planes or local memory on the same cycle if required, e.g.

R0 := P0 := R0 + R2

During a register-read operation (determined by RSF), the register contents are output through the DA and DB ports (see Table 5.1), allowing external units to access two registers simultaneously. This facility has been used on SIP with the multiplier as depicted in Figure 5.6 - the outputs of the DA and DB ports of the 29203's are connected to the inputs of the multiplier. Thus, any two arbitrary registers can be output through the ports, clocked into the multiplier by enabling XMUL and YMUL and multiplied in a single cycle. The result can be either written back into the register file or operated on (by enabling OEP) before being written back on the next cycle.

As mentioned before, the Am29203's have enhanced functionality support for arithmetic operations such as normalisation, binary-BCD conversion and multiply and divide. (Note that SIP runs at 8MHz because of the time it takes the 29203's to execute the divide instruction. However, it has been found that SIP will run at 10MHz if the divide instruction is not required - see Chapter 9 for a list of enhancements that could be made to SIP.) In addition to this, they can support a three address architecture, i.e. they are able to support functions such as

R0 := R1 + R2

in a single clock cycle.

The local memory consists of 4Kwords (1 word = 16 bits) of high-speed static RAM, this being used to hold variables, arrays and lookup tables, etc. The I/O ports are connected to the Y-bus via a buffer enabling output from the ALU, the multiplier or the image planes to be written directly into the RAM as depicted in Figure 5.6. The address supplied to the RAM is in the form of two 8-bit up/down

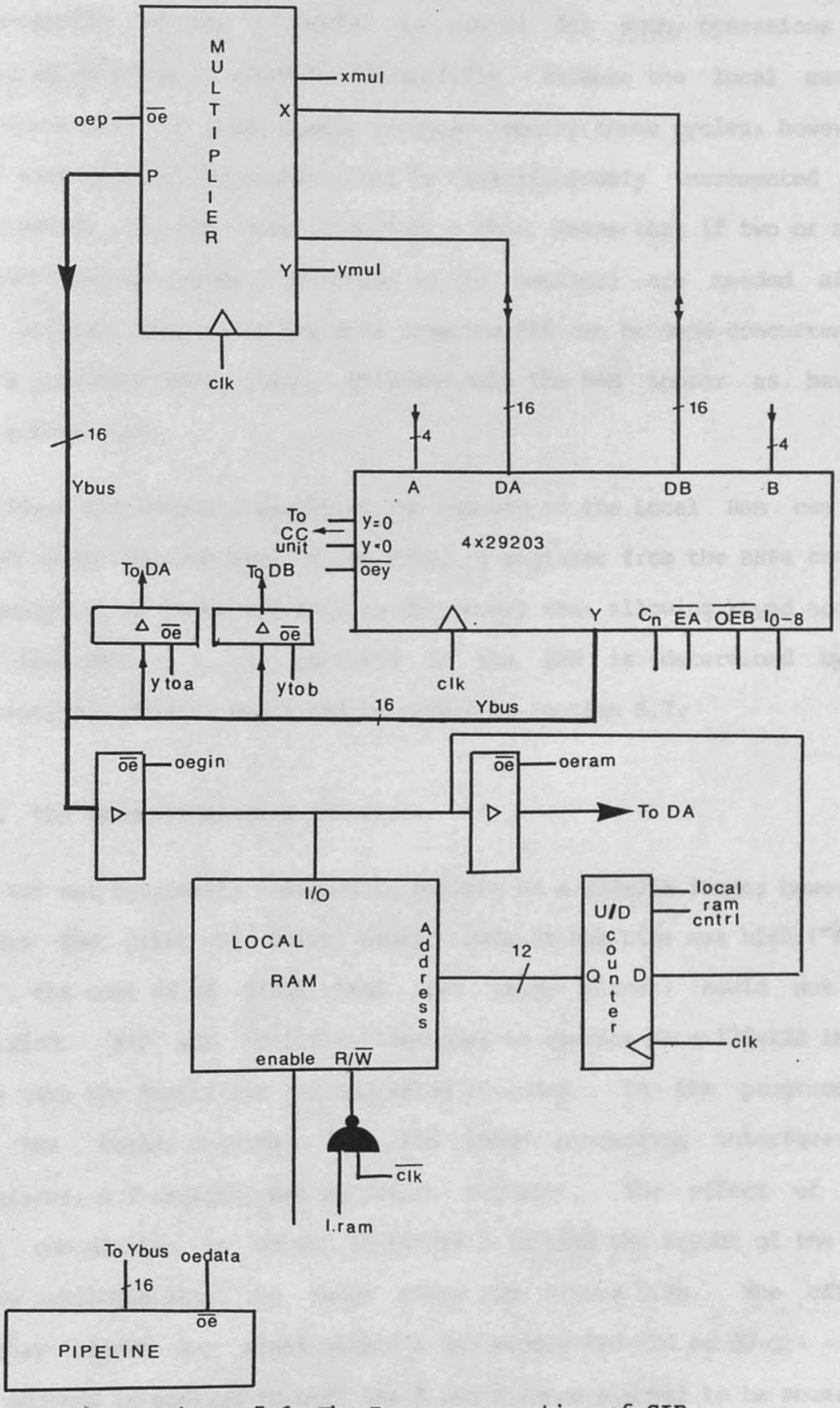


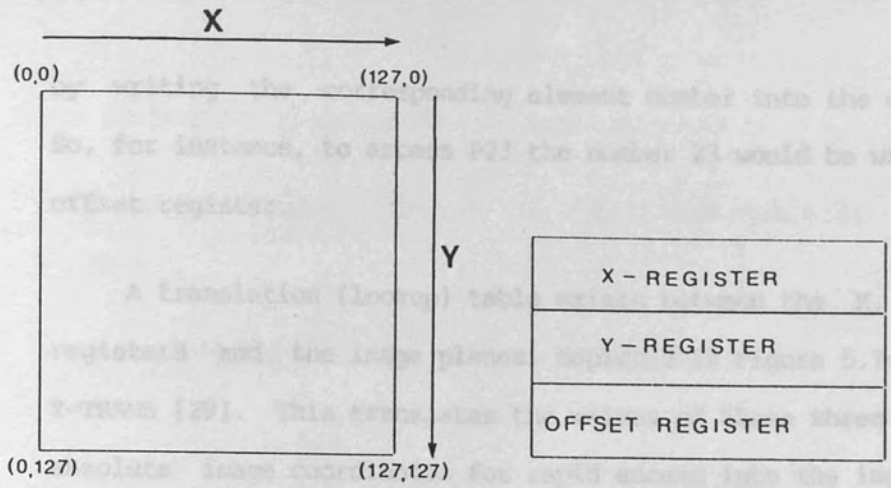
Figure 5.6 The Processor section of SIP

counters. These have the facility to be loaded from the Y-bus, count up, count down or hold (no change), this being determined by the 'Local Ram cntrl' field in Figure 5.6. The ability to increment and decrement independently of the processor is useful for such operations as accessing an array of elements sequentially. Because the local memory has common data I/O pins, memory accesses require three cycles; however, after each access, the counters may be simultaneously incremented (or decremented) to the next location. This means that if two or more instructions (or cycles - whichever is the smaller) are needed after each access, then fetching data from the RAM can be done concurrently with a processor instruction. This can make the RAM appear as having zero access time.

Since the counters supplying the address to the Local Ram can be loaded with the contents of the Y-bus, a register from the BSPs can be designated as an index variable to the memory thus allowing keyed access into the RAM, i.e. the location in the RAM is determined by a calculation. This is described in detail in Section 5.7.

5.5.2 The Image Processing Interface.

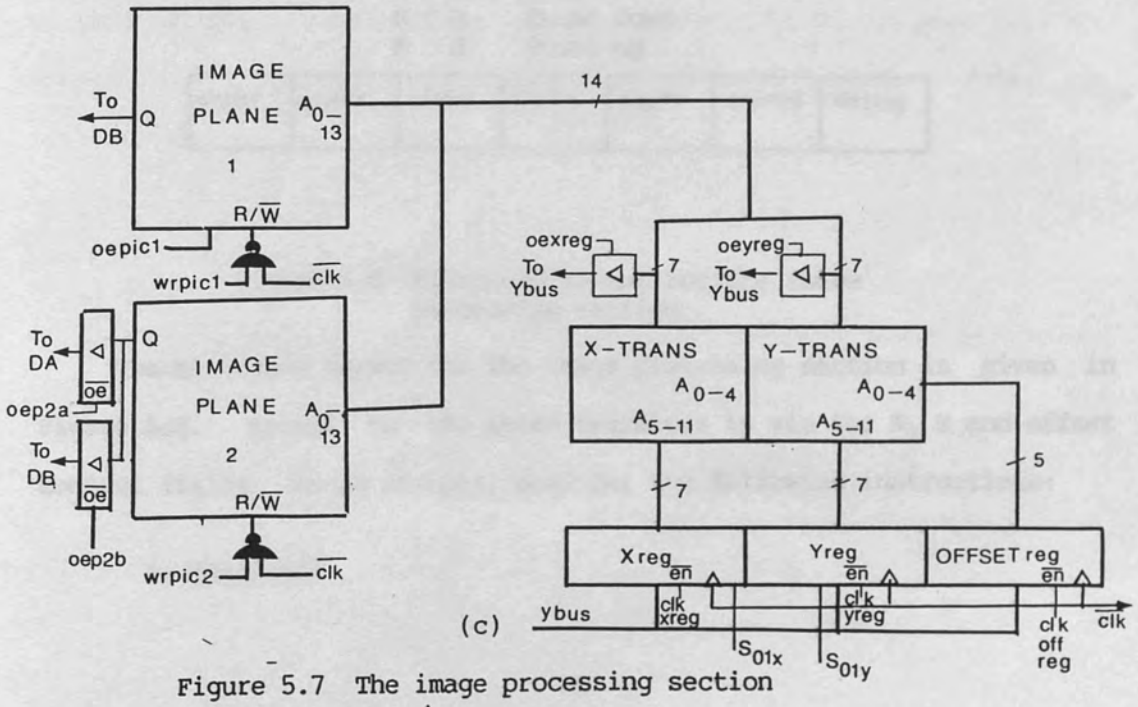
SIP was originally designed to operate on a 256x256 image; however, because the price of 64Kx1 static rams at the time was high (~£100 each), the cost of 16 64Kx1 RAMS (two image planes) could not be justified. SIP was therefore designed to operate on a 128x128 image plane with the facilities for upgrading included. To the programmer, SIP has three registers for its image processing interface: an X-register, a Y-register and an offset register. The effect of the (X,Y) coordinates is shown in Figure 5.7a with the layout of the 5x5 window available into the image plane in Figure 5.7b. The offset register allows any pixel within a 5x5 window (P0-P24 or Q0-Q24 - the same address is applied to both the P and Q image planes) to be accessed



(a)

16	15	14	13	12
17	4	3	2	11
18	5	0	1	10
19	6	7	8	9
20	21	22	23	24

(b)



(c)

Figure 5.7 The image processing section

by writing the corresponding element number into the offset register. So, for instance, to access P23 the number 23 would be written into the offset register.

A translation (lookup) table exists between the X, Y and offset registers and the image planes, depicted in Figure 5.7c as X-TRANS and Y-TRANS [29]. This translates the values of these three registers into absolute image coordinates for rapid access into the image space within the 5x5 window. Note that a 5x5 window was chosen as a suitable tradeoff between the cost of the RAMS required for the translation table and (from experience obtained earlier) the advantages to be gained from using a larger window, e.g. 7x7. As mentioned before, SIP was originally designed to operate on a 256x256 image. A 7x7 window in this case would have required four 16Kx4 (45ns access time) static rams which would have significantly increased the cost of the system at the time of designing. As the system had to be cost-effective, it was decided that a 7x7 window could not be justified.

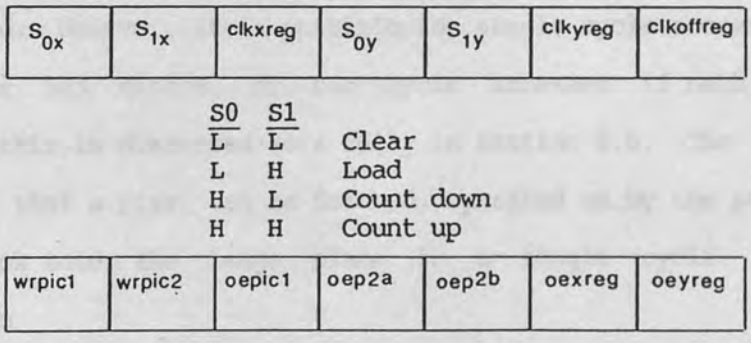


Figure 5.8 Microword format for the image processing section

The microcode format for the image processing section is given in Figure 5.8. Access to the three registers is via the X, Y and offset control fields. As an example, consider the following instructions:

```
X:=34; Y:=23
```

The corresponding microcode instructions would be:

1. Load X-field with the code for 'LOAD' and enable 34 onto the data bus.

2. Load Y-field with the code for 'LOAD' and enable 23 onto the data bus.

Each of these instructions is executed in a single cycle (125ns). Because SIP is a sequential processor, rapid access to any part of the image space is essential for high-speed manipulation of the image: this can be done in two ways. If the pixel lies within the 5x5 window then two cycles are required to fetch the data: the first cycle sets up the address to the image planes by writing to the offset register where the data is available on the next cycle. If the pixel lies outside the 5x5 window, it is necessary to change the X and Y registers. Directly changing X or Y requires two cycles to produce the required data at the output of the image planes (as before) or three cycles if both X and Y are changed. However, it is possible for single cycle accesses to occur within the 5x5 window, or two cycle accesses if both X and Y are changed - this is discussed more fully in Section 5.6. The consequence of this is that a pixel can be fetched, operated on by the processor and written back into the image plane in a single cycle. Thus, the instruction

```
P0:=(P0+R1)*2
```

can be executed in a single cycle.

To relieve the programmer of having to increment X and Y at the end of a row or column, SIP has a highly efficient auto-scan facility. For instance, in Pascal the instructions to scan an image would be:

```
y:=0;
REPEAT x:=0;
 REPEAT
    .
    .
  picture function
    .
    .
 UNTIL x=128; y:=y+1
UNTIL y=128;
```

However, SIP uses the instructions:

```
APPLY
 .
picture function
 .
END
```

This scans the image sequentially left to right, top to bottom while still allowing the X and Y registers to be manipulated. With optimisation (Section 5.8), zero cycle overhead can be achieved by the scanner, i.e. the code appears as though it were continuous. This clearly shows the advantage of designing a microcoded machine. The instruction set is described more fully in Appendix A along with the addressing modes available.

5.5.3 The Program Section

The program section consists of 4Kx80 bits of high speed static RAM, a pipeline and an AMD 2910A program sequencer as depicted in Figure 5.9. The program sequencer incorporates 16 powerful functions which are listed in Table 5.2. By code optimisation, it is possible to make instructions such as JSR and RTS appear to take zero cycles - this is discussed more fully in Section 5.8. One of the main deficiencies with a pipeline is that branches take two cycles to execute on a single level pipeline, i.e. one to execute the instruction (this sets the condition code register) and the second to test the condition code

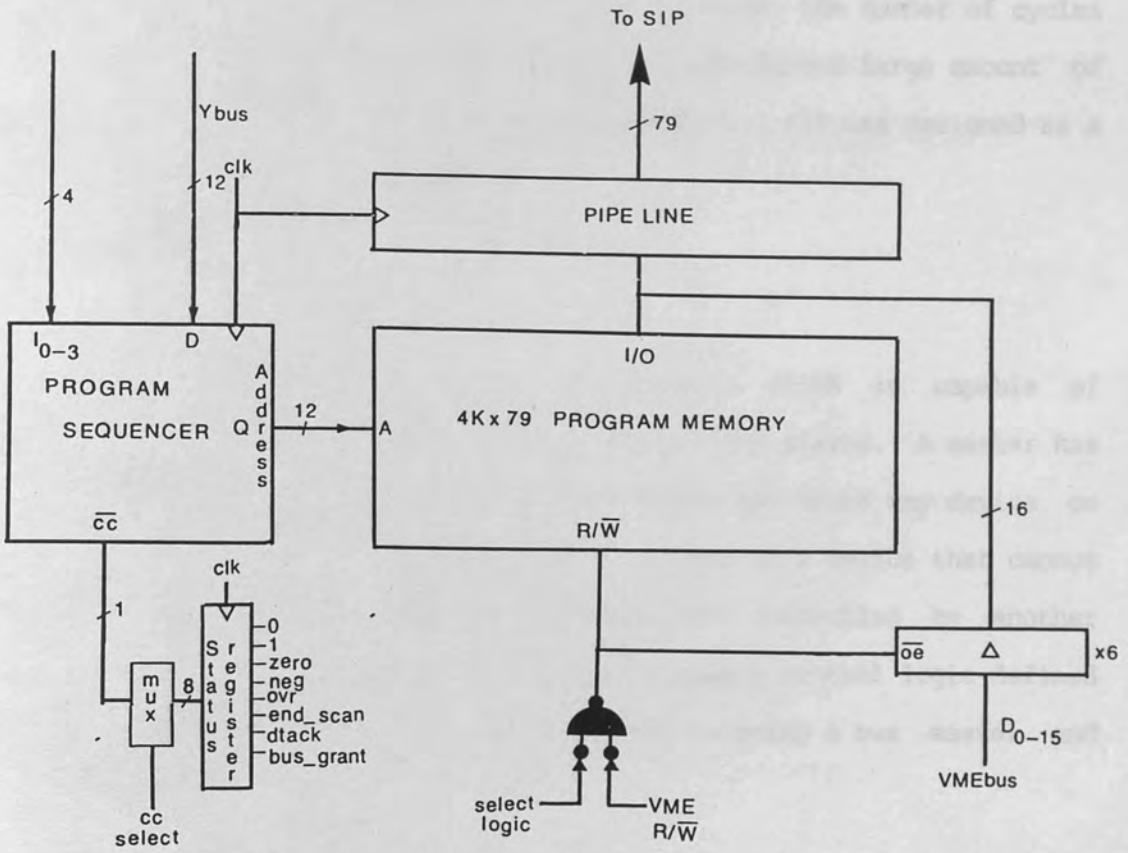


Figure 5.9 Program section of SIP

I_3-I_0	Function
0	Jump to location zero
1	Conditional JSR Ybus
2	N/A to SIP
3	Conditional jump Ybus
4	Push address & conditional load counter
5	Conditional JSR via register/pipeline
6	N/A to SIP
7	Conditional jump via register/pipeline
8	Repeat loop until counter=0
9	Repeat pipeline until counter=0
10	Conditional RTS
11	Conditional jump pipeline and pop
12	Load counter and continue
13	Test end of loop
14	Continue (NOP)
15	Three-way branch

Table 5.2 Program sequencer (AM2910A) instructions

register. As the number of pipelines increase, the number of cycles will be increased linearly. Because of the anticipated large amount of branching (common to many sequential programs), SIP was designed as a single level pipeline machine.

5.5.4 The I/O Interface.

SIP is based on the VMEbus (Section 5.9) which is capable of supporting devices commonly known as masters and slaves. A master has the capability of gaining control of the VMEbus and hence any device on the bus, i.e. another master or slave. A slave is a device that cannot control the bus and can only be accessed and controlled by another master, e.g. a frame store. SIP has the necessary control logic defined in its VMEfield depicted in Figure 5.10a for becoming a bus master and controlling the VMEbus.

SIP's I/O field (VMEfield) contains seven microcode bits to control the interaction between SIP and the VMEbus via an intelligent Field Programmable Logic Sequencer (FPLS), designed by A.I.C. Johnstone of this research group. The bits depicted in Figure 5.10a correspond to the necessary control signals as defined by the VMEbus protocol [80]. The image plane address bits from SIP (16-bits) go to the low-order VMEbus address lines (A0-A15) while SIP's internal data bus (the Y-bus) go to the VMEbus data lines (D0-D15) via the usual bus interface transceivers (Figure 5.10b). This enables a memory-mapped frame store on the VMEbus appear as though it was local to SIP. When reading from the VMEbus, data appearing on SIP's internal data bus is made available to SIP's functional units, e.g. processor, image planes, etc. By making maximum use of SIP's internal parallelism and by configuring the image addressing to be in auto-scan mode (Section 5.5.2), rapid sequential accesses can occur from the VMEbus simultaneously with on board processing. With careful programming and an optimised program, an image

REQUEST VMEbus	RELEASE VMEbus	READ/WRITE TO VMEbus	ENABLE STROBES	DS0·	DS1·	CLOCK VME ADDRESS

Figure 5.10a Microword field for SIP's interaction with the VMEbus

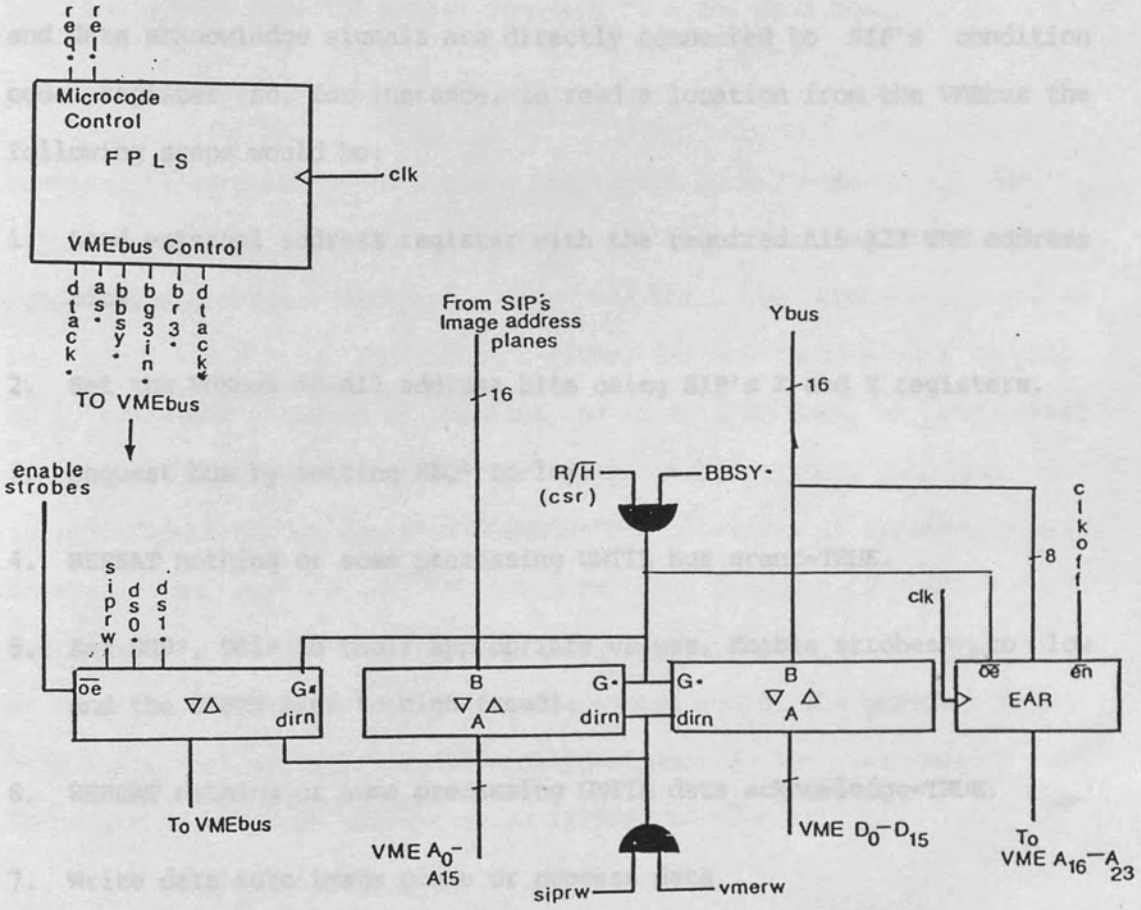


Figure 5.10b SIP's VMEbus control circuitry

can be grabbed and input from an external source, e.g. a frame store, simultaneously with the processing of a previously stored image. This eliminates the I/O bottleneck previously associated with bus-based sequential processors.

Since the VMEbus has a 24-bit address space (via the P1 connector), an additional 8 bits are supplied from the external address register (EAR) as depicted in Figure 5.10b. This must be loaded with the correct 8-bits to form bits A16-A23 of the VMEbus address space. The bus grant and data acknowledge signals are directly connected to SIP's condition code register so, for instance, to read a location from the VMEbus the following steps would be:

1. Load external address register with the required A16-A23 VME address bits.
2. Set the VMEbus A0-A15 address bits using SIP's X and Y registers.
3. Request bus by setting REQ* to low.
4. REPEAT nothing or some_processing UNTIL bus_grant=TRUE.
5. Set DS0*, DS1* to their appropriate values, Enable_strobes* to low and the SIPRW line to high (read).
6. REPEAT nothing or some_processing UNTIL data_acknowledge=TRUE.
7. Write data into image plane or process data.
8. Optionally increment SIP's image address lines and goto step 5 or
9. Release bus by setting REL* to low.

By making maximum use of SIP's internal parallelism, VMEbus accesses can take three cycles (375ns) for high-speed peripherals.

5.6 PIPELINING THE PIXEL FETCH

A feature of SIP is that it contains four independent sections, thus providing the capability of instruction parallelism. This can lead to a dramatic reduction in execution time and code length of a program. This is particularly noticed when image accesses are pipelined because image processing operations require frequent access to the image planes. There are usually two operations required for fetching a pixel

1. Load offset into the offset register from the data bus.

2. System accesses pixel on the next cycle.

However, it is possible to achieve both steps in a single cycle. During step 1 (above), one can note that as the offset register is being loaded, the processor section is idle, and while the processor section in step 2 is active, the offset register for the image planes is idle. It is therefore possible to pipeline the fetch such that, as one pixel is being processed, the next one is being fetched. Thus, only one cycle is required to access any pixel within the 5x5 window as opposed to two; however, the next pixel to be fetched must be known (Section 5.8.1). The X and Y registers also work on the same principle, i.e. altering X or Y can be done concurrently with the processing of the previous pixel. This means that the majority of instructions in the instruction set (Appendix A) take one machine cycle (125ns) to execute.

5.7 SIP'S ASSEMBLY LANGUAGE

In order to translate an intelligible form of code into microcode, an assembler was written for SIP. One of the features of a microcoded machine is that it can emulate a variety of other machines. The instruction set was designed around the PDP-11 'Macro-11' instruction set with several enhancements for image processing. Since Macro-11 was

familiar in the laboratory, the amount of learning involved in programming SIP was minimised.

The assembler consists of three parts: the intermediate code generator, the translator and the loader. It was written in the form of a P-code language where the source code is compiled to an intermediate code before being translated into microcode, this being necessary for efficient code optimisation (Section 5.8). Each of these parts will now be described.

5.7.1 The Assembler and Intermediate Code Generator

The aim of the assembler is to convert assembler mnemonics into an intermediate code suitable for the translator to convert into microcode (Section 5.7.2). Instructions can have either one or two operands, i.e.

```
MNEMONIC source, destination

e.g. ADD #4,R0   - equivalent to R0:=R0+4
     ADD R5,R9   - equivalent to R9:=R9+R5
```

or

```
MNEMONIC destination

e.g. INC R7      - equivalent to R7:=R7+1
     BRA label   - equivalent to goto label
```

(The instruction set is described more fully in Appendix A.) The intermediate code is in the form of a mnemonic representing which of the functional unit(s) (e.g. processor, image plane, etc.) is to be accessed, followed by either one or two operands required by the translator for the corresponding functional unit fields in the microword. Before we consider the intermediate code, it is necessary to describe the six addressing modes available. These are: data, register,

image, xy, indexed and memory mode.

1. Data mode (D) - this is equivalent to immediate addressing on a conventional microprocessor and is indicated by a hash (#) before the operand. The # indicates that the following data should be taken literally, e.g.

```
MOV #4,R0
```

states that the number '4' is to be written to register R0.

2. Register mode (R) - this is when a register in the bit-slice processors is to be accessed, e.g.

```
ADD R5,R7
```

states that the contents of register R5 are to be added to register R7 and written back to R7.

3. P and Q mode (P) or (Q) - this is when either the P or the Q image planes are to be accessed, e.g.

```
MOV P0,R5
```

indicates that element P0 (see window layout in Figure 5.7b) is to be written to register R5. Any element within the 5x5 window can be accessed in this way in both P and Q spaces, e.g.

```
MOV R4,P1
ADD Q23,R1
```

The first example states that R4 is to be written to element P1. The second states that the contents of element Q23 are to be added to register R1.

4. X and Y mode (X) or (Y) - this is when the X and Y registers are accessed, e.g.

```
MOV X,R3
MOV R4,Y
```

The first instruction indicates that the value of the X-register is to be written to R3. The second instruction indicates that the

contents of R4 are to be written to the Y register.

5. Indexed mode (I) - this is when the contents of the register indicated in brackets are taken as an address into the local memory, e.g.

```
MOV (R0),R1
```

If R0 contained the number 20, then location 20 in the local memory would be written to register R1. This could alternatively be written as

```
MOV 20,R1
```

(see memory mode below (6)). However, this is inflexible since it does not allow manipulation of the address, i.e. 20 is a constant whereas R0 in the preceding example is variable. The source and destination of the operands in all modes can be interchanged, e.g.

```
MOV R5,(R13)
```

If R13 contained the number 56 then the contents of R5 would be written into location 56 in the local memory.

6. Memory mode (M) - this is the default mode. If none of the above modes are encountered then it is assumed that the name or number of the operand is a memory location, e.g.

```
INC ros
MOV jim,R4
MOV tom,bill
```

In the first example, if ros was allocated location 1234 in local memory by the assembler then location 1234 would be incremented. In the second example, if jim was assigned to location 34 in the local memory, then the contents of location 34 would be written to register R4. This could alternatively be written as

```
MOV 34,R4
```

In the last example, if tom was allocated location 999 and bill was allocated location 67 then location 999 would be written to location

67. By preceding the variable with a hash (#), the allocated location of the variable will be used, e.g.

```
MOV #ros,R4
```

If ros was allocated location 67 then the number 67 would be written to R4 and not the contents of location 67. This facility is useful for array indexing, for example, to index into an array called NAME the corresponding set of instructions would be:

```
MOV #name, R0       ; base location of array
ADD R1,R0           ; R1 is the index value
MOV (R0),R0         ; contents of R0 is memory
                    ; location #name+R1
```

this is equivalent to

```
R0:=name[R1]
```

It is not good practice to designate memory locations as absolute numbers. To avoid this problem the construct VAR is available. This is similar to the Pascal VAR in that it enables single variables or arrays to be represented as absolute memory locations, e.g.

```
VAR jim, i, j, symbol
```

This defines the variables: jim, i, j and symbol and are all allocated 1 word (it is not possible to define byte locations) in the local memory. VAR also has the facility for defining arrays, e.g.

```
VAR arr:400, name:5
```

This example defines an array 'arr' of size 400 words and an array 'name' of size 5 words. (Note that only one-dimensional arrays are allowed.) The mixing of integers and arrays in the declarations is allowed, e.g.

```
VAR jim, i, arr:400, j, symbol, name:5
```

All variables and arrays are assigned memory locations in the order in which they are defined; therefore, the locations assigned to the variables in the last example are:

```
jim     - 0
i       - 1
arr     - 2
j       - 402
symbol  - 403
name    - 404
```

Variables may be declared anywhere within the program, hence enabling them to be declared at the beginning of subroutines - this can make the code easier to read.

The assembler translates the intermediate user code into a form

'OPCODE' | mode1 | (mode2) | data1 | (data2) |

where mode1 (and mode2 for a two operand instruction) are the modes for the operand (D,X,P,I, etc.) as described above and data1 (and optionally data2) are the necessary values of mode1 and mode2 respectively. This is best explained with an example. Table 5.3 below shows the previous examples translated into their intermediate code:

Opcode	Intermediate code
MOV #4,R0	MOVDR 4 0
ADD R5,R7	ADDRR 5 7
MOV P0,R5	MOVPR 0 5
MOV Q23,R0	MOVQR 23 0
MOV X,R3	MOVXR 3
MOV R4,Y	MOVRY 4
MOV (R0),R1	MOVIR 0 1
INC jim	INCM 20
MOV jim,R6	MOVMR 20 6
MOV 20,R1	MOVMR 20 1

Table 5.3 Example of translating assembler mnemonics into intermediate code

(N.B. The INC instruction assumes that jim was assigned to location 20 in the local memory.)

The assembler takes three passes to complete the intermediate code generation:

(I) Mnemonics are converted into the above intermediate code style. Full syntactic error checking is carried out here to eliminate the most common errors such as the misspelling of opcodes and use of undefined identifiers, etc. All variables and arrays declared in the VAR construct are allocated space in the local memory where error checking for multiple declaration of variables and memory overflow is checked.

(II) Identifiers are substituted for real numbers and pseudo high-level instructions (see Appendix A) are expanded. All label names and label values are read into a table for PASS-III. Full error checking is enabled for missing delimeters and multiple declaration of labels.

(III) The operands for the branch instructions are substituted for absolute locations. Full error checking for undefined labels is also carried out - this is the final phase of error checking. This step is needed after PASS-II because labels referred to by operands in PASS-II may not have been defined because of forward referencing. In other words, all the labels have to be known before absolute locations can be calculated and inserted.

5.7.2 The Translator

The translator is basically a large database of predefined routines that translate the intermediate code into microcode. At this stage, it is assumed that all error checking has been carried out and the program is assumed to be correct. (As in all programs, the order of instructions is determined by the programmer and it is beyond the

assembler's capability to check for an illogical sequence.)

Each line of the intermediate code is read. A number representing the value of the intermediate mnemonic is determined by the use of a translator table. This number is then used as an index to call a routine representing the correct micro-instruction(s) for that instruction. The operands associated with the intermediate mnemonic are passed to the routine and written into the appropriate microcode field. For example, consider the mnemonic

```
MOV #4,R7
```

this is translated into the intermediate opcode

```
MOVDR 4 7
```

This is then read by the translator where the intermediate code 'MOVDR' is translated into its corresponding position in the table, in this case '9'. This number is then used as an index into a large database of routines where, in this case, it corresponds to a routine appropriately named "MOVDR". The operands '4' and '7' are passed to this routine where the correct sequence of micro-instructions is generated with '4' written into the data field and '7' written into the B-register field. The microcode is then written to disk.

5.7.3 The Loader

The function of the loader is to load the code from disk to SIP and run the program. The advantage of having a separate loader is that code can be transported to different machines, it being only necessary to write a loader for each new machine. Once the code has been downloaded to SIP, it is not necessary to load the program again in order to run it. SIP has a 'RUN/HALT' flag in its CSR (Figure 5.3) which executes the code resident in its program memory. It is only necessary to load new code (a) after a power failure or (b) when a new program needs to be downloaded.

The assembler and the translator generate non-optimised code (note that here we are dealing with time-optimisation). As we discussed in Section 5.5, SIP's architecture exhibits a high degree of internal parallelism thus enabling several instructions to be executed simultaneously. The ability to detect the presence of the instructions that can be merged leads us onto the concept of microcode optimisation. This is discussed next.

5.8 MICROCODE OPTIMISATION

Generally speaking, a microprogram is said to be optimal if there are no other functionally equivalent microprograms that can be run on the same machine which require a smaller number of clock cycles. It is generally accepted that hand-coded optimisation (often termed compaction) produces more optimal code (fewer cycles to execute) than machine optimisation. The need to compare the differences between human and machine compaction drove Fisher et al. [22] to investigate how well a machine can produce optimal code.

There are two methods of compacting a program: local and global compaction. Local compaction is the ability to detect parallelism in straight-line program segments, whereas global compaction encompasses the analysis of microprograms that include iterations and conditionals. For example, if a variable is assigned within a loop then, assuming it is not used elsewhere within the loop, it can be taken outside without altering the logic of the sequence of instructions; this means that a cycle is saved within the execution of the loop. Fisher considered the optimisation of a floating point divide routine. Global compaction compared quite favorably with hand-compacted code; the average execution time being 16.4 and 16.1 cycles respectively. The initialisation code for the routine took 14 cycles in both cases. Similar results were achieved when tested with a floating point add; however, local

compaction required some 60% additional cycles in both cases. These results provided evidence that automated global compaction is feasible and can be competitive with hand-compacted code.

To fully optimise a microprogram, it is necessary to exploit every possible occurrence of concurrently executable micro-instructions. This is a difficult task since, even though two operations may be found to be concurrently executable by a parallelism detection technique, they may not be concurrently executed because of resource contention. This adds an extra degree of difficulty to the problem of optimising microcode. By defining a graph model of a microprogram showing the earliest and latest events at which a micro-instruction could occur and by defining a resource requirement matrix, Tsuchiya and Gonzalez [122] showed acceptable results for many microprograms.

Ramamoorthy and Tsuchiya defined a high-level language SIMPL (Single Identity MicroProgramming) for microprogramming [98]. This detected and merged concurrent micro-operations. Various techniques were used such as detection of parallel processable microstreams, optimisation of concurrent micro-instructions and minimisation of micro-instruction sequences. Probably a more ambitious task was the work done by Malik and Lewi [22]. This concerned the development of a machine independent language, VMPL, that allowed micro-instructions to be expressed as declarations within a program. In other words, a program written in VMPL is a specification for a target machine. In order to execute a microprogram written in VMPL on the target machine, the program is first translated into an intermediate code and then compiled into microcode. An optimisation scheme developed by Lewi [22] was used to optimise the intermediate code produced.

To optimise SIP's microcode, it is necessary to locate the earliest and latest times that microoperations can occur without resource contention. This will now be discussed.

5.8.1 Optimising SIP's microcode

The main purpose of optimising SIP's microcode is to reduce the execution time of an algorithm (see Section 6.4 for the results). In order to achieve this, we need to detect as much instruction parallelism (simultaneous execution of multiple instructions) as possible. This differs from the parallelism offered by the parallel language Occam (the transputer's native language) in two different ways: (1) the parallelism is explicitly stated in Occam, whereas here it is implicitly stated, and (2) parallelism occurs at the task (procedural) level in Occam while here it occurs at the instruction level.

The major reduction in execution time by optimising the code generated by the assembler is from the parallelism associated with the simultaneous processor instruction execution of a pixel with a pixel fetch (Section 5.6). In order for this operation to occur, we need to locate the next pixel to be operated on so we can merge the two instructions. This is easily detected by looking ahead in the intermediate code. When a pixel mnemonic is encountered (conveniently indicated by a P in the 4th or 5th column of the pseudo-microcode opcode), the program "looks ahead" until either a pixel mnemonic or a branch instruction occurs. If a pixel mnemonic occurs then the pixel fetch of the second instruction can be merged with the ALU function of the first instruction. If a branch is encountered first then no merging can occur since optimisation is incapable of detecting the sequence of instructions a program will execute after a branch. These constraints may appear too restrictive but analysis of many image processing algorithms shows that a series of pixel reads followed by a pixel write

frequently occurs (c.f. the Sobel where there are 12 reads followed by a write).

A second reduction in execution time (although only slight) can be gained from the ability to merge Jump-to-subroutine (JSR) and Return-from-subroutine (RTS) instructions. If an instruction previous to a JSR instruction does not make use of the data bus or 'useful' use of the program counter, then the JSR instruction may be merged with the previous instruction. This generally occurs for instructions such as register-to-register and memory-register accesses but not for data-to-register or data-memory type instructions, e.g.

```
MOV R0,R1
JSR label
```

may be merged as the MOV instruction does not make use of the data bus whereas

```
MOV #4,R0
JSR label
```

cannot be merged because they both use the data bus. RTS may also be merged in a similar manner if the PCcntrl field before the RTS is set to continue (default state). With efficient writing of code, jumping and returning from subroutines can effectively occur in zero cycles.

Multiply can also result in zero cycles. For example, if a 'MUL' instruction is encountered, then it is necessary to back track through the previously encountered section of code looking for the earliest time the required registers can be used, i.e. when they were last used as a destination. When this situation occurs, the registers used as operands in the MUL instruction may be loaded into the multiplier simultaneously with that instruction. This would eliminate the need for the MUL instruction as the two registers will have been multiplied (the multiplier multiplies the contents of its input registers on every cycle of the system clock) by the time the result is required. Thus,

multiplication in this case appears as requiring zero cycles.

Another major reduction in a program's execution time is by the use of registers rather than local memory. Register accesses only require a single cycle whereas local memory usually requires about 2-3 cycles. Thus, by transferring the required memory locations to be operated on to registers just after a JSR instruction manipulating them and re-storing them to their respective locations just before the RTS instruction, the execution time of an algorithm can be dramatically reduced. This will also assure that JSR and RTS instructions can be merged.

These are just a few examples of possible optimisations that are carried out on SIP's code. As with many optimisers, the degree that a program can be optimised relies heavily on the programmer for efficient layout of code. Chapter 6 examines the decrease in program execution time that is normally gained from optimisation of SIP's code.

5.9 THE VMEBUS

SIP is based on the public domain VMEbus [80]. This was chosen for several reasons:

1. It has a high bus bandwidth of 24Mbytes/sec

2. It is a true multiprocessor bus and is thus capable of supporting an arbitrary number of processors up to the limit of the number of backplane slots. SIP can therefore communicate with peripherals such as frame stores, external memory and disk drives - this opens up the commercial market for image processing systems.

3. The array processor LAP-II (Section 6.2.2) is based on the VMEbus.

As described in Section 5.5.4, SIP has 7 bits in the VMEfield (VMEF) in its microcode. This allows SIP to communicate with a wide range of

peripherals.

5.10 SUMMARY

This chapter has described, in detail, a high-speed sequential processor (SIP). The assembler for SIP was also described which included a brief insight into microcode optimisation. SIP was built to investigate the possibility of being combined with a parallel processor. The next chapter (Chapter 6) introduces the Linear Array Processor (LAP) and highlights the performance abilities of both SIP and the LAP when applied to image processing. This will permit us to study whether such a combination of processors has a greater cost-speed tradeoff than other designs.

CHAPTER 6

RESULTS AND THE FUTURE

"To study, to finish, to publish."

Benjamin Franklin 1706-1790

6.1 INTRODUCTION

In Chapter 5 we stated that SIP and a parallel processor (the LAP) were initially designed to be combined as part of a multiprocessing system for image processing. This chapter describes the Linear Array Processor (LAP) and its associated compiler, PPL. A discussion on factors that affect machine performance leads on to an analysis of SIP and the LAP. These results along with the machine's corresponding costs will be compared with a variety of similar machines commercially available. An analysis of the performance of the system when both processors are combined is undertaken which will show the feasibility of this kind of configuration. However, first we will describe the Linear Array Processor.

6.2 THE LINEAR ARRAY PROCESSOR

The Linear Array Processor (LAP) was developed at the National Physical Laboratory by Plummer [93] and is classed as a SIMD machine (Section 7.4). Like SIP, the LAP is a microcoded bit-slice processor.

The object of the design was to increase data throughput by parallelism in order to achieve execution times typically in the order of 100 times faster than conventional computers, while minimising the cost by using cost-effective bit-slice processors (AMD 2901). The architecture [94] (Figure 6.1) is essentially a linear array of 256 1-bit processing elements (each PE has 256 bits of local memory), operating on all pixels in a line of a 256x256 image simultaneously. The program memory of the LAP is 16K x 24 bits. Because each processor is a 1-bit processing element, the LAP is a bit-serial machine (Section 7.10), i.e. the pixels are operated on one bit at a time rather than a byte at a time as in SIP. Bit-serial devices are discussed in more detail in Chapter 7.

Because a line of an image must be loaded into an array processor before it can be operated on, a data bottleneck can often arise. The LAP is capable of loading a line of data while the previous line is being processed. However, a slight bottleneck can still exist if the loading of a line takes longer than the time spent processing each line. For ease of programming, the PPL high-level language and associated compiler was developed at the NPL. The compiler translates high-level code to the LAP's microcode. This will now be described.

6.2.1 The PPL Compiler

The PPL (Picture Processing Language) language allows image processing commands to be expressed in simple one-line commands and to translate these commands into microcode for the LAP. The user interface is via the PPL prompt '**:' where commands are entered. Programs stored on disk are run by typing the name of the program. (A program can also call other PPL programs - see later.) The body of a PPL program lies within the construct

```
apply .... end
```

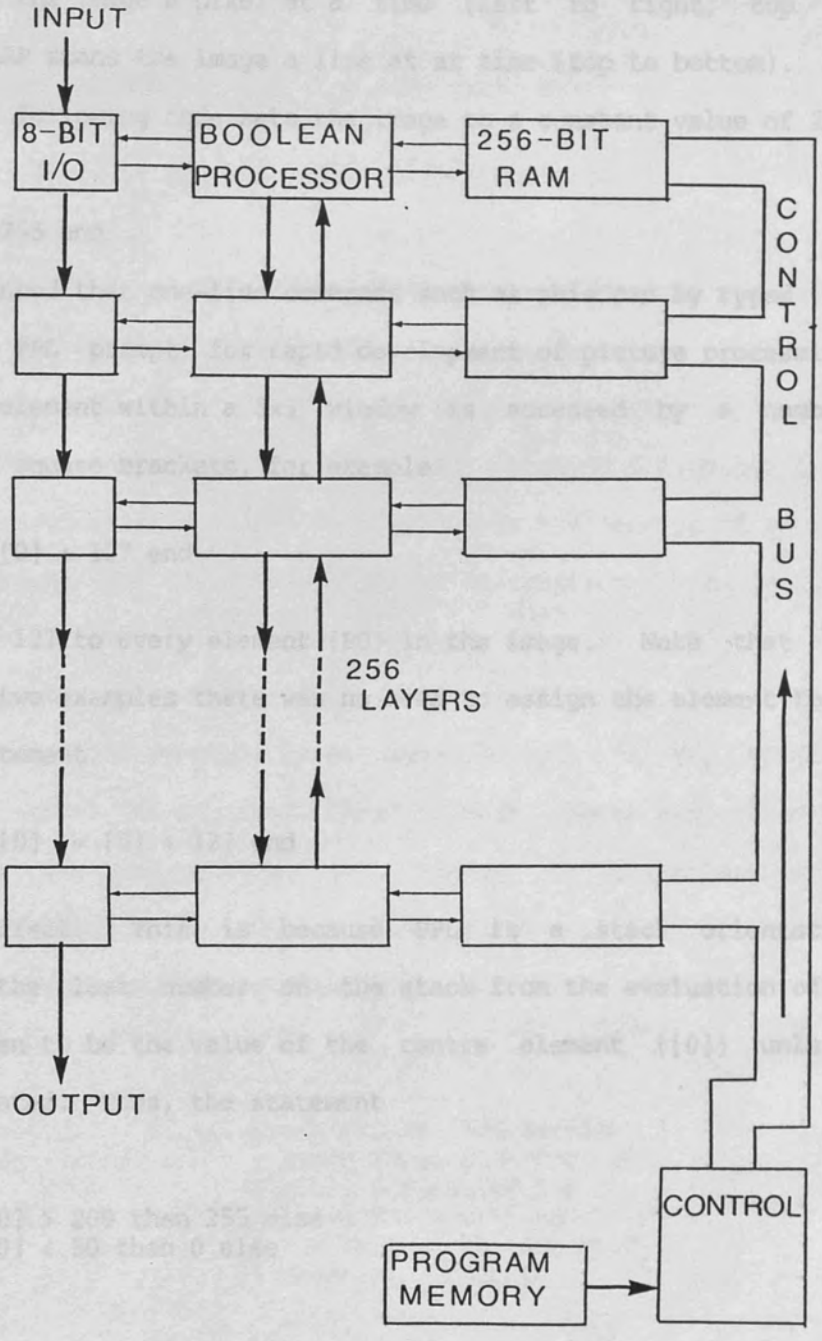


Figure 6.1 Architecture of the Linear Array Processor (LAP)

This is similar to SIP's apply...end construct (historically it was the other way around) in that it scans over the whole image space. However, while SIP scans the image a pixel at a time (left to right, top to bottom), the LAP scans the image a line at at time (top to bottom). As an example, the following code sets the image to a constant value of 255 (white)

```
apply 255 end
```

(It should be noted that one-line commands such as this can by typed in following the PPL prompt for rapid development of picture processing routines.) An element within a 3x3 window is accessed by a number enclosed within square brackets, for example

```
apply [0] + 127 end
```

adds the number 127 to every element (P0) in the image. Note that in the previous two examples there was no need to assign the element to a value. The statement

```
apply [0] := [0] + 127 end
```

has a similar effect. This is because PPL is a stack orientated compiler, so the last number on the stack from the evaluation of a function is taken to be the value of the centre element ([0]) unless otherwise indicated. Thus, the statement

```
apply
  if [0] > 200 then 255 else
  if [0] < 50 then 0 else
   127
end
```

scans the image setting the element [0] to either 0, 255 or 127, depending on the value of [0] from the original image. The PPL language contains familiar high-level constructs such as

```
if ... then ... else ...
A := operand
for i := 1 step s until N do
```

Thus, to threshold an image at 127, the code would be:

```
apply   if [0] > 127 then 255 else 0   end
```

The ability to temporarily save and store images during execution of a program is essential if the same image is to be operated on by several different operations. PPL achieves this by the commands 'get' and 'put'. Below is an example involving these commands. Although this program has no particular function, it illustrates the concept of saving and storing images in PPL. Note that the instruction "param" allows parameters to be entered via the call of the program from the PPL prompt. Thus, at the PPL prompt, if the command 'TEST(100)' was called, then T in the program below would be set equal to 100. A PPL program can also call other PPL programs stored on disk. These are called by their names (in upper case letters) with appropriate parameters. In this case the programs called are SOBEL which applies a Sobel edge detect over the image and TH(param) which thresholds the image on the value param.

```
param T;                    { get number from terminal }
put ORIG;                   { store image with name ORIG }
SOBEL;                      { apply a SOBEL of the image }
put TEMP;                   { store image with name TEMP }
get ORIG;                   { GET image with name ORIG }
TH(T);                      { threshold it at T }

apply
 [0] := [0] AND TEMP;       { threshold AND sobelled image }
end
```

This is fairly self-explanatory. By 'GETting' an image, the new image becomes the current image. By 'PUTting' an image, the current image is stored while remaining the current image.

We previously concluded that a microcoded machine appeared suitable for image processing because it allows application specific commands or functions to be defined and optimised in the language. PPL has several commands that allow frequently used image processing commands to be expressed in a single line, when they would usually be expressed in several lines of code each. Below are examples of these and their equivalent Pascal codes:

```
N := sum K = 0:8 of [K]      N:=P0+P1+P2+P3+P4+P5+P6+P7+P8

N := max K = 0:8 of [K]      IF P0>P1 THEN max:=P0 ELSE max:=P1;
                             IF P2>max THEN max:=P2;
                             IF P3>max THEN max:=P3;
                             IF P4>max THEN max:=P4;
                             IF P5>max THEN max:=P5;
                             IF P6>max THEN max:=P6;
                             IF P7>max THEN max:=P7;
                             IF P8>max THEN max:=P8;

N := min K = 0:8 of [K]      Ditto, substituting the > with a <
                                    and max with min
```

e.g.

```
apply  ( sum K = 0:8 of [K]) / 9  end
```

The above example averages the pixels in the 3x3 window by summing the centre pixel and the surrounding pixels and dividing by 9. The equivalent code in Pascal is

```
y:=0;                    { Setup scan for a 128x128 image }
REPEAT x:=0;
 REPEAT
  Q0:=(P0+P1+P2+P3+P4+P5+P6+P7+P8) DIV 9; {average of a 3x3 window}
  x:=x+1
 UNTIL x=128; y:=y+1  { increment scan counters }
UNTIL y=128;
```

The LAP-I was built in 1980/81. With the advent of VLSI and the availability of semi-custom gate arrays in recent years, the possibility arises of optimising the performance of the LAP-I. The next section describes the LAP-II which is an enhancement of the LAP-I.

6.2.2 LAP-II

The LAP-I was designed with cost-effective boolean devices; however, the LAP-II has been designed to take advantage of recent technological developments. The following enhancements have been made:

i) The AM2901 processors have been replaced by semi-custom integrated circuits. Because the processors are custom designed, they have been optimised for the LAP architecture. Each individual chip now has eight processing elements (c.f. four per chip in the LAP-I), an activity bit for control of instruction execution and a larger amount of memory (32Kx1 per PE). As a further enhancement, it is possible for a global test to be executed throughout the PE's such that they ALL execute the same function if the result in ALL PE's is TRUE.

ii) A program sequencer has been added that enables branches and loops to be executed efficiently.

iii) While the LAP-I was based on its own private bus, the LAP-II is based on the VMEbus. This allows VMEbus based products such as SIP to communicate with the LAP-II.

We will now discuss machine performance. This will enable us to examine SIP and the LAP from a performance point of view.

6.3 MACHINE PERFORMANCE

One often needs to know whether a machine is suited to an application, e.g. its cost must be within a given budget while being able to execute a program within a given time constraint. A method of achieving this is to run the program on several machines and measure the various execution times. A cost-speed choice can then be made to determine which machine is best suited to the application. However,

this method is often not practicable and one usually has to rely on figures that manufacturers state as being a measure of their machine's performance. A problem with measuring the performance of a machine is that there is no standard definition of "measure". One of the most common practices among manufacturers is to state the performance in terms of MIPS (millions of instructions per second) or MFLOPS (millions of floating point instructions per second); however, this often bears little resemblance to how well a machine will execute a user's program.

Another alternative to represent the performance of a machine is to state the execution times of several "typical" algorithms. This is, in many ways, a more suitable method as it allows us to relate a real set of results to an application; however, the problem that arises here is that "typical" will ultimately be area/application dependent. For instance, with regard to image processing, "typical" operations may represent operations that access a pixel and its neighbours within a 3x3 (or larger) window. Thus, frequent array accesses may be the most common denominator in the program and hence the most predominant factor in the execution time of the program. On the other hand, "typical" operations in a scientific environment may involve many millions of floating point operations. This would constitute a different kind of "typical" program to the image processing programs.

No attempt here is undertaken to define a machine's "measure of performance"; however, we will choose the last of the above methods of assessing the performance of SIP and the LAP, i.e. finding the execution times of several "typical" algorithms. Since we are mainly concerned with the image processing area, we have chosen a set of algorithms to analyse the performance of SIP and the LAP that contain operations that we have found common in image processing. These include point-point (pixel-pixel) operations, operations that involve a pixel and its neighbours in a 3x3 window, and algorithms that have been found

advantageous to industrial inspection, e.g. the Hough transform, thinning, etc. However, because this is not a complete representation of image processing operations, one must treat these results carefully. First let us consider, from a hardware point of view, what main factors influence the performance of a machine.

6.3.1 Machine Architecture and Machine Performance

The following five factors contribute to machine performance:

1. Type of technology (TTL, ECL, GaAs, etc.).
2. Number of processors.
3. Bit efficiency.
4. Orthogonality.
5. Addressing capability.

The type of technology used is an important factor in governing a machine's performance. ECL static RAMS can achieve access times in the region of 6ns while recent developments in the TTL range achieve access times down to 15ns. Thus, the faster the components, the faster the processor clock frequency, up to the maximum limit allowed for that processor. An interesting point to note is that, in a recent set of results published by Billig and Cronk [10], they showed that their benchmarks (from comparing various LSI-11/23 configurations (with and without floating point) with an Intel iSBC-86 (again with and without floating point) and Motorola's M68000 (running at 8MHz)) showed no relationship whatever to the processor clock frequencies. As time goes on, improvements in technology may well slow down so much, that a more concentrated effort into architectures will need to be undertaken. Some architectural features that affect machine performance will now be

discussed.

Obviously the number of processors in a machine will affect the performance of the machine. However, the topology of the system (the way the processors are connected together) is very important and usually has to "match" a task to achieve optimum performance. This is discussed more fully in Chapter 7 where a series of machines are reviewed and discussed.

The next point to note is the concept of bit efficiency. This allows a computer to execute an algorithm using fewer instruction bits, and is therefore a measure of its functionality. Bit efficiency is a function of the number of words in an instruction word and the number of operations performed for each instruction. A computer with a long instruction word may therefore be more bit efficient than a computer with a small instruction word, if the computer can do an equal number of operations with far fewer instructions. The benefits of bit efficiency are small program size, high execution rates and fewer memory references to fetch program instructions.

Orthogonality measures the ability of a computer to address different types the same way, independent of the data type it references. A possible problem with non-orthogonal computers is that they usually have difficulty addressing particular data types. Therefore, they may take several operations to perform a function on a particular data type that could otherwise be executed in a similar amount of time. This problem would cause a decrease in the performance of the machine.

The addressing capability of a computer determines how it accesses external devices. An architecture with good addressing capability uses the same instructions to address a processor register, an I/O device, main memory etc. There is also no distinction between data and address

locations. Often a processor has separate data and address registers, e.g. the 68000 series. In this case, when manipulating an index into an array, it is necessary (e.g. in some instructions with the 68000) to manipulate a data register and transfer it to an address register to form the index before the data can be retrieved. This requires additional lines of code and hence reduces total system performance.

Now that we have determined some of the factors that affect machine performance, and problems that can arise during "measuring" the performance, we shall now present the results from SIP and the LAP. As discussed before, these are based on operations and algorithms that we have frequently used and that we have found common in many inspection algorithms.

6.4 PROGRAM RESULTS

Since SIP and the LAP have been designed specifically for image processing, a suite of "typical" image processing algorithms has been used (as described in Section 6.3) to demonstrate the processing capability of the processors. Note that this is not an exhaustive set of algorithms but merely demonstrates the machine's performance in these situations. All programs have been written in the machine's own language.

Strictly speaking, a direct comparison between SIP or the LAP and the PDP-11/73 should not be made since SIP and the LAP are special purpose image processing processors whereas the PDP-11/73 is general purpose; however, it does demonstrate the advantages of special purpose hardware and bit-slice designs over commercial processors. Also, no

---

[1] Note that the PDP11/73 has a floating point unit and 8Kbytes of cache

direct comparison should be made between the LAP and SIP since SIP was specifically designed to execute the algorithms the LAP was incapable of executing. Nevertheless, a comparison is made by comparing the ratio (LAP/SIP) of the execution times of the same algorithms executed on both machines. This will show if a bit-serial parallel processor has a significant performance improvement over a bit-parallel serial processor. The results for the three machines (SIP, PDP-11/73[1], and the LAP) are given in Table 6.1.

As explained in Section 5.8.1, it is possible for SIP to execute several instructions concurrently. For this reason, SIP has been run with non-optimised code (noc) and optimised code (oc). This allows us to observe the performance increase caused by code optimisation. All times are given in milliseconds (ms) and exclude I/O (add 30ms onto the times for SIP for input and output and ~1s for the LAP). As mentioned in Chapters 3 and 5, SIP and the PDP-11/73 both operate on a 128x128 image; however, the LAP operates on a 256x256 image. The times given for SIP and the PDP-11/73 are therefore the times taken to operate on a 128x128 image plane multiplied by four. All programs were executed on the same image to ensure no biasing for data dependent operations. Those entries entered as a double asterix (**) mean that the machine was incapable of executing the algorithm for some architectural reason.

The next table of results (Table 6.2) shows the performance increase due to code optimisation for SIP both in terms of program execution time and code reduction. The following table (Table 6.3) shows the ratio between SIP and the LAP, SIP and the PDP-11/73 processor and SIP and a 68000 processor running at 8MHz. A double asterix (**) indicates that there were so few lines of code or that optimisation produced so little change in code length (usually because of many branches in the algorithm) that there was no significant increase in performance. All times are given in milliseconds.

Algorithm	SIP (noc) (ms)	SIP (oc) (ms)	LAP (ms)	PDP-11/73 (ms)
Sobel	372	224	37.4	4304
Threshold	63	45	6	1168
Complement	35	26	5.7	984
Edge	372	264	36	4900
Binary Smooth	424	237	16.6	3760
Mean (3x3)	258	258	18.7	3056
Expand/Shrink	263	117	11	2424
Robert Cross	178	134	14	1692
Median (3x3)[1]	2000	1780	166	46428
Intensity Histogram	88	64	**	2756
(a) Centre Find[2]	184	164	**	4816
(b) Centre Smooth	0.44	0.44	**	8
(c) Draw Circle	588	588	**	7976
Thin[3]				
3-iterations	1280	1180	**	25900
5-iterations	2760	1884	**	42846

Table 6.1 Table of the execution times for SIP, the LAP and the PDP-11/73

---

[1] Note that the median is also useful for such algorithms as corner finding [88], etc. as well as noise removal. It is therefore thought important enough to be included.

[2] These steps represent the first three steps of the O-ring algorithm, i.e. it involves the Hough transform. The Hough transform has previously been cited as being useful for industrial inspection and has therefore been included.

[3] The thinning algorithm used was that by Davies and Plummer [23]. This is a parallel algorithm that requires a continuous full scan over the image, applying the algorithm until no change in the image occurs. In this case, two images were used - one took 3 iterations and the other took 5 iterations.

Algorithm	Reduction in execution time (%)	Code reduction (%)
Sobel	40	48
Threshold	29	21
Complement	26	12
Simple edge detector	29	25
Binary Smooth	44	32
Mean (3x3)	**	**
Expand/Shrink	56	36
Robert Cross	25	19
Intensity Histogram	27	8
a) Centre Finder	11	
b) Centre Smooth	**	25 (for all three)
c) Draw Circle	**	
Median filter (3x3)	11	32
Thin[1]		
3-iterations	8	15
5-iterations	32	15

Table 6.2 Table of the reduction in execution time and code length after optimisation of SIP's code

Algorithm	LAP/SIP	PDP/SIP	68000/SIP
Sobel	0.16	20	13
Threshold	0.13	26	13
Complement	0.22	38	14
Edge	0.14	19	13
Mean (3x3)	0.07	12	14
Expand/Shrink	0.09	21	15
Robert Cross	0.10	13	13
Median (3x3)	0.09	26	16

Table 6.3 Table of the ratios of the average execution times for SIP, the LAP and the 68000

6.5 ANALYSIS OF THE RESULTS

Inspection of Table 6.2 shows that optimised code achieves an average of 30% speed reduction over non-optimised code with a corresponding average of 30% reduction in code size (note that these two figures are not directly related to each other). From Table 6.3, the LAP has (on average) only a factor 8 speed improvement over SIP when SIP uses optimised code whereas SIP has, on average, an algorithm execution speed of 25-30 over the PDP-11/73.

6.5.1 Performance against other machines

The performance of SIP has also been compared against a variety of other machines. The results in Table 6.3 show that SIP has an average factor of 14 speed improvement over the 68000 processor. (Note that all times are based on a 68000 processor running at 8MHz operating on a 128x128 image.) These results give a fairly realistic figure of performance as the algorithms involve both single point and 3x3 kernel operations.

Probably the most realistic comparison is comparing SIP's results with those published by Logica [97] for DIPOD (Section 7.8.1). DIPOD is similar to SIP in that it is based on a bit-slice architecture; however, their similarities extend beyond the basic architecture. DIPOD and SIP are both based on the AMD 29203 bit-slice processor running at the same clock frequency (8MHz). This demonstrates an important point in that the performance must now be governed by the architecture and not by the type of processor used or the clock frequency. Below are the results published by Logica against those achieved by SIP. Unfortunately, these are the only figures currently available for DIPOD. The Sobel is based on a 128x128 image plane.

1. Sobel: DIPOD - 85ms SIP - 55ms

2. Naive List reversal (30 entries): DIPOD - 200μs SIP - 29μs

This shows that a machine's performance has a strong dependency on its architecture. Another point to note is that the cost of a single SIP is around £620 (one off) whereas a single DIPOD node (a DIPOD system is built from many nodes) costs around £30,000 (commercially). Comparing the price/performance ratio one can see that, even assuming a commercial price of £2000 for SIP, SIP would be a more advantageous solution (see also Chapter 8). Based on the same cost analysis, the LAP costs about £9000 commercially.

6.6 A PARALLEL/SEQUENTIAL CONFIGURATION

In Section 5.2 we showed that many image processing algorithms include both sequential and parallel processes. Thus, a parallel algorithm (e.g. a Sobel) executing on a sequential machine (SIP), is inefficient relative to the same algorithm executing on a parallel machine (LAP). Also, a sequential algorithm executing on a parallel machine is inefficient when compared with the same algorithm executing on a sequential machine. Because many image processing algorithms include both sequential and parallel processes, it seems clear that either processor executing an algorithm consisting of both parallel and sequential processes will be inefficient in one way or another.

This leads on to the possibility of combining both SIP and the LAP so the LAP would execute the parallel processes and SIP would execute the sequential processes. An example of this might be:

(1) Find the edges of the object by application of a Sobel.

(2) Threshold the image.

(3) Chain code the image.

(4) Extract measurements from the chain code.

The above algorithm could be partitioned into two parts: the LAP could execute parts (1) and (2) while SIP could execute parts (3) and (4). Both SIP and the LAP would have their program code (microcode) loaded into their respective program memories by an external host. SIP would load the image into the LAP, run the LAP, then fetch the resultant image and load it into one of its image planes for further processing. Unfortunately, the LAP-II is as yet unfinished and its performance is unknown, though it is estimated that it will have a speed improvement of approximately two over the LAP-I. The following analysis is based on the results derived from the LAP-I.

In order to execute the above algorithm on the LAP-I, SIP is required to load the data (the image) into the LAP, run the LAP, then fetch the result (65,536 pixels). This is likely to cause a data bottleneck, particularly if parallel and sequential tasks are scattered throughout the algorithm. For example, considering the example above, SIP takes 25 cycles per pixel to apply a Sobel and 5 cycles per pixel for the threshold. Therefore, 30 cycles are needed for every pixel in the image. This will lead to an execution time of

256 * 256 * 30 SIP clock cycles.

on a 256x256 image. However, the LAP takes 1172 cycles for the Sobel and 210 cycles for the threshold (per line) therefore, the total time to execute the parallel part of the algorithm will be

256 * 1382 LAP clock cycles

which is ~5-6 times faster that SIP. (Note that both the LAP and SIP operate at the same clock frequency of 8MHz: hence a direct comparison can readily be made.) However, the additional overhead of transferring data from the LAP means an additional

$$(256 * 256 * 3) + (256 * 256 * 3)$$
$$\text{(to LAP)} \qquad \text{(from LAP)}$$

SIP clock cycles will be needed (assuming three cycles per pixel VME read and write). This means that a Sobel and threshold on a SIP/LAP configuration is in fact executed in

$$(256 * 1382) + (256 * 256 * 3 * 2) \text{ clock cycles}$$

Thus, about 53% of the total time to execute the parallel task is spent transferring data. This reduces the efficiency of the system to only a factor 2.6 greater than SIP as a stand-alone processor. Note that the LAP was originally designed to accept data directly from a line-scan camera, although this was never implemented. In this case, the first data transfer can be eliminated (since data need not now be transferred to the LAP); thus, the percentage of time spent transferring data is reduced to 35.7%, producing a SIP/LAP ratio of 3.6.

If parallel and sequential tasks are scattered throughout the algorithm, then SIP will need to transfer data more frequently throughout the algorithm. This will eventually cause a much larger data bottleneck in the system. However, as we mentioned in Chapter 4, there will be cases where a parallel processor will greatly enhance the performance of a system (here, we have taken rather trivial examples). For instance, there will be cases when there will be several parallel tasks and few sequential tasks. In this case, a parallel/sequential processor configuration may greatly reduce the execution time of an algorithm by several factors compared with the situation where the

algorithm is implemented on a single sequential processor. This requires a more detailed investigation; however, as the LAP-II is as yet unfinished and the performance is unknown, any further calculation will be rather speculative. For this reason, this study will not be continued here. The above calculations only serve to give a brief insight into possible timings and how the system would operate.

Another point worth mentioning is the problem that occurs when programming two different bit-slice architectures (particularly a sequential and parallel processor) when configured as a complete system. Since SIP and the LAP have different microcode formats and have, as a result, acquired different languages, they will have to be programmed accordingly - this is a situation to be avoided.

6.6.1 Conclusions

The initial results of a system composed of SIP (sequential processor) and the LAP (parallel processor) show that a bottleneck is likely to occur because of SIP having to transfer data to and from the LAP. Another problem is the need to program these machines in different languages when implementing a complete system. Because each machine has a different microword format, programming is made more difficult. An alternative is to have a common language and the parallel and sequential tasks are partitioned by the programmer. However, because the microword format for each machine is hardware dependent, knowledge of the hardware for each machine must be incorporated into the language. As the number of hardware modules increases (and hence different microword formats), this situation will become undesirable.

Chapter 8 investigates various configurations using multiple SIPs. This has the advantage that, because each processor has the same architecture and microword format and each processor executes the same

program, the simplicity of programming is maintained while the processing power is increased as the number of processors increases.

6.7 SUMMARY

This chapter has discussed the problems concerned with the problem of measuring machine performance. A MIPS figure is frequently quoted as a measure of machine performance; however, this figure often bears little relation to the execution time of a program. For this reason, it was decided to represent the performances of SIP and the LAP by stating the execution times of several image processing algorithms. These contained operations that we found frequent in image processing. A comparison against conventional processors and processors of a similar nature was also undertaken. Optimisation of SIP's code has shown that a 30% reduction in code size and a 30% decrease in execution time could on average be obtained. Comparing these results with those obtained for DIPOD which has many similarities to SIP, it was found that the architecture of the system plays a crucial role in the performance of a machine.

Initial results of a multiprocessor configuration composed of the SIP and the LAP show that this was likely to incur a bottleneck because of the data transfer to and from the LAP; however, because the performance of the LAP-II is as yet unclear, this area of investigation is incomplete. The next chapter reviews a variety of architectures that have been proposed and employed for use in image processing. These tend to be SIMD (parallel) machines for reasons that were summarised in Chapter 1. From the results gained here and an analysis of other architectures, Chapter 8 describes a variety of configurations composed of multiple SIPs (sequential processors) and analyses the performance of each configuration. This entails an analysis of processor bottlenecks.

CHAPTER 7

ARCHITECTURES

"...why they are as they are, and not otherwise"

Mysterium Cosmographicum Preface

7.1 INTRODUCTION

This chapter presents a study of computer architectures and techniques that have been adopted to eliminate some of the classic problems associated with the traditional von Neumann architecture. An attempt to clarify the question "what is a computer architecture?" is undertaken which leads to the classification of architectures. Criteria for choosing an architecture are discussed which highlight some of the pitfalls that can be avoided by measuring specific features of a proposed architecture. This includes optimisation of an architecture for a particular problem with regard to cost-speed tradeoffs.

A varied set of architectures that have been used for image processing are reviewed. Particular idiosyncratic features that have been employed in the architecture are highlighted including reasons why these features were implemented. These thoughts are applied to image processing where a high degree of both sequentialism and parallelism is frequently required. The conclusions from this chapter are applied to a hybrid architecture (Chapter 8) that attempts to execute both sequential

and parallel processes efficiently. However, first let us discuss the common source of inefficiency in most microprocessors.

7.2 ARCHITECTURES

With few exceptions, modern computers still employ the original von Neumann architecture as depicted in Figure 7.1. It consists of a single processing unit and a single memory for both instructions and data. No distinction is made between data and instructions, enabling data to be interpreted as instructions and instructions as data. The problem with this approach is that only one location in memory can be accessed at any one time: hence instructions have to be fetched and decoded before the data can be fetched. Several machine cycles are therefore needed to implement each instruction and this is highly inefficient; for example, consider a typical nonpipelined von Neumann processor that takes five machine cycles to execute an instruction. (Note that more cycles may be needed for some instructions. Also, each processor has a different instruction fetch/decode/execute scheme. This example merely serves as a typical sequence a processor will follow to implement a short instruction.)

Cycle 1 - Fetch instruction
Cycle 2 - Decode instruction
Cycle 3 - Fetch data
Cycle 4 - Execute instruction
Cycle 5 - Store result

If L operations (here L=5) are required for each instruction and T is the time required to execute each operation (usually the period of one clock cycle), then the total time taken (t) is

$$t = L*T$$

The maximum instruction execution rate (r) is therefore

$$r = 1/(L*T)$$

We can immediately see that this is highly inefficient because the

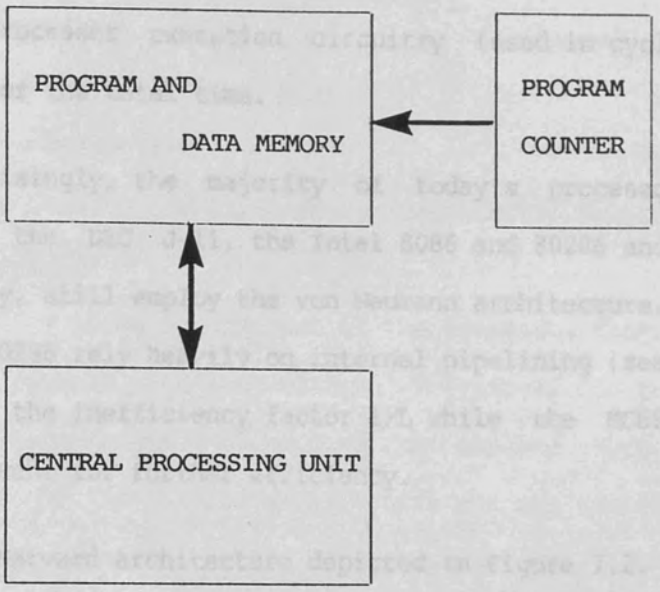


Figure 7.1 The traditional von Neumann processor. Program and data are held in the same memory

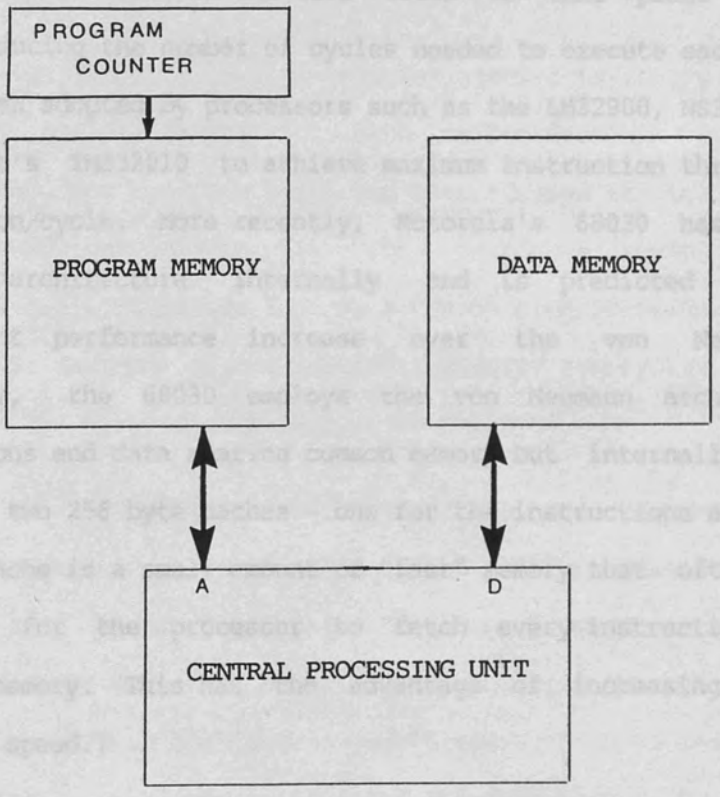


Figure 7.2 The Harvard architecture. Program and data are held in separate memory blocks

actual processor execution circuitry (used in cycle 4) is active for only 1/L of the total time.

Suprisingly, the majority of today's processors including the MC68020, the DEC J-11, the Intel 8086 and 80286 and machines like the VAX family, still employ the von Neumann architecture. Processors such as the 80286 rely heavily on internal pipelining (see Section 7.2.1) to eliminate the inefficiency factor 1/L while the MC68020 also employs on-chip cache for further efficiency.

The Harvard architecture depicted in Figure 7.2. was developed in the 1940's by Howard Aiken (inventor of the Mark I calculator) of Harvard University [128]. This diverts from the classic von Neumann architecture in that it maintains separate program and data spaces allowing the instruction and data fetch to take place concurrently, hence reducing the number of cycles needed to execute each instruction. It has been adopted by processors such as the LM32900, NS32532 and Texas instrument's TMS32010 to achieve maximum instruction throughput of one instruction/cycle. More recently, Motorola's 68030 has adopted the Harvard architecture internally and is predicted to achieve a significant performance increase over the von Neumann 68020. Externally, the 68030 employs the von Neumann architecture with instructions and data sharing common memory but internally, the 68030 features two 256 byte caches - one for the instructions and one for the data. (Cache is a small amount of "fast" memory that often eliminates the need for the processor to fetch every instruction from "slow" external memory. This has the advantage of increasing the program execution speed.)

As an example, consider the von Neumann machine above that took five cycles to execute an instruction. The Harvard architecture in a similar setup fetches the data in cycle 1 concurrently with the

instruction fetch. Cycle 3 is now eliminated making the maximum instruction execution rate, r as

$$r = 1/T*(L-1)$$

which is a performance improvement over the von Neumann processor of

$$L/(L-1)$$

This is not a large improvement if the processor takes a large number of cycles to execute each instruction. Therefore, for processors that take a large (>3) number of cycles to execute each instruction, the advantages of the Harvard architecture are not realised, e.g. for a von Neumann processor that takes five cycles to execute, adopting the Harvard architecture will mean that the processor circuitry will only be active 25% of the time, which is not a great difference. (Note that this assumes a register-memory operation. Memory-memory operations will obviously take longer.)

This inefficiency has led computer architects to try to reduce the number of cycles needed for each instruction in order to improve performance. One approach which has been adopted in both von Neumann and Harvard processors is the technique known as pipelining (Section 7.2.1). This can improve a von Neumann processor's instruction rate to a maximum of one instruction every two cycles or to a single cycle when the Harvard architecture is employed.

Some of the differences between the von Neumann and Harvard architectures are not immediately obvious. Since instructions and data share the same memory space in the von Neumann architecture, the instructions and data must have lengths that are equal to or are factors of each other. With the Harvard architecture, program and data spaces are separate, enabling instruction and data lengths to be of whatever size is appropriate. The same also applies to the various registers in the processor. For instance, the program counter must be of sufficient length to address only the instruction memory, the size of the data

memory being determined by other factors.

In this chapter, we mentioned a technique known as pipelining. This will now be discussed.

7.2.1 Pipelining

The fundamental property of a pipeline is that it can decompose a process into dedicated subprocesses as long as the subprocesses are independent of each other. Dasgupta [22] described a pipelined system not as an architecture but as an architectural style. An architectural style is a feature which is exhibited by an architecture.

Processors such as the 80286 and the LM32900 incorporate a pipeline to obtain simultaneous instruction fetch and execute. For example, in the 3-level pipeline of the LM32900 (Harvard architecture), as an instruction is being executed, the next one is being decoded and the one after that is being fetched. This is depicted in Figure 7.3. In many cases, a maximum throughput of one instruction/cycle can be achieved.

	T_1	T_2	T_3	T_4	T_5	T_6
Fetch	1	2	3	4	5	6
Decode		1	2	3	4	5
Execute			1	2	3	4

Figure 7.3 A 3-level pipeline producing a rate of 1 instruction per cycle. The numbers in the boxes represent an instruction at time T_n

Ideally, one would require a pipeline for every independent operation to be performed. However, a serious disadvantage of this is that the processor assumes that the next instruction required for processing is the next one in sequence in the program memory. This is not true in the case of a branch. If a branch is encountered at the execution unit, the instruction in the decoder and the instruction being fetched in the same time interval will be wrong, as control is now being passed to a different part of the program. In this case, the instructions being decoded and fetched have to be discarded and the pipeline must be filled with the new instructions before correct operation can resume. Thus, if a program has a large number of unconditional or conditional branches and the number of pipelines is excessive, the performance of a pipelined processor can degrade rapidly. (The same applies to the start of a program - the pipeline must be filled before the initial result is delivered.) Therefore, the time to perform a series of operations in a pipelined system is

$$t = [\ S + L\ ]*T$$

where S*T is the initial start-up time required to set up the pipeline. The pipeline thus has a maximum instruction rate of

$$r = 1/T$$

which is a factor L increase in performance over the original von Neumann architecture. As we shall see later, for image processing where a large amount of data has to be processed, computer architects have traditionally adopted the Harvard approach and replicated the processor section for maximum throughput.

The next section explores what is meant by the term 'computer architecture' and how the way we view an architecture can influence the design of a computer system.

7.3 WHAT IS AN ARCHITECTURE?

What constitutes a computer architecture has been a point of some debate in computer science. The crux of the problem lies in the complex nature of computers. A convenient way of representing a complex system is by the use of hierarchy. For instance, a computer consists of a processor, memory, control unit and I/O unit. The processor may contain a pipeline, ALU and register file; the memory may contain local data and dual-port memory; and the I/O section may consist of a bus and serial interfaces.

Dasgupta [22] noticed that an architecture can be viewed in different ways. The machine language programmer views a machine from a logical structure and functional capability point of view. He does not need to know the details of the machine's physical components, the logical structure of their interconnections or the nature of the information flow between components, all of which are necessary knowledge for the hardware designer. These two views (known as 'levels' in this context) have been termed exoarchitecture and endoarchitecture respectively [22]. Exoarchitecture is in effect an abstraction of endoarchitecture. One can view the relationship between exoarchitecture and endoarchitecture as a means of "hiding" information.

These two levels will often suffice for a complete abstract specification of a computer system; however, there is a third level that becomes discernible in systems with a writeable control store. This is termed microarchitecture. Many aspects of microarchitecture will also be a part of endoarchitecture (ALU's, local memory, etc.) but endoarchitecture may not include details of the microprogram control unit which are essential knowledge for the microprogrammer. Using such systems, the microprogrammer may create new endoarchitectures and hence new exoarchitectures. We can thus view an endoarchitecture as the

result of implementing a particular microprogram on a microprogrammable machine.

We can therefore conclude that an architecture is an abstraction of the hardware that can be viewed from different levels: at each level (starting from the system as one component and working down), the information revealed becomes more detailed. The next section introduces classification of architectures. Two classifications are presented followed by a discussion.

7.4 CLASSIFICATIONS OF ARCHITECTURES

Flynn [40] based his classification of architectures, not on the structure of the machines but on how the machines relate their data to the instructions being processed. By doing this he produced four classes of machines:

1. SISD - single instruction stream/single data stream. This is the typical von Neumann serial computer (Section 7.2) where there is only one stream of instructions. Examples are: CDC6600 (nonpipelined), CDC7600 (pipelined), AMDAHL 470V/6, PDP-11, VAX, 68000, SIP (Chapter 5), etc.

2. SIMD - single instruction stream /multiple data stream. This is a computer that distributes the same instruction from a single stream to many processors all operating on different data. This class of machine includes most types of array processors, e.g. the ILLIAC IV, ICL DAP, CLIP4 and the LAP (Chapter 5).

3. MISD - multiple instruction stream/single data stream. These systems usually consist of special streaming organisations, e.g. a pipelined processor system. Here, a processor processes the data then passes it on to the next processor for further processing, etc.

Thus, the data is pipelined between processors.

4. MIMD - multiple instruction stream/multiple data stream. As its name asserts, multiple instructions are operating on multiple data streams. An example of this is DIPOD (Section 7.8.1), the Connection Machine (Section 7.8.3) and the Pyramid architecture (Section 7.8.4).

Shore [107] on the other hand, based his classification on how the computer was organised into its constituent parts. From this he produced six different types of organisation as follows:

1. Machine I - is the conventional von Neumann architecture with a single processor unit, a single control unit and a single instruction and data memory unit. This class includes both the pipelined scalar computer, e.g. CDC 7600, and the pipelined vector computer, e.g. CRAY-1.

2. Machine II - is similar to machine I except that the data is fetched as a bit-slice rather than a word-slice and the data is performed on in a bit-serial like fashion, e.g. STARAN and the ICL DAP. The main difference between Machine I and Machine II is that Machine I processes its data: word serial, bit parallel while Machine II processes its data: word parallel, bit-serial.

3. Machine III - is a combination of both Machines I and II. Known as an orthogonal computer, the data memory is organised as a two dimensional array which may be read as either words or bit slices and thus contains both a horizontal and vertical processing unit. It has the advantages of both Machine I and Machines II resulting in a higher throughput; however, the cost is increased significantly since the arithmetic hardware of Machines I and II is needed. Complications also arise because of the need for dual-ported memory.

An example of such a machine is the Sanders Associates OMEN-60 series.

4. Machine IV - this machine is obtained by replicating the processor section and data memory of Machine I. All instructions are issued from a single control unit. There is no communication between the PE's except through the control unit which tends to limit the applicability of the machine, but addition of further PE's is fairly simple. An example is the PEPE [55].

5. Machine V - similar to Machine IV except the PE's are connected to their nearest neighbours. This means that a PE can access its own memory and data from its neighbours. Most array processors (Section 7.6) are examples of machine V.

6. Machine VI - this is called a logic-in-memory array (LIMA) and is an alternative approach of distributing the processor logic throughout the memory. Machine VI ranges from simple associative memories to complex associative processors. The NRL AP is a typical example of a LIMA machine [107].

By comparing Shore's classification with Flynn's (and ignoring the awkward case of the pipelined vector computer), Hockney and Jesshope [55] noticed that Machines II to V are subdivisions of Flynn's SIMD machine while Machine I is equivalent to the SISD machine. Hence, Flynn's classification is rather too broad. Shore's classification allows us to draw a finer line between the parallel and sequential computer. It is often stated that Machine II is a SIMD machine and Machine I is not. In fact, they are both SIMD machines because Machine I processes multiple-bit streams a word-slice at a time, whereas Machine II processes multiple-word streams a bit-slice at a time. If a line between sequential and parallel processors does exist, it will probably

be between Machines III and IV and not between Machines I and II as one would expect [107].

As we can see, either taxonomy does not cover all possible cases. For instance, a pipelined vector computer falls into the same category as a nonpipelined scalar computer. More recent attempts include a nomenclature [8] that aims to bring out the diversity in parallel processor designs such as the incorporation of different levels of parallelism.

7.5 CRITERIA FOR CHOOSING AN ARCHITECTURE

With all of these architectures and architectural styles at his disposal, the designer has to have solid criteria for selecting between them. The choice will ultimately depend on the application and the factors involved. Quite often, it is necessary to optimise a system either on a cost-speed basis or on a performance basis. We will now examine both of these approaches below.

7.5.1 Optimising an Architecture on a Performance Basis

The most useful method for optimising an architecture on a performance basis is to measure the degree to which the processing hardware is being usefully kept busy. An incorrect design could cause processing power to be used inefficiently. For instance, it is often implied that if a parallel processor can be programmed for an application, then a throughput higher than a sequential processor will result. This is not always the case, especially if the parallel processor consisting of N processors was writing to a single data word from memory. This would result in one useful processor and N-1 idle processors. The sequential processor in this case would be more suitable for the application since it can perform a single operation

using less processor hardware. However, a parallel processor will usually be superior to a sequential processor when the application can be fully parallelised and the execution time must be much less than that possible by an optimally programmed sequential processor.

7.5.2 Optimising an Architecture on a Cost-Speed Basis

Optimising an architecture on a cost-speed (cost of hardware-speed in software) basis will here be taken to mean distinguishing between those parts of a system that should be implemented in hardware and those that should be implemented in software, taking into account cost and speed factors. One extreme situation is when algorithms are too slow for the application; another is when the total cost to implement the algorithm in hardware is too great.

Suppose now that we are considering whether to implement a given algorithmic function in hardware to make it run faster. We need to obtain guidelines to decide whether the hardware can be justified. As an example, consider using hardware accelerator modules to speed up the O-ring algorithm presented in Chapter 3. The times (in milliseconds) are derived from those given in Section 3.3.5; with the absence of actual costs of the hardware modules, notional costs will be used for this example.

For each task in the algorithm, the procedure adopted by Davies and Johnstone [29] was to tabulate the execution time in software (t) and the corresponding hardware implementation cost (c). From this a cost-time ratio (c/t) can be derived for each task (Table 7.1). This figure gives us the cost per unit time, e.g. pounds/millisecond.

task	time (ms)	cost (£)	c/t ratio (£/ms)
1. Find Centres	2500	5200	2.1
2. Sort List	30	2000	66.7
3. Find Centres	20	1500	75.0
4. Median filter	20	2000	100.0
5. Radial Histogram	270	2500	9.3
	2840	13200	

Table 7.1 Breakdown of the O-ring algorithm and its c/t ratio for each task

It is fairly clear that those tasks which should be implemented first are those with a low c/t ratio, since this helps to give low cost complete with high speed. On the other hand, those parts with particularly high c/t ratios (the threshold depends on the application) may quickly be eliminated for the purpose of hardware implementation. A high c/t ratio may arise either because the task is performed quite fast initially or because the circuitry required is relatively expensive for the functionality it achieves (or else because of a combination of these factors).

Next, it is necessary to find where best to stop implementing these tasks in hardware. We now define C,T as being the running totals of the c and t figures down the list (Table 7.2): the aim will be to minimise the cost-speed tradeoff product, C*T. According to this criterion, the best tradeoff in this case arises when all tasks are implemented in hardware. In spite of optimising a cost-speed tradeoff for a system in this way, it may nonetheless be necessary to tailor a system for a specific speed or cost constraint. In this case, those parts of the (C,T) list down to the specific cost or speed limit should be implemented. For example, consider the system above with a £10,000 budget. Those parts up to (and including) the £9700 row should be

implemented in hardware, implying an O-ring execution time of 80ms.

task	time (ms)	cost (£)	C £	T (ms)	C*T ($x10^6$) £-ms
		2000	2000[1]	2850[2]	5.7
1	2500	5200	7200	350	2.5
5	270	2500	9700	80	0.8
2	30	2000	11700	50	0.6
3	20	1500	13200	30	0.4
4	20	2000	15200	10	0.2

Table 7.2 C*T table for the O-ring algorithm

We shall now discuss the use of the SIMD machine in image processing. Following this, architectures that have been employed in image processing will be reviewed.

7.6 SIMD MACHINES FOR IMAGE PROCESSING

A large volume of data processing is needed for image processing but because of the constraints of technology at the time, designers have resorted to the use of multiple processors for increased data throughput. Combining multiple processors on a single bus usually incurs major drawbacks:

1. The data bus can become saturated for as few as ten processors, hence producing a data bottleneck. The onset of this bottleneck is governed by the bandwidth of the bus, and in any case is highly data dependent.

---

[1] An initial cost of £2000 has been included to cover the cost of the basic system, e.g. host computer, backplanes, power supply, etc.

[2] 10ms has been added to the original times in Table 7.1. This implies that if all tasks were implemented in hardware, the algorithm would execute in 10ms.

2. Partitioning and scheduling of tasks into subtasks with similar execution times is difficult and is sometimes impossible. This can lead to processors lying idle if its task is finished early, making inefficient use of processing power.

3. Because of this, expansion to an arbitrary number of processors and maintaining efficient use of processing power is generally difficult.

The bus organisation for interprocessor communication is thus unsuited for image processing where the problems are varied and, at times, unpredictable. Ideally, one would like a performance increase proportional to the number of processors in the system. Computer architects have traditionally gone for the SIMD array processor approach for increased data throughput. This has the particular advantage that the data format (i.e. the image) maps onto the processor array. This suggests that the performance of the machine will rise linearly with the number of processors. However, Minsky's conjecture [40] states that the gain in performance for a SIMD machine is roughly proportional to $\log_2 N$ and not N as one would hope. This was interpreted as being due predominantly to the way a SIMD processor executes its branch instructions, although this must inevitably be data dependent.

In general, array processors consist of an NxN array of processors (usually referred to as processing elements (PE's)), each with its own local memory as depicted in Figure 7.4. A central controller broadcasts instructions to all PE's simultaneously: hence all PE's execute their instructions in lockstep. Each processor is usually connected to either its four or its eight nearest neighbours. When used in image processing, each PE often represents a single pixel in the image. The pixel is loaded into the PE's local memory, operated on, then transferred either back to local memory for further processing or to an

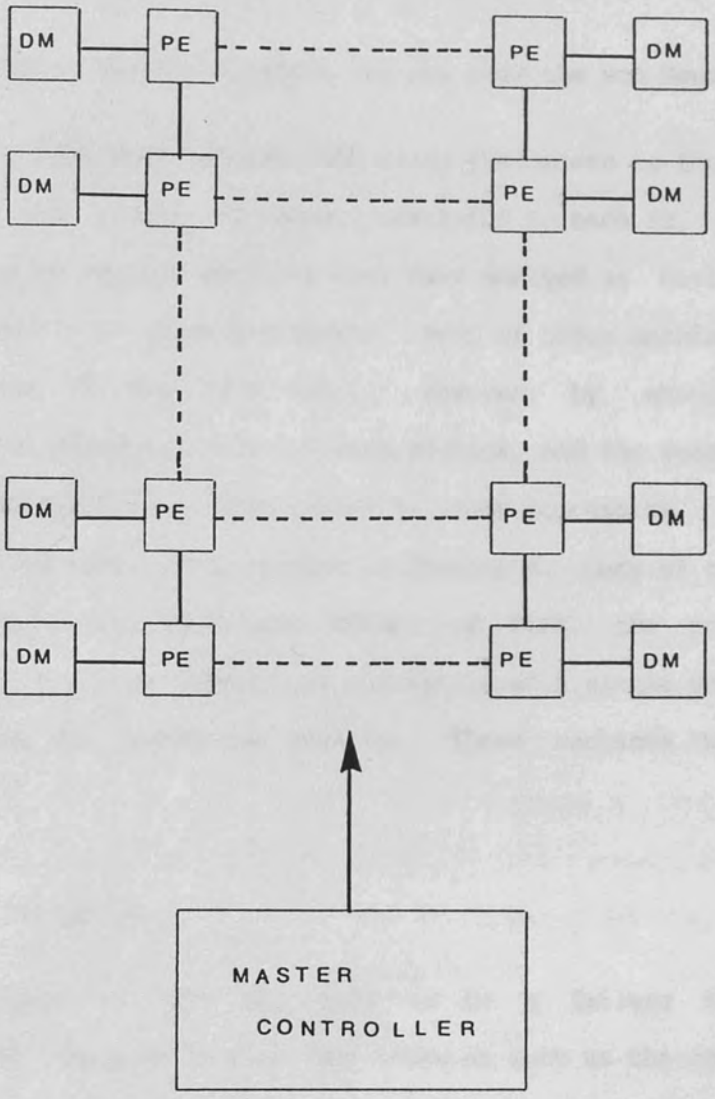


Figure 7.4 Typical representation of a SIMD machine consisting on an NxN array of P.E's, each with its own local memory and receiving the same instructions

I/O device such as a frame store. Following the notations of Section 7.2, the instruction throughput is now

$$r = N/(L*T)$$

where N is the number of processors in the system. For the majority of array processors, L=1, i.e. the Harvard approach has been adopted with a pipelined instruction fetch. Thus, the maximum instruction rate for an array processor is N/T. This has a performance increase of N over the

single processor Harvard approach and N*L over the von Neumann approach.

A main difference between SIMD array processors is the size of the array and the amount of memory available to each PE. The following sections review several machines that have emerged as having the most prominent effect in image processing. Most of these machines tend to be in the form of the SIMD array. However, by showing specific architectural idiosyncracies for each machine, and the reasons why these were implemented and how they relate to image processing, we can then build on this work. This is done in Chapter 8. Many of these machines are now old; however, with the advent of VLSI, the possibility of fabricating a large number of processors on a single chip to produce larger arrays has become a reality. These machines will also be reviewed.

7.6.1 The ILLIAC IV

The ILLIAC IV [37] was said to be a failure in its time (circa. 1970) because it cost four times as much as the contract figure and did not come within a factor 10 of its original proposed performance: nevertheless, it had a profound and lasting influence on architectures. The architecture was too advanced for the technology at the time; however, what emerged from the experience was the significant advancement and finally the introduction of ECL chips and specialist CAD tools to cope with the 15-layer circuit boards that were required in the PE section [6].

The basic layout was a square array of 8x8 PE's (each PE was a 64-bit PE) and a control unit (CU). The original specification was an array of 16x16 PE's, each with 2Kx64 RAM (120ns access time) for local data storage. A system clock cycle time of 40ns gave it an estimated peak processing rate of 1GFlops/s. The system was to be composed of 4

8x8 quadrants; however, only one quadrant was ever built and the clock cycle time was eventually increased to 80ns, only ever producing a peak rate of 50Mflop/s (one floating point operation every 240ns) and a typical rate of 15Mflops/s. This still compares favourably with the CDC7600 which had a peak processing rate of 10MFlops/s and a typical processing rate of 5MFlops/s.

Although the ILLIAC IV was essentially a 64-bit machine, its most common mode was for 32-bit floating point arithmetic. It was capable of performing 64-bit and 32-bit floating point operations but in the 32-bit mode, two floating point operations could occur concurrently in each PE. It was thus capable of performing 32-bit arithmetic on vectors of length 128. Each PE had an enable/disable bit which controlled the instruction execution of that PE, and 4 64-bit registers: A (accumulator), B (operand register), R (multiplicand and data routing register) and S (general purpose). The PE section was developed so that arrays of PE's could be partitioned into subprocessors of 64x1, 32x2 or 8x8 under software control.

The main purpose of the CU depicted in Figure 7.5 was to control and decode instruction streams and to generate and broadcast those instructions that were common to all processors. One of the most prominent features of the ILLIAC IV was the instruction fetch/execute sequence. Each instruction is 32-bits in length and is used either by the CU (for simple single operations) or by the PE array (for more complex operations). Initially, each instruction entered the instruction buffer. As the control advanced, each instruction was sent to the advanced instruction station (ADVAST) unit where it was decoded from a repertoire of 260 operational instructions into a set of microsequences chosen from the 720 micro-instructions available. If it was an instruction local to the CU then it was executed: otherwise, in the case of a PE instruction, ADVAST constructed the necessary address

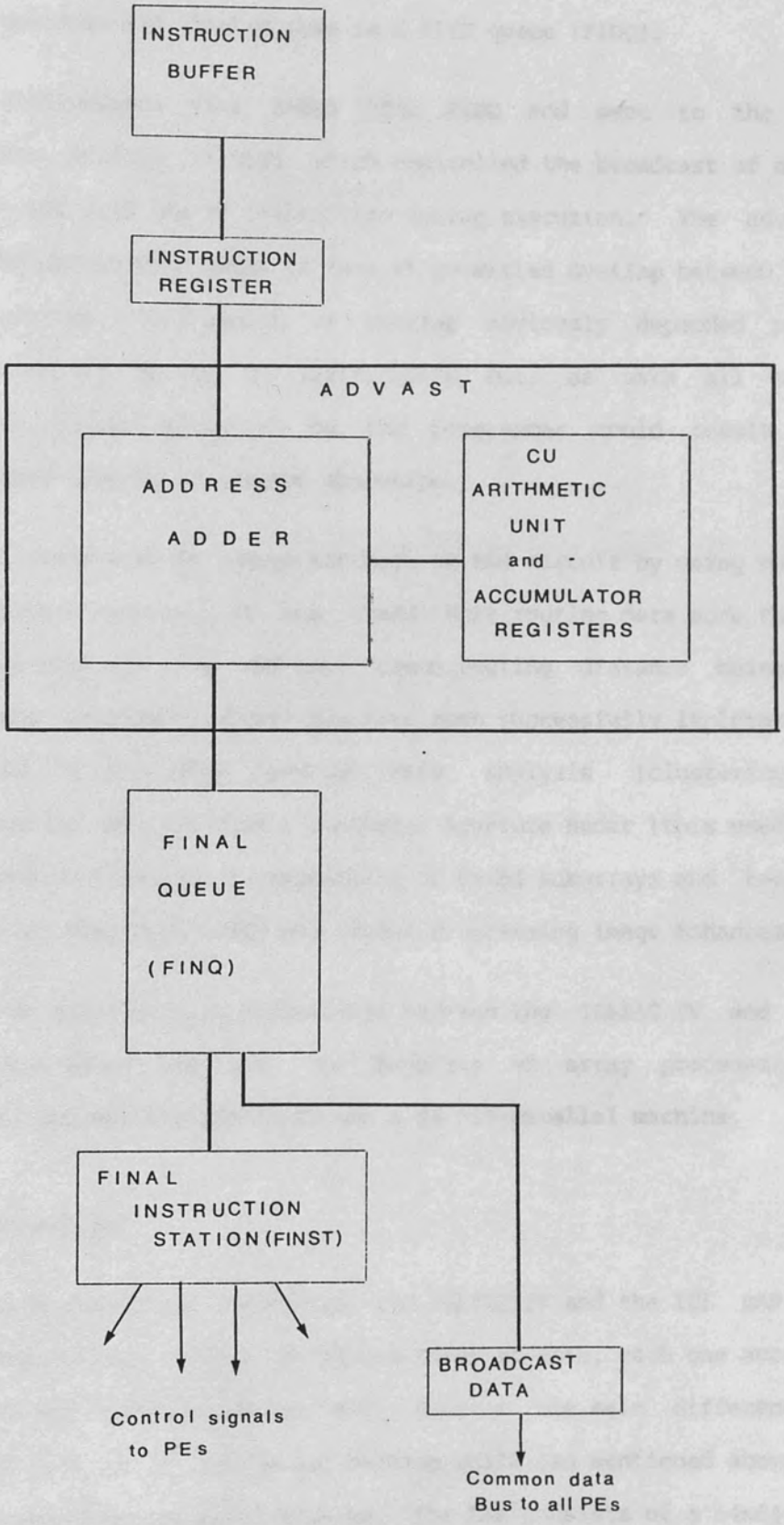


Figure 7.5 The ILLIAC IV Control unit

or data operands and stacked them in a FIFO queue (FINQ).

PE instructions were taken from FINQ and sent to the final instruction station (FINST) which controlled the broadcast of address and data, and held the PE instruction during execution. The advantage of the PE instruction queue is that it permitted overlap between CU and PE instructions. The amount of overlap obviously depended on the distribution of PE to CU instructions but, as with all overlap strategies, careful attention by the programmer could result in a considerable speedup of program execution.

Data could also be routed anywhere in the circuit by using multiple instructions; however, it was found that routing data more than two processors away was rare, the most common routing distance being one. Many image processing algorithms have been successfully implemented on the ILLIAC IV including Landstat data analysis (clustering and classification of the data), Synthetic Aperture Radar (this used range and azimuth correlation, transposition of 64x64 subarrays and texture), Fast Fourier Transform (FFT) and Linear Programming Image Enhancement.

One of the startling differences between the ILLIAC IV and other array processors was that the majority of array processors are bit-serial whereas the ILLIAC IV was a 64 bit-parallel machine.

7.6.2 The ICL DAP

As with most array processors, the ILLIAC IV and the ICL DAP have many similarities, namely a square array of PE's, each one accepting control from a common broadcast unit. However, one main difference is that the DAP is a bit-serial machine while (as mentioned above) the ILLIAC IV was a bit-parallel machine. The DAP consists of a 64x64 array made up on 256 boards, each board containing a 4x4 array of PE's. The whole 64x64 array is controlled by an ICL 2900 host mainframe which

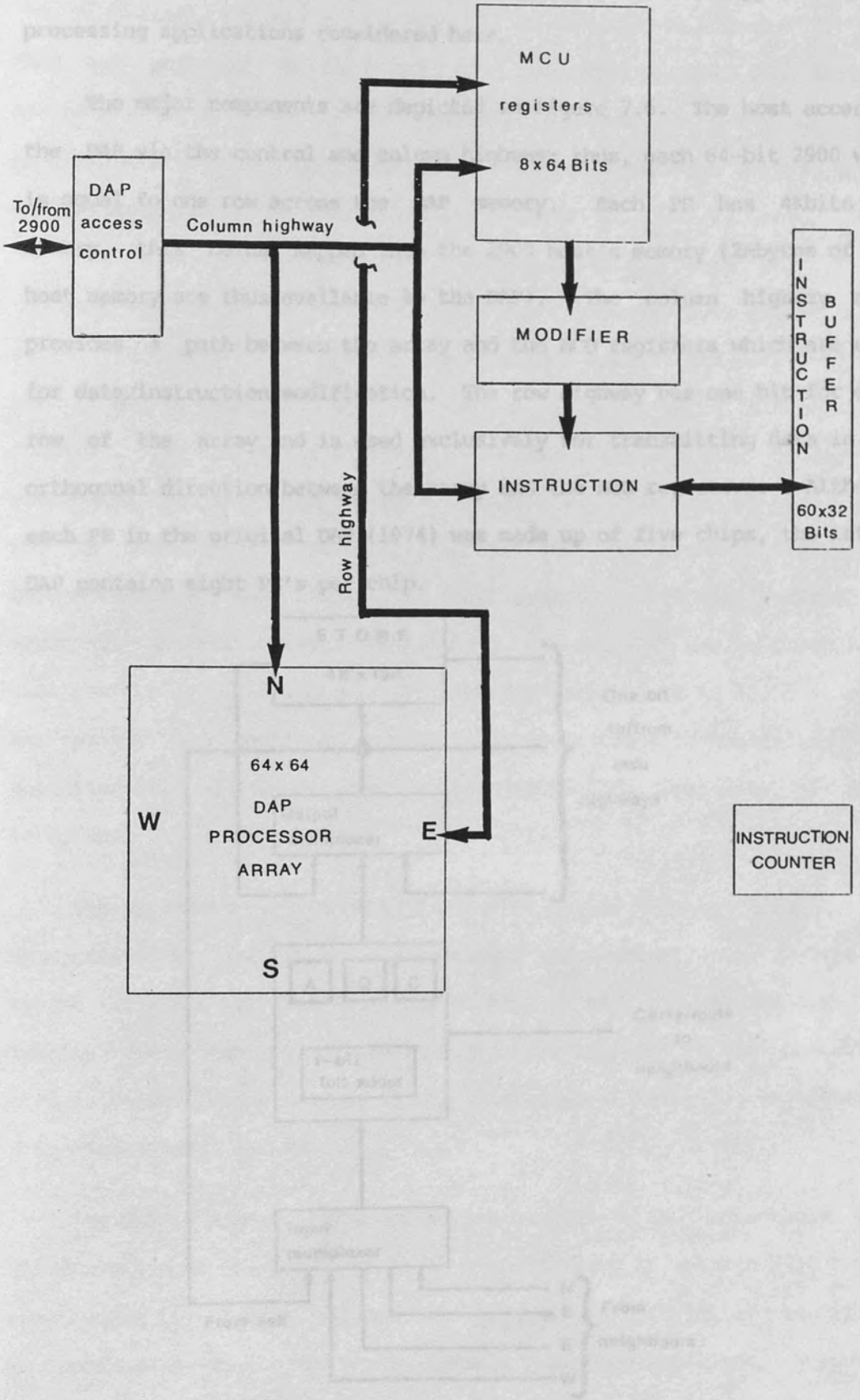


Figure 7.6 The major components of the DAP

consequently makes the system too expensive for the types of image processing applications considered here.

The major components are depicted in Figure 7.6. The host accesses the DAP via the control and column highway; thus, each 64-bit 2900 word is equal to one row across the DAP memory. Each PE has 4Kbits of memory, this being mapped into the 2900 host's memory (2Mbytes of the host memory are thus available to the DAP). The column highway also provides a path between the array and the MCU registers which are used for data/instruction modification. The row highway has one bit for each row of the array and is used exclusively for transmitting data in the orthogonal direction between the array and the MCU registers. Although each PE in the original DAP (1974) was made up of five chips, the latest DAP contains eight PE's per chip.

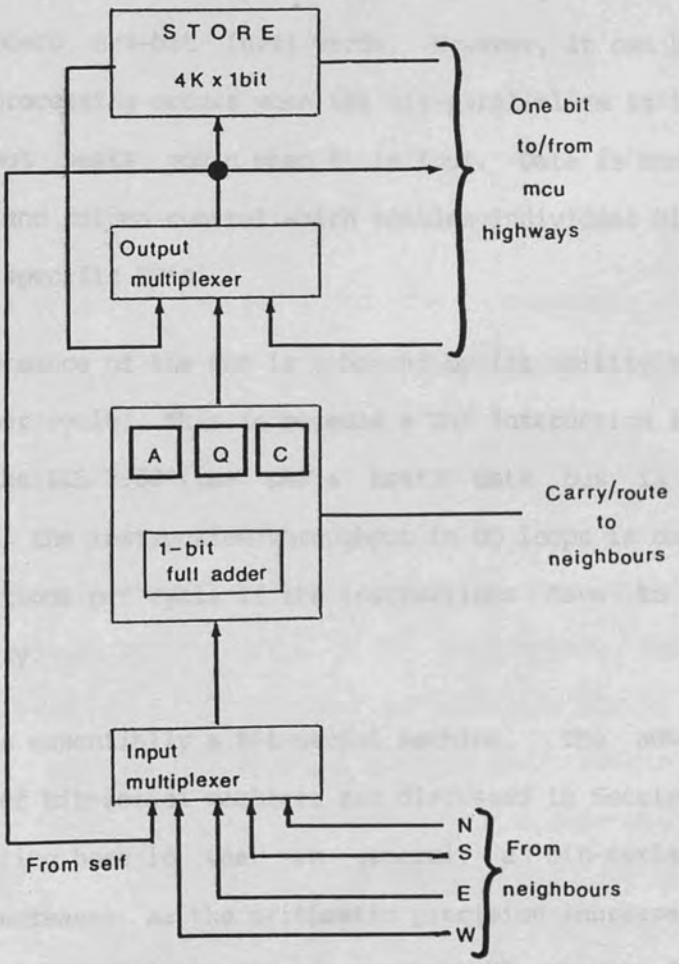


Figure 7.7 A DAP processing element

A schematic diagram of a single DAP PE is shown in Figure 7.7. The PE's are arranged so they are all connected to their four nearest neighbours: N, S, E and W. Each has three registers: A, Q and C. The programmable activity enable register (the A reg) prevents certain store instructions from writing into the store unless this bit is set. This has the effect of selectively enabling/disabling each PE. The Q register represents the accumulator and the C register acts as the carry store. A feature of the DAP (as with many array processors [55]) is that it may be configured as a bit-serial machine or a bit-parallel machine. To act as a bit-parallel machine, the carry bit (C reg) is propagated (rippled) down the PE's in that row. Thus, 64 64-bit words may be processed in parallel. Since four positions of carry are guaranteed in a single clock cycle [55], several cycles are required in order to process n/4-bit (n>4) words. However, it can be shown [55] that maximum processing occurs when the bit-parallelism is 1, i.e. pure bit-serial, but peaks again when it is four. Data is controlled by a selective row and column control which enables individual bits of data to be sent to specific PE's.

The performance of the DAP is enhanced by its ability to fetch two instructions per cycle. This is because a DAP instruction is 32-bits in length while the ICL 2900 (the DAP's host) data bus is 64-bits in length. Thus, the instruction throughput in DO loops is one per cycle, or 1.5 instructions per cycle if the instructions have to be fetched from main memory.

The DAP is essentially a bit-serial machine. The advantages and disadvantages of bit-serial machines are discussed in Section 7.10 but a point worth noting here is that in general, a bit-serial machine's performance decreases as the arithmetic precision increases. However, Reddaway [55] showed that the DAP gives acceptable results for precision as high as 32-bit floating point arithmetic (about 20Mflops/s), this

being due to the high data memory bandwidth, it being 4-6 times faster than a CRAY-1.

7.6.3 The GOODYEAR MPP

The Goodyear Massively Parallel Processor (MPP) [95] was originally designed to process LANDSAT-D satellite images in real-time. In order to do this, a processing rate of greater than 1000 million operations/second is required. The estimated performance is 2-4 Gflops for 8-12 bit integers or 200-400 Mflops for 32-bit floating point arithmetic.

The MPP is modelled on the DAP architecture, the main difference being that it has a 128x128 array of PEs and that the I/O system is configured differently. Whereas the DAP used the 2900 for I/O, each row in the MPP array acts as a 128-bit shift register thus enabling data to flow left to right across the columns of the array. Each MPP PE has 1Kbit of RAM for local use and six registers (A,B,C,P,G,S).

The main addition is the provision of a programmable shift register S. This was added to improve the multiplication time which is implemented using shifts and adds. Each PE in the MPP has a cycle time of 100ns, half that of the DAP, yet it can perform 32-bit floating point multiplication 16 times faster. The figure one would expect, considering the array size (4 times larger) and clock cycle time, is 8. The additional factor of two is due to the programmable shift register in each PE. The data connections to a single MPP PE are shown in Figure 7.8 where each PE is connected to its four neighbouring PE's. This layout is typical of most SIMD machines. Another major difference between the DAP and the MPP is that the MPP does not have selective row or column broadcast facilities.

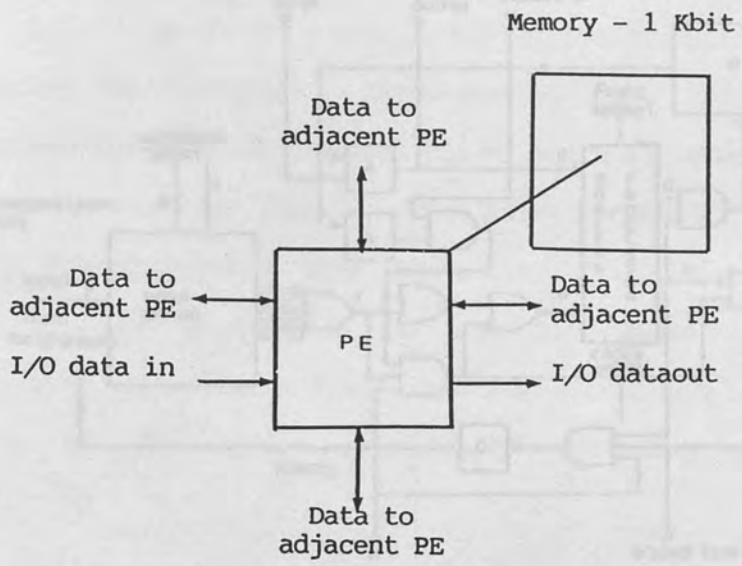


Figure 7.8 Connections to an MPP PE

The MPP is currently being implemented in CMOS VLSI but, even though half of the PE logic is required by the programmable shift register, it is still possible to incorporate eight PE's per chip (c.f. 5 chips/PE on the original DAP and eight PE's per chip on the latest version).

We shall now review SIMD machines that have been specifically developed for image processing use.

7.6.4 CLIP4

The Cellular Logic Image Processor (CLIP4) has a 96x96 array of processing elements controlled by a single broadcast unit. Each integrated circuit (the CLIP4B chip) contains eight PE's controlled by a 4-phase clock running at 1MHz, each with 32 bits of RAM. Unlike the ILLIAC IV and the DAP, CLIP4 was specifically tailored for image processing, so each PE has direct access to its eight neighbouring PE's rather than only four. A PE section of the CLIP4 is shown in

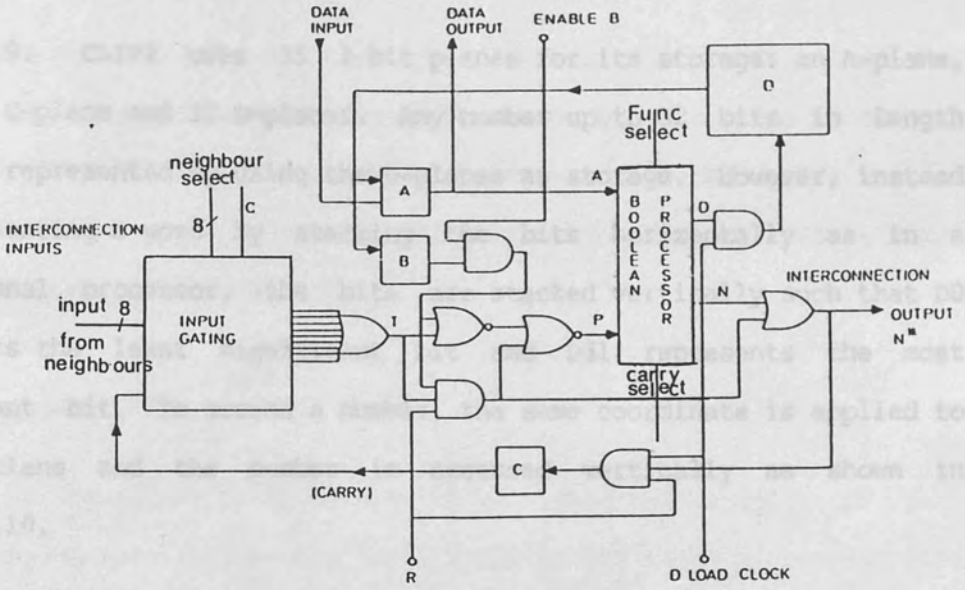


Figure 7.9 A CLIP4 PE

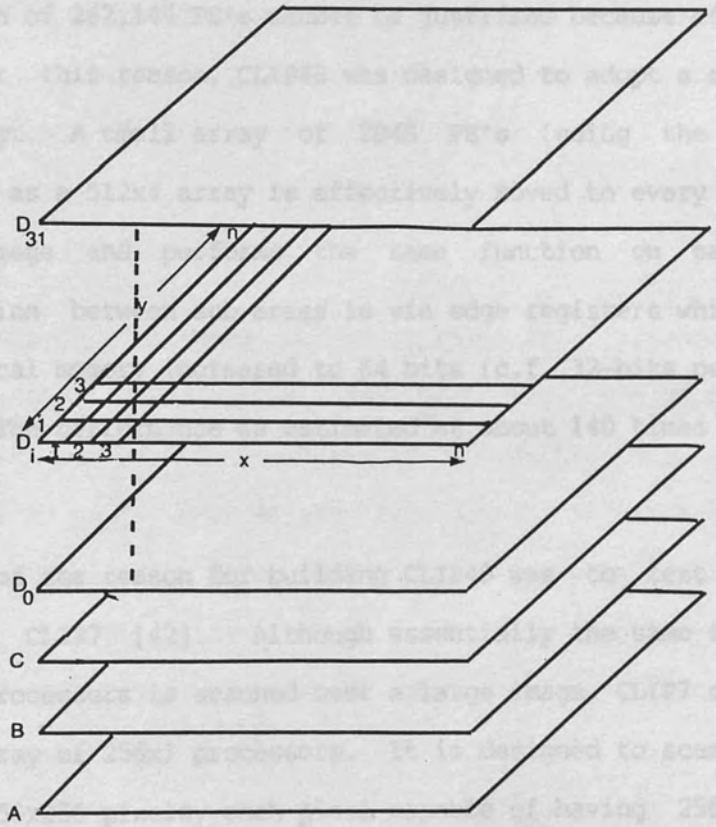


Figure 7.10 Data storage in the CLIP4 array

Figure 7.9. CLIP4 uses 35 1-bit planes for its storage: an A-plane, B-plane, C-plane and 32 D-planes. Any number up to 32 bits in length can be represented by using the D-planes as storage. However, instead of representing a word by stacking the bits horizontally as in a conventional processor, the bits are stacked vertically such that D0 represents the least significant bit and D31 represents the most significant bit. To access a number, the same coordinate is applied to every D-plane and the number is accessed vertically as shown in Figure 7.10.

7.6.5 CLIP4S and CLIP7

CLIP4 [52] was designed to operate on a 96x96 image with 1 PE/pixel. Many applications require a 512x512 image but the realisation of 262,144 PE's cannot be justified because of the enormous cost. For this reason, CLIP4S was designed to adopt a one-dimensional scan concept. A small array of 2048 PE's (using the CLIP4B chip) organised as a 512x4 array is effectively moved to every 512x4 sub-area of the image and performs the same function on each sub-area. Communication between sub-areas is via edge registers while each PE has had its local memory increased to 64 bits (c.f. 32-bits per PE in the CLIP4). The performance is estimated at about 140 times less than that of CLIP4.

Part of the reason for building CLIP4S was to test some of the ideas for CLIP7 [42]. Although essentially the same in that a small array of processors is scanned over a large image, CLIP7 consists of a linear array of 256x1 processors. It is designed to scan over an input image of 256x256 pixels, each pixel capable of having 256 grey levels (CLIP4 only had 64 grey levels). The main difference between CLIP7 and CLIP4S is that each processor is 16-bits wide (as opposed to 1-bit wide) and has a much greater degree of autonomy. CLIP7 is still in the

development stage and should now be nearing completion; but with so many factors affecting performance, it is impossible at this stage to do more than state that "the typical performance expected of the CLIP7 system over its 256x256 pixel data images is similar to that achieved by CLIP4 over 96x96 pixels" [Fountain 42].

7.6.6 The GRID Chip

One of the latest developments for realising the benefits of VLSI and employing parallelism on a large scale in the image processing area is the GEC Rectangular Image and Data (GRID) chip [124]. The concept is the same as the array processors described above in that a master controller broadcasts the same instruction to every PE in the array and each PE executes the same instruction in lockstep. Previous machines such as CLIP and the DAP have demonstrated that very high throughputs could be obtained using a high degree of processor parallelism. However, because of the limitations of MSI logic with which these machines were built, they required cabinets of electronics to hold the large numbers of PE's needed.

The GRID chip has been designed with 64 PE's, arranged as an 8x8 array on a single chip with a 150ns cycle time. The PE's were designed as bit-serial processors consisting of a one-bit ALU and a one-bit communication path. Each PE also includes a 32x1 bit register file and various I/O paths. Two other communication paths exist: the X-bus and the Y-bus. These allow the PE addressing circuitry to selectively broadcast data to every PE on every row and column - this is useful for matrix transpositions.

As with the CLIP, GRID has access to its eight nearest neighbours in its local 3x3 neighbourhood via the nearest neighbour select circuitry (NNS). A complete 64x64 PE array can be mounted on 16 cards

(each card has a 16x16 array of PEs) and can achieve a throughput in the order of 10^9 instructions/second. See Section 7.12 for the implementation of image processing algorithms on GRID.

7.6.7 Linear Array Processors

Linear array processors are similar to NxN arrays described above except that they consist of a one-dimensional array of processors (Nx1). In this case, a linear array can be regarded as a compression of a two-dimensional array of processors, all processors in the one-dimensional array executing their instructions in lockstep. A linear array of 256 PE's can process a line of image data at a time while accepting common instructions from a master controller. Linear arrays have the ability to manipulate symbolic information which is of a similar form and are hence useful for such operations as linear database searching, matching and correlation of unusual data items.

Fountain [43] has proposed a 256x1 processor array with a processor of 32-bit complexity. The CLIP7 [42] array processor is based along these lines with each processor having a 16-bit word length. Here, each processor also has a greater degree of autonomy. Plummer of the National Physical Laboratory (NPL) has developed the "Linear Array Processor" (Chapter 6) which is a bit-serial processor composed of a 256x1 array of cost-effective, readily available 1-bit processors. The outcome of this is that the cost is kept low while the processing power is maintained because of the advantages bit-serial processors have to offer. The LAP is therefore efficient for picture point-picture point processing but, as with most bit-serial devices, suffers when high precision numbers are required.

SLAP [39] is a 512x1 scan line array processor based on CMOS VLSI technology. Each chip contains four PE's, each PE having an ALU, 32 registers, an instruction decoder and a high-speed video shift register for video data. Input to the shift register is via an 8-bit input bus, the output of which feeds the ALU and register file in each PE. The ALU produces 16-24 results which are available to both PE's connected to it in the linear array. All processing occurs in lockstep as in a SIMD machine. In real-time mode, video data is shifted through the array by the shift registers, then latched in after a line has been loaded, independently of the processor. Thus, processing can be carried out on one line while the next line is being loaded.

Space precludes further descriptions of LAPs; however, more exist and can be readily found in the literature [35].

7.7 PIPELINED ARRAY PROCESSORS

Another concept frequently used is in operations where the same instruction has to be executed on all data. This differs from the previous types of array processors in that several ALU's are pipelined as depicted in Figure 7.11. Each ALU in the chain is loaded with a specific instruction. Data is entered into the first ALU (ALU1) on every clock cycle. Simultaneously, the output of every ALU in the chain is sent to the next ALU. This process is repeated for all ALU's in the chain. The advantage of this is that complex instructions that consist of N internal steps can be executed in a single cycle with N processors. Thus, in Figure 7.11, the operation ((A*B)/C-4)*56 is executed in a single cycle; however, one must ensure that the data appears at the right place at the right time, i.e. at ALU2 in Figure 7.11, C must appear at the input on the following clock cycle data was entered into ALU1. One could also consider that each processor could contain a part of an algorithm - this is discussed more fully in Section 8.2.

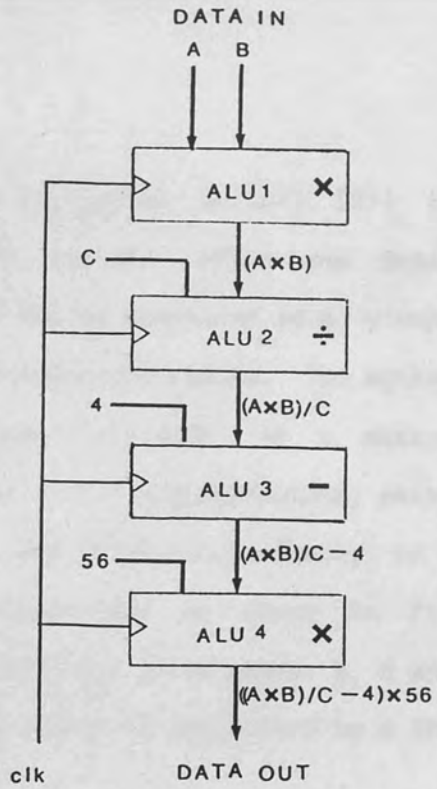


Figure 7.11 An ALU pipelined system

7.8 MIMD MACHINES

MIMD (Multiple Instruction, Multiple Data) machines are classed as multiprocessor machines. Each processor in the system operates on a different set of data with a set of instructions (tasks). The problem with MIMD machines is finding the optimum processor connection topology for the application. In the simplest case, each processor runs the same program on a different set of data. In a more complex form, each processor executes a different task on a different set of data. However, complications arise here when scheduling of tasks needs to be carried out effectively. Inefficient scheduling could mean that many tasks lie idle while waiting for other tasks to complete. Various MIMD architectures have been proposed for use in image processing, several of which will be discussed here.

7.8.1 DIPOD

DIPOD (Dedicated Image Processing Device) [97] is a high-speed microcoded bit-slice MIMD system. DIPOD was designed for image processing algorithms which can be expressed as a group of tasks and performed in parallel and pipelined stages. The system consists of 15 FENS (each FEN (FIFTH Execution Node) is a microcoded bit-slice processor) loosely coupled over a high-bandwidth, packet organised bus where each processor on the bus works independently on its own task. The architecture of a single FEN is shown in Figure 7.12. The functional modules are accessed via three buses: A, B and Y. The code is downloaded over the Y-bus which is controlled by a 68000 host running UNIX.

The bit-slice approach was chosen because is shows a significant speed improvement over conventional microprocessors. The hardware multiplier (MAC) is independently controlled by the microcode; therefore, a significant degree of internal parallelism can occur when performing MAC operations in data-independent operations. For instance, the Sobel operator involves a '*2' operation so the data fetch from memory and the MAC operation can be performed simultaneously.

This approach to parallelism differs from the systolic array approach of WARP (Section 7.9) in that each processor executes different instructions on different sets of data whereas WARP executes the same instructions on different sets of data, although in both cases, each processor is autonomous. (Autonomous is here taken to mean the ability of a processor to execute its own instructions on its own set of data, independently of other processors.) The advantage of the MIMD approach over the SIMD approach is the ability to handle sequential operations. Each FEN in the system is a sequential processor executing a sequential task. DIPOD gains its speed from partitioning an algorithm into several

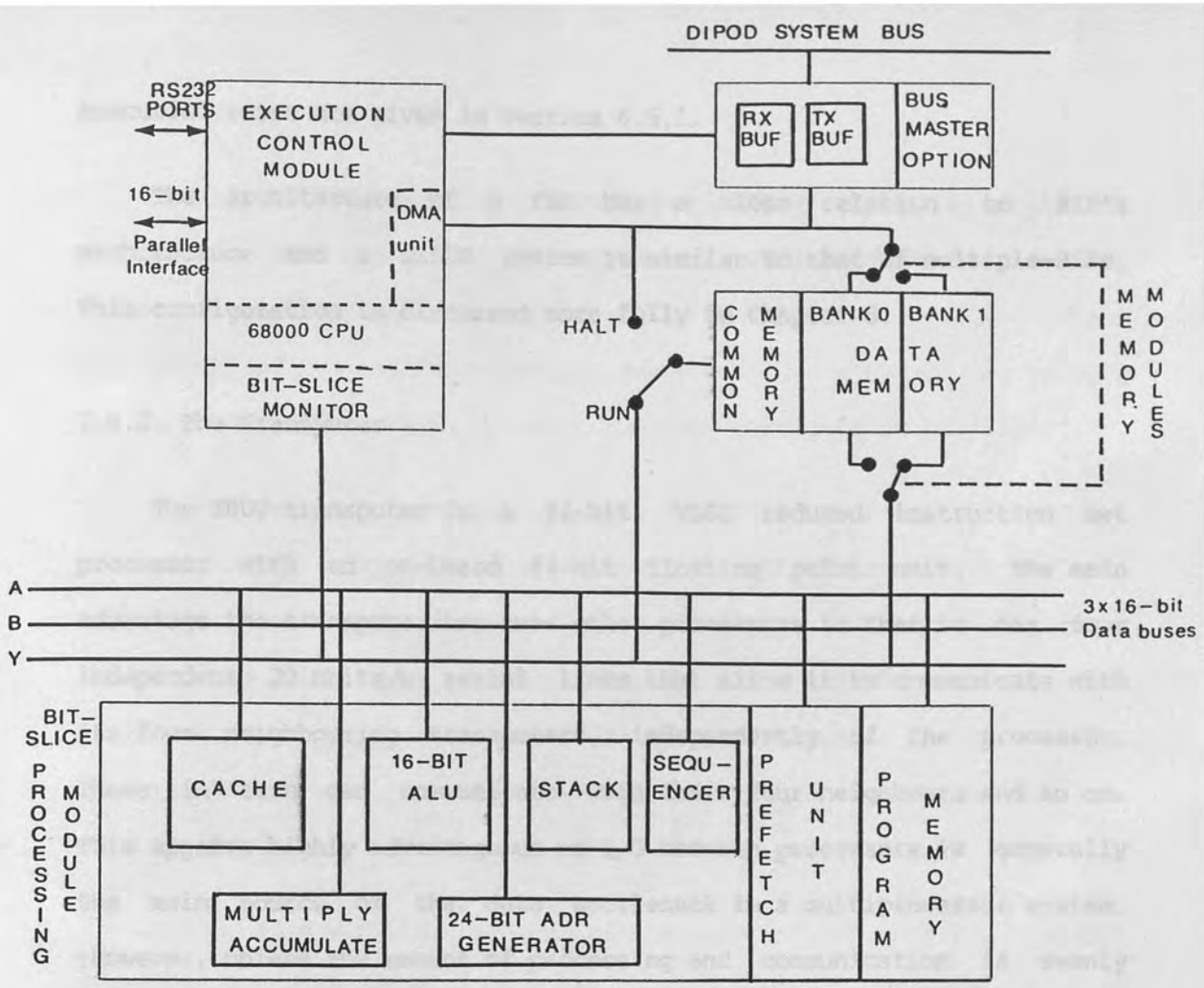


Figure 7.12 Schematic diagram of a DIPOD FEN

tasks.

Bit-slice processors are not known for their ease in programming so for this reason, a language FIFTH (as mentioned in Chapter 5) has been developed which allows an algorithm development environment to be implemented in which modifications can be easily made. (Note that the language is called FIFTH as it represents a fifth generation language.) As with many microcoded machines, all instructions are translated into short sequences of bit-slice microcode. Parallelism is stated explicitly in the code and is downloaded to the FEN's. Typical

execution rates are given in Section 6.5.1.

The architecture of a FEN has a close relation to SIP's architecture and a DIPOD system is similar to that of multiple-SIPs. This configuration is discussed more fully in Chapter 8.

7.8.2 The Transputer

The T800 transputer is a 32-bit, VLSI reduced instruction set processor with an on-board 64-bit floating point unit. The main advantage the transputer has over other processors is that it has four independent 20 Mbits/s serial links that allow it to communicate with its four neighbouring transputers, independently of the processor. These in turn can communicate with their four neighbours and so on. This appears highly advantageous as I/O between processors is generally the main source of the data bottleneck in a multiprocessor system. (However, unless the amount of processing and communication is evenly balanced, I/O may still be a problem.) In order to allow parallelism to be exploited, a parallel language Occam was developed in conjunction with the transputer.

Often, it is difficult to partition an image processing algorithm into independent tasks. Thus, although the transputer appears to be an optimum solution for parallel processing, it still does not solve the problem of matching the task to the architecture. One solution would be to partition the image such that each transputer executes the same program independently of the others on its section of image - this is analysed more fully in Chapter 8. However, a point worth mentioning is that the transputer is a device that is used more at the implementation level rather than at the architectural level, i.e. it is a useful PE to use with which to implement architectures rather than an architecture per se.

7.8.3 The Hypercube

The hypercube scheme [105] (sometimes referred to as the n-cube or the cosmic cube) has a powerful interconnection feature that allows any number, N ($N=2^n$) of processors (nodes), to be connected together. These are organised so that the maximum number of nodes a signal has to pass to reach any other node is n, where n is the dimension of the cube.

An example of a hypercube is the "Thinking Machines' Connection Machine" (TMCM). This has a large array of 1-bit PEs each with a single ALU, 16 registers and 4-Kbytes of external memory. Each PE operates by reading two bits from external memory and one flag, combining them according to a specified operation and then producing a two-bit result. This is then written to external memory producing a total execution time of 3 clock cycles.

The problem with the topology of the hypercube is that the fan-out from the microprocessor needs to be n. Thus, with increasing number of nodes, the hardware becomes increasingly complex. Pease [90] suggested a switching network where every pair of processors is connected to a two-state switch. The switch can either be set as "straight through" enabling the signal to pass directly opposite or "crossed" enabling the signal to cross over to a similar switch. Figure 7.13 depicts this concept for a 4-dimensional cube (16 processors). If the cube is of n-dimensions then the nesting level of the switches is n. The final output is fed back into the destination processor. This layout is called the "indirect" binary n-cube because the nodes are not connected according to the topology of the binary n-cube.

Another feature of the n-cube is that, unlike the tree or shuffle-exchange structures [52], no node in the n-cube plays a particular role; thus, the n-cube can adopt to many topologies easily. Pease also suggested the switching control system shown in Figure 7.14.

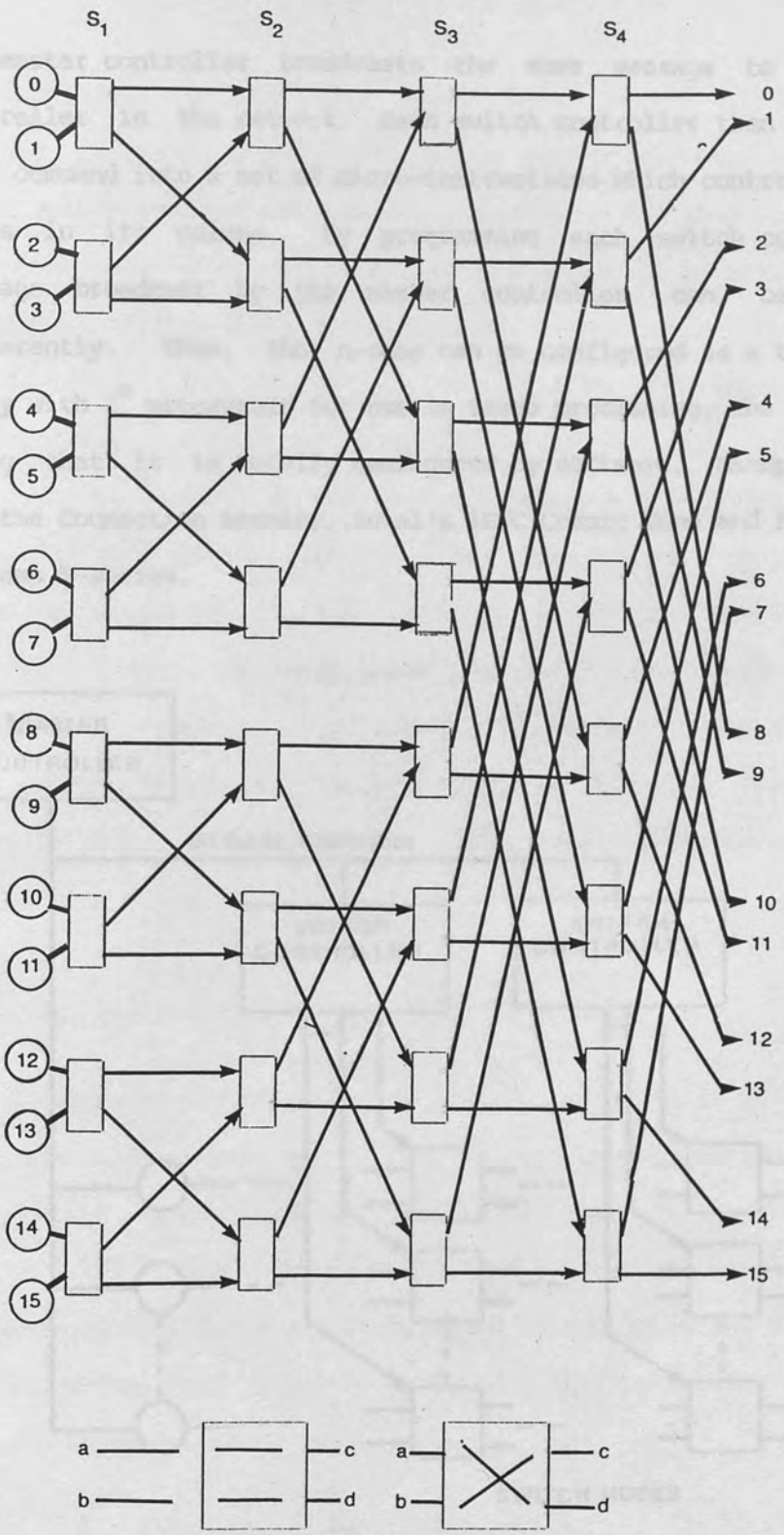


Figure 7.13 The indirect binary 4-cube array and the switch node (a) direct connection and (b) crossed connection

The master controller broadcasts the same message to every switch controller in the network. Each switch controller then interprets the same command into a set of micro-instructions which controls the switch nodes in its column. By programming each switch controller, each message broadcast by the master controller can be interpreted differently. Thus, the n-cube can be configured as a two-dimensional array with 2^n processors for use in image processing, the main advantage being that it is totally configured by software. Examples of n-cubes are the Connection machine, Intel's iPSC Cosmic Cube and Floating Point Systems T-series.

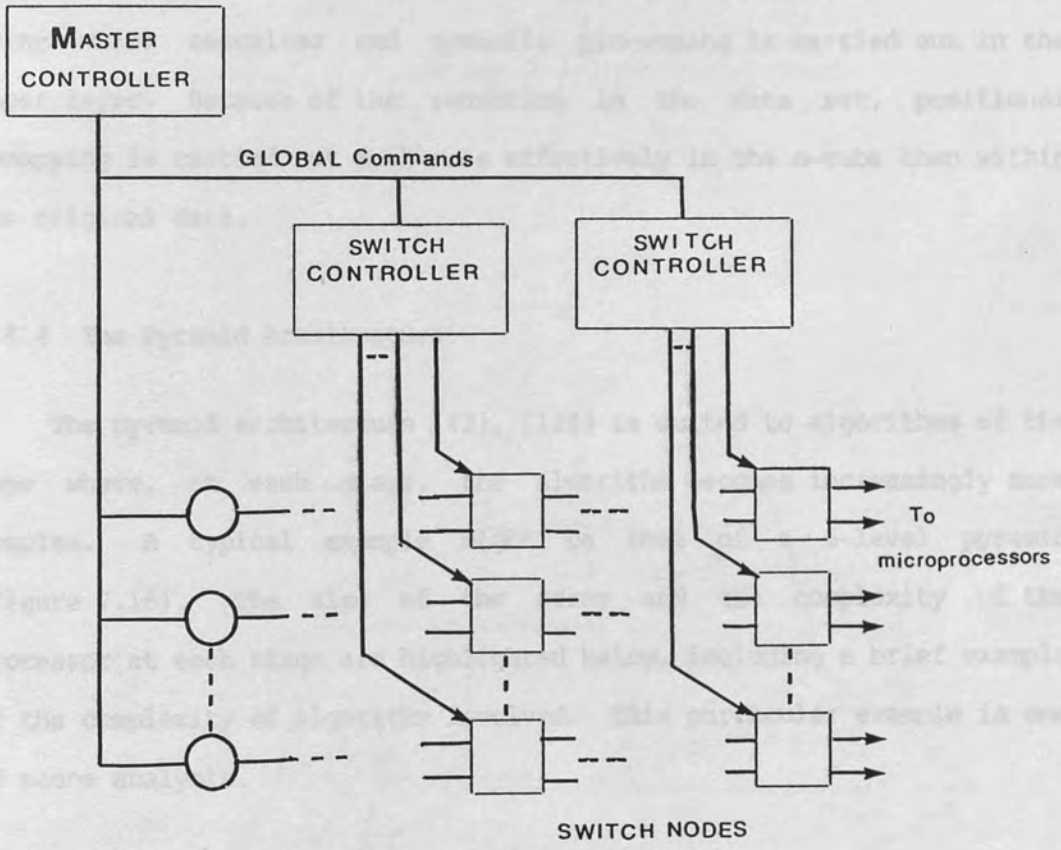


Figure 7.14 Switching control system for the indirect binary cube

Fountain [43] suggested a two-level N-cube architecture for tracking purposes. This problem is efficiently carried out with the binary n-cube because image data can easily be rescaled by a factor of two in very few operations. The scene consists of an object whose range is changing through a series of images. This example requires the use of low-level image processing, data shifting and symbolic processing. The architecture chosen for this problem is depicted in Figure 7.15. The array of elements on the lower layer are a two-dimensional array of bit-serial processors. Each 4x4 array communicates with the upper layer which is connected in the form of an n-cube consisting of 8-bit processors. Thus, low-level image processing is achieved in the lower array while rescaling and symbolic processing is carried out in the upper layer. Because of the reduction in the data set, positional remapping is carried out much more effectively in the n-cube than within the original data.

7.8.4 The Pyramid Architecture

The pyramid architecture [43], [120] is suited to algorithms of the type where, at each stage, the algorithm becomes increasingly more complex. A typical example might be that of a 5-level pyramid (Figure 7.16). The size of the array and the complexity of the processor at each stage are highlighted below, including a brief example of the complexity of algorithm involved. This particular example is one of scene analysis.

1.	256x256	1-bit	find edges
2.	64x64	8-bit	fit edges to line segments
3.	16x16	16-bit	2-D objects
4.	4x4	32-bit	3-D objects
5.	1	VAX	relationships

Although many machines exist using the pyramid topology [120], space does not permit more to be said on this interesting development.

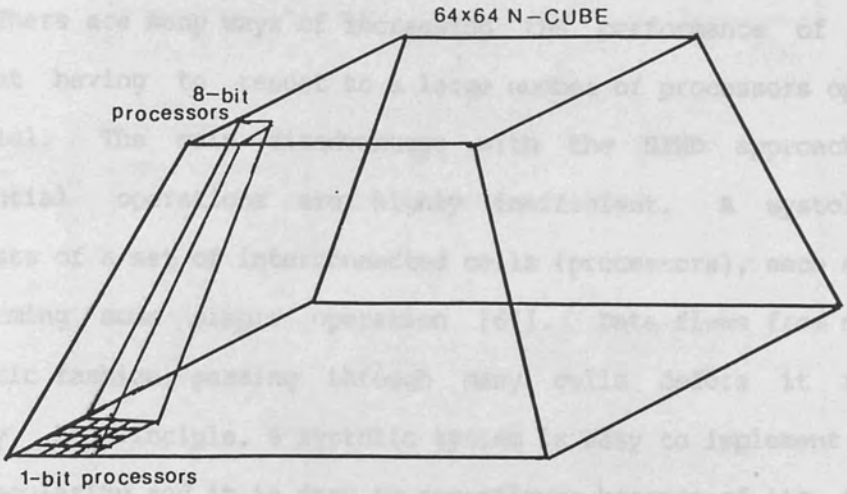


Figure 7.15 A two-level N-cube architecture

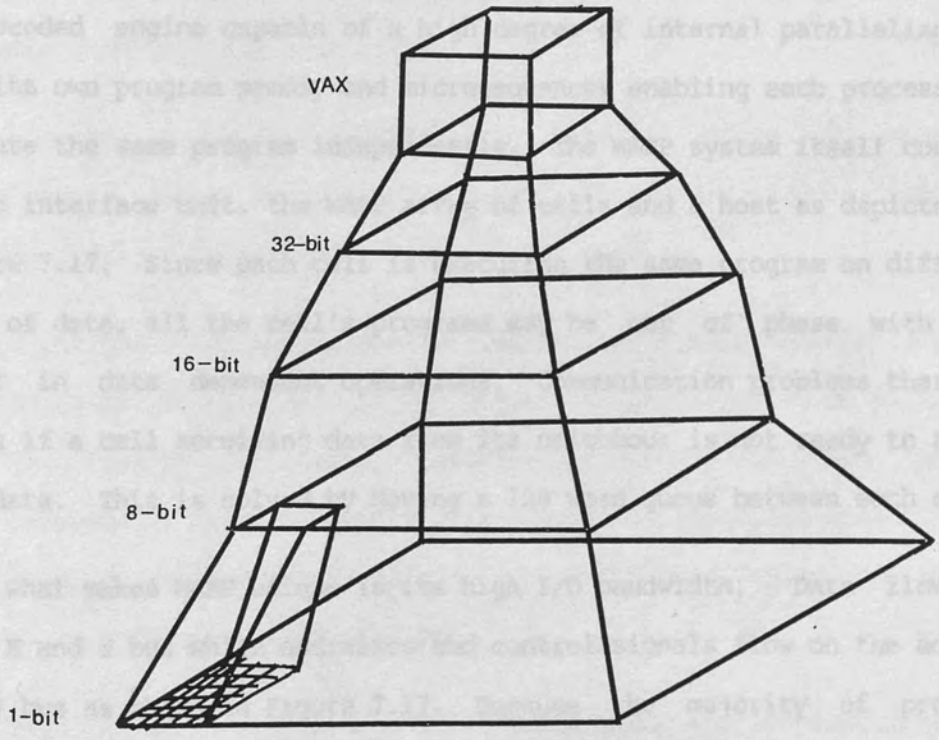


Figure 7.16 Pyramid architecture consisting of different levels of processor complexity

7.9 WARP - A SYSTOLIC ARRAY PROCESSOR

There are many ways of increasing the performance of a machine without having to resort to a large number of processors operating in parallel. The main disadvantage with the SIMD approach is that sequential operations are highly inefficient. A systolic system consists of a set of interconnected cells (processors), each capable of performing some simple operation [67]. Data flows from memory in a rhythmic fashion, passing through many cells before it returns to memory. In principle, a systolic system is easy to implement because of its regularity and it is easy to reconfigure because of its modularity. Pipelining is a simple example of a systolic architecture.

WARP [3] is a high-performance systolic array computer consisting of a systolic linear array of 10 or more identical cells. Each cell is an independent, 10Mflop, 32-bit floating point programmable horizontal microcoded engine capable of a high degree of internal parallelism. It has its own program memory and microsequencer enabling each processor to execute the same program independently. The WARP system itself consists of an interface unit, the WARP array of cells and a host as depicted in Figure 7.17. Since each cell is executing the same program on different sets of data, all the cell's programs may be out of phase with each other in data dependent operations. Communication problems therefore arise if a cell receiving data from its neighbour is not ready to accept the data. This is solved by having a 128 word queue between each cell.

What makes WARP unique is its high I/O bandwidth. Data flows on the X and Y bus while addresses and control signals flow on the address (ADR) bus as shown in Figure 7.17. Because the majority of programs implemented on a systolic array need to communicate intensively, WARP has been designed to transfer 20 million words (80 Mbytes) between cells per second. It operates on a 200ns clock cycle and is capable of I/O

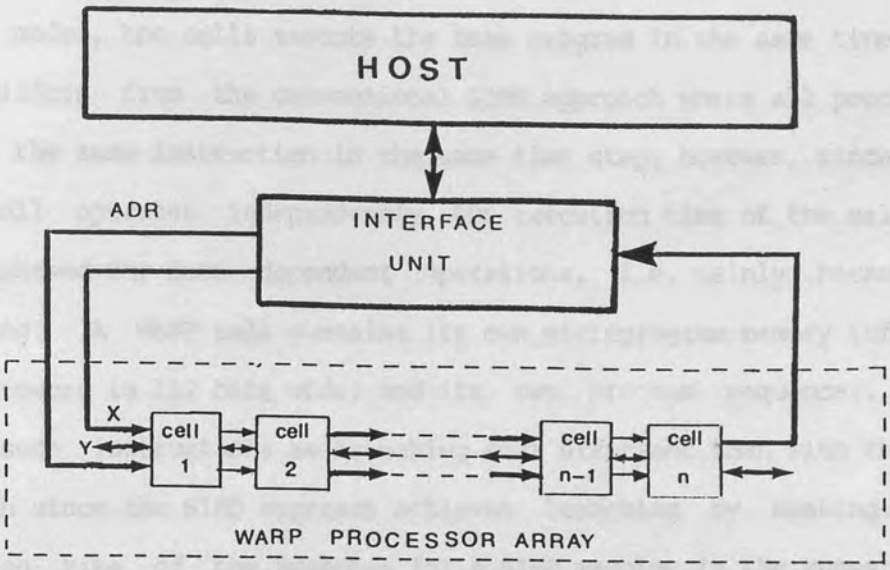


Figure 7.17 An overview of the Warp machine

(often the bottleneck in multiprocessor systems) at a rate of 10Mbytes/sec because of the simplicity of the linear interconnection structure between the WARP cells.

To the programmer, WARP looks like an array of sequential processors. An array of WARP processors is capable of operating in one of two modes: pipelined mode and parallel mode.

1. Pipelined mode: each processor constitutes a stage in the pipeline. As data is processed by one processor, it is passed on to the next for further processing. Repetitive computation can thus be decomposed into a number of identical pipelined stages. As an example, complex matrix multiplication has been implemented on WARP using an 8-stage WARP pipeline.

2. Parallel mode: the data is partitioned among the processors and each processor executes the same function on data resident in its local memory. This description will be expanded in Chapter 8.

In both modes, the cells execute the same program in the same time slot. This differs from the conventional SIMD approach where all processors execute the same instruction in the same time step; however, since each WARP cell operates independently, the execution time of the cells may become skewed for data dependent operations, i.e. mainly because of branching. A WARP cell contains its own microprogram memory (of which the microword is 112 bits wide) and its own program sequencer. This makes such instructions as branching more efficient than with the SIMD approach since the SIMD approach achieves branching by masking. The execution time of the branches for a SIMD machine is the summation of the execution time of each branch. With local program control on each cell, different cells may follow different branches; hence the execution time is the maximum execution time of the different branches.

The microcode in WARP is completely horizontal, giving the user complete control over the amount of parallelism needed in the program. Because each component in WARP is controlled by a dedicated field in the microword, scheduling is made easier, since there is no interference in the schedule of different components caused by conflicts in the micro-instruction field assignment. The internal data bandwidth is often a bottleneck in systolic array architectures. For this reason, a crossbar switch has been implemented between the functional modules of the WARP array. As an example of the performance of WARP, a 10-cell WARP can execute a 1024-point complex FFT at a rate of one FFT every 600μs. It is worth noting that a 70-cell version of WARP is currently being implemented in VLSI.

7.10 BIT-SERIAL MACHINES

Because of the large number of processors needed for parallel processors in image processing, the view of having a 32-bit processor per pixel for a 256x256 image size is still unrealistic. Because most

array processors have a large number of processors, they usually inherit the bit-serial architecture. Instead of processing a word horizontally (bit-parallel) as with a conventional processor, the words are processed vertically (bit-serial). The advantage of this is that the processors only need to be simple 1-bit wide processors; however, several machine cycles are required in order to process a multi-bit word, e.g. an 8-bit pixel value. With the emergence of VLSI, one can now consider incorporating several 1-bit processors onto a chip. The outcome of this is that bit-serial devices process data at a higher rate though they will be less efficient when floating point calculations need to be done.

Since image processing is largely concerned with 8-bit pixel data, it would appear that a bit-serial device is in many ways an optimum choice. This goes some way in explaining its wide use, e.g. CLIP4, DAP, MPP. etc.

7.11 HYBRID ARCHITECTURES

As we have seen in the previous sections, the SIMD architecture is mainly suited to pixel-parallel tasks[1] where only short-range interactions between pixels need be considered. On the other hand, a MIMD architecture is more suited to tasks that require access to any part of the image. Thus, SIMD is well suited for the early stages of an algorithm where pre-processing an image (e.g. histogramming, smoothing, filtering, etc.) needs to be carried out, whereas MIMD is more suited for the later stages such as those that deal with lines, contours, regions and image descriptions [113].

We have already seen that multiprocessor machines tend to be

---

[1] Here we mean that all pixels in an image can be processed in parallel

application driven. An example of this was the pyramid structure [120]. When applied to image-understanding, the machine topology matches the character of the data and the flow of the task up the pyramid. Siegel et. al. [113] are currently involved in the design of PASM (Partitionable SIMD/MIMD). This is being developed to meet the needs of both the low-level and the high-level tasks commonly found in image processing.

7.11.1 PASM - Partitionable SIMD/MIMD

PASM differs from the SIMD and MIMD approaches described in previous sections in that it can dynamically configure itself to operate in either SIMD or MIMD mode, thus adopting the configuration most suited for the current task (c.f. the mesh and pyramid topologies where the configuration is fixed).

A schematic diagram of the PASM prototype is given in Figure 7.18. It consists of a system control unit, a memory management system, four microcontrollers (MC's), 16 PE's, 16 memory banks and a switching network in order to connect the processors to the memories. (The final version is expected to consist of 1024 PE's and 32 MC's.) As its name asserts, PASM is partitionable and can hence appear as a number of independent machines. Figure 7.18 shows PASM partitioned into four sub-systems, each with its own MC.

The interconnection network (the network that connects the PE's to the memories) is similar to that described by Pease (Section 7.8.3) in that it adopts a switch-based arrangement as in Figure 7.13; however, the switches in the PASM network can also be set to 'broadcast', where either the upper or the lower input connection is connected to both the upper and the lower output. In order for a PE to communicate with a memory, it must initially set up a path by configuring the switches.

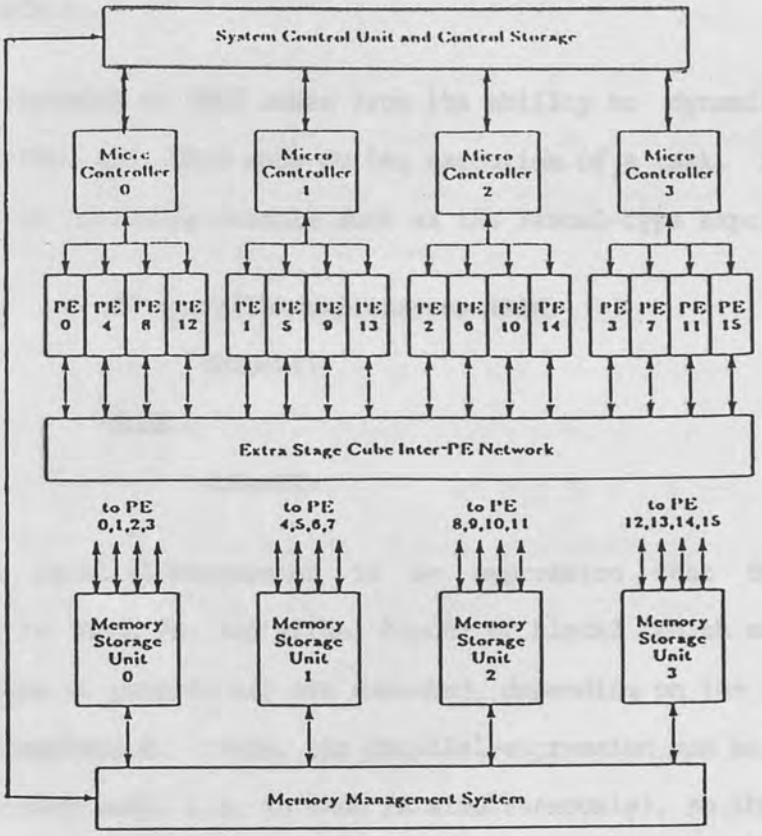


Figure 7.18 Schematic diagram of PASM showing the arrangement between the Microcontrollers, PE's, memories and the interconnection network

This is achieved by sending a routing address, R (bit i in R sets switch i to either straight or crossed) and a broadcast tag, B. If bit i in B is '0' then no broadcast is performed and the switch is set either to straight or to crossed, depending on the state of bit i in R. However, if bit i in B is '1' then a broadcast is performed, the upper or lower input being broadcast, depending again on the state of bit i in R. When a connection is made, the PE can communicate freely along the connected path with the memory until the connection is released by the PE (note that all this happens transparently to the user). An interesting point to note is the design consideration in adopting a hardware switch based system. This was chosen rather than a packet based system because of the ease of implementation and the anticipated large "conflict-free"

data transfers.

The strength of PASM comes from its ability to dynamically change between SIMD and MIMD mode during execution of a task. For instance, consider the following example such as the Pascal-type expression

```
IF <parallel-expression> THEN
        <block1>
ELSE
        <block2>
```

where the parallel-expression is an expression that depends on a variable in each PE, and either block1 or block2 (which may be several instructions or procedures) are executed, depending on the result of the parallel-expression. Here, the parallel-expression can be evaluated in parallel (SIMD mode, i.e. in each PE simultaneously), so the result will be true in some PE's and false in others. In a SIMD machine, branching is achieved by masking thus: block1 must be executed with the appropriate PE's disabled, then block2 must be executed, enabling and disabling the appropriate PE's. However, PASM can avoid this inefficiency by 'jumping' to MIMD mode. In MIMD mode, each PE executes either block1 or block2 (depending on the result from the previous operation) independently of each processor and then returning to SIMD mode. Thus, the execution time of the above expression is the greater of the block1 time and the block2 time rather than their sum.

Each PE's memory is divided into two sections: MIMD space and SIMD space. In MIMD mode, each PE operates as though it were a normal von-Neumann computer, fetching data and instructions from MIMD space in its own memory. Data from other PE's can be accessed via the interconnection network by setting the source and broadcast tags as described before, all of which takes place asynchronously under DMA control. A transfer to SIMD mode is initiated by executing a

"jump-to-subroutine" to an address in SIMD instruction space. This indicates that instructions are now to come from the MC rather than from the local memory. A request is made to the MC for an instruction, and the MC waits until all PE's in its partition are requesting before an instruction is sent. This is indicated by the result of ANDing all the request signals (if multiple MC's are used then the request signals from all MC's are further ANDed). All PE's then latch in the same instruction at the same time; however, each executes it independently. A new instruction is only issued when all PE's are requesting again. (While in SIMD mode, instruction latching and switch setting take place synchronously.) A return to MIMD mode is indicated by executing a "return-from-subroutine" instruction.

The only difference between SIMD mode and MIMD mode is that the PE program counter points to SIMD space in SIMD mode and MIMD space in MIMD mode. While in SIMD mode, the program counter merely serves to indicate that the PE is in SIMD mode: its actual value is irrelevant (as the instructions are now coming from the MC) as long as it accesses SIMD instruction space.

With this flexibility, PASM can emulate a variety of architectures including ring, mesh, pyramid and tree. With (for example) the mesh topology, each processor is physically hardwired to its neighbouring processors whereas PASM only has one outgoing link per processor. PASM thus has to communicate with each neighbour in turn and will therefore emulate different topologies with varying degrees of efficiency (and clearly sometimes with poor efficiency). One particular aspect of emulating a machine is the ease with which it can calculate the address of its neighbours. This is trivial for ring and mesh type topologies but is often more difficult for the pyramid topology; however, since the 16-PE prototype PASM is still in the preliminary stages of construction, the usefulness of this configuration in practical situations is as yet

undetermined.

Although the concept of a SIMD/MIMD architecture appears to have distinct advantages, the benefits of such systems are not realised without efficient ways of programming them. Indeed, the high degree of flexibility of PASM necessarily leaves many choices for the programmer and hence makes it quite difficult to program optimally. APLISP (A Parallel Language for Image and Speech Processing) is being developed for use on PASM [108]. The syntax of APLISP is similar to Pascal but for a few additions to the language specifically suited to image processing, e.g. the type BYTE to represent a pixel. Most current parallel systems [108] attempt to parallelise programs written in serial languages automatically or require users to structure their algorithms based on knowledge of the system architecture. APLISP allows the programmer to express parallelism in a natural way, independent of the machine's architecture. The compiler is given some information as to which operations in the program can be executed in parallel. APLISP follows the philosophy "the expression of parallelism in the language should be problem orientated rather than machine orientated" [108].

There are three possible ways of distributing the data in PASM:

1. A subarray per PE. Here, the image is divided into N subimages, each of $N^{\frac{1}{2}} \times N^{\frac{1}{2}}$.

2. A row per PE, i.e. PE i gets row i.

3. A column per PE, i.e. PE i gets column i.

Various algorithms have been proposed for suitable implementation on PASM including contour extraction [112], image correlation [111], image coding [110] and contextual classification [109]. Applications cited for PASM include remote sensing and the inspection of printed circuit boards [112]. The ability of PASM to execute these algorithms

efficiently is essential. This aspect is discussed in more detail in Chapter 8 when we consider the implementation of the O-ring algorithm on a series of multiprocessor architectures.

7.12 A COMPARISON OF SEQUENTIAL vs PARALLEL MACHINES

Many algorithms exhibit a high degree of both parallelism and sequentialism. For this reason, no individual algorithm is likely to be executed efficiently (efficiently being defined in this context as the proportion of processors that are processing information usefully) on one particular machine. In other words, a SIMD machine consisting of N processors executing a sequential algorithm will have N-1 idle processors, while a sequential machine executing a parallel algorithm will have to process the whole image (which may include a lot of redundant information) in order to achieve its goal. As the performance of PASM is unknown (though some simulation results have recently been published [114]), we will only discuss results that are available. However, since PASM appears to offer the ability to execute both parallel and sequential tasks efficiently in a multiprocessing environment, this may offer an effective solution to this problem.

Daniel Slotnick, father of the ILLIAC IV, states that parallelism is a very special case and that only certain computations can be usefully carried out by a parallel machine [37]. With regard to image processing, parallel computers are often employed as pre-processors except in special applications where the whole algorithm can be parallelised. We saw in Chapter 6 that a parallel and a sequential processor on a bus based system is impractical as it introduces a data bottleneck, while the PASM approach tackles the problem "head on". We therefore require a simpler more cost-effective solution that could be used in industrial applications. More will be said about this later.

In a recent study, several segmentation algorithms which exhibited a high degree of sequentialism and parallelism were studied on the GRID parallel processor [124]. GRID is of a similar nature to most SIMD array processors in the way it organises its data and instructions. This is a useful study because, as we saw in Chapter 2, segmentation techniques are probably the most widely used routines in image pattern recognition. The algorithms described below were implemented on a 64x64 array of GRID processors working on a 128x128 image size.

7.12.1 Bartliff's Algorithm

This algorithm consists of applying a combination of a Marr-Hildreth operator [75] and the Sobel operator [33] to an image. Because this algorithm is fairly complex, the details will be omitted here for space reasons; however, the following table of results lists the operations involved in the algorithm and the execution times for each operation as implemented on GRID. Note that all of the operations are parallel in nature except the chain code routine.

1.	Sobel	0.5ms
2.	Marr-Hildreth (width 4)	30ms
3.	Two multiplications	0.8ms
4.	Three thresholds	20μs
5.	Sort	30ms
6.	Interpolation	20ms
7.	Thinning	20ms
8.	Chain Coding	10s

The whole of the algorithm took 10.1s, the most prominent time being the Chain Code which took 10s.

7.12.2 Nevatia-Babu Algorithm

This algorithm uses a mask matching approach to edge detection (Section 2.5.2). It consists of applying six different 5x5 masks (each one corresponding to a different edge orientation of an ideal edge) to

every point in the image. The one that gives the highest response is chosen as the result. The result is a list of magnitudes and directions for each pixel. The edges are then thinned as a post-processing stage. The time to execute the whole algorithm was 20ms.

7.12.3 Merge

This algorithm was based on that described by Gupta and Wintz [124]. It involves dividing the image into elementary regions and then merging statistically similar regions into homogeneous 'blobs'. When a point is reached such that the blob will not merge with an elementary region, a new blob is started. This algorithm involves a large amount of communication over large parts of the image and is probably better suited to a sequential machine. The only part of the algorithm which could be parallelised was the initial stage where the statistics of the elementary regions were calculated. The algorithm took 7.46s, being reduced to 4.92s when the initial part was parallelised.

7.12.4 Conclusions

The Nevatia-Babu algorithm is a typical example of how a parallel machine will significantly improve the execution time of a program. However, examination of the Bartliff program shows that out of the 10.1s taken to execute the algorithm, 10s was spent in the chain code routine which accounts for 99% of the total execution time. This is a typical example of an inherently sequential process being implemented on a SIMD machine, and thus renders the SIMD approach useless for sequential image processing tasks. For this reason, SIMD machines have mainly been applied to low-level image processing tasks such as edge detection, thresholding and to tasks that exhibit a high degree of parallelism. Analysis of many common image processing algorithms (typically those in Chapters 2 and 3) show that many consist of both parallel and sequential

parts suggesting that the SIMD machines mentioned will do poorly for "typical" tasks. One of the main disadvantages is that each processor only has access to its local neighbours. If each processor had access to the whole image, this would dramatically improve the performance of the machine. However, with several thousand processors in such systems, this is impractical. Chapter 8 investigates this idea for a small set of processors.

Reconfigurable machines such as PASM can dynamically configure themselves to operate in either SIMD or MIMD mode, the configuration chosen being the one better suited to meet the needs of the current task. This appears to be an optimum approach; however, the performance of PASM is as yet unknown and optimal methods of programming this kind of flexible architecture are still in their early stages.

7.13 SUMMARY

The von Neumann architecture suffers from many deficiencies, most of which have been overcome by the use the Harvard architecture and pipelining. Both of these techniques are aimed at achieving the ultimate performance of one instruction/cycle. For a further increase in performance, computer architects have replicated the processor section for increased data throughput while maintaining the single instruction unit for global instruction broadcasts (a SIMD machine). A single instruction unit simplifies processor control, eliminating such problems as interprocessor communication, task scheduling and data bottlenecks - all traditionally associated with MIMD machines. This allows maximum instruction and data throughput to be achieved. The processor organisation is usually in the form of a linear or 2-D array of processors.

Image processing would appear to be an ideal application for a SIMD machine since the processor organisation can be mapped directly onto the data representation, i.e. the image. However, with image sizes typically of 256x256 pixels, matching a single 16-bit processor to every pixel is still unrealistic. Only now, with machines like the CLIP7, are 16-bit processors being used for the 1 processor/pixel concept. However, with this complexity, only a linear array is currently implemented using 16-bit processors. For this reason, SIMD processors have usually been designed as bit-serial processors. Processors like the DAP (1974) had 5 chips/PE, while current VLSI-based schemes, such as GRID, show that 64 PE's (eight in the case of the new DAP and the MPP) can be incorporated onto a single chip. Unfortunately, GEC (the designers of the GRID chip) are still suffering from problems which suggests that the number of PE's on a single chip may be approaching the limit achievable using traditional methods of production.

Analysis of sequential tasks executing on a SIMD machine shows that the performance of the machine rapidly deteriorates. Bearing in mind that a criterion for choosing an architecture was to measure the degree to which the processing hardware is being usefully kept busy (Section 7.5.1), the SIMD approach fails drastically because, by definition of a serial task, only one processor is usefully being employed. From this, we can draw the conclusion that a sequential processor is more useful executing programs that exhibit both sequential and parallel tasks than a SIMD machine is, mainly because the processor is fully utilised. In other words, a SIMD machine is not cost-effective unless the task can be fully parallelised. We may wonder therefore, if the SIMD architecture is the correct approach to take. Obviously in some situations it will be but, in general, we have found that many algorithms cannot be fully parallelised.

The ultimate question at this stage must be "where do we go from here?". The above arguments for a sequential machine suggest that we should concentrate on developing the fastest possible sequential processors and combine them in some way so as to maintain high performance levels for the execution of both sequential and parallel tasks, while still satisfying the criterion laid down in Section 7.5.1.

Such machines as the systolic array of WARP and the reconfigurable approach of PASM have been proposed which attempt to achieve this goal. In both cases, each processor in the array is a high speed sequential processor. With respect to WARP, each processor is able to communicate with its two neighbours, while in PASM, any processor can communicate with any other processor's memory via a switching network. The main difference between these methods and the SIMD approach is that each processor executes the same program but not in lockstep with its neighbours (however, when PASM is in SIMD mode, instruction latching takes place synchronously). Interprocessor communication obviously arises in data dependent operations. WARP uses a queue for such cases while PASM uses interrupts.

WARP aims to achieve the processing power of a MIMD machine without the problems associated with MIMD architectures (scheduling of tasks and communication, etc.); it also aims to maintain the simplicity of SIMD in that every processor executes the same program. This appears to be an optimal solution (PASM requires SIMD and MIMD tasks to be partitioned into separate processors). With the advent of VLSI, one would expect that the processing power of WARP could be reduced from a whole board to a few chips. Processors such as the 68030, although powerful in their own right, cannot deliver the necessary performance per processor or have the required flexibility. However, microcoded processors can achieve speeds far beyond those of conventional processors and yet remain a cost-effective and flexible solution.

The next question to raise is "how can an architecture like this help us in image processing?". Probably the most effective method of maintaining the performance of a parallel processor, yet still executing sequential tasks efficiently is by partitioning the image into equal areas and allocating each processor to an area as in PASM. This will undoubtably involve communication between processors: if the areas are small then the communication factor may dominate the execution time of the task. Hence, the problem remains of finding an efficient means of communicating between processors. We cited that the transputer appears to be an effective solution.

Unlike WARP which is a linear array of independent sequential processors, the next chapter (Chapter 8) investigates the application of a two-dimensional array of independent sequential processors and attempts to tackle the problem of interprocessor communication. This architecture is applied to one of the image inspection algorithms introduced in Chapter 3 that exhibits both parallel and sequential tasks.

CHAPTER 8

DEVELOPMENT OF A MULTI-SEQUENTIAL ARCHITECTURE

"You can't invent a design. You recognise it, in the fourth dimension. That is, with your blood and your bones, as well as with your eyes."

David Herbert Lawrence 1885-1930

8.1 INTRODUCTION

The aim of this chapter is to present a series of investigations into a number of multiprocessor architectures that are similar in nature but are essentially different in their interprocessor communication arrangement. These architectures are similar to that of WARP (Section 7.9) in that they are composed of multiple sequential processors, but the concept has been extended to the two-dimensional case. A novel method for minimising data bottlenecks usually associated with interprocessor communication is derived with little increase in circuit complexity. This entails an analysis of the architecture to determine the source of data bottlenecks. A discussion on the practical use of each system is undertaken and examines the expected performance relative to other machines of a similar topology.

8.2 REASONS FOR A HYBRID ARCHITECTURE

Many inspection algorithms consist of parallel and sequential tasks (c.f. Chapter 3). As we discussed in Chapter 5, a typical inspection algorithm might scrutinise an object by applying a Sobel, thinning the edges, chain coding the resultant image and manipulating the chain code. The first two steps would be executed more efficiently on a parallel processor whereas the third and fourth steps would be executed more efficiently on a sequential processor.

In Chapter 5 we described a bit-slice sequential processor (SIP) that was capable of executing the inspection algorithms presented in Chapter 3 in real-time. Chapter 6 proposed a dual-processor system composed of a sequential processor (SIP) and a parallel processor (the LAP); however, the initial results showed that this was likely to incur a large data bottleneck. This configuration was also limited in its expansion capabilities. Ideally, we require a system whose performance increases linearly with the number of processors for both parallel and sequential tasks.

We may recall that the processors in a SIMD machine execute the same instructions in lockstep and that this proved to be inadequate for sequential operations, while the MIMD machine executes different instructions on different sets of data. This increases the complexity of the system and large data bottlenecks are likely to occur without algorithms to make optimum use of the processors and their interprocessor communication arrangement. We showed that the approach of WARP in Chapter 7 appeared in some ways to be an efficient solution for use in image processing. This contained the same arrangement as a MIMD machine in that each processor is autonomous, while maintaining the simplicity of a SIMD machine in that each processor executes the same instructions. This can be thought of as a hybrid architecture between a

SIMD and a MIMD machine.

Because many image processing algorithms contain a high degree of both parallelism and sequentialism, a hybrid architecture is necessary in order to execute both parts efficiently. Machines such as DIPOD (Section 7.8.1) and PASM (Section 7.11.1) can adopt a SIMD/MIMD approach and appears highly suitable. However, as we shall see, without the right communication arrangement, a system can incur data bottlenecks that dramatically reduce the performance of the system to a less efficient level than intended. This chapter investigates several processor configurations each composed of multiple SIPs. An important point to note is that these configurations are aimed at the industrial inspection area that require a higher level of performance than can be achieved by a single processor. Although the concept of PASM and WARP display similarities to these systems, their costs are unknown to this research group. However, one suspects that they will cost appreciably more than the systems described in this chapter with an equivalent number of processors, and in any case they will be too expensive for certain industrial applications. Indeed, as we mentioned in Chapter 7, a single DIPOD processor has a similar performance to SIP, yet costs around £30,000.

Throughout this chapter we shall use the O-ring algorithm presented in Chapter 3 to examine the performance of each proposed topology. This was chosen as it is considered a "difficult" algorithm to execute efficiently on a machine because of its inherently high degree of sequentialism and parallelism. First, we must determine what we should parallelise in the system.

8.2.1 Exploiting Instruction and Data Parallelism

Three options for distributing the processors are immediately

obvious:

1. Partitioning the algorithm into data independent tasks so each processor executes its allocated task(s) in parallel with the other processors on its own data.

2. Pipelining the processors so each processor executes its allocated task(s) in parallel. This is a special case of option 1. Here, each processor operates on the output of another processor. The tasks may therefore be data dependent.

3. Partitioning the image into equal areas so each processor executes the same task(s) on a different part of the image.

These three possibilities are interesting in the sense that they provide three very different ways of parallelising a problem. In options one and two the instructions are parallelised while in the third case, the data is parallelised. Parallelising the tasks as in option 1 is of little use to us since the majority of image processing algorithms cannot be partitioned effectively except by pipelining (option 2). For instance, in the O-ring algorithm which involved the tasks: Sobel, Hough transform, sort, list manipulation and radial histogram, the tasks must be executed in a sequential order. Another point worth mentioning is that with this method, the number of processors applied to a problem is limited by the number of tasks in the algorithm. Ideally, we require a system that can be expanded to any desired degree.

Option 2 suggests parallelising the tasks by the use of pipelining. Consider a three-level pipeline as in Figure 8.1 where each processor represents a task from the algorithm described above. As processor 1 operates on the input image, processor 2 operates on the result from processor 1, and processor 3 operates on the result from processor 2. Hence, the time to execute the algorithm is the time of the slowest

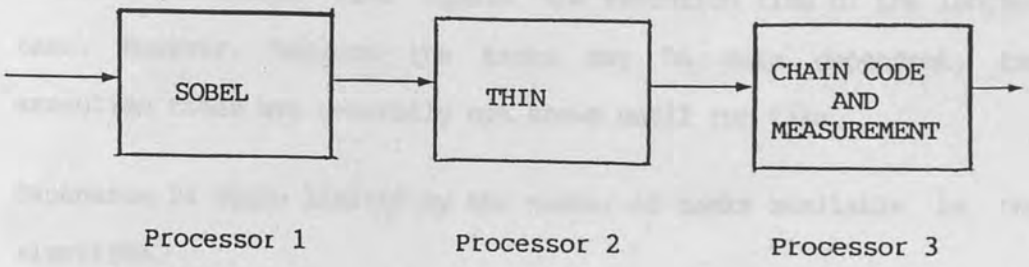


Figure 8.1 A three-level pipeline of processors. The time to execute the algorithm is the time of the slowest processor in the chain

processor in the chain. This method has the advantage that any algorithm consisting of several tasks may be parallelised in this way; however, it gives rise to several constraints:

1. To gain the benefits of a pipeline, an image must be constantly input. This occurs in industrial inspection applications where the same algorithm is applied to each frame of data coming off the camera. If only a single image was input, or images were input intermittently, then the time for it to travel through the pipeline is equal to the total time taken by each processor to execute its task, plus the communication overhead. This would violate the criterion given in Section 7.5.1 for inefficient use of processing power as only one processor would be active at any point in time.

2. To achieve maximum throughput, each processor must take the same amount of time to execute its task. If at one stage in the pipeline a processor's task takes a longer time to execute than the others, there would be idle processors and again, the criterion in Section 7.5.1 would be violated (c.f. initial parts of the algorithms in Chapter 3 which take ~70% of the total processing time). Alternatively, several tasks could be combined so their

combined execution time equals the execution time of the longest task. However, because the tasks may be data dependent, the execution times are generally not known until run time.

3. Expansion is again limited by the number of tasks available in the algorithm.

Points 1 and 2 have arisen because of the idle processor criterion in Section 7.5.1. However, this is ultimately application dependent; for instance, consider point 2 where two processors may be involved and, because of the structure of the algorithm, one processor is idle 50% of the time. One could argue that this is inefficient; however, if the algorithm must execute within a given time constraint and a single processor is incapable of achieving this, then two processors would need to be adopted. If the cost of two processors is within the budget available, then this would satisfy the requirements of the application and hence be cost-effective.

The third option of partitioning the image into equal areas and allocating each processor to an area initially appears sound for the following reasons:

1. Since each processor executes the same instructions, partitioning the algorithm is no longer a burden to the programmer.

2. The criterion given in Section 7.5.1 is satisfied since all processors are automatically (on average) kept busy.

From these two statements, it is clear that we should investigate this third option. From the above, we can conclude that if N processors are used on an LxL image then each processor should operate on an area (A) of:

$$A = L^2/N$$

where L is generally 128, 256 or 512 and N is 1, 4, 16, 256, etc.

Having decided that the image is to be partitioned, we now need to decide on how to arrange the processors, i.e. determine the topology of the system.

8.2.2 Matching the Task to the Architecture

The ability of the proposed architecture to handle both sequential and parallel tasks efficiently is essential. In the last section, we determined that the data (i.e. the image) should be distributed between the processors. Let us assume at this stage that each processor only holds the part of the image it needs to access, i.e. if four processors are used then each processor's image memory only contains a quadrant of the image (this is in fact one means by which PASM can distribute its data). Parallel tasks are straightforward in that no communication outside a processor's own area is required. (Note that problems will occur at the borders of the section of image when a 3x3 (or larger) window is used. Throughout this section, we shall assume that the borders are also available so interprocessor communication is not necessary for this reason.) However, sequential tasks, which are usually data dependent, may need to access <u>any</u> part of the image (c.f. the centre finding algorithm in Chapter 3) or another processor's results. This means that some form of interprocessor communication is necessary - this is where the data bottleneck usually occurs.

8.2.3 Simulation

In this chapter, all configurations are simulated on a PDP-11/73 processor using the Pascal language, operating on Figures 3.9a-3.9h (128x128 images). (Note that these are indicated on the graphs in Figures 8.3,8.7 and 8.9.) However, in reality a high speed sequential processor such as SIP would be used in the system. This will offer a factor in the region of ~27 times speed improvement over the PDP-11/73

(Chapter 6). Note that it was necessary to simulate a SIP-based system on the PDP-11/73 as it would have been difficult and time consuming to program in SIP's native language. All times and costs referred to throughout this chapter are thus targeted at SIP.

There are two important points to note about the simulation study. First, the PDP-11/73 and SIP have entirely different architectures namely, the PDP-11 has has a von Neumann architecture while SIP has a Harvard architecture. However, the PDP-11 is found to execute the majority of its instructions from cache (the PDP-11/73 has an 8Kbyte cache), and these accesses are comparably fast relative to frame store (data) accesses. Thus, the PDP-11/73 turns out to perform much as it would if it had a Harvard architecture (more will be said about this later). Second, there is the problem of emulating N processors on a single processor. In these simulations, a task (in effect a Pascal procedure) is presented with the data appropriate to one of the processors, and the time to execute this procedure is taken. This process is repeated N times, once for each processor. Thus, if a procedure is executed in time T_i by processor i, then the time for the task to execute on N processors is given by:

$$T = MAX(T_1, T_2, \ldots, T_N)$$

Because the following configurations involve buses (used for interprocessor communication) some means of measuring the communication time is required. Here, a processor's memory is represented as an array in the program; thus, the time it takes to transfer data from one processor to another is taken as the time it takes to transfer an array of data to another array. This is justified, as the time for a SIP processor to access data on another SIP processor (based on the VMEbus) is the same as the time it takes a SIP processor to access data from its on-board memory, i.e. three clock cycles (Chapter 5). (Note that the

time to distribute an image to the other processors in the system is included in the simulation in order to obtain a realistic measure of performance in a real situation.)

The following sections investigate various topologies that attempt to solve the interprocessor communication problem and, although a topology may appear satisfactory, only rigorous analysis will show if it suffers from bottlenecks. The first of these configurations highlights this problem and also shows the problems that occur for a pipelined system where tasks have to be explicitly partitioned.

8.3 CONFIGURATION 1 - THE MASTER/SLAVE ARRANGEMENT

Analysis of the timings from the algorithms given in Chapter 3 show that the parallel tasks consume ~70% of the total execution time on a single sequential processor. One could therefore consider reducing the execution time of the parallel tasks while maintaining the execution time of the sequential tasks. Because a parallel task requires no communication outside its own area, it is not unreasonable to suggest having multiple sequential processors (called slaves) to execute the parallel tasks. Each slave need only contain and operate on its allocated part of the image as shown by the shaded quadrant in Figure 8.2, depicting a four processor configuration. A single sequential processor (called the master) would then fetch the data from the slaves and execute the sequential tasks. (It is assumed here that the master and slaves are connected by a common bus and the master accesses the slaves through dual-port RAM.)

This configuration will have the effect of distributing the computation of the parallel tasks evenly over a number of processors. (Note that this is a pipelined system, i.e. as the slaves are operating on one image, the master could (in principle) be operating on the

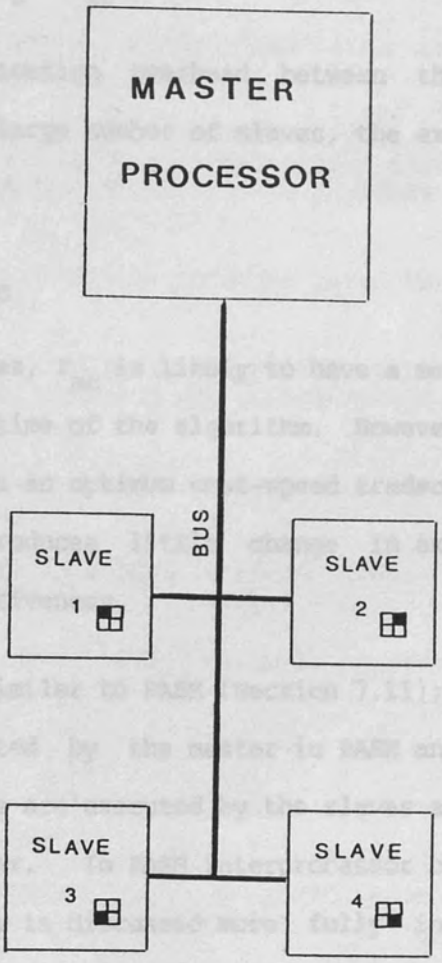


Figure 8.2 A 1-Master/4-Slave arrangement. The slaves only operate on their allocated section of image (indicated by the shaded square)

previous result from the slaves.) Thus, for an algorithm whose parallel tasks execute in time T_p and whose sequential tasks execute in time T_s, the time to execute an algorithm T_{tot} on a single processor system, is

$$T_{tot} = T_p + T_s$$

Therefore, for the Master/Slave configuration

$$T_{tot} = T_p/N + T_s + T_{sc}$$

where T_{sc} is the communication overhead between the master and the slaves. Thus, for a large number of slaves, the execution time of the algorithm will be

$$T_{tot} \simeq T_s + T_{sc}$$

Therefore, as N increases, T_{sc} is likely to have a more prominent effect on the total execution time of the algorithm. However, there will be an N for which there exists an optimum cost-speed tradeoff where addition of more processors produces little change in execution time, hence reducing its cost-effectiveness.

This topology is similar to PASM (Section 7.11); however, whereas SIMD tasks are executed by the master in PASM and MIMD tasks by the slaves; here, SIMD tasks are executed by the slaves and MIMD tasks are executed by the master. In PASM interprocessor communication occurs during MIMD tasks - this is discussed more fully in Section 8.4. We chose this method for hardware simplicity. The rest of this section is devoted to analysing the Master/Slave configuration in order to determine the optimum number of slaves that should exist in the system, and quantifying the values of T_p, T_s and T_{sc}. First, let us consider the implementation of the O-ring algorithm on the system.

8.3.1 The O-ring Algorithm and the Master/Slave Arrangement

The algorithms below represent the O-ring algorithm for a single processor (algorithm 1) and the modified version for the Master/Slave configuration (algorithm 2). (See Chapter 3 for a detailed explanation of the O-ring algorithm.)

1. For the whole image in P-space, apply a Sobel operator.

2. For each edge point, obtain a candidate centre point and mark in Q-space.

3. After completion of the whole image, get the (x,y) coordinates and value of those points greater than a fixed threshold (peaks). These are the possible centres of the O-rings.

4. Sort list into a decreasing order based on the values of the peaks.

5. Starting with the highest value peak, determine the centres of the rings, ignoring any points within a fixed distance of each other.

6. For each centre found, apply a filter in Q-space to eliminate noise.

7. Put true centre in a "Centre found" list.

The O-ring algorithm for a single processor system

1. Let each slave apply a Sobel to its part of the image in P-space.

2. For each edge found, obtain a candidate centre position and put the (x,y) coordinates into an array.

3. On completion of all slaves, the master gathers the (x,y) data from each slave and reforms Q-space.

4. The reformed image is now scanned and the coordinates of those points with a value greater than a fixed threshold (peaks) are stored. These are the possible centres of the O-rings.

5. Sort list into a decreasing order based on the values of the peaks.

6. Starting with the highest value, determine the centres of the rings, ignoring any points within a fixed distance of each other.

7. For each centre found, apply a filter in Q-space to eliminate noise.

8. Put true centre in a "Centre found" list.

The O-ring algorithm for the Master/Slave configuration

The modified version of the algorithm highlights three major properties of the system:

1. Parallel tasks and sequential tasks must be explicitly partitioned for the master and slaves to serve their purpose.

2. Although no parallel tasks exist after the data has been fetched by the master, if any did exist, they would either have to be executed by the master, or the data would have to be redistributed to the slaves before processing could continue. This leads on to point three.

3. A data bottleneck may occur when the master either fetches data from the slaves or redistributes its data to all the slaves.

The Master/Slave configuration is therefore best suited to an algorithm that exhibits either entirely parallel tasks or one composed of entirely parallel tasks followed by entirely sequential tasks. What we now need to determine is the optimum number of slaves that should exist in the system for optimum performance.

8.3.2 Analysis of the Master/Slave Arrangement

Figure 8.3a shows a graph of the square root of the number of slaves (y-axis)[1] against the time taken for both the master (A lines) and the slaves (B lines) to execute the modified version of the O-ring algorithm (x-axis) on Figures 3.9a to 3.9h. As discussed before, the algorithm was simulated with 1, 4, 16, 64 and 256 slave processors. (Note that the times here relate to a PDP-11/73 processor. These are

---

[1] Because of the non-linear nature of the number of slaves involved, the square root of the number of slaves produced a visually clearer graph than if the y-axis represented the number of slaves directly, i.e. it has no special significance except for presentation.

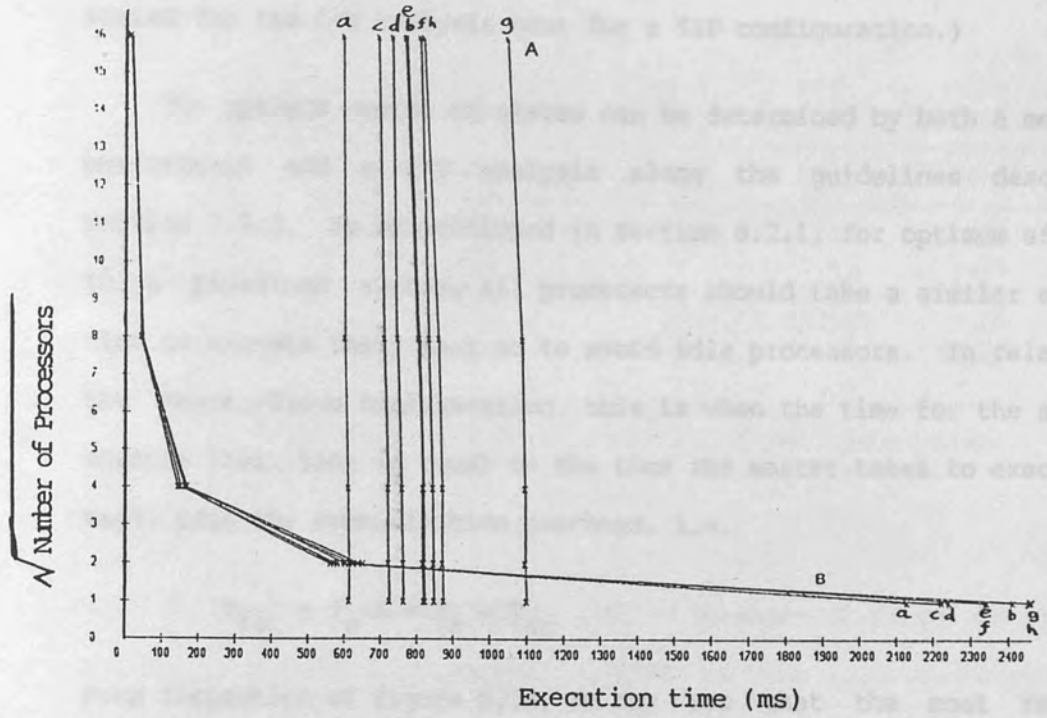


Figure 8.3a Master(A)/Slave(B) execution times of the O-rings algorithm for 1,4,16,64 and 256 SLAVE PROCESSORS

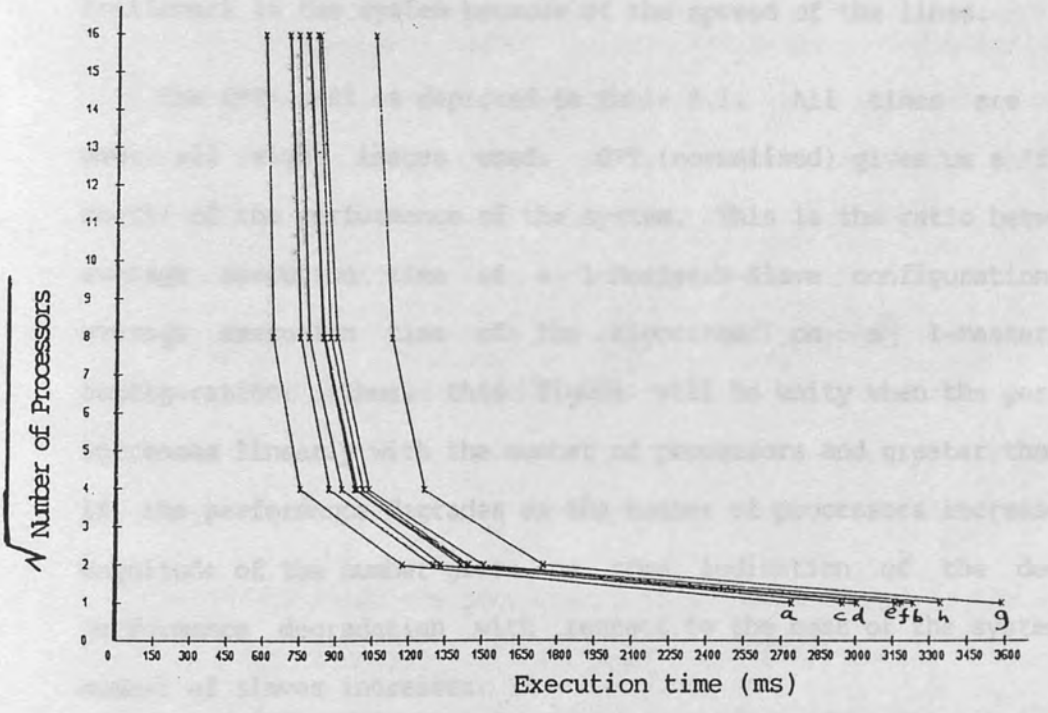


Figure 8.3b Graph of the execution times of the Master/Slave configuration on eight images of test data for 1,4,16,64 and 256 SLAVE PROCESSORS

scaled for the C*T analysis next for a SIP configuration.)

The optimum number of slaves can be determined by both a measure of performance and a C*T analysis along the guidelines described in Section 7.5.2. As we mentioned in Section 8.2.1, for optimum efficiency in a pipelined system, all processors should take a similar amount of time to execute their task so to avoid idle processors. In relation to the Master/Slave configuration, this is when the time for the slaves to execute their task is equal to the time the master takes to execute its task, plus the communication overhead, i.e.

$$T_{tot} = T_p/N = T_s + T_{sc}$$

From inspection of Figure 8.3a, we can see that the most reasonable choice of slaves is four, i.e. where the lines intersect; however, from inspection of the graph, it is clear that the master (A lines) is a bottleneck in the system because of the spread of the lines.

The C*T[1] test is depicted in Table 8.1. All times are averaged over all eight images used. C*T (normalised) gives us a 'figure of merit' of the performance of the system. This is the ratio between the average execution time of a 1-Master/N-Slave configuration and the average execution time of the algorithm on a 1-Master/1-slave configuration. Thus, this figure will be unity when the performance increases linearly with the number of processors and greater than unity if the performance degrades as the number of processors increases. The magnitude of the number gives us some indication of the degree of performance degradation with respect to the cost of the system as the number of slaves increases.

---

[1] The cost of a single processor (SIP) is assumed to be £620 based on July 1987 prices.

Number of slaves	Cost C (£) (master+slaves)	Time T (ms)	C*T x100	C*T (normalised)
0[1]	620	105	****	****
1	1240	117	1.45	1.00
4	3100	53	1.64	1.13
16	10,540	36	3.79	2.61
64	40,300	31	12.50	8.62
256	159,340	28	46.20	31.86

Table 8.1 Cost-Time breakdown of the Master/Slave configuration for 1,4,16,64 and 256 slave processors

Strangely, Table 8.1 shows that a 1-Master/1-Slave configuration actually impedes the execution time of the algorithm over a single processor (one processor executing both the sequential and parallel tasks). This is because of the communication overhead, i.e. $T_1 > T_0$ and $T_1 > T_4$. In order to improve the speed of the algorithm with this architecture, we must adopt a 1-Master/4-Slave configuration. This gives us an approximately linear increase in speed with the number of slaves. However, this linear variation soon deteriorates indicating a bottleneck. The C*T figures show a corresponding deterioration. One can therefore conclude from this analysis that the O-ring algorithm is only suited to a 4-slave or a 16-slave configuration or intermediate numbers of slaves: here and elsewhere, this will tend to mean dividing the image in powers of two, but not necessarily powers of four. Clearly, it will probably be unsuitable for many other algorithms.

Table 8.2 shows the performance improvement factor of the execution time of the algorithm on the Master/Slave configuration relative to a 1-Master/1-Slave system. Also shown is the percentage of the total

---

[1] Note that this is merely provided for comparison purposes.

execution time (t.e.t.) of the algorithm spent communicating (T_{sc}).

Number of slaves	Performance increase	Tsc (% of t.e.t.)
1	1.00	13
4	2.21	30
16	3.25	48
64	3.77	56
256	4.18	60

Table 8.2 Table of the performance increase and the communication bottleneck for the Master/Slave configuration

As we can see from Table 8.2, for a 256 processor configuration, 60% of the time is spent in transferring the data. Figure 8.3b shows the graph of the total time taken to execute the algorithm - this allows us to view the processing times of each complete system.

From the results given in Table 8.1, it is possible to derive a mathematical model of the system. Referring back to our previous equation of the system, i.e.

$$T_{tot} = T_p/N + T_s + T_{sc}$$

we can now quantify T_p, T_s and T_{sc}. For 256 processors (N=256), T_{tot}=28ms which can be approximated to $T_s + T_{sc}$ since T_p/N is negligible when N=256. Thus, subtracting this from the time for one processor to execute the algorithm (i.e. both sequential and parallel parts), we calculate the value of T_p to be 89ms. Therefore,

$$T_{tot} \simeq 89/N + 28 \qquad \simeq T_0(3/N+1)$$

where $T_0 \simeq$30ms. Thus,

$$T_s + T_{sc} \simeq 30ms.$$

From the results in Table 8.2, we can calculate that

$$T_{sc} \simeq 16ms.$$

Therefore, the master processor spends about half of its time communicating, and this is not very efficient. We can calculate the theoretical values and compare these results. Assume that the time to access all pixels in an image plane is equal to 1 time unit. The tasks for the master can be divided into the following:

1. The Master clears Q-space.
2. It then loads the image into the slaves. (The slaves are then started.)
3. The results from the slaves are fetched and Q-space (Hough space) is formed.
4. Q-space is scanned to locate the centres.
5. Processing continues.

Points 1, 2 and 4 add up to the equivalent of three image plane accesses, i.e. three time units. Points 3 and 5, from the timings given in Chapter 3, add up to the equivalent (in time) of a single time unit. Thus, the Master takes approximately four time units to execute its tasks. Now assume that there are N slaves in the system. The tasks for the slaves can be divided into:

1. Each slave operates on its section of image applying a Sobel.
2. For each edge point, the centre is calculated.
3. These centre points are written into a RAM.

Here, application of a Sobel is equivalent to 12 image plane accesses. Thus, each slave takes 12/N time units. (Note that we must also consider the time to write the edge points to the RAM; however, since this is of the order of 100 points, it is considered negligible relative to a single time unit.) Thus, the time for the Master and Slaves to execute the algorithm (in time units) is

$$4 + 12/N = 4(1+3/N)$$

The actual value of a time unit on a SIP processor (in this simulation) is 7ms. Thus, the above equation becomes

$$7*4(1+3/N) = 28(1+3/N)$$

The equation we derived from the simulation results (when we assumed SIP was 27 times faster than a PDP-11/73) was 30(1+3/N). This close agreement between the model and the observed timings shows that we have a good understanding of the process involved in running this algorithm, and lends support to the statement made earlier that the PDP-11/73 is here acting as if it had a Harvard architecture: it goes some way to justifying performing a simulation.

We can carry on with the analysis and calculate the (theoretically) most cost-effective system which is when d(C*T)/dN=0, i.e.

$$d(C_0(1+N)*T_0(1+3/N))/dN = 0$$

thus,

$$1-3/N^2 = 0 \quad \text{i.e. } N = 1.73$$

Hence, in this case, the optimal number of slaves is two. The above figures suggest that we should investigate methods for reducing T_{sc} - this is discussed in Section 8.4.

8.3.3 Topologies Related to the Master/Slave Configuration

This configuration of processors has been used by DIPOD (Section 7.8.1) for shape classification. The algorithm consisted of a Sobel followed by tracing, segmentation, data manipulation and classification. The topology of the algorithm mapped well on to the Master/Slave configuration: the slaves executed the Sobel while the master executed the sequential operations. Although no analysis has been carried out on DIPOD, the above investigation suggests that the

usefulness of this configuration is application dependent.

In order to improve the performance of the system, three further configurations have been investigated called ARCH-1, ARCH-2 and ARCH-3 respectively. The first of these will now be described.

8.4 CONFIGURATION 2 - ARCH-1

In order to eliminate the master from the Master/Slave configuration, it is necessary for the slaves to execute the sequential tasks. This means that each slave must have access to the results from all other processors. The total execution time of an algorithm would now become

$$T_{tot} = T_p/N + T_s/N + T_K$$

where T_K is a constant which is the sum of the communication time (T_{sc}) and the execution times of those tasks that are independent of the number of processors in the system (T_{pi}), i.e. the reformation of Q-space and the sort task in the case of the O-ring algorithm. Therefore, for a large number of processors

$$T_{tot} \simeq T_K = T_{sc} + T_{pi}$$

This section aims to minimise T_{sc} and determine (1) the optimum number of processors that should exist in the system, (2) T_{sc} as a proportion of the total execution time of the algorithm and (3) the performance ratio P_N where

$$P_N = T_{one\text{-}processor}/T_{N\text{-}processors}$$

Ideally, if N processors are used we should achieve a decrease in execution time of an algorithm by a factor N (i.e. P_N=N).

As we mentioned before, each processor must have access to all results derived by the other processors, whether in the form of part of an image or data, e.g. a list of edge coordinates. This may appear to be a complex task without avoiding memory contention (and hence a data bottleneck), i.e. if several processors access the same memory on the same processor at the same time, but can in fact be solved quite simply as follows.

Rather than each processor having access to only part of an image, each holds the whole image. This immediately solves the problem of image memory contention; however, each processor still only operates on its designated area of image as depicted in Figure 8.4 for a four processor configuration. This may appear not to be cost-effective, for instance, if 256 processors are used on a 128x128 image then each processor would only operate on an 8x8 area of the image. Thus, for operations that do not access data outside a processor's allocated area, only 0.4% of the memory would actually be used. However, one must also consider that memory is becoming increasingly cheap and, if interprocessor communication was necessary, the extra cost and board space required by the additional communication logic would be similar to that of the required memory. Logic is therefore kept to a minimum which is appealing from both a hardware and software point of view.

At some point during a program a processor may need to access another processor's memory, e.g. if four processors are each operating on a quadrant of an image and the next stage requires that a processor access the results from another processor, then interprocessor communication must occur. (Note that intermediate results (e.g. a subimage) can be held locally in each processor, and data need not be transferred unless another processor requires that data. In the case of the O-ring algorithm, data is in the form of a series of (x,y) coordinates, thus eliminating the need to transfer the whole Sobelled

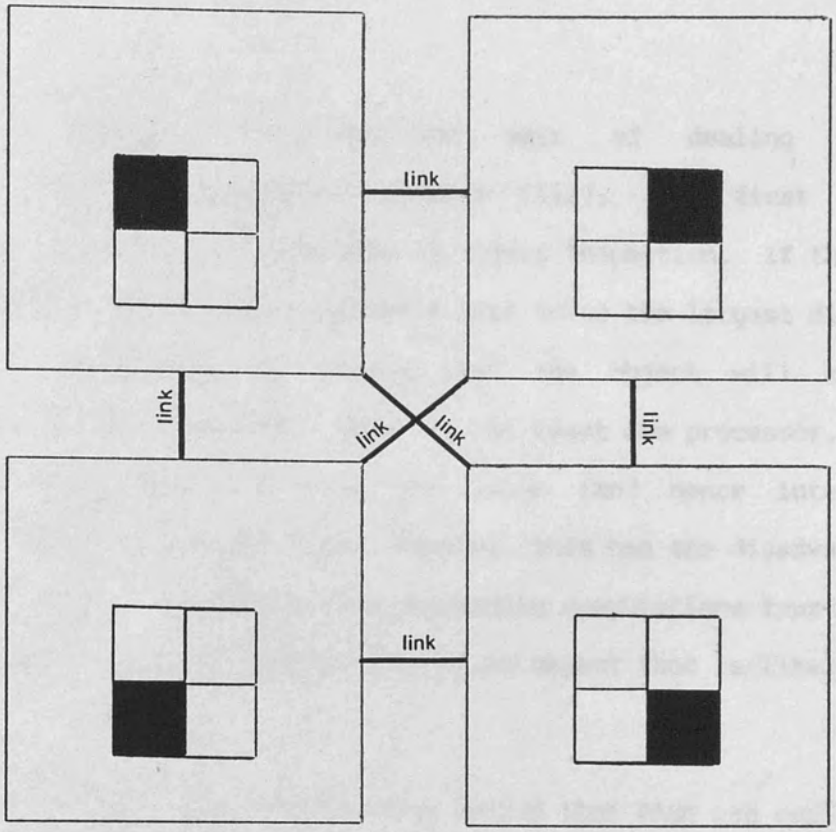


Figure 8.4 Each processor holds the whole image plane but only operates on its allocated part of the image

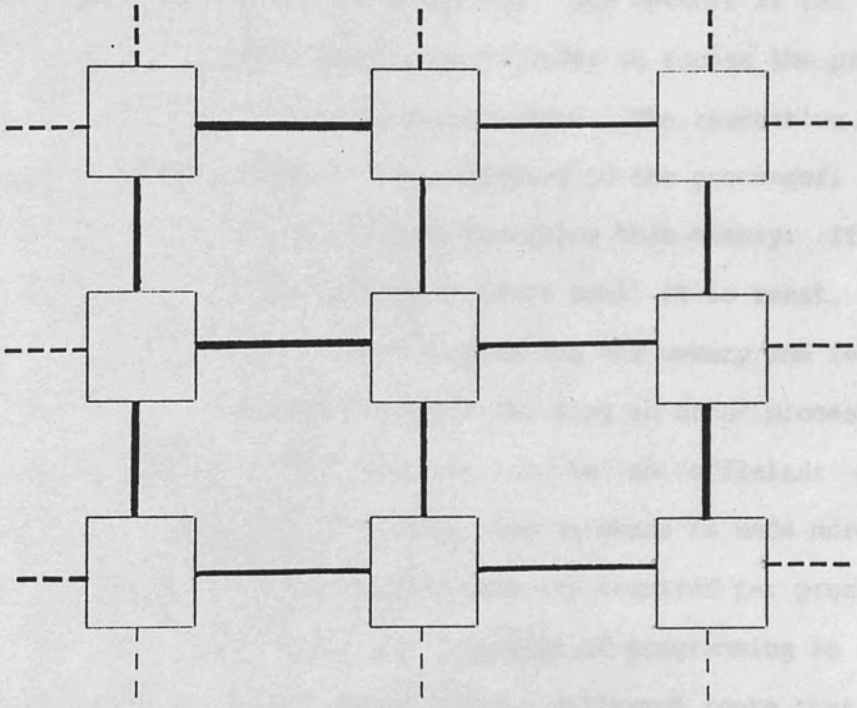


Figure 8.5 ARCH-1 - each processor is only connected to its four nearest neighbours

image.)

PASM (Section 7.10.1) has two ways of dealing with the interprocessor communication problem [112]. The first (and more restricted method) is in the case of object inspection. If the size of the object is known, then a subimage size twice the largest dimension of the object is chosen. This ensures that the object will be totally enclosed in the subimage area in at least one processor. External references to other parts of the image (and hence interprocessor communication) need not occur; however, this has the disadvantage that it is necessary to perform image processing computations four times for each pixel. Also, the maximum size of an object that is likely to occur must be known.

The second and less restrictive method that PASM can employ is by the use of semaphore flags. We may recall that PASM accesses another processor's memory by setting up the switching network [113] by the use of a routing tag and a broadcast tag. The network is set up by the requesting processor using these tags in order to access the processor's memory which contains the required subimage. The requesting processor then checks a "memory locked" flag belonging to the processor. This is set if another processor is already accessing that memory. If the flag is set, then the requesting processor waits until it is reset, else it sets the flag to indicate it is accessing the memory and fetches the data. When it has finished, it resets the flag so other processors can access that memory. This appears to be an efficient means of interprocessor communication; however, the hardware is made more complex as an additional two VME-standard boards are required per processor for the communication logic [113]. As this kind of programming is still at the research stage, we have investigated a different route that requires less complex hardware. Two alternatives for interconnecting the processors have been investigated:

1. Every processor is connected to its four nearest neighbours as depicted in Figure 8.5.

2. Each processor is connected to every other processor (maximally connected).

The next section investigates the first option in that each processor is connected to its four nearest processors.

8.4.1 Sharing the Data in the System

Let us assume that each processor has a memory scheme such that it has access to a byte-wide, dual-port memory on each of its four neighbours, and that each processor is able to transmit a data byte to each of these memories simultaneously. This is depicted in Figure 8.6 where the main data bus of each processor is connected to the input of the dual-port memories on its neighbouring four processors, and also to the outputs of its four local dual-port memories.

In order to transfer the data throughout the system, each processor must transmit its data to its four neighbours; hence each processor will receive data from its four neighbours. This received data is then re-transmitted to its four neighbours, avoiding transmitting data received from processor i back to processor i. After application of this step $\sqrt{N}/2$ times, all processors will have received the data initially transmitted by every processor. It is therefore clear that, as the number of processors in the system increases, the time to transmit the data (T_{sc}) throughout the system will also increase. In order to investigate this further, we will now consider the implementation of the O-ring algorithm on this configuration.

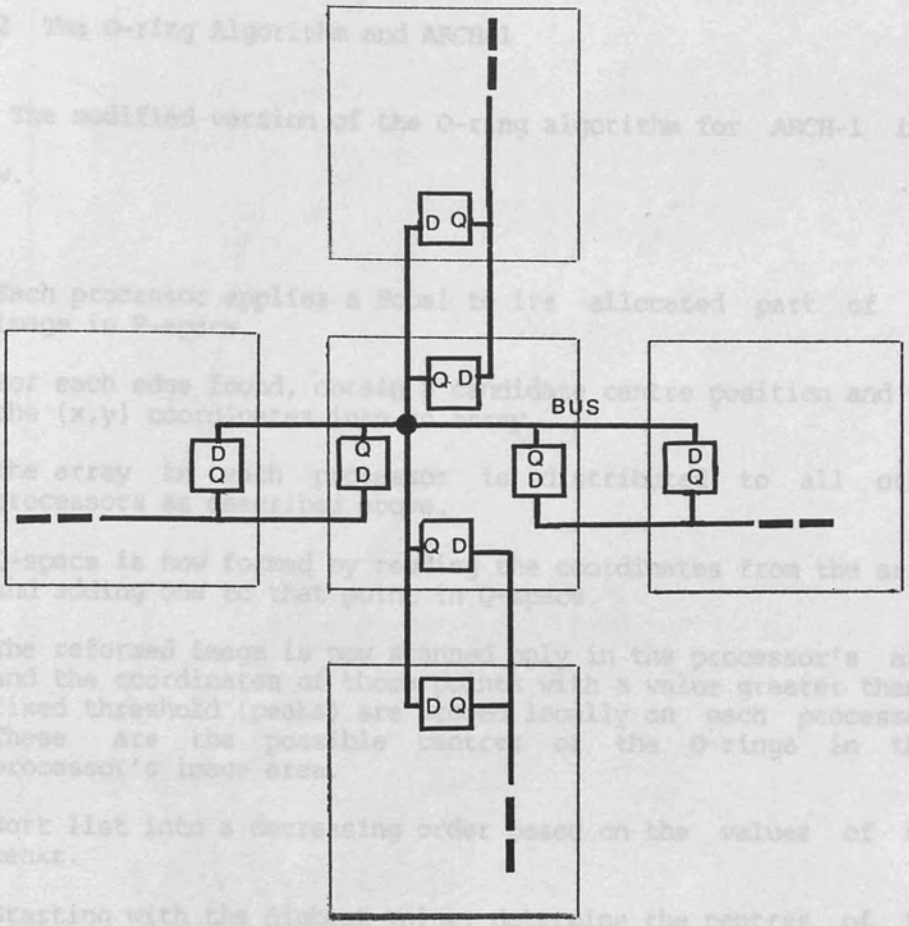


Figure 8.6 Each processor has access to a dual-port memory on each of its four neighbouring processors and also its four local dual-port memories

8.4.2 The O-ring Algorithm and ARCH-1

The modified version of the O-ring algorithm for ARCH-1 is given below.

1. Each processor applies a Sobel to its allocated part of the image in P-space.
2. For each edge found, obtain a candidate centre position and put the (x,y) coordinates into an array.
3. The array in each processor is distributed to all other processors as described above.
4. Q-space is now formed by reading the coordinates from the array and adding one to that point in Q-space.
5. The reformed image is now scanned only in the processor's area and the coordinates of those points with a value greater than a fixed threshold (peaks) are stored locally on each processor. These are the possible centres of the O-rings in that processor's image area.
6. Sort list into a decreasing order based on the values of the peaks.
7. Starting with the highest value, determine the centres of the rings, ignoring any points within a fixed distance of each other.
8. For each centre found, apply a filter in Q-space to eliminate noise.
9. Put true centre in a "Centre found" list.
10. Redistribute "Centre found" list in order to determine those centres that lie on the border of two processors.

The O-ring algorithm for ARCH-1

The graph of the execution time of the algorithm is depicted in Figure 8.7. Ideally, we expect a performance ratio (P_N) of 4.0 for a 4-processor configuration (an average execution time of 105ms). However, the average execution time is 36ms producing a P_N of three. Further analysis reveals that a P_N of 7.5 (average execution time of 14ms) exists with a 16-processor configuration and a P_N of 11.6 (average execution time of 9ms) exists with a 64-processor configuration. Not

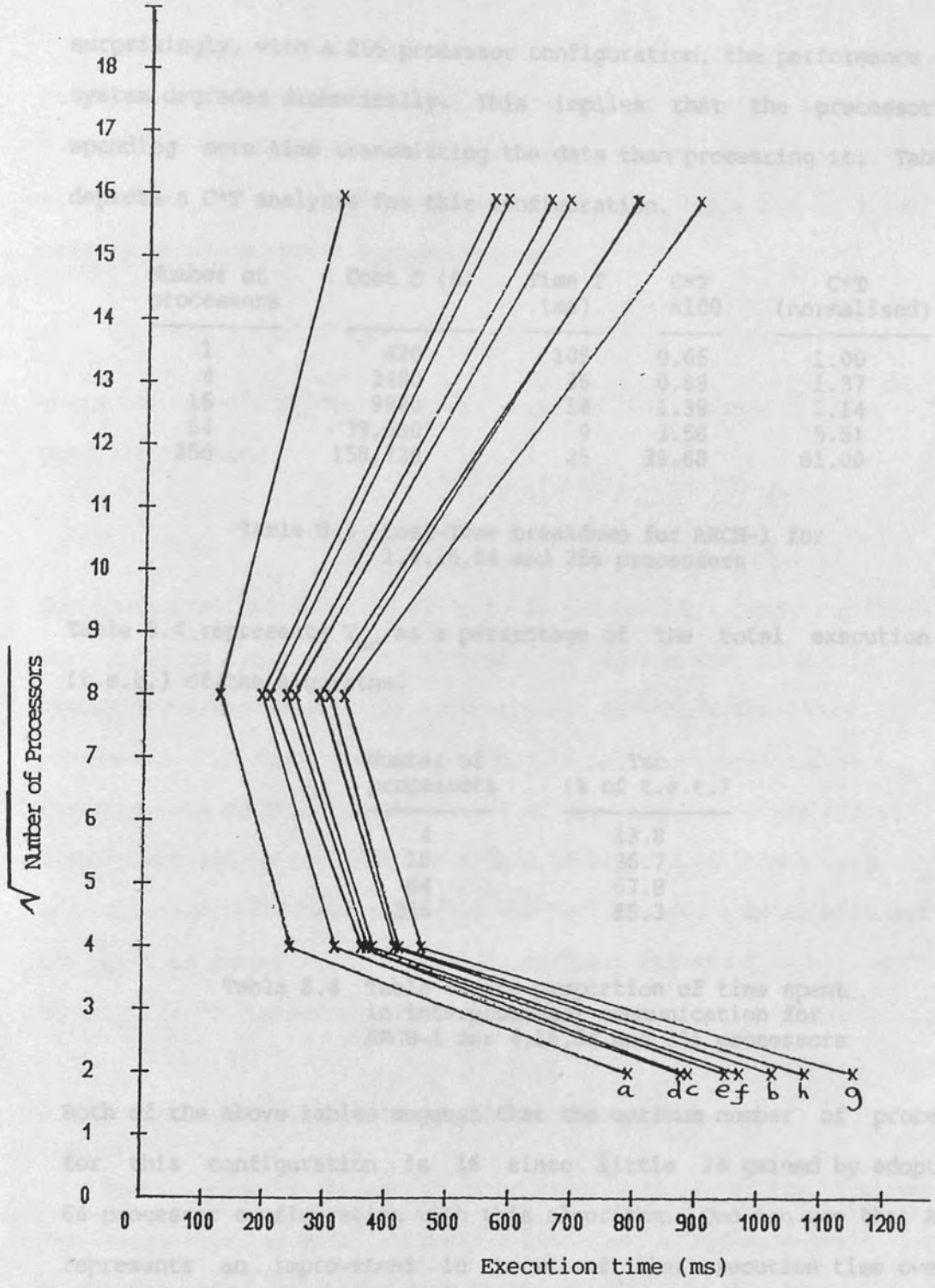


Figure 8.7 Graph of the execution times of the O-ring algorithm with ARCH-1 using 1,4,16,64 and 256 processors, operating on eight images of test data

surprisingly, with a 256 processor configuration, the performance of the system degrades dramatically. This implies that the processors are spending more time transmitting the data than processing it. Table 8.3 depicts a C*T analysis for this configuration.

Number of processors	Cost C (£)	Time T (ms)	C*T x100	C*T (normalised)
1	620	105	0.65	1.00
4	2480	36	0.89	1.37
16	9920	14	1.39	2.14
64	39,680	9	3.58	5.51
256	158,720	25	39.68	61.00

Table 8.3 Cost-Time breakdown for ARCH-1 for 1,4,16,64 and 256 processors

Table 8.4 represents T_{sc} as a percentage of the total execution time (t.e.t.) of the algorithm.

Number of processors	Tsc (% of t.e.t.)
4	13.8
16	36.7
64	67.8
256	85.3

Table 8.4 Table of the proportion of time spent in interprocessor communication for ARCH-1 for 4,16,64 and 256 processors

Both of the above tables suggest that the optimum number of processors for this configuration is 16 since little is gained by adopting a 64-processor configuration with this algorithm. One can see that ARCH-1 represents an improvement in terms of the execution time over the Master/Slave configuration, even though the amount of time spent communicating increases as N increases. This is because T_{sc} has now been distributed over the N processors.

For optimum efficiency, the data transfer must be explicitly stated in the algorithm (although details about the transfer can be hidden from the user). One advantage of this configuration is that expansion of the processors is straightforward as only four links are involved. As before, we can derive a mathematical model. Here,

$$T_{tot} = T_p/N + T_s/N + T_K$$

where the symbols represent their meaning as defined before. This produces the model

$$T_{tot} \simeq 100/N + 10$$

for configurations composed of up to 64 processors. Comparing this with our previous equation of 89/N+28, we can see that ARCH-1 is in fact faster for all realisable N; however, the model breaks down for 256 processors. A more complex model is therefore required as T_{sc} is a function both of N and of the amount of data to be transferred. The discrepancy can be explained by a term in N^2 arising from interprocessor communication. However, as stated earlier, the above model will suffice for up to 64 processors. From this and Table 8.3 and 8.4, we can deduce that T_{sc} is ≃5.5ms and T_{pi} is ≃4.5ms.

8.4.3 Topologies Related to ARCH-1

The topology of the above system maps well onto the transputer. The transputer is a RISC processor with four independent serial links. Unlike SIP, the transputer can transmit and receive data from all four links simultaneously, independent of the processor. However, in spite of the immediate attraction of the transputer, the above analysis shows that a data bottleneck will exist with a large number of transputers: further investigation of this problem is beyond the scope of this thesis.

The following sections investigate two architectures to attempt to ameliorate the problem of explicit communication and performance degradation for a larger number of processors.

8.5 CONFIGURATION 3 - ARCH-2

This configuration is based on the idea that all processors should have access to the same required information without memory contention. Here, each processor transmits its data (whether in the form of a result or a pixel value) to all other processors simultaneously. The next section proposes a solution to this problem.

8.5.1 Improving the Communication Bottleneck

In order to allow each processor transmit its data to all other processors without memory contention, we shall apply the same principle as that in Section 8.4.1 in that each processor has access to a dual-port RAM on-board a processor and each executes the same task. However, in this case each processor has access to a RAM on all processors in the system, via a set of data buses. Clearly, for a large system this approach would break down, but in what follows, we pursue the idea in order to see how practical it is for moderate-sized systems, and at what stage it actually breaks down.

If we consider that every processor has access to two image planes, a data RAM (for local storage such as arrays) and N-1 Communication RAMs (CRAMs) - one CRAM (C_n) for each processor as depicted in Figure 8.8a for a 4-processor configuration. Here, each processor's CPU (P) outputs data onto the bus which is simultaneously received by all N-1 CRAMs connected to the bus. (Note that a CRAM must be at least the size of the section of image that the processor operates on.) Thus, for a parallel task (e.g. a Sobel), a processor's portion of an image can

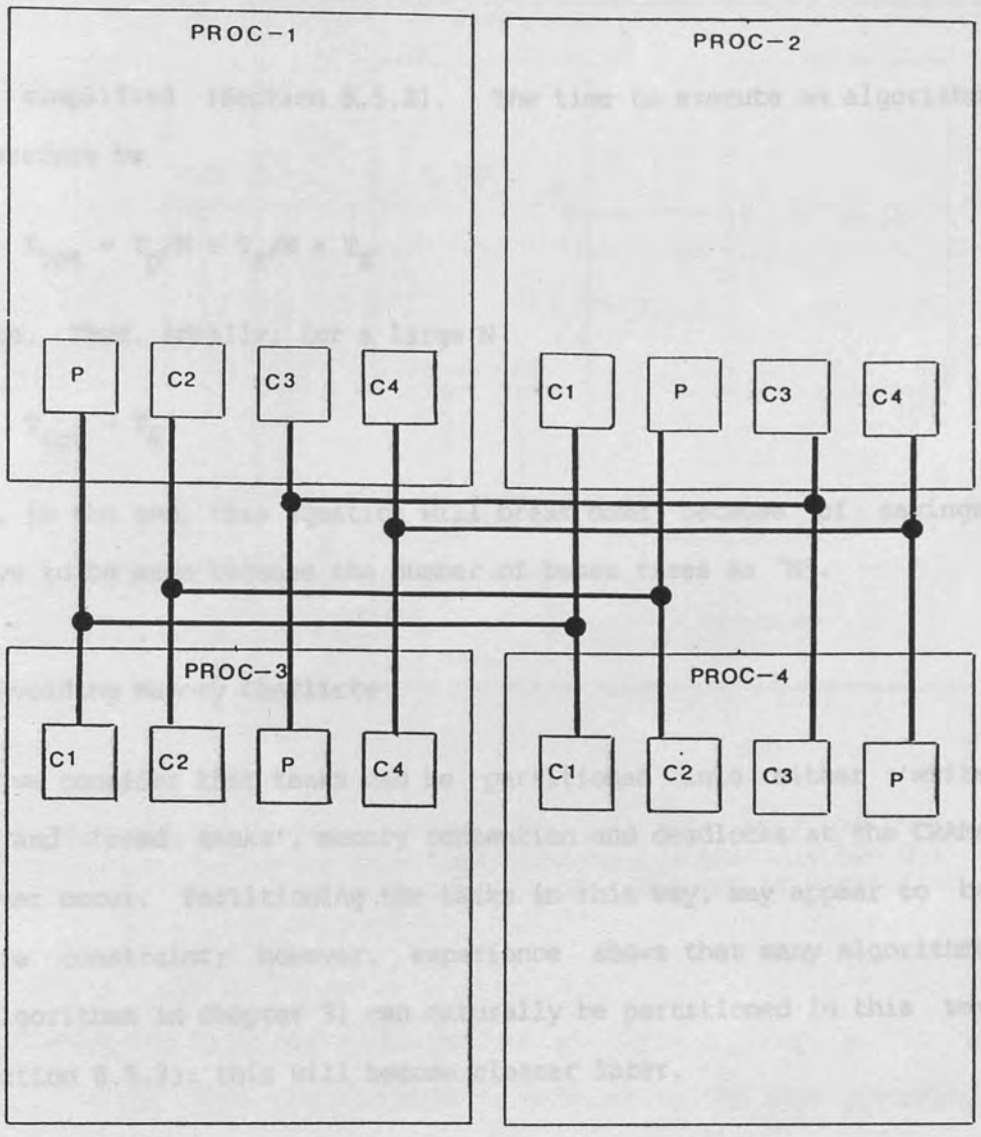


Figure 8.8a ARCH-2 - each processor can transmit its data to all other processors simultaneously via the CRAMs

essentially be transferred to all other processors simultaneously. Typically, when a processor is required to read data from the CRAMs, it will cycle through its on-board CRAM, either storing the results or manipulating the data directly (each processor either reads or writes data since each is executing the same task). Because only one processor can write to a CRAM and only one processor can read from it at any particular moment in time, the logic for the communication circuitry is

greatly simplified (Section 8.5.2). The time to execute an algorithm will therefore be

$$T_{tot} = T_p/N + T_s/N + T_K$$

as before. Thus, ideally, for a large N

$$T_{tot} \simeq T_K$$

However, in the end, this equation will break down because of savings that have to be made because the number of buses rises as $\sim N^2$.

8.5.2 Avoiding Memory Conflicts

If we consider that tasks can be partitioned into either 'write tasks' and 'read tasks', memory contention and deadlocks at the CRAMs will never occur. Partitioning the tasks in this way, may appear to be a severe constraint; however, experience shows that many algorithms (c.f. algorithms in Chapter 3) can naturally be partitioned in this way (see Section 8.5.3): this will become clearer later.

Figure 8.8b represents the control logic for one of the CRAMs. During a 'write task', data to be transmitted (here from processor i) is enabled onto the CD bus with the appropriate control signals, so all other processors receive the data. (Note that just processor j (which also has N-1 CRAMs on-board attached to N-1 processors) is depicted but in fact is replicated N-1 times.)

Control of the multiplexor and the buffering is controlled by the receiving processor (here processor j). Each processor sets its CCNTRL bit in its microword low throughout the execution of a 'write to CRAM task' enabling external processors to write data into the CRAMs. This would then be set high during a 'read from CRAM task', enabling the local processor to read from the CRAMs. The advantage of this is that

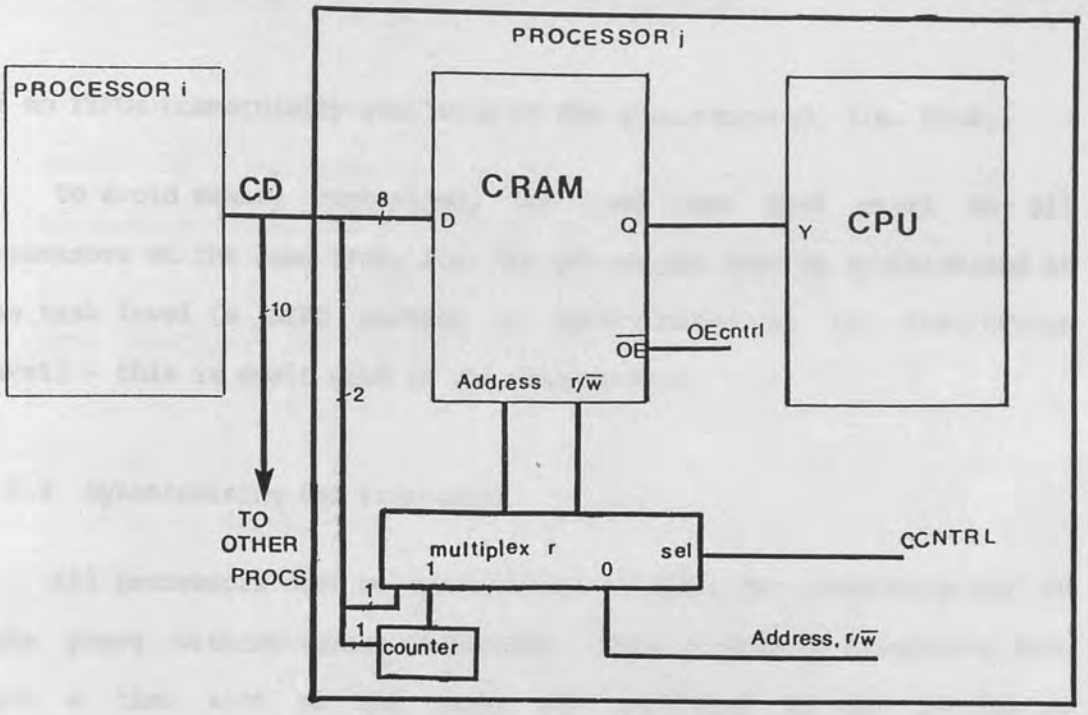


Figure 8.8b Dual-ported logic for the CRAM. Only one CRAM is depicted here; however, processor j has N-1 CRAMS on board and each processor is replicated N times

partitioning the tasks and code generation from a compiler is made easier. (Note that the same task is executed in the same processor in the same time slot, yet the processors still remain autonomous in the execution of each task; thus, at any one time, all CCNTRL bits in the system will be high or low as appropriate.) Logic is also kept to a minimum because only external processors require access to the 'D' port of the CRAM and local processors only require access to the 'Q' port. Another important point to note is that one of the control lines supplies the clock to a counter which supplies the address to the CRAM during a 'write task'. This minimises the number of wires for the links as data will usually (it is reasonable to assume) be written sequentially. All counters are cleared by the local processor at the start of a 'write task'. (Note that the CRAM is in effect acting as a large FIFO. This is needed as (at the time of writing) there appear to

be no FIFOs commercially available of the size required, i.e. 8Kx8.)

To avoid memory contention, the same task must start in all processors at the same time, i.e. the processors must be synchronised at the task level (a SIMD machine is synchronised at the instruction level) - this is dealt with in the next section.

8.5.3 Synchronising the Processors

All processors must be synchronised in order for communication to take place without memory contention. This is done by allocating each task a time slot so the tasks are initiated in all processors simultaneously. After the completion of every task (each processor is executing the same task independently), each processor sets a flag and waits until all other processors have finished their respective tasks (detected by all flags being set). The next task is only invoked when all processors have completed their respective tasks. If the processors were not synchronised, a processor with little processing in a write task may finish before the others. If the second task involves fetching the data from the CRAMs (a read task) and the tasks were not synchronised, the second task will be initiated and the processor may attempt to fetch non-existent data, since the other processors have not finished the previous task. WARP adopts a queue and efficient compilation of code to solve this problem while PASM involves interrupts. However, by allocating time slots to each task, hardware and software are kept to a minimum with little loss in efficiency. This is because:

1. If a parallel task is being executed (e.g. a Sobel) then all processors will take approximately the same amount of time.

2. If a sequential task is being executed (which often takes a small amount of time relative to a parallel task), then the time lag between processors finishing will be relatively small anyway.

The hardware for synchronising the processors is relatively straightforward. As each processor completes its respective task, it sets a flag (in effect a microcode bit). All flags from all processors are logically ANDed where the output is available at each processor's condition code register. Thus, on completion of the task, the processor sets its flag and executes the following loop:

```
REPEAT UNTIL all_flags_set=TRUE
```

Thus, the next task is only invoked when all flags have been set. The next section modifies the O-ring algorithm for this configuration.

8.5.4 The O-ring Algorithm and ARCH-2

Below is the O-ring algorithm modified to be implemented on ARCH-2. Note that the tasks are easily partitioned into 'read' and 'write' tasks. Figure 8.9 depicts the graph of the execution times for this configuration.

1. Each processor applies a Sobel to its allocated section of image in P-space. This not only writes it locally but also to all its associated CRAMs on all other processors.

2. Q-space is reformed in each processor by reading each processor by reading the data from all of the local CRAMs. Thus, in a 4-processor system, processor one will read data from the CRAMs associated with processors two, three and four.

3. Each processor now scans the reformed image (each has its own copy); however, the area scanned is limited to the processors's allocated area of image. The coordinates of those points with a value greater than a fixed threshold (peaks) are stored locally on each processor. These are the possible centres of the O-rings in the processor's image area.

4. Each processor sorts its list of peak values into a decreasing order.

5. Starting with the highest value, determine the centres of the rings, ignoring any points within a fixed distance of each other.

6. For each centre found, apply a filter in Q-space to eliminate noise.

7. Put true centre in a "Centre found" list.

8. Redistribute "Centre found" list in order to determine those centres that lie on the border between two processors. Executed in time T; however, this is usually negligible.

The O-ring algorithm for ARCH-2

From inspection of the Table 8.5, a system with more than 16 processors is unlikely to be cost-effective. In order to quantify these results, a C*T test is carried out below. The extra CRAMs needed for each processor have now been taken into consideration - these are assumed to cost £4.00 each, this being the current cost of a 1 off 8Kx8 memory chip (July 1987), which is expected to be adequate for our purposes. The cost of the buses (ribbon cable) has been omitted since this is minimal compared with the rest of the system (£10 for 30m of 10-way cable - July 1987). (Here we take the view that ribbon cable will suffice for (realistically) 64 processors, since the output from each processor is buffered.) The additional cost of the components for the bus logic has now been included. These increase as $\sim N^2$ and are

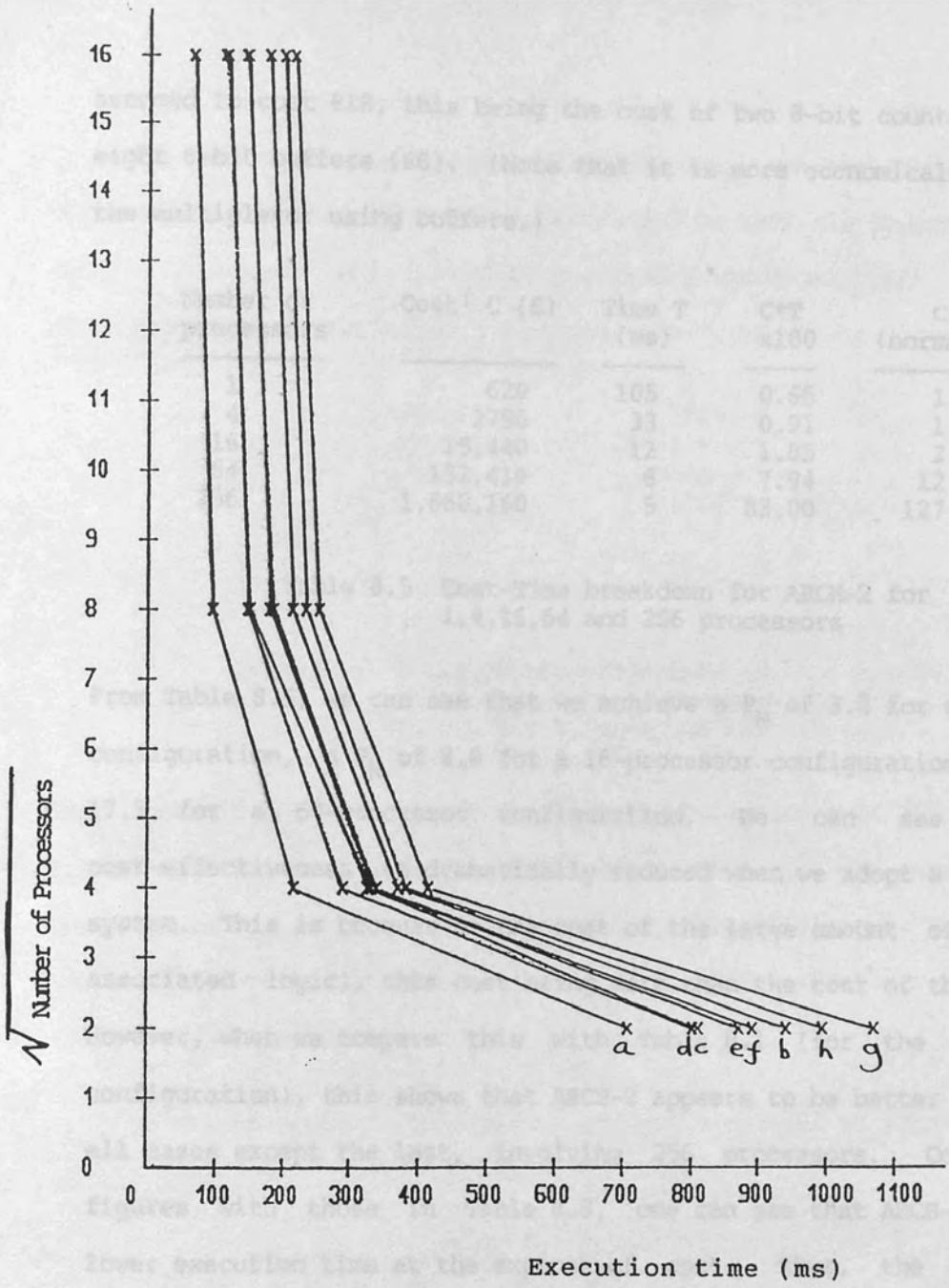


Figure 8.9 Graph of the execution times of the O-ring algorithm with ARCH-2 using 1,4,16,64 and 256 processors, operating on eight images of test data

assumed to cost £18, this being the cost of two 8-bit counters (£12) and eight 6-bit buffers (£6). (Note that it is more economical to implement the multiplexor using buffers.)

Number of processors	Cost[1] C (£)	Time T (ms)	C*T x100	C*T (normalised)
1	620	105	0.65	1.00
4	2756	33	0.91	1.40
16	15,440	12	1.85	2.85
64	132,416	6	7.94	12.21
256	1,660,160	5	83.00	127.70

Table 8.5 Cost-Time breakdown for ARCH-2 for 1,4,16,64 and 256 processors

From Table 8.5, we can see that we achieve a P_N of 3.2 for a 4-processor configuration, a P_N of 8.8 for a 16-processor configuration and a P_N of 17.5 for a 64-processor configuration. We can see that the cost-effectiveness is dramatically reduced when we adopt a 64-processor system. This is because of the cost of the large amount of CRAM (and associated logic), this cost being more than the cost of the processor. However, when we compare this with Table 8.1 (for the Master/Slave configuration), this shows that ARCH-2 appears to be better solution for all cases except the last, involving 256 processors. Comparing the figures with those in Table 8.3, one can see that ARCH-2 achieves a lower execution time at the expense of cost. Thus, the decision on whether one should choose ARCH-1 or ARCH-2 will ultimately depend of the application. ARCH-1 should be chosen when cost is the predominant factor while ARCH-2 should be chosen where speed is the predominant factor. The main advantage with 'ARCH-2' is that communication between processors can be made transparent to the user.

---

[1] This ignores such factors as reduction in price for bulk quantities and manufacturing costs, etc.

Table 8.6 depicts the percentage of the total execution time (t.e.t.) the processors spend reading the data from the CRAMs. This is independent of the number of processors and is entirely dependent on the data. Thus, as the number of processors increases, this will have a more prominent effect on the total execution time of the algorithm.

Number of processors	Tsc (% of t.e.t.)
4	2.6
16	7.2
64	14.5
256	17.6

Table 8.6 Table of the proportion of time spent in interprocessor communication for ARCH-2 for 4,16,64 and 256 processors

This shows that ARCH-2 spends less time transferring data than ARCH-1; hence, achieving a faster algorithm execution time. The reason for the decrease in T_{sc} is because ARCH-1 requires several steps to transfer the data to all processors in the system while ARCH-2 transfers the data to each processor simultaneously.

We now have enough information to derive a model as before. We may recall that

$$T_{tot} = T_p/N + T_s/N + T_K$$

Thus, for a large N, $(T_p + T_s)/N$ will be negligible; therefore, from Tables 8.5 and 8.6 we can deduce

$$T_{tot} \simeq 100/N + 5$$

where T_K is the sum of the time it takes to read the CRAMs (0.8ms) and the times of those tasks that are independent of the number of processors in the system (4.4ms). This compares favourably with ARCH-1 since the amount of time spent communicating has decreased from 5.5ms to 0.8ms. As a further step in the analysis procedure, it is interesting

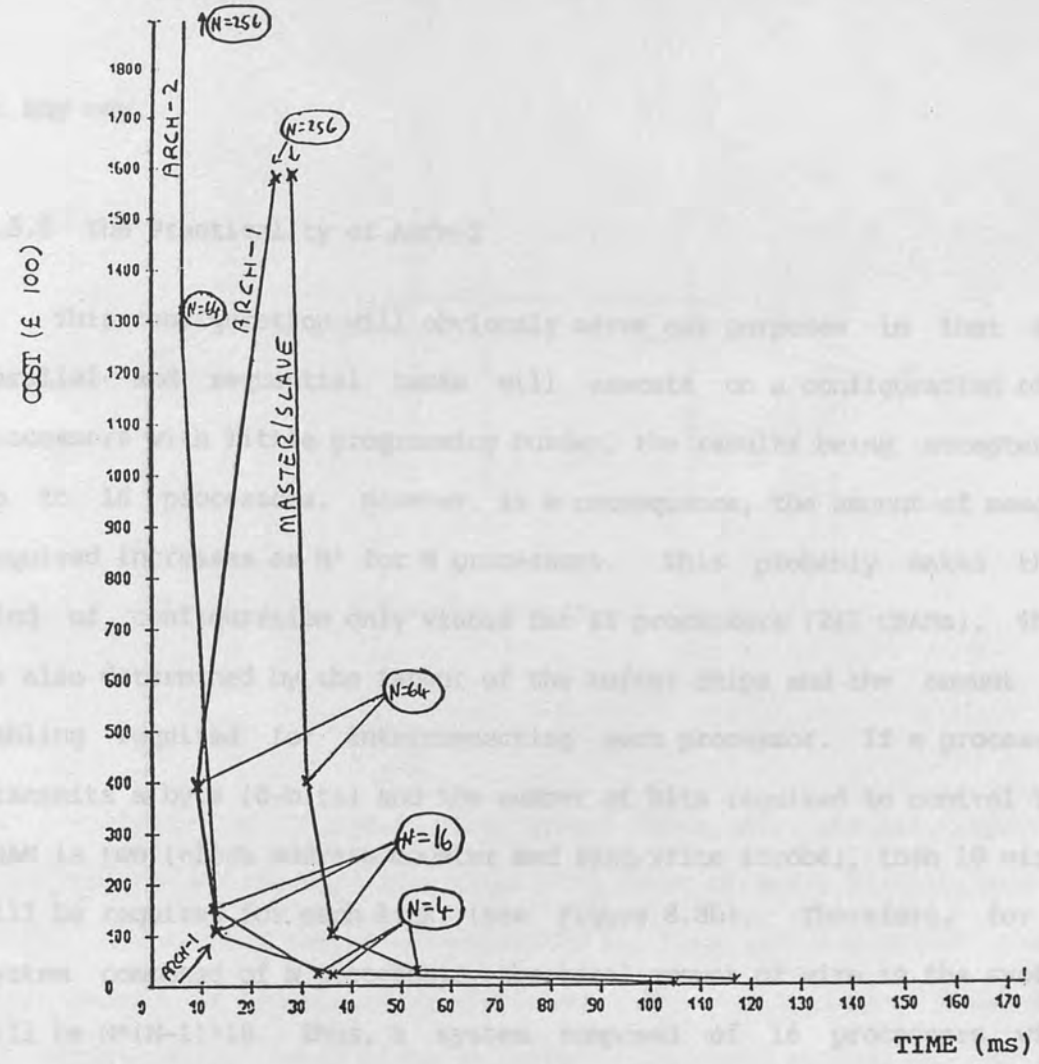


Figure 8.10 Graph of the cost of an N processor configuration (C) vs. execution time of the O-ring algorithm on that system (T). The numbers ringed indicate the numbers of processors at various points on the graph

to draw the graph of C vs. T for all three architectures (Figure 8.10). (Note that the number of processors corresponding to each point is marked on the graph.) Inspection of the graph shows that ARCH-2 will, in general, will execute the O-ring algorithm faster than the other architectures, simulated with the same number of processors. However, a 64-processor ARCH-2 costing £132,416 has little speed improvement over a 64-processor ARCH-1 (£39,680), although the cost has trebled. Thus, for 64-processors, one should consider ARCH-1. For 16 processors, the costs vary relatively little; thus, in this case, ARCH-2 could probably be justified. The Master/Slave configuration appears not to be beneficial

in any way.

8.5.5 The Practicality of ARCH-2

This configuration will obviously serve our purposes in that all parallel and sequential tasks will execute on a configuration of N processors with little programming burden, the results being acceptable up to 16 processors. However, as a consequence, the amount of memory required increases as N^2 for N processors. This probably makes this kind of configuration only viable for 16 processors (240 CRAMs). This is also determined by the fanout of the buffer chips and the amount of cabling required for interconnecting each processor. If a processor transmits a byte (8-bits) and the number of bits required to control the CRAM is two (clock address counter and read/write strobe), then 10 wires will be required for each link (see Figure 8.8b). Therefore, for a system composed of N processors, the total amount of wire in the system will be N*(N-1)*10. Thus, a system composed of 16 processors will require 2400 wires, which is not too excessive.

8.5.6 Topologies Related to ARCH-2

ARCH-2 is similar in concept to the Heidelberg POLYP system [7] in that it is a multi-bus configuration. POLYP is based on the same concept as ARCH-2 in that any processor can access any other processor's data. A schematic diagram of the POLYP system is given in Figure 8.11. Here, if a processor (a 68000 processor in the actual implementation) is required to access another processor's memory, a bus grant is made to the bus arbitration logic. This allocates one of the buses (a POLYBUS) to the requesting processor. Communication between two processors can then proceed until the bus is released by the processor.

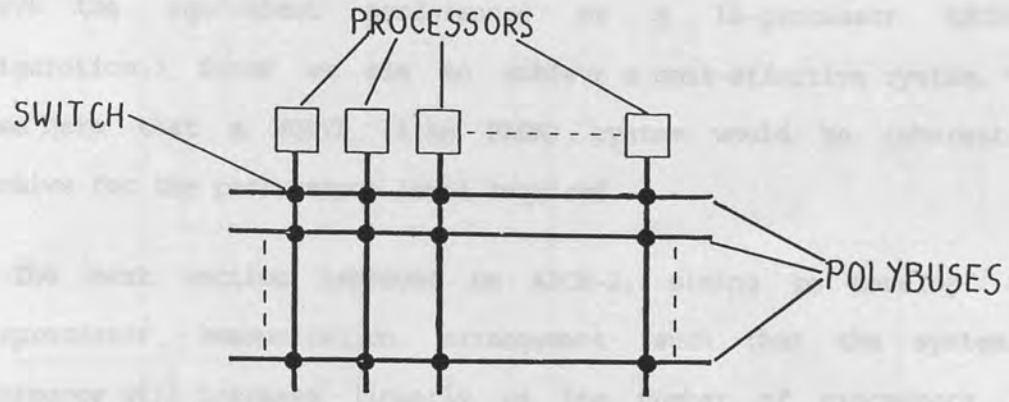


Figure 8.11 Schematic diagram of the POLYP system

Although the concept is the same, there are several important differences in the implementation. The first of these is that, whereas ARCH-2 has one bus per processor, POLYP can have any number of buses, independent of the number of processors. The performance of POLYP thus increases as the number of buses increases, up to a maximum of one bus per processor. However, a POLYP processor consists (typically) of five boards whereas an ARCH-2 processor consists of two boards (including CRAM) up to a 16-processor configuration. Therefore, a tradeoff between the number of boards per processor and the number of buses per processor exists. For a large number of processors (~100), the POLYP system would probably be a more cost-effective (and realistic) solution; however, for a small processor system (~16), ARCH-2 may be more cost-effective.

An important point to note is that POLYP and PASM are 68000 processor based systems. As we saw in Chapter 6, a SIP based processor can achieve ~14 times speed improvement over a 68000 running at 8MHz. Therefore, a large number of processors may be needed in a POLYP system to achieve comparable performance to a 16-processor ARCH-2 system. (From the figures given in [7], over 100 POLYP processors are needed to

achieve the equivalent performance of a 16-processor ARCH-2 configuration.) Since we aim to achieve a cost-effective system, we assume here that a POLYP (like PASM) system would be inherently expensive for the performance level required.

The next section improves on ARCH-2, aiming to develop an interprocessor communication arrangement such that the system's performance will increase linearly as the number of processors is increased for parallel tasks; it is also capable of executing sequential tasks quite efficiently.

8.6 CONFIGURATION 4 - ARCH-3

A more complex design based on the results of the above analysis shows that a data bottleneck still exists in the system because of Task-2 in the algorithm. An improvement on this architecture is presented in Figure 8.12 for a 4-processor configuration. It consists of Image RAM (IRAM) physically partitioned into N individual chips for an N processor system, so each processor in the system has unique 'write only' access to only one of the chips. This corresponds to its allocated section of image; however, the N chips appear as a contiguous block of memory to the local processor. For example, if four processors were operating on a 128x128 image, the image would be partitioned into four 4Kx8 memory chips (Figure 8.12). Each section of image (except the section being written to by the local processor, e.g. IRAM4 in Figure 8.12) is either written to by an external processor or read from by the local processor. The control section is organised in a similar way to the CRAMs in that a multiplexor is controlled by the local processor. This way, all processors have access to their sections of image without contention. (Note that problems will occur at the borders of an image. Here, we will assume that the borders are available without memory contention (Section 8.2.2)).

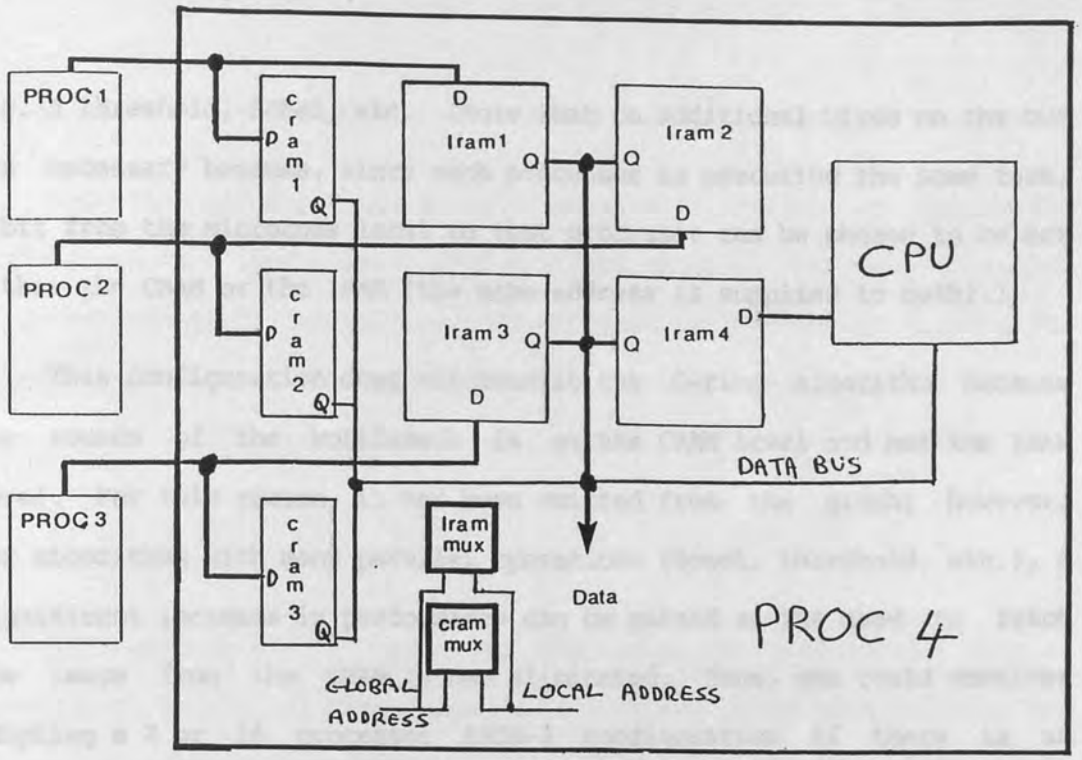


Figure 8.12 ARCH-3 - Each processor has access to a section of the physically partitioned image ram and a CRAM on board each processor

Here, tasks are again partitioned into read tasks and write tasks as in ARCH-2. This ensures that the local ALU (Figure 8,12) will never read one of the other IRAMs while they are being written to; thus memory contention and deadlocks are avoided. ARCH-3 also contains CRAMs as before for non-image data (e.g. a list of edge coordinates).

As a processor writes to its allocated IRAM (e.g. IRAM4 for PROC4 as in Figure 8.12), it simultaneously writes to its equivalent section of IRAM on all other processors. However, the main difference between ARCH-3 and ARCH-2 is that the need to read the CRAMs in order to reform the image (as what would have to be done using ARCH-2) is now eliminated, since a processor's section of image is transferred to all other processor's IRAM simultaneously. Thus, the processing time decreases linearly with the number of processors for parallel tasks,

e.g. a threshold, Sobel, etc. (Note that no additional wires on the bus are necessary because, since each processor is executing the same task, a bit from the microcode local to that processor can be chosen to select either the CRAM or the IRAM (the same address is supplied to both).)

This configuration does not benefit the O-ring algorithm because the source of the bottleneck is at the CRAM level and not the IRAM level. For this reason, it has been omitted from the graph; however, for algorithms with many parallel operations (Sobel, threshold, etc.), a significant increase in performance can be gained as the need to fetch the image from the CRAM is now eliminated. Thus, one could consider adopting a 4 or 16 processor ARCH-3 configuration if there is an anticipated large number of parallel tasks to be executed. (To calculate the cost of an ARCH-3 system, add £4*N*(N-1) (for buffers) to the cost of an ARCH-2 system.) However, sequential tasks can still be executed quite efficiently (relative to a SIMD machine) for (practically) up to 16 processors, thus offering distinct advantages over a SIMD machine.

8.7 CONCLUSIONS

The choice between the configurations is ultimately application dependent. For industrial inspection, a 4/16-processor ARCH-1 system may be desired for practical reasons as this is more easily configurable and will consume less power. For more complex tasks such as industrial 3-D inspection where a higher throughput is required, ARCH-2 or ARCH-3 may be chosen with up to 16 processors - the choice depending on the cost and the execution time required. ARCH-2 is more economical than ARCH-3 as it does not require the partitioning of the IRAMs; however, ARCH-3 with 16 processors gives a greater throughput for parallel operations. ARCH-2 and ARCH-3 thus have distinct advantages over a SIMD machine because sequential tasks can still be executed quite efficiently

(although a bottleneck does exist), and yet remain comparable in cost.

Both PASM and POLYP appear too expensive to achieve a comparable performance of a 16-processor ARCH-2/3 configuration; however, this area of investigation is incomplete since no costs or performance figures for either system seem to be available. They may well prove to be more cost-effective if a large number of processors (>100) are involved.

8.8 SUMMARY

This chapter has investigated the idea of configuring several autonomous sequential processors capable of executing both parallel and sequential tasks with little programmer burden. A simulation of various multiprocessor topologies was undertaken which, although a topology intuitively appeared sound, in fact incurred data bottlenecks. From an analysis of the source of the bottlenecks, successive attempts to reduce the bottleneck were carried out by adopting a configuration based on the results of the previous configuration. This eventually led to ARCH-2 and ARCH-3. Both of these incurred a bottleneck which was acceptable up to 64 processors (although practically, a 16-processor version would probably be the limit). During the analysis of each architecture, a C*T test aided us in our decisions along with a simulation of the architecture for 4, 16, 64 and 256 processor configurations using the O-ring algorithm.

Whether ARCH-1, ARCH-2 or ARCH-3 should be chosen will ultimately depend on the application, the cost and the execution time required. The bit-slice processor SIP was used merely to provide a fast and cost-effective means of investigation. Machines like PASM and POLYP, although powerful in their own right, are likely to be too costly for industrial inspection.

Apart from designing elaborate multiprocessor architectures, one should also consider optimising the processors themselves. By upgrading the processors and decreasing the access time of the memory chips, a significant speed up in performance can be obtained without changing the architecture. This could provide a basis for a 'standard' architecture while relying on technology for further performance.

CHAPTER 9

CONCLUSIONS AND SUGGESTIONS FOR FURTHER WORK

"It isn't that they can't see the solution.
It is that they can't see the problem"
The Point of a Pin in The Scandal of Father Brown
(London: Cassell, 1935)

9.1 INTRODUCTION

The first part of this chapter reviews the inspection algorithms given in Chapter 3. Following this is a study of the SIP machine and the Linear Array Processor (LAP) which were developed to be part of a multiprocessing system. Subsequent sections summarise the results derived from both of these machines (including the results from the implementation of the above algorithms) and how they relate to the industrial inspection area. An overview of the experimental work carried out on a series of multiprocessor configurations is also given. The final part of this chapter describes current limitations and makes suggestions for future work.

9.2 INDUSTRIAL INSPECTION ALGORITHMS

Chapter 3 described two industrial inspection algorithms, one to inspect O-rings (circular objects) and the other to inspect chocolate

biscuits (rectangular objects). Both of these algorithms took of the order of a few seconds to execute on a PDP-11/73 operating on a 128x128 image. Although both algorithms employed different inspection methods, the same fundamental technique was used for deriving the initial measurements (calculating the centre in the case of the O-ring, and determining the orientation of the biscuit) namely, the Hough transform. This provided a robust, accurate and inherently fast method for extracting information from an image. The general conclusion is that the Hough transform is well suited for industrial inspection purposes.

Both of these algorithms contained a high degree of internal sequentialism and parallelism; hence, it is difficult to execute either efficiently on most machines. It is generally accepted that it is easier to implement a parallel algorithm sequentially than it is to implement a sequential algorithm in parallel. Also, the cost of a sequential processor is generally much less than that of a parallel processor. One would therefore expect that the sequential implementation of a parallel algorithm on a sequential machine would produce a highly cost-effective solution.

In order to investigate this further, Chapter 4 adapted the chain code (which is generally restricted to binary images) to the grey-scale case. This involved the introduction of a novel method, again using the Hough transform, for improving results derived from the chain code when applied to noisy images. When this was applied to the edge extraction task in both algorithms in Chapter 3 (which are essentially parallel tasks), it was found that a reduction in execution time of the order of 5 times was obtained for extracting the edges and all relevant information. This produced an overall reduction in execution time for the whole algorithm of the order of 2-3 times. This showed that a definite speed improvement could be obtained by implementing an algorithm sequentially on a sequential processor, while incurring little

loss in accuracy, and led to the question of whether a parallel processor is appropriate for industrial inspection. Ultimately, inspection algorithms have to run in real time, this being governed by the line speed, but this compares to "large" pixel rates of the order 10^6/second and not "huge" rates of the order 10^8/second.

9.3 SIP - A BIT-SLICE IMAGE PROCESSOR AND THE LAP

Chapter 5 described a high-speed, microcoded bit-slice Sequential Image Processor - hence the acronym SIP. This was designed with two objectives in mind:

1. To design a processor based system capable of efficient execution of frequently used image processing functions. The hardware was designed with an image processing interface and the flexibility to execute several instructions concurrently.

2. To design a processor as a cost-effective solution for use in industrial inspection, i.e. capable of achieving inspection at typical industrial product rates while remaining affordable.

Essentially, it was a processor designed to execute algorithms such as those described in Chapter 3 at rates of 5-10 products/second. However, it was also designed to be combined with a parallel processor, namely the LAP described in Chapter 6, for inclusion in part of a multiprocessor system.

The final design was based around the AMD 29203 bit-slice processor. This was combined with a dedicated multiplier chip and a pipelined image plane, enabling the majority of instructions to be executed in a single cycle (125ns). Performance figures showed that SIP achieved a gain in performance of 25-30 times when compared with a PDP-11/73 for certain types of image processing algorithms.

Implementation of the algorithms described in Chapter 3 on SIP showed that SIP executed these algorithms in ~130ms - including I/O. This corresponds to the inspection of 7-8 products/second - a speed which is suitable for many industrial inspection applications - and shows that bit-slice designs can provide cost-effective solutions for automated industrial inspection systems. Indeed, it is noteworthy that industrial processes rarely work at rates exceeding 30 products/sec.

As mentioned before, SIP was also designed to be combined with the Linear Array Processor (LAP) - the LAP would execute the parallel tasks of an algorithm while SIP would execute the sequential tasks. Such a configuration seems to be highly attractive for situations where an algorithm consists of many parallel tasks and sequential tasks, and where a single sequential processor is incapable of executing the algorithm at a suitable rate. Initial estimated results showed that a quite large bottleneck was likely to occur for many image processing algorithms because SIP was required to transfer the data to and from the LAP; however, these results were deduced from LAP-I performance data while the actual implementation was to occur with the LAP-II. The LAP-II has a superior performance to the LAP-I; however, because the detailed performance of the LAP-II is as yet unknown, this investigation could not be completed.

As mentioned before, SIP was designed to execute several instructions concurrently. Performance figures showed that a 30% reduction in code with a corresponding 30% reduction in execution time was achieved with many of the routines in Chapter 6. This gave a favourable SIP/LAP ratio of execution times of about eight.

9.4 MULTIPROCESSOR ARCHITECTURES

Chapter 7 reviewed a series of multiprocessor architectures. This

attempted to show that the systolic array approach appeared to be beneficial for image processing while the SIMD and MIMD approaches were inadequate for typical algorithms. (Typical algorithms are here assumed to include both parallel and sequential tasks.) The systolic array approach maintained the processing power of MIMD by having autonomous processors (hence allowing execution of sequential algorithms), while maintaining the simplicity of SIMD in that each processor executes the same instructions. Machines such as DIPOD, PASM and POLYP could adopt this approach, although these were cited as being rather too expensive for our purposes. A notable example was that of DIPOD where a single processor costs of the order of £30,000. However, their architectures enabled us to highlight some important points about multiprocessor systems. Indeed, when several hundred processors are to be configured together, these systems may become more feasible than say, several hundred SIPs.

Chapter 8 extended this idea of combining sequential processors together, in an attempt to investigate several multiprocessor configurations, and capable of executing "typical" algorithms. At each stage of the process, an analysis of the architecture with the O-ring algorithm (chosen as a typical algorithm) using 4, 16, 64 and 256 processors was carried out. This allowed us to locate the bottlenecks in the system and to carry out further analysis. Note that these architectures were simulated on a PDP-11/73 which has an entirely different architecture to that of SIP. However, analysis of the simulation results at each stage enabled us to model the results and show that the models and the theoretical values gave very good agreement, indicating that the simulations were in fact quite realistic.

Two main architectures were derived: ARCH-1 and ARCH-2. ARCH-1 was based on the philosophy "every processor is connected to its four nearest processors" while ARCH-2 was based on the idea that "every processor is connected to every other processor". Dual-port RAMS were used in ARCH-1 for interprocessor communication while ARCH-2 used Communication RAMs (CRAMs). Here, each processor in an N processor ARCH-2 configuration had write-only access to a CRAM on each of the other N-1 processors. After transmitting the data (image or non-image data), each processor read its local CRAMs to obtain the results from the other processors. Partitioning tasks into read and write tasks eliminated memory contention and deadlock.

ARCH-2 achieved a higher execution rate than ARCH-1 at the expense of cost and flexibility; however, many parallel tasks would have caused quite a large bottleneck because of the large amount of data that would have to be read from the CRAMs. This would have been particularly severe if there were a large number of processors involved, since the time to read the data from the CRAMs is independent of the number of processors in the system, and constant for a particular image. In order to improve on this, ARCH-2 was modified to produce ARCH-3. ARCH-3 contained both CRAMs (for non-image data) and an IRAM (Image RAM) for image data. Here, the IRAM was partitioned into N chips for an N processor system; however, the IRAM appeared as a single image plane to the local processor. As a processor wrote to its section of image, it simultaneously transferred it to its equivalent section on all other N-1 processors, hence eliminating the need to read the CRAMs to reform an image. This meant that for ARCH-3, the processing rate could, in the case of parallel tasks, increase linearly as the number of processors was increased, and in addition the system was also capable of executing sequential tasks quite efficiently. Since tasks were again partitioned into read tasks and write tasks, memory contention at the CRAMs and the

IRAM did not occur. However, in all ARCH configurations, a large bottleneck occurred at 64 processors, which made a 64 processor system non-cost-effective. The reason for this was because the source of the bottleneck was at the CRAM level rather than the IRAM level for the O-ring algorithm.

Because the results from a sequential process may be random (e.g. a list of edge coordinates), they must be written to the Communication RAM (CRAM). Therefore, a read from a CRAM will always be invoked whenever a sequential process is executed; thus, a linear increase in performance as the number of processors is increased for sequential tasks cannot be achieved with the present configurations. A decision on whether ARCH-1, ARCH-2 or ARCH-3 should be used ultimately depends on the application and the predominant influence over the system, i.e. speed or cost. A graph of the cost of these architectures against the execution times enabled us to draw the following conclusion: for 4 processors, ARCH-2 should be chosen; for 16 processors, ARCH-1 should be chosen. However, if parallel tasks are to be common (e.g. Sobel), one could consider adopting a 16 processor, ARCH-3 configuration.

9.5 SUGGESTIONS FOR FURTHER WORK

SIP has shown that a bit-slice architecture provides a means of producing a high-speed, cost-effective machine suitable for industrial inspection purposes. During the development of SIP, various possible enhancements emerged which are to be incorporated on the next version - SIP-II. These reflect the technological changes and the reduction in cost of many of the components used, and the experience gained since the development of SIP. The basic architecture (interconnections between the components) would remain the same except for the following changes:

1. A more powerful bit-slice. Either the Texas's 16-bit SN74AS888 or IDT's 49C404 32 bit-slice processor.

2. A more sophisticated program sequencer able to support interrupts will be used (as opposed to the AMD 2910A currently used in SIP).

3. A clock generator and microcycle length controller, hence producing a variable system clock.

4. 256x256 image planes as standard.

5. 32K or 64K (words) of RAM.

6. A shortened microword width achieved by a combination of horizontal and vertical microcoding.

7. Provision for the user to define mnemonics and the corresponding microcode instructions relatively easily.

8. Capability of doubling as a frame store, capable of data acquisition, display and manipulation. This will allow SIP to operate on incoming data from the video signal hence achieving true real-time processing.

9. A private interconnect bus for communication with other SIP-IIs.

Although the above features sound very attractive, an important aspect of a commercial product is board space and cost. The prototype of SIP involved a relatively large amount of miscellaneous logic. It should be possible to reduce board space significantly on the current version by the use of programmable logic devices such as PALs, etc. However, additional functionality as described in the above list would increase the board space and cost. It is estimated that SIP-II will occupy the same amount of board space as SIP-I with an increase in cost of about 30% and a 30-40% corresponding increase in performance. This will give

an approximate cost of about £1000 (1-off cost-price) for SIP-II and shows the suitability of bit-slice architectures and programmable logic in high-speed image processing systems.

When multiple SIPs are to be configured in relation to the four configurations investigated, SIP-II would be particularly suited for single processor based systems and adequate when configured with 4 and 16 processors. When more processors are to be configured together, processors such as the transputer may offer a more cost-effective solution than multiple SIPs, and we are currently investigating this possibility.

We have shown that autonomous sequential processors in a system can decrease the execution time of an algorithm and perform parallel and sequential tasks quite efficiently. However, under the present conditions, a large data bottleneck is produced for a large number of processors during sequential tasks. One should therefore concentrate on developing more efficient methods for interprocessor communication in order to reduce this bottleneck. The CRAM may not be the most efficient method of communicating non-image data.

9.6 CONCLUSIONS

The main aim of this thesis was to produce a cost-effective bit-slice processor that, when combined with a particular type of parallel processor (namely the LAP), would achieve a throughput suitable for real-time industrial inspection at a fraction of the cost of equivalent multiprocessor systems. Several industrial inspection algorithms were to be implemented in order to analyse the system more fully. The Hough transform was cited as a useful method for use in industrial inspection, and was well suited for such a configuration. Unfortunately, because of delays with the LAP-II being built at the NPL,

it has not proved possible to make a full implementation of the target system. However, it has been shown that SIP is capable of real-time execution of industrial algorithms (namely those in Chapter 3) as a stand-alone processor.

After an attempt to combine multiple, autonomous sequential processors all executing the same algorithm, it was shown that architectures along the guidelines of "every processor has access to the same required information at all times, yet remains autonomous" are quite practicable and appear suitable for many image processing algorithms. However, much work has yet to be carried out in this area. This thesis serves to provide some initial results for four configurations that have been investigated; an important conclusion is that the configuration of processors of each architecture investigated is application dependent. Ideally, one would like to build 16 SIP-II's and combine them in order to investigate these architectures more fully - this should be the next step. However, in the present work, these architectures had to be simulated on a PDP-11/73 and the results were encouraging for small numbers of processors.

At this point it is clear that there are no general purpose architectures for which the performance increases linearly with the number of processors for all algorithms. We have therefore sought to find an architecture that is efficient and cost-effective at the lower end of the cost scale, using modest numbers of processors.

ACKNOWLEDGEMENTS

I am indebted to my supervisor Dr. Roy Davies for his constant encouragement, guidance and perseverance. Next, I would like to express my gratitude to the National Physical Laboratory for sponsoring the project, in particular Dr. Piers Plummer of the NPL for his valuable advice and assistance throughout the project. I am also grateful to Mr. Adrian Johnstone for his helpful suggestions on hardware early on in the project and for looking at the final draft. Finally, I would like to express my gratitude to Rosalind Singer for her help in bringing the whole thesis together.

Glossary

ARCH-1 - a configuration where each processor is connected to its four nearest neighbours.

ARCH-2 - a configuration where each processor is connected to all other processors via a bus.

ARCH-3 - another configuration where each processor is connected to all other processors via a bus (however, this differs from ARCH-3 in its internal organisation).

CRAM - a RAM which is used for interprocessor communication of data in the case of ARCH-3 and data and image in the case of ARCH-2.

IRAM - an Image RAM which is used for interprocessor communication of image data in the case of ARCH-3.

LAP - a microcoded bit-slice parallel processor.

Master/Slave - a configuration consisting of N+1 sequential processors, N of which operate on a section of the image executing a parallel task (slaves) while one (master) executes the sequential tasks.

P/Q space - two 128x128 image planes identified by the letters P and Q.

Parallel task - an operation which, in principle, can be applied to each pixel simultaneously.

Sequential task - an operation which is carried out pixel by pixel, and the result for each pixel depends on the results from previous pixels.

SIP - a microcoded bit-slice sequential processor.

T_s - execution time of a sequential task simulated on a sequential machine.

T_p - execution time of a parallel task simulated on a sequential machine.

T_{all} - sum of the execution times of the sequential tasks and the parallel tasks simulated on a sequential machine.

T_{sc} - time to transfer data from one processor to another along a private bus.

REFERENCES

1. Abdou, I.E. and Pratt, W.K. (1979)
"Quantitative Design and Evaluation of Enchancement/Thresholding Edge Detectors"
Proc. IEEE, Vol. 67, No. 5, May 1979, pp. 753-763

2. Agin, G.J. (1980)
"Computer Vision Systems for Industrial Inspection and Assembly"
IEEE Computer, May 1980, pp. 11-20

3. Annaratone, M. et. al., (1986)
"WARP Architecture and Implementation"
IEEE Computer, 1986, pp. 346-356

4. Arvind, D.K., Robinson, I.N. and Parker, I.N. (1983)
A VLSI Chip for Real-Time Image Processing"
Proc. IEEE Int. Symposium in Circ. and Sys., May 1983, pp.405-408

5. Ballard, D.H. (1981)
"Generalizing the Hough Transform to detect Arbitrary Shapes"
Pattern Recognition Vol. 13, No. 2, 1981, pp. 111-122

6. Barnes, G.H. et. al. (1968)
"The ILLIAC IV Computer"
IEEE Trans. Comput., C-17, No. 8, Aug. 1968, pp. 746-757

7. Bartels, P.H., Männer, R., Shoemaker, R.L., Paplanus, S. and Graham, A. (1986)
"Computer Configurations for the Processing of Diagnostic Imagery Histpathology""
Evaluation of Multicomputers for Image Processing, ed. Uhr et. al. , Academic Press, U.K., 1986, pp. 239-278

8. Basu, A. (1987)
"Parallel Processing Systems: A Nomenclature Based on Their Characteristics"
IEE Proceedings, Vol. 134, Pt. E, No. 3, May 1987, pp.143-147

9. Batchelor, B.G. and Cotter, S.M. (1984)
"Inspecting complex parts and assemblies"
Proc. 4th Int. Conf. Robot Vision and Sensory Controls, ed. A. Pugh, pp. 447-468

10. Billig, R and Cronk, R. (1986)
"A System/Architecture Approach to Microcomputer Benchmarking"
Digital Information Sheet, 1986

11. Bolles, R.C. and Cain, R.A. (1983)
"Recognising and Locating Partially Visible Objects: The Local-Feature-Focus Method"
Robot Vision, ed. A. Pugh, IFS Publications, U.K., 1983

12. Brook, R.A. and Purll, D.J. (1979)
"On-Line Image Acquisition and Analysis for Automatic Product Inspection"
Inst. Phys. Conf. Ser., No. 44, Chapter 4, 1979, pp. 137-150

13. Cantoni, V. and Levialdi, S. (1982)
"Matching the Task to an Image Processing Architecture"
Proc. 6th Int. Conf. on Pattern Recognition, Munich, Vol. 1, 1982, pp. 254-257

14. Cheng, H.D. and Fu, K.S. (1987)
"VLSI Architectures for String Matching and Pattern Matching"
Pattern Recognition Vol. 20, No. 1, 1987, pp. 125-141

15. Chin, R.T. (1982)
"Automated Visual Inspection Techniques and Applications: A Bibliography"
Pattern Recognition Vol. 15, No. 4, 1982, pp. 343-357

16. Chin, R.T. and Harlow, C.A. (1982)
"Automated Visual Inspection: A Survey"
IEEE Trans. Patt. Anal. Mach. Intell., Vol 4, No. 6, Nov. 1982, pp. 557-573

17. Clarke, K.A. and Ip, H. H-S. (1982)
"A Parallel Implementation of Geometric Transformations"
University College London, Internal Report 82/5, 1982

18. Courtney, J.W., Magee, M.J. and Aggarwal, J.K. (1984)
"Robot Guidance using Computer Vision"
Pattern Recognition Vol. 17, No. 6, 1984, pp. 585-592

19. Cronshaw, A.J. (1982)
"Automatic Chocolate Decoration by Robot Vision"
Robot Vision, ed. A. Pugh, IFS Publications, New York, 1982

20. Danielsson, P-E. (1981)
"Getting the Median Faster"
Computer Graphics and Image Processing, Vol. 17, 1981, 71-78

21. Dasgupta, S. and Tartar, J. (1976)
"The Identification of Maximal Parallelism in Straight Line Programs"
IEEE Trans. Comput., C-25, No. 10, Oct. 1976, pp. 986-992

22. Dasgupta, S. (1984)
"The Design and Description of Computer Architectures"
John Wiley & Sons, U.S.A., 1984

23. Davies, E.R. and Plummer, A.P. (1981)
"Thinning Algorithms: A Critique and a New Methodology"
Pattern Recognition Vol. 14, No. 1, 1981, pp. 53-63

24. Davies, E.R. (1983)
"Image Processing - its Milieu, its nature, and Constraints on the Design of Special Arcitectures ofr its Implementation"
Computer Structures for Image Processing, ed. M.J.B. Duff, Academic Press, 1983, Chap. 5, pp. 57-76

25. Davies, E.R. (1984)
"Design of Cost-Effective Systems for the Inspection of Certain Food Products during Manufacture"
Proc. 4th Int. Conf. Robot Vision and Sensory Controls, ed. A. Pugh, pp. 437-446, 1984

26. Davies, E.R. (1984)
"Circularity - A New Principle Underlying the Design of Accurate Edge Orientation Operators"
Image and Vision Computing, Vol. 2, No. 3, Aug. 1984, pp. 134-142

27. Davies, E.R. (1985)
"Radial histograms as an aid in the inspection of circular objects"
IEE Proceedings, Vol. 132, Pt. D, No. 4, July 1985

28. Davies, E.R. (1986)
"Constraints of the Design of Template Masks for Edge Detection"
Pattern Recognition Letters, No. 4, 1986, pp. 111-120

29. Davies, E.R. and Johnstone, A.I.C. (1986)
"Engineering Trade-offs in the Design of a Real-time System for the Visual Inspection of Small Products"
Proc. IMechE Conf. on UK Research in Advanced Manufacture, 1986, pp. 15-22

30. Davis, L.S. and Rosenfeld, A. (1978)
"Noise Cleaning by Iterated Local Averaging"
IEEE Trans. Systems, Man, Cybernetics, Vol. 8, No. 9, Sept. 1978, pp. 705-711

31. Deutsch E.S. (1972)
"Thinning Algorithms on Rectangular, Hexagonal, and Triangular Arrays"
Comm. ACM, Vol. 15, No. 9, Sept. 1972, pp. 827-837

32. Duda, R.O. and Hart, P.E. (1972)
"Use of the Hough Transformation To Detect Lines and Curves in Pictures"
Comm. ACM, Vol. 15, No. 1, Jan. 1972, pp. 11-15

33. Duda, R.O. and Hart, P.E. (1973)
"Pattern Classification and Scene Analysis"
Wiley, 1973, p. 271

34. Dudani, S. and Luk, A. (1978)
"Locating Straight-Line Edge Segments On Outdoor Scenes"
Pattern Recognition Vol. 10, 1978, pp. 145-157

35. Duff, M.J.B. (1982)
"Special Hardware for Pattern Processing"
Proc. 6th Int. Conf. on Pattern Recognition, Munich, Vol. 1, 1982, pp. 368-379

36. Elliott, C.J. (1986)
"High Speed Image Processing for Fingerprint Recognition"
Smiths Assoc. Tech. Report presented at Proc. 2nd Int. Conf. on Image Processing, 1986

37. Falk, H. (1976)
"Reaching for a Gigaflop"
IEEE Spectrum, Vol. 13, Part 10, 1976, pp. 65-70

38. Fan, T-J. and Tsai, W-H. (1984)
"Automatic Chinese Seal Identification"
Computer Vision, Graphics, Image Processing, Vol. 25, 1984, pp. 311-330

39. Fisher, A.L. (1986)
"Scan Line Array Processors for Image Computatuion"
IEEE Trans. Patt. Anal. Mach. Intell., Vol 3, No. 5, Aug. 1986, pp. 338-345

40. Flynn, M.J. (1972)
"Some Computer Organizations and Their Effectivness"
IEEE Trans. Comput., C-21, No. 9, Sept. 1972, pp. 948-960

41. Förster, K.D., Lohmann, A.W., Weigelt, G. (1982)
"Optical Processing - Recent Developments and Future Trends"
Proc. 6th Int. Conf. on Pattern Recognition, Munich, Vol. 1, 1982, pp. 57-65

42. Fountain, T.J. (1983)
"The Development of the CLIP7 Image Processing System"
Pattern Recognition Letters, No. 1, 1983, pp. 331-339

43. Fountain, T.J. (1986)
"Array Architectures for Iconic and Symbolic Image Processing"
Proc. 8th Int. Conf. on Pattern Recognition, 1986, 24-33

44. Fountain, T.J. (1987)
"The ICL DAP Programme"
Processor Arrays: Architectures and Applications, Academic Press, 1987, pp. 43-48

45. Freeman, H. (1961)
"On the Encoding of Arbitrary Geometric Configurations"
IEEE Trans. Comput., C-10, June 1961, pp. 260-268

46. Freeman, H. (1961)
"Techniques for the Digital Computer Analysis of Chain-Encoded Arbitrary Plane Curves"
Proc. National Electronic Conference, Part. 17, 1961, pp. 421-432

47. Freeman, H. (1974)
"Computer Processing of Line-Drawing Images"
Computing Surveys, vol. 8, No. 1, March 1974, pp. 57-97

48. Freeman, H. and Davis, L.S. (1977)
"A Corner-Finding Algorithm for Chain-Coded Curves"
IEEE Trans. Comput., C-26, No. 3, March 1977, pp. 297-303

49. Freeman, H. (1978)
"Shape Description Via the Use of Critical Points"
Pattern Recognition Vol. 10, 1978, pp. 159-166

50. Frei, W. and Chen, C.C. (1977)
"Fast Boundary Detection: A Generalisation and a New Algorithm"
IEEE Trans. Comput., C-26, No. 10, 1977, pp. 988-998

51. Fu, K.S. and Mui, J.K. (1981)
"A Survey on Image Segmentation"
Pattern Recognition Vol. 13, 1981, pp. 3-16

52. Fu, K.S. and Ichikawa, T. (1982)
"Special Computer Architectures for Pattern Processing"
CRC Press, Florida, 1982.

53. Hall, E.H. (1979)
"Computer Image Processing and Recognition"
Academic Press, New York, 1979

54. Hilditch, C.J. (1983)
"Comparison of Thinning Algorithms on a Parallel Processor"
Image and Vision Computing, Vol. 1, No. 3, Aug. 1983, pp. 115-132

55. Hockney, R.W. and Jesshope, C.R. (1981)
"Parallel Computers"
Adam Hilger Ltd. (IOP), Bristol, 1981

56. Horowitz, E. and Sahni, S. (1984)
"Fundamentals of Data Structures in Pascal"
Pitman, London, 1984, pp. 338-341

57. Hough, P.V.C. (1962)
"Method and Means for Recognizing Complex Patterns"
Patent No. 3,069,654, 1962

58. Hueckel, M.H. (1971)
"An Operator Which Locates Edges in Digitized Pictures"
J.ACM, Vol. 18, No. 1, Jan. 1971, pp. 113-125

59. Hueckel, M.H. (1973)
"A Local Visual Operator Which Recognises Edges and Lines"
J.ACM, Vol. 20, No. 4, Oct. 1973, pp. 634-647

60. Isenor, D.K. and Zaky, S.G. (1986)
"Fingerprint Identification using graph matching"
Pattern Recognition Vol. 19, No. 2, 1986, pp. 113-122

61. Jarvis, J.F. (1980)
"A Method for Automating the Visual Inspection of Printed Wiring Boards"
IEEE Trans. Patt. Anal. Mach. Intell., Vol 2, No. 1, Jan. 1980, pp. 77-82

62. Kaufmann, P., Medioni, G. and Nevatia, R. (1984)
"Visual Inspection Using Linear Features"
Pattern Recognition Vol. 17, No. 5, 1984, pp. 485-491

63. Kimme C., Ballard, D. and Sklansky, J. (1975)
"Finding Circles by an Array of Accumulators"
Comm. ACM, Vol. 18, No. 2, Feb. 1975, pp. 120-122

64. Kittler, J. (1983)
"On the Accuracy of the Sobel Edge Detector"
Image and Vision Computing, Vol. 1, No. 1, Feb. 1983, pp. 37-42

65. Kirsch, R.A. (1971)
"Computer Determination of the Constituent Structure of Biological Images"
Computers and Biomed. Res., Vol. 4, 1971, pp. 315-328

66. Kitchen, L. and Rosenfeld, A. (1982)
"Grey-level Corner Detection"
Pattern Recognition Letters, No. 1, 1982, pp. 95-102

67. Kung, H.T. (1982)
"Why Systolic Architectures"
IEEE Computer, Jan. 1982, pp. 37-45

68. Kushnir, M., Abe, K. and Matsumoto, K. (1983)
"An Application of the Hough Transform to the Recognition of Printed Hebrew Characters"
Pattern Recognition Vol. 16, No. 2, 1983, pp. 183-191

69. Lai, M.T.Y. and Suen, C.Y. (1981)
"Automatic Recognition of Characters by Fourier Descriptors and Boundary Line Encodings"
Pattern Recognition Vol. 14, 1981, pp. 383-393

70. Laycock, L.C. (1987)
"Optical Computers for Image Processing"
Physics Bulletin, Vol. 38, No. 11, Nov. 1987, pp. 408-409

71. Lee, C.C. (1983)
"Elimination of Redundant Operations for a Fast Sobel Operator"
IEEE Trans. Systems, Man, Cybernetics, Vol. 13, No. 3, March/April 1983, pp. 242-245

72. Lev, A., Zucker, S.W. and Rosenfeld, A. (1977)
"Iterative Enhancement of Noisy Images"
IEEE Trans. Systems, Man, Cybernetics, Vol. 7, No. 6, July 1977, pp. 435-442

73. Lin, W.C. and Chan, C-F. (1975)
"Feasibility Study of Automatic Assembly and Inspection of Light Bulb Filaments"
Proc. IEEE, Vol. 63, No. 10, Oct. 1975, pp. 1437-1445

74. Mantas, J. (1987)
"Methodologies in Pattern Recognition and Image Analysis-A Brief Survey"
Pattern Recognition, Vol. 20, No. 1, 1987, pp. 1-6

75. Marr, D. and Hildreth, E. (1980)
"Theory of Edge Detection"
Proc. Roy. Soc. Lond. B, Vol. 207, 1980, pp. 187-217

76. Martelli, A. (1976)
"An Application of Heuristic Search Methods to Edge and Contour Detection"
Comm. ACM, Vol. 19, No. 2, Feb. 1976, pp. 73-83

77. McQueen, M.P.C. (1981)
"A Generalization of Template Matching for Recognition of Real Objects"
Pattern Recognition Vol. 13, No. 2, 1981, pp. 139-145

78. Mick, J. and Brick, J. (1980)
"Bit-slice Microprocessor Design"
McGraw-Hill, New York, 1980

79. Montanari, U. (1969)
"Continuous Skeletons from Digitized Images"
J.ACM, Vol. 16, No. 4, Jan. 1969, pp. 534-549

80. Motorola (1985)
"VMEbus Specification Manual - Revision C"
Motorola, MVMEBS/D2, Feb. 1985.

81. Murase, H. and Wakahra, T. (1986)
"Online Hand-Sketched Figure Recognition"
Pattern Recognition Vol. 19, No. 2, 1986, pp. 147-160

82. Naccache, N.J. and Shinghal, R. (1984)
"SPTA: A Proposed Algorithm for Thinning Binary Pictures"
IEEE Trans. Systems, Man, Cybernetics, Vol. 14, No. 3, May/June 1984, pp. 409-418

83. Nevatia, R. (1982)
"Machine Perception"
Prentice-Hall, U.S.A., 1982.

84. Nixon. M. (1985)
"Application of the Hough Transform to Correct for Linear Variation of Background Illumination in Images"
Pattern Recognition Letters, No. 3, May 1985, pp. 191-194

85. Oda, M., Womack, B.F., and Tsubouchi, K. (1971)
"A Pattern Recognising Study of Palm Reading"
IEEE Trans. Systems, Man, Cybernetics, April 1971, pp. 171-175

86. Oshima, M. and Shirai, Y. (1979)
"A Scene Description Method Using Three-Dimensional Information"
Pattern Recognition Vol. 11, 1979, pp. 9-17

87. Paler, K. and Kittler, J. (1983)
"Greylevel Edge Thinning: A New Method"
Pattern Recognition Letters, No. 1, 1983, pp. 409-416

88. Paler, K., Föglein, Illingworth, J. and Kittler, J. (1984)
"Local Ordered Grey Levels as an Aid to Corner Detection"
Pattern Recognition Vol. 17, No. 5, 1984, pp. 535-543

89. Panda, D.P. and Rosenfeld, A. (1978)
"Image Segmentation by Pixel Classification in (Grey Level,Edge Value) Space"
IEEE Trans. Comput., C-27, No. 9, Sept. 1978, pp. 875-879

90. Pease III, M.C., (1977)
"The Indirect Binary n-Cube Microprocessor Array"
IEEE Trans. Comput., C-26, No. 5, May 1977, pp. 458-473

91. Pen-Shu, Y. and Antoy, S. and Litcher, A. and Rosenfeld, A. (1987)
"Address Location on Envelopes"
Pattern Recognition Vol. 20, No. 2, 1987, pp. 213-227

92. Pfaltz, J.L. and Rosenfeld, A. (1967)
"Computer Representation of Planar Regions by Their Skeletons"
Comm. ACM, Vol. 10, No. 2, Feb. 1967, pp. 22-33

93. Plummer, A.P.N. (1982)
"The NPL Linear Array Processor"
NPL Internal Report, ITC h 23, 1982

94. Plummer, A.P.N. (1982)
"The NPL Linear Array Processor: technical details"
NPL Internal Report, ITC h 24, 1982

95. Potter, J.L. (1983)
"Image Processing on the Massively Parallel Processor"
IEEE Computer, Jan. 1983, pp. 62-67

96. Prewitt, J.M.S. (1970)
"Object Enhancement and Extraction"
Picture Processing and Psychopictorics, Academic Press, 1970, pp. 75-149

97. Pritchard, S., Cohen, D. and Sleigh, A.C. (1986)
"DIPOD: An Advanced Multiptocessor System for Image Analysis"
Proc. 2nd Int. Conf. on Image Processing, 1986, pp. 134-138

98. Ramamoorthy, C.V. and Tsuchiya, M. (1974)
"A High-level Language for Horizontal Microprogramming"
IEEE Trans. Comput., C-23, No. 8, Aug. 1974, pp. 791-801

99. Richard, C.W. and Hooshang, H. (1974)
"Identification of Three-Dimensional Objects Using Fourier Descriptors of the Boundary Curve"
IEEE Trans. Systems, Man, Cybernetics, Vol. 4, No. 4, July 1974, pp. 371-378

100. Roberts, L.G. (1965)
"Machine Perception of Three-dimensional Solids"
Electro-optical Information Processing, MIT Press, Chap. 9, 1965, pp. 159-197

101. Rorres, C. and Anton, H. (1984)
"Least Squares Fitting to Data"
Applications of Linear Algebra, Wiley and Sons, Chap. 15, 1984, pp. 195-205

102. Rosenfeld A. (1970)
"Connectivity in Digital Pictures"
J.ACM, Vol. 17, No. 1, Jan. 1970, pp. 146-160

103. Rosenfeld, A. (1981)
"Image Pattern Recognition"
Proc. IEEE, Vol. 69, No. 5, May 1981, pp. 596-605

104. Rosenfeld, A. and Kak, A.C. (1982)
"Digital Picture Processing"
Academic Press, New York, 1982

105. Seitz, C.L. (1985)
"The Cosmic Cube"
Comm. ACM, Vol. 28, No. 1, Jan. 1985, pp. 22-33

106. Shapiro, S.D. (1980)
"Use of the Hough Transform for Image Data Compression"
Pattern Recognition Vol. 12, 1980, pp. 333-337

107. Shore, J.E. (1973)
"Second Thoughts on Parallel Processing"
Computer and Electronic Engineering, Vol. 1, 1973, pp. 95-109

108. Siegel, H.J., Siegel, L.J. and Mueller, P.T.Jr (1980)
"A Parallel Language for Image and Speech Processing"
Proc. IEEE Comp. Socs. Fourth International Computer Software and Applications Conference, Oct. 1980, pp. 476-483

109. Siegel, H.J. and Swain, P.H. (1981)
"Contextual Classification on PASM"
IEEE Comp. Soc. Conf. on Pattern Recognition and Image Processing, August 1981, pp. 320-325

110. Siegel, H.J., Siegel, L.J., Mudge, T.N. and Deip, E.J. (1982)
"Image Coding Using the Multiprocessor System PASM"
IEEE Comp. Soc. Conf. on Pattern Recognition and Image Processing, June 1982, pp. 200-205

111. Siegel, L.J., Siegel, H.J. and Feather, A.E. (1982)
"Parallel Approaches to Image Correlation"
IEEE Trans. Comput., C-31, No. 3, Mar. 1982, pp. 208-218

112. Siegel, H.J., Tuomenoksa, D.L., Adams III, G.B. and Mitchell, O.R. (1983)
"A Parallel Algorithm for Contour Extraction: Advantages and Architectural Implications"
Proc. IEEE Comp. Soc. Conf. on Computer Vision and Pattern Regognition, June 1983, pp. 336-344

113. Siegel, H.J. and Kuehn, J.T. (1986)
"Multifunction Processing with PASM"
Intermediate-Level Image Processing, ed. M.J.B. Duff, Academic Press, 1986, Chap. 13, pp. 209-229

114. Siegel, H.J. and Kuehn, J.T. (1986)
"Simulation Based Performance Measures for SIMD/MIMD processing"
Evaluation of Multicomputers for Image Processing, ed. Uhr et. al. , Academic Press, U.K., 1986, pp. 139-158

115. Silberberg, T.M., Davis, L. and Harwood, D. (1984)
"An iterative Hough Procedure for Three-Dimensional Object Recognition"
Pattern Recognition Vol. 17, No. 6, 1984, pp. 621-629

116. Sklansky, J. (1978)
"On the Hough Technique for Curve Detection"
IEEE Trans. Comput., C-27, No. 10, Oct. 1978, pp. 923-926

117. Smith, R.W. (1987)
"Computer Processing of Line Images: A Survey"
Pattern Recognition, Vol. 20, No. 1, 1987, pp. 7-15

118. Stefanelli, R. and Rosenfeld, A. (1971)
"Some Parallel Thinning Algorithms for Digital Pictures"
J.ACM, Vol. 18, No. 2, April 1971, pp. 255-264

119. Stentiford, F.W.M. and Mortimer, R.G. (1983)
"Some New Heuristics for Thinning Binary Handprinted Characters for OCR"
IEEE Trans. Systems, Man, Cybernetics, Vol. 13, No. 1, Jan/Feb 1983, pp. 81-84

120. Tanimoto, S.L. (1986)
"Architectural Issues for Intermediate-Level Vision"
Intermediate-Level Image Processing, ed. M.J.B. Duff, Academic Press, 1986, Chap. 1, pp. 3-17

121. Toshifumi, T. and Huang, T.S.(1987)
"Motion Stereo for Navigation of Autonomous Vehicles in Man-made Environments"
Pattern Recognition Vol. 20, No. 1, 1987, pp. 105-114

122. Tsuchiya, M. and Gonzalez, M.J. (1974)
"An Approach to Optimisation of Horizontal Microprograms"
Micro 7-PPS. 7th Annual Workshop on Microprogramming Preprints, 1974, pp. 85-90

123. Wahl, F.M. and Biland, H.P. (1986)
"Decomposition of Polyhedral Scenes in Hough Space"
Proc. 8th Int. Conf. on Pattern Recognition, 1986, 78-84

124. Ward, C.G. (1986)
"A Comparison of Parallel Implementations of Image Segmentation Techniques"
Proc. 2nd Int. Conf. on Image Processing, 1986, pp. 102-106

125. Wechsler, H. and Sklansky, J. (1977)
"Finding the Rib Cage in Chest Radiographs"
Pattern Recognition Vol. 9, 1977, pp. 21-30

126. Weiman, C.F.R. (1976)
"Highly Parallal Digitized Geometric Transformations Without Matrix Multiplication"
Proc. Int. Conf. on Parallel Processing, pp. 1-10

127. Williams, T. (1987)
"Optics and Neural Nets: Trying to Model the Human Brain"
Special Report on Advanced Supercomputers, Computer Design, March 1 1987, pp. 47-62

128. Willis, N. (1986)
"Architectural Considerations"
Computer Architecture and Communications, Paradigm, 1986, pp. 134-135

129. Wu, C.K., Wang, D.Q. and Bajcsy, R.K. (1984)
"Acquiring 3d Spatial Data of a Real Object"
Computer Vision, Graphics, Image Processing, Vol. 28, 1984, pp. 126-133

130. Zhang, T.Y. and Suen, C.Y. (1984)
"A Fast Parallel Algorithm for Thinning Digital Patterns"
Comm. ACM, Vol. 27, No. 3, March. 1984, pp. 236-238

APPENDIX A

INSTRUCTION SET FOR SIP

Addressing Modes

#	data	(D)
Rx	register	(R)
Px/Qx	picture	(P/Q)
X/Y	x/y mode	(X/Y)
(Rx)	indexed	(I)
name/number	memory	(M)

Program instructions

BEGIN	always start a program with this
HALT	stops the program at this point
ENDPROG	last statement in program
VAR	declare variables/arrays
; text	comment

Processor instructions

CLR dest	clear location dest
INC dest	increment location
DEC dest	decrement location
SHL dest	arithmetic shift left location
SHR dest	arithmetic shift right location
NOT dest	invert location
NEG dest	negate (R0:=-R0) location
MOV src,dest	move src to dest
ADD src,dest	dest:=dest+src
SUB src,dest	dest:=dest-src
CMP src,dest	src-dest
BIC src,dest	dest:=NOT source AND dest
BIT src,dest	src AND dest
BIS src,dest	dest:=src OR dest
AND src,dest	dest:=src AND dest

PC instructions

```
BRA dest                branch unconditionally to dest
BEQ dest                branch 'if equal' to dest
BLT dest                branch 'if less than' to dest
BLE dest                branch 'if less than or equal' to dest
IFEQ dest1,dest2        if equal then goto dest1 else goto dest2
IFLT dest1,dest2        'if less than' then goto dest1 else goto dest2
JSR dest                jump to subroutine dest
JSREQ dest              jump to subroutine if equal
JSRLT dest              jump to subroutine 'if less than'
JSUBEQ dest1,dest2      'if equal' JSR dest1 else JSR dest2
JSUBLT dest1,dest2      'if less than' then JSR dest1 else JSR dest2
RETURN                  return unconditionally from subroutine
RETEQ                   return 'if equal' from subroutine
RETLT                   return 'if less than' from subroutine
REPEAT                  repeat the following code
UNTILLT                 until 'less than'
UNTILEQ                 until equal
UNTILF                  until false
REPLOOP register        repeat loop 'contents of reg' times (max=4096)
ENDLOOP                 end of repeat loop
```

Picture functions

```
APPLY                   apply to every point in the image
 .
 .
END                     end apply

PGET P                  get an image from VMEbus into P(Q)-space
POUT P                  send P(Q) to VMEbus for display
OUT Rx                  output register to the VMEbus at location Rx-1
```

Example Program

As an example here is a program to threshold an image with 127

```
        BEGIN                   ; always start with this
        PGET P                  ; get a picture into P-space

        APPLY                   ; apply over the picture
        CMP     #127,P0         ; 127-P0
        IFLT    SET255,SET0     ; if P0<127 then set255 ELSE set0
SET255: MOV     #255,P0         ; set P0 to 255
        BRA     GOON            ; goto goon
SET0:   MOV     #0,P0           ; set P0 to 0
GOON:   END                     ; carry on until finished picture

        POUT P                  ; Display thresholded picture (p-space)
        HALT
        ENDPROG                 ; END
```

The microcode consists of five parts MC1, MC2, MC3, MC4 and MC5. These correspond to

MC1 - processor instructions
MC2 - picture and local memory functions and condition code select
MC3 - data (for the data bus)
MC4 - Program counter instructions, output enables, and 'done bit'
MC5 - vme access + datapro

This is at a primitive stage; however, tailor made instructions can be defined by entering into the program:

```
MC1(rama,ramb,source,alufunc,dest,cn)
MC2(wrpic1,wrpic2,wrglbl,xcom,yxom,gcom,oeg,eng,clkoffreg,ccsel)
MC3(data)
MC4(pc_instruction,output enables,done)
MC5(req,rel,rw,vsen,type,clkextad,datapro)
```

with the appropriate values for rama, ramb, etc. To assemble a program, execute the following instructions:

```
ASSEM input_filename     (default is .MAC)
TRANS output_filename    (default is .SIP)
DLOAD filename           (download code to SIP)
```

Two qualifiers (L & N) can be added to ASSEM, i.e. ASSEM filename/L/N.

1. L - produces a listing file of all three passes of ASSEM, including error messages signified by " error message".

2. N - produces the length of the corresponding microcode instruction.

APPENDIX B

IMPLEMENTATION OF THE O-RING ALGORITHM ON SIP

```
        VAR peaks:3000, found:80
        VAR xmm, ymm
        VAR oldnum, old:100

        BEGIN
GRAB:   PGRAB           ; Grab picture
        PGET P          ; Get picture into P-space
        JSR INIT        ; Initialise arrays
        JSR DOT         ; Find possible centres
        JSR GETLST      ; Get list
        JSR DEDUCE      ; Deduce true centres
        JSR WIPE        ; Wipe old crosses
        JSR CROSS       ; Put new crosses on the image
        BRA GRAB        ; Loop forever !!

INIT:   MOV #21,R0      ; Radius = 21
        MOV #7,R1       ; Peak at centre - defines circle !!
        MOV #0,R2       ; R2 is PEAK_COUNT
        MOV #0,R10      ; R10 is the number of rings found
        RETURN

DOT:    APPLY           ; Wipe Q space to zero
        MOV #0,Q0
        END

        APPLY           ; Setup scan
        MOV P1,R6       ; Do a Sobel
        SHL R6
        ADD P8,R6
        ADD P2,R6       ; R6=(P8+2*P1+P2)
        MOV P5,R7
        SHL R7
        ADD P4,R7
        ADD P6,R7       ; R7=(P4+2*P5+P6)
        SUB R7,R6       ; R6:=R6-R7

        CMP #0,R6
        BLE NONEG
        NEG R6
```

```
        SHR R6
        SHR R6
        NEG R6
        BRA CONT
NONEG:  SHR R6
        SHR R6

CONT:   MOV P7,R7
        SHL R7
        ADD P6,R7
        ADD P8,R7       ; R7=(P6+2*P7+P8)
        MOV P3,R8
        SHL R8
        ADD P2,R8
        ADD P4,R8       ; R8=(P2+2*P3+P4)
        SUB R8,R7       ; R7 HOLDS DY

        MOV R6,R8
        CMP #0,R8
        BLE LOOP3
        NEG R8

LOOP3:  MOV R7,R9       ; R8=ABS(DX)
        CMP #0,R9
        BLE LOOP4
        NEG R9          ; R9 = ABS(DY)
LOOP4:  CMP R8,R9
        BLT BIG9
        MOV R8,R9

BIG9:   CMP R9,#150     ; IS DD>THR?
        BLT ENDDOT      ; IF YES THEN END ELSE
        CMP #0,P0       ; SET ADDRESS BACK TO P0
        MOV X,R3        ; SAVE X & Y
        MOV Y,R4

        MUL R0,R6       ; DX:=RADIUS*DX
        MOV R6,R13
        MOV R9,R14      ; R9 IS DD
        JSR DIVIDE
        MOV R13,R6      ; DX:=DX/DD
CONT1:  MOV R3,R5       ; R3 and R5 now holds X
        SUB R6,R5       ; R5:=X-DX      NEW X !!!!

        MUL R0,R7       ; DY:=radius*DY
        MOV R7,R13
        MOV R9,R14
        JSR DIVIDE
        MOV R13,R7      ; DY:=DY/DD
        MOV R4,R8       ; USE R8 AS A TEMP REGISTER
        SUB R7,R8       ; R8:=Y-DY      NEW Y !!!!

        MOV R5,X        ; X:=R5
        MOV R8,Y        ; Y:=R8     new X and Y
        INC Q0          ; now increment Hough space Q0:=Q0+1
        MOV R3,X
        MOV R4,Y
ENDDOT: END
        RETURN
```

```
GETLST: APPLY
        CMP Q0,R1        ; IS Q0>THRESHOLD
        BLT ENDLST
        MOV #peaks,R15   ; peaks[0]
        MOV R2,R14
        SHL R14
        ADD R14,R15
        MOV X,R14        ; Save X and Y
        MOV R14,(R15)
        INC R15
        MOV Y,R14
        MOV R14,(R15)
        INC R2           ; Bump peak_count
ENDLST: END
        DEC R2           ; One over actual value
        RETURN


DEDUCE: MOV R2,R11
        SHL R11
        MOV #peaks,R15
        ADD R11,R15
        MOV (R15),R13    ; R13 and R14 at peak position
        INC R15
        MOV (R15),R14
        MOV #found,R15
        MOV R13,(R15)
        INC R15
        MOV R14,(R15)    ; First centre found and stored

GOON2:  MOV #peaks,R15
        MOV R2,R14
        SHL R14
        ADD R14,R15      ; peaks[peak_count]
        MOV (R15),R3     ; xx
        INC R15
        MOV (R15),R4     ; yy
        MOV #1,R5        ; Peak_exists := TRUE

        MOV R3,X         ; GET Q0 value from (xx,yy)
        MOV R4,Y
        MOV Q0,R0

        MOV #0,R6        ; R6 is temp variable for loop (i)
GOON1:  MOV #found,R15
        MOV R6,R7        ; R7 is R6
        SHL R7
        ADD R7,R15
        MOV (R15),R8
        MOV R8,R7        ; peaks x value
        INC R15
        MOV (R15),R9
        MOV R9,R11       ; peaks y value
        SUB R3,R8
        SUB R4,R9
        MUL R8,R8
        MUL R9,R9
        ADD R8,R9
        CMP #40,R9
        BLT SKP3
```

```
        MOV R7,X        ; Get Q0 value
        MOV R11,Y
        CMP R0,Q0
        BLT NOCH

        MOV R6,R14
        SHL R14
        MOV #found,R15
        ADD R14,R15
        MOV R3,(R15)
        INC R15
        MOV R4,(R15)
NOCH:   MOV #0,R5       ; Peak_exists := FALSE
SKP3:   INC R6
        CMP R6,R10
        BLE GOON1

        CMP #0,R5
        BEQ NOSTRE
        INC R10
        MOV R10,R14
        SHL R14
        MOV #found,R15
        ADD R14,R15
        MOV R3,(R15)
        INC R15
        MOV R4,(R15)
NOSTRE: DEC R2
        CMP #0,R2
        BLE GOON2
        RETURN


WIPE:   MOV oldnum,R13  ; Wipe old centres
        MOV #0,R4
NXTIT:  MOV #old,R15
        SHL R4
        ADD R4,R15
        MOV (R15),R0
        INC R15
        MOV (R15),R1
        MOV #0,R3
        OUT R0
        INC R4
        CMP R4,R13
        BLE NXTIT
        RETURN


CROSS:  MOV R10,oldnum  ; Put a dot at the centre of the O-ring
        MOV #0,R13
PLOT:   MOV #found,R1   ; apply a median filter at each one
        MOV R13,R2
        SHL R2
        ADD R2,R1
        MOV (R1),R14
        INC R1
        MOV (R1),R15

        CLR R2          ; R2 = SIGMY
        CLR R3          ; R3 = SIGMYY
```

```
        MOV R14,R4
        MOV R15,R5

        MOV R5,R6
        MOV R5,R7       ; MAKE A COPY FOR LATER
        SUB #8,R5       ; R5=YM-8
        ADD #8,R6       ; R6=YM+8

AGAIN1: MOV R4,X
        MOV R5,Y
        ADD Q0,R2
        MOV Y,R5
        INC R5
        CMP R5,R6
        BLE AGAIN1

        MOV R14,R4      ; R4 IS XM
        MOV R7,R5
        SUB #8,R5       ; R5 IS YM-8
        SHR R2          ; SIGMY/2

AGAIN2: CMP R2,R3       ; WHILE SIGMYY<=SIGMY/2 DO
        BLT ENDBIT
        MOV R4,X
        MOV R5,Y
        ADD Q0,R3
        MOV Y,R5
        INC R5
        MOV Y,YMM
        CMP R5,R6
        BLE AGAIN2

ENDBIT: CLR R0          ; R0 = SIGMX
        CLR R1          ; R1 = SIGMXX
        MOV R14,R2      ; R2 IS XM
        MOV R2,R3
        MOV R2,R4       ; MAKE A COPY FOR LATER
        MOV YMM,R5      ; R5 = YMM
        SUB #8,R2       ; R2 IS XM-8
        ADD #8,R3       ; R3 IS XM+8

AGAIN3: MOV R2,X
        MOV R5,Y
        ADD Q0,R0
        MOV X,R2
        MOV Y,R5
        INC R2
        CMP R2,R3
        BLE AGAIN3

        MOV R14,R2
        SUB #8,R2       ; R2 IS XM-8
        SHR R0          ; R0=SIGMY/2

AGAIN4: CMP R0,R1       ; WHILE SIGMYY<=SIGMY/2 DO
        BLT END4
        MOV R2,X
        MOV R5,Y
        ADD Q0,R1
        MOV X,R2
        INC R2
```

```
        MOV X,XMM
        MOV Y,R5
        CMP R2,R3
        BLE AGAIN4

END4:   MOV XMM,R0
        MOV YMM,R1
        SUB #2,R0
        MOV #old,R2     ; Get old coordinates into R2 and R5
        SHL R13         ; Get x position
        ADD R13,R2
        MOV (R2),R4
        MOV R4,R11
        INC R2
        MOV (R2),R5
        MOV R5,R12
        SHR R13         ; Restore R13
        SUB R0,R4       ; Find difference
        SUB R1,R5
        CMP #0,R4       ; R4 := ABS (R4)
        BLE LLP1
        NEG R4
LLP1:   CMP #0,R5       ; R5 := ABS (R5)
        BLE LLP2
        NEG R5
LLP2:   CMP #2,R4
        BLT RPLACE
        CMP #2,R5
        BLT RPLACE

        MOV R11,R0
        MOV R12,R1

RPLACE: MOV R1,(R2)     ; Save new position
        DEC R2
        MOV R0,(R2)
        MOV #255,R3
        OUT R0          ; Draw dot, directly accessing VFS1
ENDCNT: INC R13
        CMP R13,R10
        BLE PLOT
        RETURN


DIVIDE: MOV R14,R12      ; SAVE R14
        CMP #0,R14
        BLE POSDIV
        NEG R14
POSDIV: CMP #0,R13
        BLE FILL1
        MOV R13,R15
        NEG R15
        BRA DIVIT
FILL1:  MOV R13,R15
DIVIT:  MVDVQ           ; Q=ABS(R13)
        CLR R15
        DVRR            ; DO DIVISION HERE
        CMP #0,R15
        BEQ RESLT
        CMP #0,R15
        BLT RESLT
```

```
TESTQ:  GETQ
        BLT INCQ
        QSUB1
        ADD R14,R15
        BRA RESLT
INCQ:   QPLUS1
        SUB R14,R15
RESLT:  CMP #0,R13
        BLE TRYNEG
        CMP #0,R12
        BLE TRYIT
        QTOR
        NEG R15
        BRA ENDDIV
TRYIT:  QTOR
        NEG R13
        NEG R15
        BRA ENDDIV
TRYNEG: CMP #0,R12
        BLE ENDMIN
        QTOR
        NEG R13
        BRA ENDDIV
ENDMIN: QTOR            ; ANSWER NOW IN R13
ENDDIV: MOV R12,R14     ; RESTORE R14
        RETURN
        ENDPROG
```