Development and Implementation of Real Time Image ~~Processing~~ ANALYSIS Algorithms

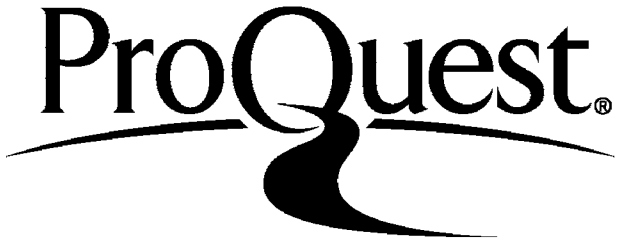
A Thesis submitted for the degree of
Doctor of Philosophy
of the University of London

by

Adrian Ivor Clive Johnstone

Computer Science Department
Royal Holloway and Bedford New College
University of London

December 1988



Abstract

This work concerns the development and implementation of real-time image processing algorithms. Such systems may be applied to industrial inspection problems, which typically require basic operations to be performed on 256×256 pixel images in 20 to 100ms using systems costing less than about £20000.

Building such systems is difficult because conventional processors executing at around 1MIPS with conventional algorithms are some 2 orders of magnitude too slow. A solution to this is to use a closely coupled array processor such as the DAP, or CLIP4 which is designed especially for image processing. However such a space-parallel architecture imposes its own structure on the problem, and this restricts the class of algorithms which may be efficiently executed to those exhibiting similar space parallelism, *i.e.* so-called 'parallel algorithms'.

This thesis examines an alternative approach which uses a mix of conventional processors and high speed hardware processors. A special frame store has been built for the acquisition and display of images stored in memory on a multiprocessor backplane. Also described are an interface to a host mini-computer, a bus interface to the system and its use with some hardwired and microcoded processors. This system is compared to a single computer operating with a frame store optimised for image processing.

The basic software and hardware system described in this thesis has been used in a factory environment for foodproduct inspection.

Contents

List of Figures

List of Tables

Table of Abbreviations

ABI	Asynchronous Backplane Inteconnect
CSR	Control and Status Register
IMP	Image-handling MultiProcessor
IPOFS	Image Processing Oriented Frame Store
LAP	Linear Array Processor
PPL	Picture Processing Language
PSW	Processor Status Word
SP1	Sequential Processor 1
V1	Video board 1
VAR	VME Address Register

Chapter 1

Introduction

1.1 The problem of robot vision

Robots are no longer the creatures of fantasy but are being installed on production lines up and down the country. However, it is clear that the currently available machines are a far cry from the free-roving creatures beloved of science fiction writers. Typically, a paint spraying robot is merely a mimic device — a human operator guides the arm around the lines of the workpiece whilst a computer monitors the path, and subsequently retraces it automatically. If the workpiece is misaligned or damaged, then the robot will continue slavishly painting thin air. The missing element in current commercial systems is a satisfactory vision system. Programming computers to recognise objects in an image is an inherently difficult task, and is made more difficult by the enormous amounts of data that must be processed.

A free-roving robot will need to be able to navigate itself and avoid hazards. Consider the operations required in crossing the road. The robot is being approached on two sides by massive objects which are possibly accelerating. To cross safely, a gap between the cars must be recognised. This entails solving simultaneous differential equations before the situation has altered so significantly that the results are useless (that is in *real time*). This problem is dwarfed by the task of identifying the cars and approximating their speed and direction, that is obtaining the differential equations in the first place.

A simple way to identify the cars might be to store in the robot's memory some pictures of cars and make point to point comparisons between them and the incoming visual data. It is axiomatic that a representative sample of cars will need to be stored. It is reasonable to expect the robot to interpolate between images, but extrapolation is much riskier and is likely to lead to erroneous results.

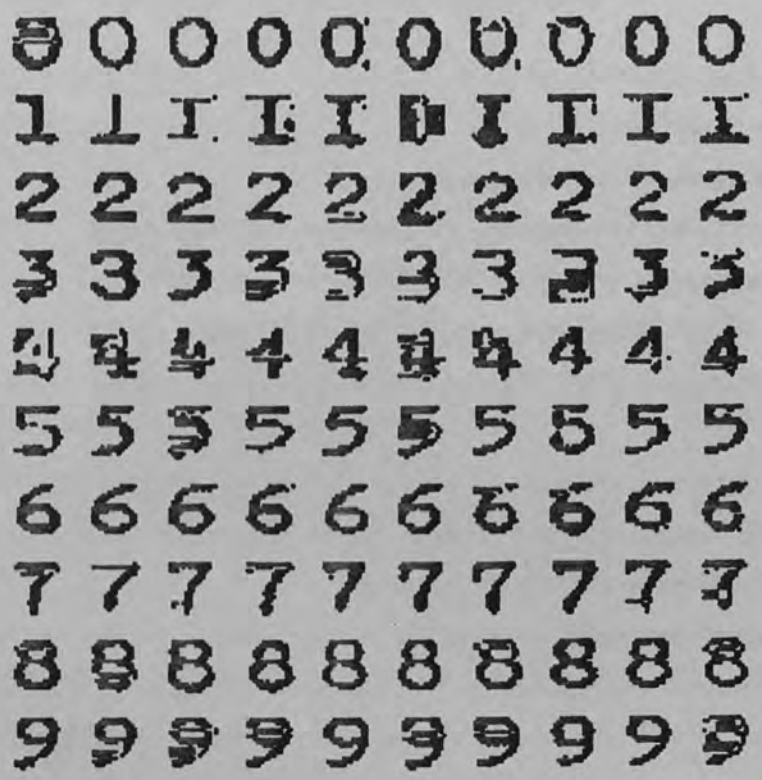


Figure 1.1: Postcode characters

This glosses over exactly what 'interpolation of images' means. If a simple direct comparison approach is used then the stored data set will need examples not only of all the cars on the road, but also pictures of them at different orientations and in different lighting conditions. Near infinite amounts of memory could be consumed in this way, and even if such a database could be accumulated, the search time would be immense.

To underline this, consider a much simpler problem. Figure 1.1 shows a selection of characters generated from photographs of typewritten postcode characters on envelopes. Each pattern comprises 256 dots arranged in a 16 by 16 matrix. Even with this restricted problem domain it is not possible to employ a logical AND gate for recognition, because there are 2^{256} possible patterns constructable from 256 black-or-white dots. This is about 10^{77}, and even at a search rate of 1MHz, 10^{71} seconds would be required for an exhaustive linear search. Cosmologists disagree on the age of the Universe since the Big Bang, but most estimates are around 2×10^{10} years [Ber76], or 6×10^{17}s, which is about 53 orders of magnitude less than our search time. Clearly point to point comparison has limited application.

1.2 Classical pattern recognition

Any practical pattern recognition scheme will have to work with extremely compacted reference data. It may be possible to select representative members of each class and use these as templates for comparison with incoming data according to some matching criterion which is more relaxed than direct point to point comparison. However, it will be difficult to find a comprehensive set of templates in situations where large variations are expected in the classes, since they will not be clustered around a limited number of class prototypes. This is the case in almost all non-trivial applications.

Classical pattern recognition can be considered a two-part problem. Initially a *feature detector* extracts measurements from the image. These are then applied to a *discriminator* which holds information concerning the various classes and their feature distributions. Figure 1.2 shows an image comprising n_p individual pixel brightness values which is reduced by the feature detector to n_f scalar feature measurements. For a practical scheme, $n_p \gg n_f$. The n_f features trigger a 1-of-n_c boolean class output at the discriminator.

The template matching approach described above is a special case in which the features are the templates and the discriminator is the matching criterion. In general the individual feature measurements define vectors in an N-dimensional feature space, and the discriminator will be a surface in that space. Obviously it is desirable for the discriminator to be linear (*i.e.* a hyperplane); however, other functions often used are the minimum distance, nearest neighbour (piecewise linear) and low order polynomial discriminators [DH73].

1.3 The pattern recognition hierarchy

It is difficult, and perhaps unwise, to present a taxonomy of pattern recognition techniques. Nevertheless, a clear hierarchy can be discerned both in terms of required processing power and the levels of abstraction provided by various approaches.

In general, a vision system must first analyse the raw data into some primitives and then synthesize a global description of the field of view. Typically the following steps would be performed:

1. Raw image data is segmented into regions which share some characteristic.

2. Each region is investigated to extract some measurement such as colour,

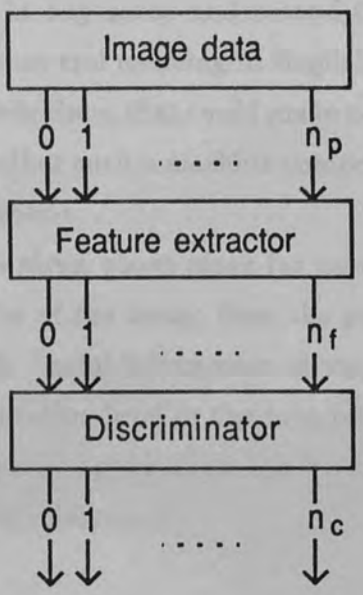


Figure 1.2: Classical pattern recognition machine

texture, shape etc.

3. Relationships between regions are established in terms of these measurements.

4. The properties of objects in the field of view are inferred from the relations between regions.

Processing levels above these might use stored contextual information to 'understand' the image. A text reading machine provides a useful example of the above process.

1. Regions can be extracted from a page of text by looking for connected dark areas and in most cases these would correspond to individual characters, although some characters such as i, j, % and punctuation symbols such as the semicolon would require special consideration.

2. Each region would be analysed to count branch points and the lengths of limbs. At this point the regions could be compared with known character data, and most characters could be identified.

3. Relational properties of the characters based on the spacing of individual characters could be used to identify separate word units and deal with non-connected characters.

Although the system should now be able to answer queries such as 'is this word in the text', it does not in any sense understand the text, and cannot in fact distinguish between nonsense and meaningful English. One can envisage a system with a large stored knowledge base that could make comments on the grammatical standard of the text. Whether such a machine comprehends the text is a question outside the scope of this thesis.

The descriptions given above place far more emphasis on the measurement of properties of parts of the image than the process of pattern recognition as described in section 1.2. Useful information about the field of view is produced even at the region segmentation level in the text reading machine, and it is reasonable to assume that useful applications can be constructed without invoking the full machinery of scene analysis.

1.4 N-tuple pattern recognition

In the classical pattern recognition system, it is possible for either the feature extraction or discrimination parts to be trivial operations. An example of trivial feature extraction is the technique of Bledsoe and Browning first reported in the 1950's [BB59], now known as n-tuple recognition, in which randomly selected groups of n bits are used as the 'features'. An n-tuple based recognition system for characters of the type shown in Figure 1.1 has been programmed by the author and can achieve a recognition rate of over 90% using 6-tuples and ten classes.

1.5 Structural pattern recognition

The case in which the discriminator is simple is interesting because it describes many real life situations. For example a robot arm may need the co-ordinates of the centre of a workpiece. This is a measurement rather than a classification problem since the type of workpiece is already known. Similarly, coated foodproducts such as fish fingers should show a uniform surface and any defects in the covering will show up as areas of the underlying material. Many industrial inspection problems may be formulated in this way where the presence or magnitude of a feature is all that is required to identify faulty products. In such cases the result appears directly from the feature extraction stage.

A more interesting situation occurs where the feature detection algorithm exhibits considerable 'intelligence' of its own. Such algorithms make sequential sets of measurements on an object which take different courses depending on the

results. The text reading machine above might use such an algorithm to work its way round an object counting branch points and measuring the lengths of limbs. This kind of algorithm is referred to as *structural* since the form of the algorithm will reflect the structure of the object being scanned. In its purest form, this kind of algorithm contains a complete description of the object and in a sense the algorithm 'parses' the object in much the same way as a compiler parses text. The formal study of this kind of pattern recognition is termed *syntactic* pattern recognition, and is concerned with the search for *picture grammars* that adequately describe scene content.

This thesis is particularly concerned with the efficient implementation of industrial inspection algorithms of the structural type that use image processing techniques to bypass the classical pattern recognition stage.

1.6 The human paradigm and artificial intelligence

Unfortunately, introspective analysis of our own capabilities does not provide much information useful to the robot engineer. Many workers have attempted to synthesize models of brain behaviour by implementing those models as computer programs. Pre-eminent in the vision field has been the work of Marr which follows an *information transfer* approach. Three principal representations of the image data are used:

> "(1) the primal sketch, which is concerned with making explicit properties of the two dimensional image, ... (2) the 2 1/2-D sketch, which is a viewer-centered representation of the depth and orientation of the visible surfaces ...; and (3) the 3-D model representation, whose important features are that its coordinate system is object centered, that it includes volumetric primitives (which make explicit the organisation of the space occupied by an object and not just its visible surfaces), and that primitives of various size are included, arranged in a modular, hierarchical organisation." [Mar82]

These models require prodigious amounts of processor time using conventional sequential computers, but this is not surprising in view of the highly parallel nature of the brain.

Apart from trying to model brain systems directly, many workers in the artificial intelligence field have used heuristic techniques to program problems which are combinatorially too large for normal analysis, such as chess and other

game playing, question and answer systems, and expert systems. 'Artificial intelligence' is an unfortunate name for these techniques which perhaps promises more than can be truly delivered — for some years the artificial intelligence community seems to have been in a 'jam tomorrow, never jam today' situation, which is possibly the result of overambitious targets.

Although artificial intelligence provides useful techniques for some pattern recognition problems, technology constraints usually rule out the use of complex heuristic methods for real time work and it may well be some time before Marr's work becomes directly usable in industrial environments. Therefore this thesis concentrates on the efficient implementation of relatively simple algorithms. Note that 'simple' and 'trivial' are not synonymous — many classical algorithms (such as Hoare's Quicksort) are both simple and elegant, but great insight was required for their discovery.

1.7 Overview of following chapters

This work is concerned with the systematic design and implementation of image processing algorithms. It falls into three parts: part 1 (Chapters 2–4) is concerned with data representation and algorithm design for image processing; part 2 (Chapters 5 and 6) looks at development systems for image processing bearing in mind the needs of the programmer; and part 3 (Chapters 7–10) looks at parallel and hardware implementation of algorithms in high speed multiprocessor systems.

Chapter 2 looks at some fundamental properties of digital images and their representations, concentrating on the use of hierarchical structures for industrial problems.

Chapter 3 discusses algorithm analysis and design, and the evaluation of systems using standardised algorithms and image data. Throughput requirements for industrial image processing are derived.

Chapter 4 gives some novel algorithms for the generation of quadtree data structures from images, and their application to real time image processing.

Chapter 5 concerns the design of framestore hardware that allows a host processor to efficiently access image data. Four frame stores designed by the author are described along with utility and application software.

Chapter 6 examines the major trends in sequential processor design and software systems to support image processing including PIPE, the software system used for the applications work described in chapter 8.

Chapter 7 reviews parallelism in hardware with a special emphasis on parallel processors designed for image processing applications.

Chapter 8 looks at various parallel programming paradigms and the programming constructs available in various languages.

Chapter 9 describes the design and implementation of a high speed multiprocessor system called IMP and its use in a realtime grey scale industrial inspection application.

Chapter 10 describes a full custom VLSI implementation of the Sobel filter designed to form the heart of an IMP hardware processor.

Chapter 11 summarises the results of the previous chapters and looks ahead to future work.

Chapter 2

Image representations for real time processing

"A picture is worth a thousand words."

2.1 Introduction

Application of computers to real world problems requires that the necessary processing be formulated as well defined algorithms, and that the physical dimensions and concepts of problems be mapped to suitable internal representations. Choice of representation is intimately connected to algorithm performance. This is because it is generally not possible to make all aspects of the data simultaneously explicit. Any collection of data in the computer has some kind of topology in which information is implicitly carried. As a result, any representation imposes its own order on the image.

The simplest example of this is the basic array representation of an image in which spatial information is carried in the topology of the array. This implicit information is expensive to retrieve: it is easy to find out what colour a pixel with certain coordinates has, but to find the coordinates of all pixels of a certain colour requires an exhaustive search of the whole structure. Similarly, the location of all 3×3 areas in the image with a certain distribution of pixel colours will require quite complex processing, because the information is even more implicit. However it is possible to envisage an image representation that directly enumerates all such features. This might take the form of a simple list, or some complex hierarchical structure.

The simplest image representation is an array, in which a one-to-one relationship exists between points in the visual field and the stored data points.

The discrete nature of digital representations may cause ambiguities and noise in subsequent processing. The array representation is examined in some detail here because it often forms the basis for more complex representations.

Early attempts to reduce storage and processing requirements in binary systems generated representations such as chain code [Fre61] and skeletonisation [Blu67].

There is a growing interest in 'hierarchical' tree structures for image processing [TK80]. These may be generated by 'bottom up' processes which generate successively reduced resolution versions of an image, or 'top down' processes which successively decompose an image into sub-images. Top down structures use global information available at each level to subdivide the image, whereas bottom up structures use the local properties of pixels to group them together. A structure of particular interest is the quadtree [Sam84], which can be used to provide global information to inherently local image processing operations, allowing increases in processing speed for certain classes of algorithm.

2.2 Information in images

Several types of information are present in an image. Most fundamentally there is displacement in two dimensions, and simple identification is possible using silhouette or 'binary' images. However there is also depth (*i.e.* displacement in a third dimension) and colour/brightness information. Typically depth information is less precise than X/Y position, and colour information is often compressed into a monochrome image.

2.3 Digitisation

Digital systems require information to be presented in digital form, and a continuum must be converted into a structure containing numbers. Most applications make use of a monochromatic representation where image data is stored in a 2-dimensional array of values corresponding to sample brightnesses across the image. Colour and depth information may be incorporated by storing colour and depth values along with the brightness data.

2.4 Spatial quantisation

The spatial relationships in the image are represented by splitting the visual plane into a number of picture elements or 'pixels'. Through familiarity with cartesian coordinates it is natural to make these pixels rectangular in shape, but this is not necessarily the best tessellation available. By considering angles at a point, it may be shown that squares, triangles and hexagons are the only regular polygons capable of tiling the plane (the so called 'regular tessellation'). It can also be shown that there are only 8 'semi-regular' tessellations which are tilings using a mix of regular polygons but with all vertices congruent [CR61].

For image representation, the characteristics of a good tessellation are:

1. it should have a fine enough net to avoid losing important detail in the quantisation noise and to avoid aliasing,
2. it should be regular and have as many axes of rotational symmetry as possible,
3. it should be able to describe primitive image properties without introducing geometrical paradoxes.

2.4.1 Spatial resolution

Characteristic 1 is a property of the spatial resolution (*i.e.* pixel size) and is not related to the actual tessellation. The 625 line PAL colour television system used in the UK specifies a line time of 64μs of which 12.05μs $\pm$ 250ns is blanking time used for flyback [IB71]. This leaves approximately 52μs of active video time. The normal video bandwidth is 5.5MHz although many cameras are capable of higher performance, say 6MHz.

According to the sampling theorem, in order to capture the full bandwidth of a signal, samples must be taken at twice the maximum frequency in the signal [GW87] — *i.e.* at 11MHz for broadcast video. Over the length of a 52μs display line, this corresponds to 572 pixels. This is an interesting result because 600 of the available 625 lines are used for display, which means that the spatial resolution of broadcast video is much better in the Y direction (600 pixels in unit length) than the X direction (572 pixels in 4/3 unit length). This is a result of the bandwidth limiting imposed on the signal to reduce demand on broadcast frequencies. If the sampling criterion is violated, aliasing can occur.

2.5 Regularity

Regularity is important if features are to be invariant under translation. If one part of the field of view is stored at higher resolution, then identical objects in different parts of the image may map to different amounts of image memory. There are cases where this is desirable, such as increasing horizontal resolution to show vertical edge profiles in more detail. It is also desirable that features should remain invariant under rotation. Any tessellation will introduce distortions in the image, but multiple axes of rotational symmetry will help reduce inconsistencies between representations of identical shapes at different angles to the coordinate axes.

2.6 Geometrical paradoxes

2.6.1 The crossing paradox

In most cases, accurate representation is a property of resolution only, but there is a deeper problem with rectangular tessellation. If four points are arranged in a block:

```
1 0
0 1
```

then a natural interpretation is that the two '1' points are parts of a connected line bisecting two background areas shown as '0's. However the black and white areas are spatially congruent, and there is no reason why this should not be interpreted as a discontinuity between two black areas. Thus connectivity is ill defined. This is known as the crossing paradox and was first noted by Rosenfeld [Ros70]. A rule often used to circumvent this problem is to require background connected areas to show four-connected adjacency, and allow foreground areas eight connectivity. This is equivalent to enlarging the size of each foreground dot in the analogue image, so that it occupies a larger area than one pixel as shown in Figure 2.1. This obviously removes generality and symmetry from the representation, and shows another problem with the rectangular tessellation: that the four and eight connected points are different distances from the centre of a window, even though they all border the centre.

The crossing paradox also affects the definition of crossing number, which is a measure of the order of connectivity at a point. Within a 3×3 window, four cases can be distinguished as shown in Figure 2.2.

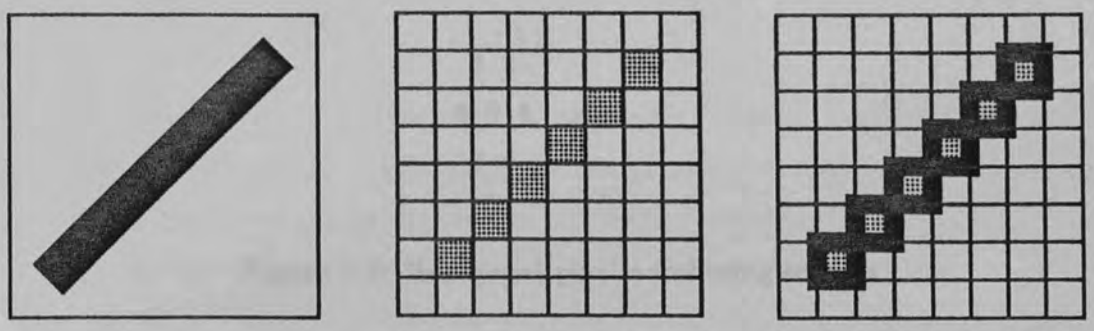


Figure 2.1: Connectivity of rectangular tessellation

- - -	X X X	- X -	- X -	X - -
- 0 -	X 0 X	- 2 -	- 4 -	- 6 X
- - -	X X X	- - -	- X -	X - X
isolated	isolated	end	mid	cross

Figure 2.2: Crossing number (rectangular tessellation)

Crossing number may be calculated as twice the number of lines or wider bodies meeting at a point. Calculation for the hexagonal tessellation is straightforward, and some examples are shown in Figure 2.3.

The crossing number may be calculated by counting the number of black/white transitions in the ring surrounding the central point. Let the pixels in the neighbouring ring be named $p_1, p_2 \ldots p_6$ giving the pixel numbering scheme shown in Figure 2.4.

Here the crossing number formula is $\sum_{i=1}^{6} p_i \,\mathrm{XOR}\, p_{i+1}$.

In the rectangular case, the situation is complicated by the two level hierarchy of points that exists within a window. The situation where three lines meet in the vicinity of a 3×3 window can map to two different 'types' of window

- -	X X	- -	- X	X -
- 0 -	X 0 X	- 2 -	- 4 -	- 6 X
- -	X X	X -	X -	X -
isolated	isolated	end	mid	cross

Figure 2.3: Crossing number (hexagonal tessellation)

```
3 2
4 0 1
5 6
```

Figure 2.4: Hexagonal pixel numbering scheme

```
X - -          - X -
- 6 X          X 4 -
X - -          - - X

cross          mid
```

Figure 2.5: Square tessellation crossing number

with different crossing numbers (Figure 2.5).

Using the connectivity criterion noted above, the two upper points in the mid case are connected, and therefore generate only one transition in the neighbouring ring of pixels. The formula for the rectangular case must be modified to

$$\sum_{i=1}^{4} p_{2i-1} \text{ XOR } p_{2i+1} + 2(\text{ NOT } p_{2i-1} \text{ AND } p_{2i} \text{ AND NOT } p_{2i+1})$$

The first term generates the crossing number defined over the four-connected points, and the second adds in any isolated corner points [DP81].

2.6.2 Geometrical paradoxes in hierarchical representations

The rectangular tessellation can tessellate itself in a regular fashion, that is a square may be simply tiled with squares. An equilateral triangle can be tiled with equilateral triangles, but the centre triangle will be upside down with respect to the immediately enclosing triangle. A hexagon cannot be tiled with complete hexagons. These properties become important when considering the hierarchical representations which will be discussed below. Problems arise with the hexagonal tessellation and also to some extent the triangular tessellation when deciding which level of a hierarchy a particular pixel belongs to.

2.7 Choice of tessellation

The semi-regular tessellations violate the need for translational invariance and it is clear that of the regular tessellations the hexagonal representation is simplest for connectivity analysis in a non-hierarchical representation. However, nearly all image processing work is done using square or rectangular pixels. Since the aspect ratio of a standard video picture is 4:3 [IB71] and since it eases indexing if a square array is used in memory, many systems use pixels with a 4:3 aspect ratio so as to make maximum use of the available field of view. Therefore an algorithm looking for circles in the image must look for ellipses in the memory array by applying a 4/3 correction factor to all Euclidean distances in the Y axis. This is an unfortunate state of affairs.

2.8 Distance information

Range finder devices may be used to detect distance information. It is not necessary or desirable to extend the array representation to a third dimension (which would require enormous amounts of storage) because occlusion of objects gives rise to only one distance datum for each point in the visual plane. Hence only one entry is required per pixel. Therefore distance information is stored as a pixel attribute like colour or brightness, rather than as an extension to the spatial digitisation scheme. If full scene representation is required (*i.e.* occlusion may not be used to reduce storage requirements) a higher level description is normally used.

2.9 Grey scale digitisation

Many early image processing systems operated on binary images only. In this case the incoming analogue signal is converted into a series of black-or-white pixels by a single comparator. The threshold is varied by changing the reference voltage to the comparator.

Most applications require the use of a grey scale with at least 16 levels. It is often convenient to work on 1 byte pixels, giving 256 possible grey levels. This corresponds to a dynamic range of 48.2dB, which is in fact beyond the capabilities of all but the most expensive video cameras.

Colour information is usually encoded separately for the red, green and blue components which requires a tripling of the basic system. Since three times

as much information is being stored, processing times will be three times longer, all else being equal. In fact extra information is different in kind (the three planes of colour information are not the same as a 24-bit grey scale), and this will inevitably make algorithms more complex: thus processing times may be even more extended. An alternative possibility is to store colour and luminance information separately. A compact code might be to display one of eight colours using three bits, and have a five-bit brightness code.

2.10 Simple structures for binary processing

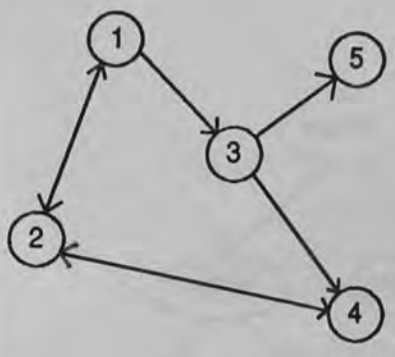
There is a great deal of redundant information present in an image, and in principle it should be possible to store only those parts of the picture that are strictly relevant to the task in hand. As well as reducing storage space, this will also allow more rapid searching and manipulation of the image as long as the required features have been efficiently coded. If the structure is not directly suitable then the conversion to a more appropriate form may impose unacceptable overheads.

Inspection problems usually require objects to be found and then analysed. Only in scene analysis or other rather complex situations is the background detail important. Several redundancy reducing techniques exploit this to remove background data and areas of uniform intensity in the foreground. Two fundamental approaches may be followed: direct representation of edges from which area information may be inferred (*e.g.* chain code, edge descriptors for shape approximation), or direct representation of areas from which edge information may be constructed (*e.g.* intensity threshold regions, texture segmentation).

2.11 Data structures

Any assembly of data forms a data structure consisting of nodes which store the data items, and pointers to other data items. The pointers may be implicit in the structure, especially if the topology of the data structure is invariant throughout its life, in which case a fixed mapping function is usually used to recover data rather than following a chain of links. The simplest example is a two dimensional array in which the mapping is

```
[column × column_dimension + row]
```



1: 2 • 3 /

2: 1 • 4 /

3: 4 • 5 /

4: 2 /

5: - /

	1	2	3	4	5
1	0	1	1	0	0
2	1	0	0	1	0
3	0	0	0	1	1
4	0	1	0	0	0
5	0	0	0	0	0

Figure 2.6: Graph representations

If a fixed allocation of nodes is used then the data structure is static, but if varying amounts of storage are required during execution, then the structure is dynamic.

The array representation can be considered as a static data structure, the form and storage requirements of which are fixed, in which only the data held in the nodes varies. As a result, no pointer information need be held within the structure, and a simple mapping is used to recover each pixel. On the other hand, the chain code has a dynamic topology and invariant 'data' — since the object is fully described by the pointers, no data field is required.

The most general data structure is the graph in which no restrictions are placed on the layout of the links. A graph is consists of a finite, nonempty set of vertices and a set of edges. If the edges are ordered pairs then the graph is said to be directed. If the edges are unordered pairs (*i.e.* sets) then the graph is said to be undirected. A graph may be represented in the form of a series of linked lists or an adjacency matrix: an $n \times n$ bit table indicating which of the n possible directed edges exist. These representations are shown in Figure 2.6.

A nondirected graph can be represented as $n(n-1)/2$ bits since the adjacency matrix is symmetrical about the leading diagonal.

Since a graph is a general structure, it follows that any structure can be represented in linked or tabular form. If there are n nodes in the graph and e links, or edges, then the linked representation requires storage proportional to $n+e$, and the adjacency table storage proportional to n^2. All else being equal, the number of edges in the structure dictates which representation will be the most efficient. However, execution time of algorithms is also dependant on representation.

In many structures, a hierarchy of nodes exists, and this gives rise to directed edges that point from one node to another. If there is a path along

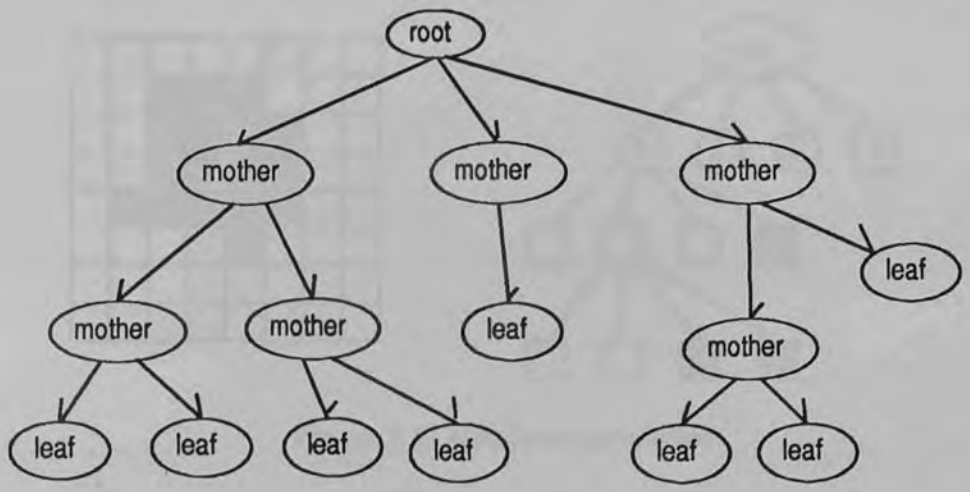


Figure 2.7: Tree structure

directed edges that links any two nodes in both directions the graph is said to be *cyclic*. An interesting subclass of directed data structures are the trees.

2.11.1 Trees

A tree is an acyclic directed graph with exactly one node (called the root) which has no edges entering it, and an arbitrary number of other nodes which have exactly one edge entering them [AHU75]. The root has *daughter* nodes which are themselves mother nodes to the nodes beneath them. At the top of the tree exist nodes which have no siblings: these are called *leaves* (Figure 2.7).

Trees have been used in image processing for the hierarchical segmentation of images. It is possible to have a tree which is built up using maximal blocks (blobs) in the image, with the largest uniform block as the root and the smaller blocks forming sibling nodes. However, such a general structure would only produce a more complicated representation of the image than the array. A more specialised tree structure that imposes some order on the image may speed up some processes, given an algorithm tailored to that representation.

2.12 Bottom up and top down representations

Many different hierarchical representations have been used for vision processing [Tan80]. They can be classified into two types: bottom up in which regions in the image are identified, grown and linked into larger regions that are again linked; and top down in which the image is decomposed by successive passes into more and more detailed representations. These correspond to local

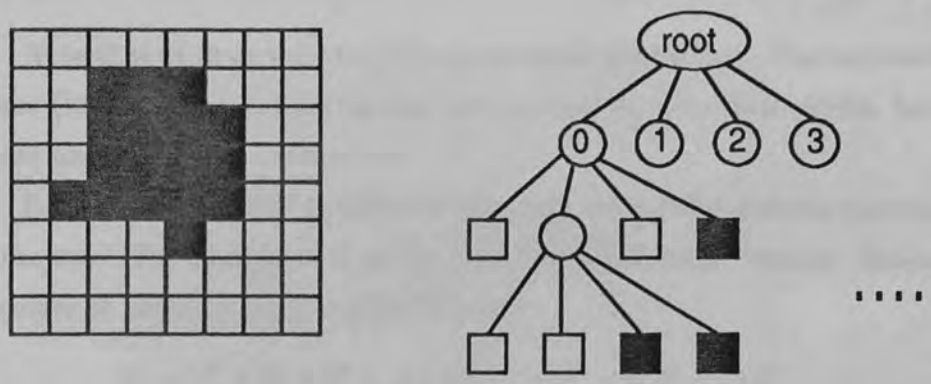


Figure 2.8: Quadtree generation

processing building up a global understanding of the image, and global processing that 'homes in' on specific local features of interest.

Note that although a particular representation may be most easily described in terms of its top down or bottom up emphasis, it may be possible to construct a top down representation using a bottom up algorithm. This will be illustrated in Chapter 4 using quadtrees.

2.13 Quadtrees

Chain code removes redundancy from the image by simply discarding all but edge information. An alternative approach is to encode an image on several hierarchical levels containing varying levels of redundancy. A simple representation called the quadtree has received considerable interest. The quadtree is unfortunately named since 'quadtree' is really the generic name for all tree structures of order 4 (that is all mother nodes have 4 daughters), and these occur in many areas apart from image decomposition.

The quadtree of a $2^n \times 2^n$ binary image may be constructed as follows.

1. If the entire image is black then the root is a black leaf, and therefore this is the only node in the tree.

2. If not, then create a mother node at the root, ascend one level in the tree and subdivide the image into quadrants. For each quadrant, create a leaf if it is uniform, otherwise create a mother node.

3. Subdivision of quadrants continues recursively until the entire image is stored in the structure.

This process is shown in Figure 2.8.

Note that at most only $n+1$ levels can exist in the tree. The bottom level of the tree (level 0) contains leaves that correspond to individual pixels. Level n of the tree maps to $n \times n$ pixel areas.

It would be possible to allocate adequate storage for a static representation of the tree. For each level n of the tree, there will be 2^{2n} nodes. Hence the total number of nodes N_p in a p level tree is:

$$N_p = 2^0 + 2^2 + 2^4 + \cdots + 2^{p-2} + 2^p = 4(2^p - 1)/3$$

hence for an image with $P = 2^p$ pixels, the maximal quadtree will require $4P/3 - 1/3$ nodes.

For a 128 $\times$ 128 pixel image, the largest possible tree would have 16384 leaves and 21845 nodes would be required. In this static representation, a straightforward mapping function could be used to access the nodes. Each node would need one of three values: black leaf, white leaf and parent, which may be encoded into 2 bits. Thus about 5.3K bytes of storage would be required, some of which would not be used in any other than the worst case.

Another form of direct mapping that is useful in controlling some image processing operations is to use multiple image planes to store (a) the original image, (b) the depth of each pixel in the tree, and (c) the grey-value of each pixel. Although requiring considerable memory space, this arrangement makes available to a conventional raster scanning operation all relevant information from the quadtree at each pixel without searching the tree.

The alternative is to use a full dynamic pointer storage scheme. On a PDP-11 or other 16-bit computer, this might be implemented with one word to each node. The bottom bit distinguishes between parent and leaf nodes. For a parent node, the other 15 bits form the word address of the first of four consecutively stored daughters. In a leaf node, the top 15 bits would contain grey-scale information. Although the binary quadtree would only require 1 bit, generalisation to grey scale and other 'simplicity' measures may make use of the extra bits. In the worst case, this scheme requires more storage (21845 $\times$ 2 bytes) than the static representation, but for many images with large uniform areas, and therefore fewer nodes, less storage will be required.

2.13.1 Extension to grey scale

The quadtree may be generalised by applying a 'simplicity' criterion other than simple uniformity. In particular, the strict uniformity measure is not suitable for use in the grey scale case because noise and texture in the image will

Figure 2.9: Q-image at threshold 16

allow very few truly uniform areas, that is regions of identical brightness. This will give rise to very many nodes, and result in large quadtrees. If the simplicity measure allows a range of pixel brightnesses to be present within a leaf region, then the quadtree will become smaller, but fine detail will be lost.

The quadtree of a binary image contains all the information required to reconstruct the original image, that is the image generated by deconstruction of the quadtree (the Q-image) is the same as the source image (the S-image). The grey scale quadtree does not contain all the information because of the smearing out of pixel intensities, and the resulting Q-image will tend to have a blocky appearance. Q-images with leaves spanning various brightness ranges are shown in Figures 2.9 – 2.12.

2.13.2 Properties and applications of quadtrees

Hunter and Steiglitz [HS79a] describe various limits on the complexity of the quadtree for polygonal figures and algorithms for the location of a point within a polygon and the filling of polygons in time linear with the number of nodes in the tree. These algorithms arise from cartographic work. A series of papers by workers at the University of Maryland describe algorithms for smoothing [RS81]; threshold selection [WHR82]; connected component labelling [Sam81]; skeletonisation [Sam83]; and edge enhancement [Ran81].

Figure 2.10: Q-image at threshold 32

Figure 2.11: Q-image at threshold 64

Figure 2.12: Q-image at threshold 128

2.13.3 Shift invariance

The quadtree is not a shift invariant structure. As a general rule, if a given representation is data driven (like the chain code) then its overall form will be independent of position. Only the coordinates of the leading link will need to be updated as an object makes its way across the field of view. On the other hand if a representation is coordinate driven, then positional information is imbedded implicitly or explicitly in the structure itself, and movement of objects will cause alteration of that structure. Of course even the basic array structure suffers from this in that an array of pixels effectively describes how well an analogue image matches the tessellation, and shifting of features will create quantisation noise as those features come in and out of alignment with the tessellation [LT82]. This problem is especially acute with the quadtree because the tessellation is of variable resolution, and in some cases the 'pixels' (leaves) are very large. This indicates that major corruption of the spatial relationships in the image may be caused. This means that any subsequent region oriented processing will be disrupted. Figures 2.13 – 2.20 show a rectangle at four positions within the image plane and the corresponding Q-images. The Q-image does preserve edges because busy regions of the image generate minimum size leaves, *i.e.* the original array representation is preserved. Thus the quadtree can be regarded as an edge oriented representation, even though at first sight it appears to be area oriented. The area information encoded in the tree is only an imperfect measure of the busy-ness of that part of the image within the limits set by the degree of matching to the quadtree leaves. This information may be useful in controlling certain algorithms

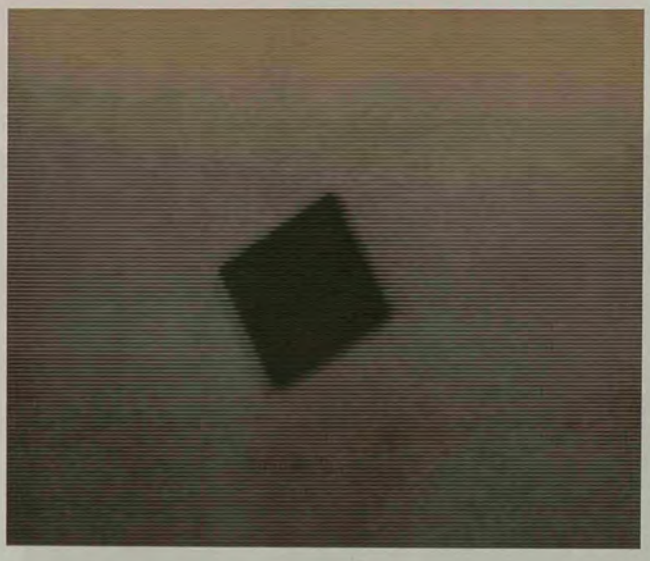


Figure 2.13: Rectangle 1

dynamically.

2.14 Metrics

Any representation imposes its own structure on the data. A 'grain' of some sort will be superimposed on a continuum, and therefore distance measures may be distorted. This is the case even for the array representation: the displayed image is noticeably pixellated, and the set of equidistant points around a point only approximates a circle. Both of these effects may be minimised by increasing the resolution of the representation because the underlying geometry of the representation is closely analagous to physical reality. Apart from the granular nature of the array representation, Euclidean distance measures applied to array coordinates will map to Euclidean measures of the coordinates in the original image. However, linked data structures do not necessarily provide the original geometrical relationships in an easily retrievable way, and other metrics may be appropriate.

A metric is a function, $d(x, y)$, that maps ordered pairs of coordinates into positive distances. For all points P, Q and R the following must be satisfied:

1. $d(x_P, y_Q) \geq 0$
2. $d(x_P, y_Q) = 0$ iff $x = y$
3. $d(x_P, y_Q) = d(y_P, x_Q)$
4. $d(x_P, z_R) \leq d(x_P, z_Q) + d(y_Q, z_R)$ { The triangle inequality }

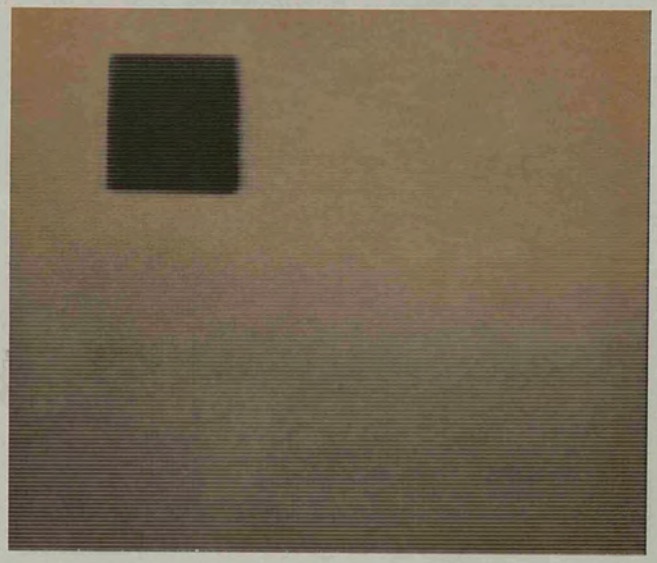


Figure 2.14: Rectangle 2

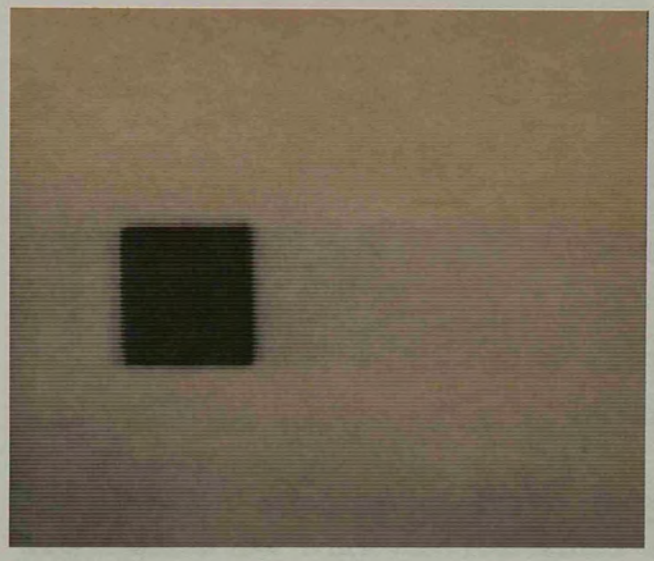


Figure 2.15: Rectangle 3

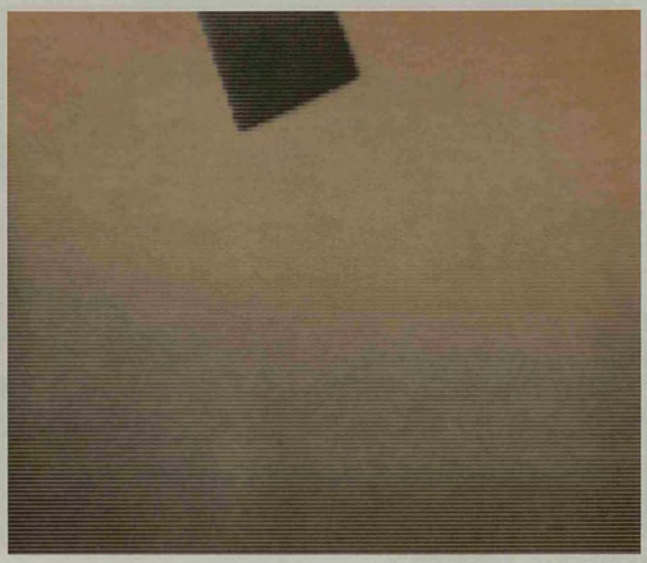


Figure 2.16: Rectangle 4

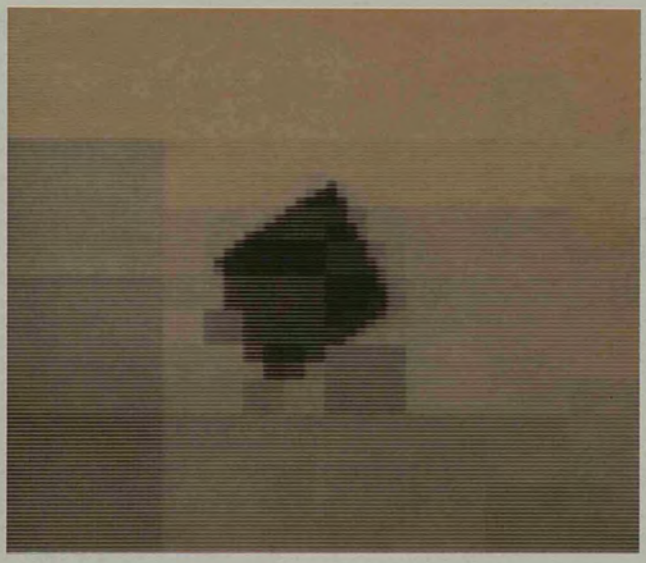


Figure 2.17: Q-image of rectangle 1

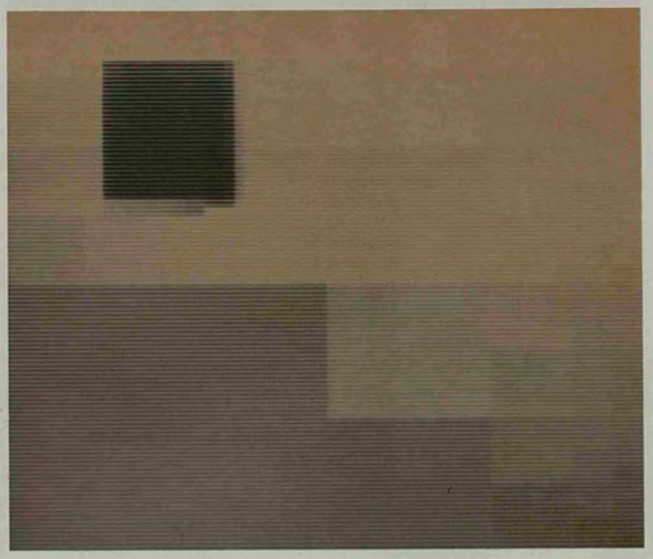


Figure 2.18: Q-image of rectangle 2

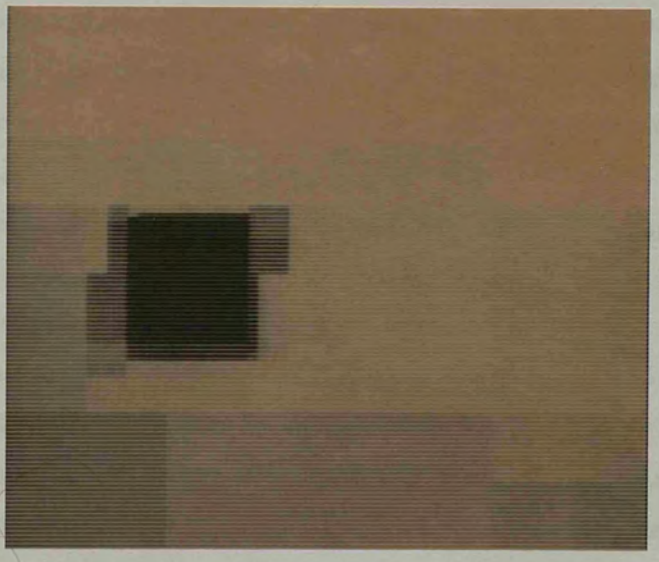


Figure 2.19: Q-image of rectangle 3

Figure 2.20: Q-image of rectangle 4

The most common metric is Euclidean distance which corresponds to the everyday measure of distance $d_E(P,Q) = \sqrt{(x_P - x_Q)^2 + (y_P - y_Q)^2}$

This can be used within a digitised image, but all the x and y coordinates must be integers, and so a general circle cannot be drawn. An approximation can be made by rounding the solutions of the circle equation to integers.

Two other metrics which are used in image processing are the City Block distance $d_{CB}(P,Q) = \mid x_P - x_Q \mid + \mid y_P - y_Q \mid$ and the Chessboard distance $d_C(P,Q) = \max(\mid x_P - x_Q \mid, \mid y_P - y_Q \mid)$

The chessboard metric has the property that the set of points distant Q from a given point P yields a square centred at P with side length $2Q$. Samet [Sam79] has proposed the use of the chessboard distance for quadtree based work and presents an algorithm for calculating it efficiently.

Within a structured representation of an image, two types of metrics may be distinguished: those that relate directly to distances in the original image, and those that describe the separation of nodes in the structure. For instance, Figure 2.21 shows a quadtree where two leaves that are adjacent in the Q-image are separated by 6 levels in the tree representation. This is another kind of 'grain' imposed on the data, and may cause gross inefficiencies in the retrieval of pixels if inappropriate structures are applied to a problem. Samet [Sam82] describes neighbour finding techniques for images represented by quadtrees, that is the conversion of intra-image distances to intra-structure distances.

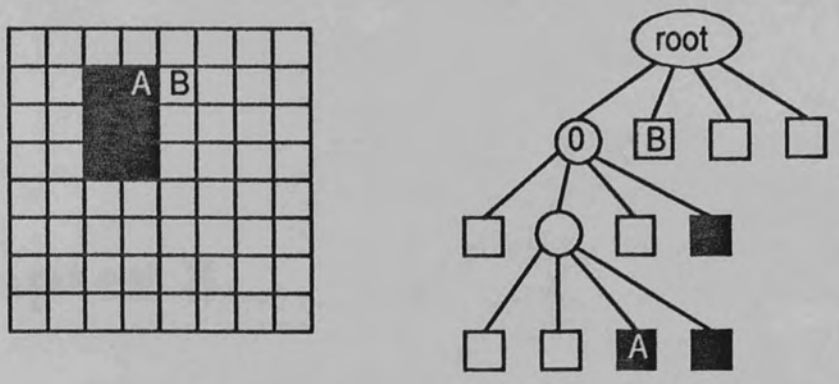


Figure 2.21: Distances within a quadtree

2.15 Conclusions

Digitisation of the visual field has been described with particular emphasis on spatial quantisation. It has been shown that the hexagonal tessellation has desirable properties, but noted that for hierarchical representations the rectangular tessellation is probably preferable. Of the hierarchical representations, the quadtree have been described along with some of the broader, and in some ways problematic, implications of its use in image analysis. Algorithms for quadtree generation and their uses in real time image processing will be described in Chapter 4.

Chapter 3

Algorithm analysis and design

> " 'Begin at the beginning', the King said gravely, 'and go till you come to the end; then stop' "
>
> Lewis Carroll, 'Alice's adventures in Wonderland' (1865)

3.1 Introduction

A procedure in the widest sense (not as a syntactic construct in programming languages) is

> "a finite sequence of well defined steps or operations, each of which requires only a finite amount of memory or working storage and takes a finite amount of time to complete" [GH77].

The definition of a true algorithm is essentially the same, but more precise in that

> "...an algorithm must terminate in finite time for any input".

Algorithms may be classified by function, *i.e.* what they do; by strategy, *i.e.* how they do it; or by goodness, that is how well they do it. The goodness measure varies according to application, but accuracy, execution time and memory requirements for different inputs will be the main criteria.

A topic of universal interest in computing is the 'benchmarking' or comparative testing of systems. Typically this is attempted by the adoption of benchmark algorithms which are run on the competing systems to give measures of execution times. For these figures to be useful suitable algorithms must exercise all parts of the system in a way which is consistent with actual usage.

In this chapter, benchmarking of image processing systems is considered after a discussion of algorithm analysis and design.

3.2 Algorithm analysis

In image processing it is often difficult to say whether an algorithm is 'correct' — two different edge detection operators will usually give different output images, and the assessment of which better defines the edge may be primarily subjective. The rigour of image processing algorithm analysis would be improved if universal measures of accuracy for fundamental operations could be agreed. Davies [DP81,Dav84] has presented useful results in thinning and edge detection.

Assuming that the algorithm under study does actually present correct answers to the problem, the most important aspects of its behaviour are the amount of storage space required and the execution time. These may be characterised for varying inputs as the space complexity and the time complexity of the algorithm.

The first step in determining the space or time complexity of an algorithm is to define an integer number called the size of the problem. In a tree type problem the size might be the number of leaves or edges in the input data; with an adaptive filtering problem the size might be the number of samples falling within the range of values which trigger the filter. The asymptotic complexity of an algorithm is the limiting behaviour as the size of the input increases, and this ultimately limits the size of the problem solvable on a given system.

Both the worst case and the average complexity of an algorithm are of interest. It may be that a particular algorithm performs well for most inputs, but almost grinds to a halt with others. It would be useful to identify dangerous inputs and filter them out. In general the worst case complexity is easier to calculate.

An upper bound on the number of basic instructions, performed by the algorithm for a given size of problem n, is defined as the work function, $f(n)$. If $f(n)$ grows at or below the speed of a simple (finite length) polynomial in n then the algorithm is said to be polynomial. If not then the algorithm is exponential. In many cases this simple division is all that is required since most systems seem to be capable of executing polynomial algorithms in a reasonable time, whereas an exponential algorithm can be guaranteed to stall for all but the smallest of problems. A simple example of this has already been seen in Section 1.1 where the point to point comparison of all 16×16 binary patterns against a single test pattern was shown to require around 10^{71} seconds on a fast processor. This problem is exponential in the size of the image, since the number of images is 2^n where n is the number of points in the pattern.

A notation from limit theory is often used in algorithm analysis. A

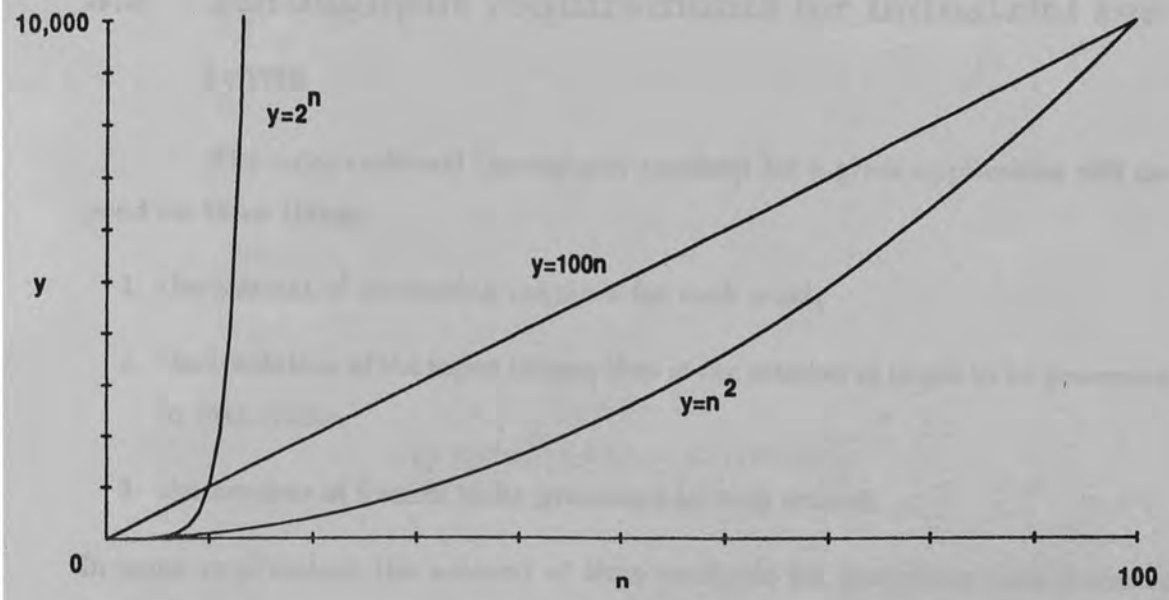


Figure 3.1: Behaviour of functions

function $f(n)$ is said to be order $g(n)$ for large n if

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = \text{constant } (\neq 0)$$

This is written $f(n) = O[g(n)]$

If for some function $h(n)$ and large n

$$\lim_{n\to\infty} \frac{f(n)}{h(n)} = 0$$

then $f(n) = o[h(n)]$

These terms are spoken "big o" and "little o". If $f(n)$ is $O[g(n)]$ then the two functions increase at similar rates as $n \to \infty$. If $f(n)$ is $o[g(n)]$ then $g(n)$ grows much more rapidly than $f(n)$.

The 'upper bound' definition of the work function given will generate the worst case behaviour. If some kind of average number of operations is used instead, then the complexity of the average run will be formed.

Figure 3.1 shows the behaviour of several kinds of function for increasing n. It will be seen that for small n, the constant of proportionality in the complexity relation is important; for instance an algorithm which is $O[n^2]$ is preferable to one which is $O[n]$ if the constant of proportionality is 100 for $n < 100$. In fact even an algorithm $O[2^n]$ is better for $n < 10$.

Indiscriminant use of order relations in algorithm analysis can be misleading when selecting algorithms for actual applications because the constants of proportionality may dominate.

3.3 Throughput requirements for industrial systems

The computational throughput required for a given application will depend on three things:

1. the amount of processing required for each pixel,
2. the resolution of the input image, that is the number of pixels to be processed in each frame,
3. the number of frames to be processed in each second.

In some applications the amount of time available for processing each frame is limited only by the patience of the programmer. Images returned from outer space can be subject to very long processing times. However the processing of weather satellite images to provide cloud cover and wind speed observations must proceed fairly rapidly because the data will be useless for forecasting within a day or so. Although the weather forecasting must proceed in real time, it is a different order of time to industrial real time, where control information is typically required from second to second. A particularly taxing situation occurs when 100% inspection of a line is required.

In this case the production line must be completely mapped into images. Figure 3.2 shows the simple case of a single camera covering the entire width of the belt with a strip s metres deep. Typically a 50% image overlay will be required to resolve registration problems and avoid splitting of objects between successive frames. The belt is moving at speed v, and $1.5\ v/s$ images will be required in unit time. Belt speeds of above $1ms^{-1}$ are unusual due to slippage of products on the conveyor when traversing bends at such speeds, and a figure of $0.5ms^{-1}$ is far more typical [Gre84]. For the study described in Chapter 9, each image subtended 0.12m on the belt which was moving at just over $0.4ms^{-1}$. In this case around 3.3 images a second would need to be processed for 100% coverage.

The amount of processing per frame is of course totally dictated by the algorithm in use. Even the simplest operation such as a global threshold for a 128 $\times$ 128 image will require 16384 pixel-read, compare, pixel-write operations and a corresponding number of coordinate counter updates.

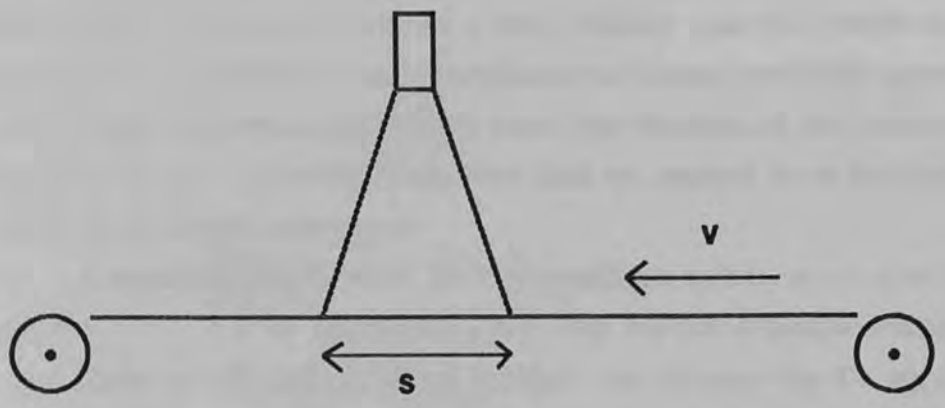


Figure 3.2: Conveyer belt coverage

3.4 Algorithm design

Formalising algorithm design is extremely difficult because of the intuitive and creative powers used in the design process.

Analysis of successful techniques can at best provide pointers to useful strategies. However, the ad hoc construction of algorithms using intuition or badly justified heuristics should be avoided.

The definition of an algorithm given earlier is rather wide ranging. On most computers addition is part of the machine code instruction set and an 'algorithm' to perform addition would be rather trivial — '`ADD a TO b`'. This is really no more than a restatement of the problem and hardly qualifies for rigorous analysis. On the other hand, division is a fundamental instruction on some machines and not others. In the absence of the relevant hardware, division can be quite complicated to implement, especially in floating point.

This raises the question of what exactly the fundamental 'unit' of an algorithm is and how to measure the operation rate of real systems. One has a strong intuitive feeling that any normal computer can be programmed to 'do anything', but that some machines require more detailed instructions than others. In fact Gödel's Incompleteness Theorem [God62] show that there are problems that cannot be proved or computed in any given formal system including that of a digital computer. However this still leaves the problem of what constitutes an elemental computation within the class of computations that are possible. This study is the province of automata theory, and an effort to derive space and time complexity for an algorithm from first principles might involve programming that algorithm on a Turing machine [Tur36] or other low level automaton. Some higher level models of computation than the pure Turing machine can be just as general

[AHU75] but for practical purposes a real machine must be considered. The constants of proportionality in an order relation are almost completely governed by the actual implementation and in some cases even the form of the function itself. Apart from these, it is undoubtedly true that an analysis from first principles would be lengthy and error prone.

A secure approach would be to characterise each instruction of the real machine in terms of a Turing machine, and treat the actual machine instructions as macro calls to a library of Turing routines. In this case the Turing machine is emulating the real machine, and a different set of macros will be required for any other machine. Thus it is clear that when analysing from first principles, only the general form of the relation may be discovered, and exact details of the complexity of an algorithm will always be implementation dependant. Here accuracy militates against generality.

An alternative and popular approach is to take a real or a paper architecture and use that as the standard machine. Reduced instruction set machines (which are discussed in Chapter 6) are especially useful for this because a richly featured architecture provides more opportunities for unexpected side effects and can lead to the use of 'short cuts' which fail in unexpected ways. Knuth [Knu79] designed a paper computer called MIX and used it to describe an encyclopediac collection of algorithms. A more modern approach would to be to use one of the so called algorithmic languages such as Algol-68 or Pascal. However these languages all introduce semantic ambiguities to one degree or another because of failings in their specification or implementation. Such details may well tie a program to a given machine because of the lack of consistency across different compilers. Some workers [AHU75] have used a subset of Algol-60 for control structures and simple statements and described more complex actions in English. Whilst this can be very useful for teaching, and for mapping out algorithms at the design stage, it is hardly a rigorous means of communicating algorithms.

The work presented here is almost all implemented on PDP-11s in several high and low level languages, and as a result this provides the implementation model. Although this is unsatisfactory for the reasons already described, it has the advantage of providing a real system which may act as a semantic arbiter in all cases, that is in cases of ambiguity the actual operation of the system described will resolve that ambiguity. The PDP-11 is especially useful in this role because it a very widely available machine.

3.4.1 Operations and algorithms

Many operations in image processing involve the straightforward calculation of functions of pixels, and involve little or no transfer of control. Similarly, many image processing programs are made up of sequential combinations of these operations. For instance, an object location routine might perform a median filter to remove noise from the image, detect edges using a Sobel filter and then threshold to leave an edge description of each object. Simple propagation functions can then label each object in turn. Although this program clearly constitutes an algorithm, its simple control flow constitutes a trivial design strategy which might be labelled *direct computation* of the result. The situation here is analagous to that noted in Section 3.4 where operations such as addition and multiplication form the elemental units of algorithm design on conventional computers. Processes which require only trivial branching (such as termination of loops) can be called *operations*, as opposed to complex algorithms. Thus by definition an operation uses a degenerate 'strategy' that proceeds to the result by direct computation.

It should be noted that this distinction is relative and context dependant. For instance, although the straightforward implementation of the Sobel filter is undoubtedly an operation, removal of redundant calculations and retention of window points for later processing can significantly improve performance, and generate a routine that is too complex to be considered as a simple operation [Lee83,Pic84].

A window operation is defined as an operation acting over a limited number of pixels that is repeatedly applied across the image in a regular fashion. Window operations are especially important because of the availability of parallel processors that can apply a window operation at all points in an image simultaneously.

3.5 Problem solving strategies

3.5.1 Hill climbing strategies

A knowledge of the form of the solution space of an algorithm allows successive passes of an algorithm to 'home in' on a required solution. In classical pattern recognition, training on a set of n features will result in clusters of training points in an n dimensional space. If the features have been chosen well, each cluster will correspond to one of the training classes. Discriminators for the classes may be derived by finding the peaks and saddle points of cluster densities, and hill

climbing techniques are readily suited to this. True hill climbing can at best only detect local maxima, and so an unsuitable choice of starting point may result in spurious results. Once a local maximum has been discovered, further exploration around that maximum may be used to discover contours and other local maxima. These iterative processes can be very time consuming, but this is not a problem if they yield discriminators that may be rapidly calculated when the system is operating in testing mode.

3.5.2 Backtracking and recursion

The quadtree generation algorithm described in Section 4.5 reserves decisions about whether a node should be a leaf or a parent until it has examined at least one point in the succeeding node. If a leaf is required, it steps back to that node, creates the leaf and then picks up again where it left off. This strategy is called backtracking, and often appears in so-called 'bottom up' algorithms. In a top down algorithm, the entire problem domain is scanned, and decisions as to further processing or classification made in a global fashion. These decisions are then passed on to lower level processing that continues the algorithm. An interesting class of top down algorithms implement the same processing at each level, with the sub-algorithm calling itself recursively. The definition of a quadtree given in Chapter 2 constitutes a recursive top down algorithm.

In a bottom up algorithm, elements of the problem domain are examined sequentially until enough information has been built up to decide on a higher level action. Backtracking is also often used for language parsing, where the lexical analyser scans source code characters until a decision can be made concerning the symbol in use. Good language design can result in significantly reduced overheads. Pascal has been designed to be parsable using one symbol lookahead, which effectively requires no backtracking. In practice this means that during the scanning of the source text no ambiguities can be created that would require parsing to be deferred.

When faced with a new problem, the natural strategy is to subdivide it in a top down fashion into sub-algorithms. This allows separation of the global aspects of a problem from the low level details. As such, most modern programming languages seek to enforce top down programming. However, this does not necessarily generate efficient algorithms. The quadtree generation algorithms described later in this chapter are good examples of the benefits of bottom-up programming.

3.6 The benchmarking problem

As noted in Section 3.4, working from first principles will not produce the absolute complexity for a real machine and each machine/algorithm combination must be analysed separately. Conversely, universal benchmarks cannot be found. However comparison of actual machines is vital for progress to be maintained, and so some benchmarking methodology must be arrived at even in the absence of strict rigour.

Any problem has boundary conditions, and the programmer rapidly becomes aware that these are the 'danger zones' for an algorithm. Unexpected input may cause attempted division by zero, or a normally well behaved work function may show a discontinuity around certain magic numbers. Identification of boundary conditions may be sufficient to generate worst case behaviour, and therefore these are attractive from a benchmarking point of view. For instance, a simple edge detector might have a time complexity of $O[n]$ where n is the number of points in the image which are sufficiently busy to trigger the edge marking routine. The worst case behaviour therefore occurs for a maximally busy picture. This image should have full dynamic range discontinuities at every point, *i.e.* a black and white checkerboard pattern at the spatial resolution of the frame store.

Although identification of worst case behaviour is important for disaster prediction, or for throughput calculations when 100% inspection must be guaranteed, the average behaviour is often of more practical interest. The problem here is that average behaviour is not well defined. Some industrial inspection problems may be tightly constrained to dark objects on a light-coloured conveyer belt, and others may involve more complex images. So benchmarking is a doubly ill-defined problem — not only are the implementation details critical and varying for any set of machines, but the input data can be critical and will vary across runs as well as across applications.

Some efforts have been made to set up standard benchmarks for scientific and other mixes of computing [Wei84a]. An alternative to standard benchmark algorithms is to characterise the processor speed in terms of the instruction execution frequency. Two popular measures are 'MIPS' (millions of instructions per second) and the 'megaflops' (millions of floating point operations per second) ratings. Since instruction sets vary widely in power (that is the amount of useful work that can be performed in one instruction) the MIPS rate is not a useful measure across processors with different instruction sets (MIPS is said to be an acronym for 'meaningless indicator of processor speed'), but it may be useful in

comparing different implementations of a single architecture (although even this can be misleading) [Bel78].

The problem here is that a MIPS rating describes how a processor goes about a task rather than what it achieves. Consider two PDP-11s, one with a hardware multiply instruction and one without. The Pascal fragment

```
VAR i:integer; BEGIN i:=i*23 END;
```

will generate very different code for the two machines:

For the machine with multiply, we have this code:

```
$2:             ;entry point from run time initialisation
        MOV     i,r0    ;get variable to register
        MUL     #23,r0  ;multiply by constant (10.69--12.29 us)
        MOV     r1,i    ;put result in variable area
        JMP     $3      ;exit to RT-11 through run time library
```

In the other case, we have

```
$2:             ;entry point from run time initialisation
        MOV     i,r5    ;get variable to register
        MOV     #23,r0  ;get constant to register
        BR      3$      ;skip
1$:     ADD     R0,R1   ;shift-and-add multiply (2.16us)
2$:     ASL     R0      ;                       (2.31us)
3$:     CLC             ;                       (2.16us)
        ROR     R5      ;                       (2.16us)
        BCS     1$      ;             (2.31us on branch,
                        ;           1.76us if no branch)
        BNE     2$      ;             (2.31us on branch,
                        ;           1.76us if no branch)
        MOV     R1,i    ;put result in variable area
        JMP     $3      ;exit to RT-11 through run time library
```

All timings are for PDP-11/34a with MOS memory.

Now on a PDP11/34a, the hardware `MUL #N,R0` instruction can take between 10.69 and 12.29μs depending on the data. A long sequence of these instructions would therefore execute at between 0.081 and 0.093 MIPS. On the other hand, the software implementation of `MUL #N,R0` uses a loop containing instructions that execute very quickly (individual instruction timings are noted in the listing) and these generate a composite MIPS rate of about 0.44 for a sequence of emulated multiplies.

If the processors were compared on the basis of how long it took to perform the multiply, then naturally the hardware multiply wins. Although this is a simple example, confusion can and does arise. The extreme case occurs when

proponents of reduced instruction set computers (RISC architectures) directly compare their MIPS rates with those of a VAX or other conventional machine. Since RISC architectures are specifically designed to implement a small number of simple operations that execute rapidly, they naturally exhibit high MIPS rates. Using this analysis, the non-hardware multiply PDP-11 would therefore be more 'powerful'. Various works have defined standard mixes of instructions for different applications (commercial, scientific, time shared editing etc.) in an attempt to circumvent this problem. However, it is unusual for manufacturers to quote the mix used when deriving their figures.

Clearly, what is required is a functional measure of processor performance. The megaflop is a more useful measure because it starts to move away from the machine dependency. However, since the particular floating point operation is not defined, a spread of rates may still be present. Even for a single operation, the execution rate may be data dependant, as in the case of the integer multiply noted above.

To summarise: benchmark programs are unsatisfactory because of their lack of generality and dependence on complex interactions between data and architecture. Attempts to reduce interactions lead to direct comparison of individual instruction rates, but even at this level precise measurements are not possible because of the lack of universally available and comparable operations. The only reliable measure of processor speed is completely functional (*i.e.* how long does it take to do a particular job) and this is naturally non-general. The best compromise may well be to use a suite of benchmark programs and to treat results conservatively.

3.7 Benchmark images

Data for benchmarking programs is always important, but especially so in image processing. As already noted, systems tend to exhibit their pathological behaviour when faced with boundary conditions. The boundary conditions for a sorting algorithm may be fairly easy to recognise, but for complex algorithms or pieces of hardware, the limits of operation may be only empirically determinable. In such a case it is probably best to supply an image with a rich mix of image features, and 'home in' on anomalous behaviour. Four images are presented here for use in benchmarking:

1. Abingdon Cross (CROSS),

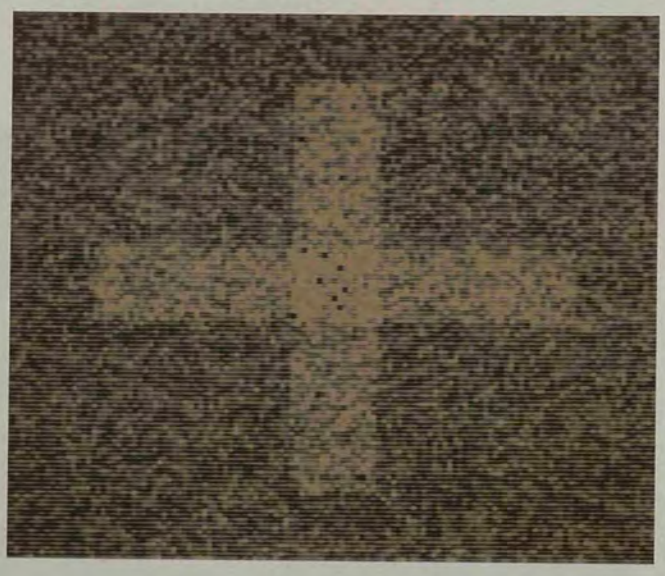


Figure 3.3: Abingdon Cross

2. Nuts and Bolts (ANB),

3. Pen (PEN),

4. Biscuit (BISCUIT).

These images are shown in Figures 3.3 – 3.6. The Abingdon Cross is an artificially constructed image formed from two orthogonal bars overlaid with noise. It was proposed as the basis of a skeletonisation benchmark at the Abingdon Workshop on Multi-Computers for Image Processing in 1982. The program to generate this image was rewritten from a routine proposed there which unfortunately contained several ambiguities, such as the use of uninitialised variables.

The nuts and bolts were photographed by the author to give a scene with relatively few objects on an unevenly lit background with shadows and a significant amount of noise. The objects also have glints and varying surface textures. This image has been in use as a benchmark in this laboratory for some years.

The pen picture is interesting because it contains some writing that is only just readable. When used to benchmark smoothing algorithms, any degradation of the image sharpness is immediately apparent in the readability of the lettering.

The biscuit picture is an example of the foodproducts inspected using the IMP system described in Chapter 9. It has some three dimensional structure and is not well defined like the nuts and bolts.

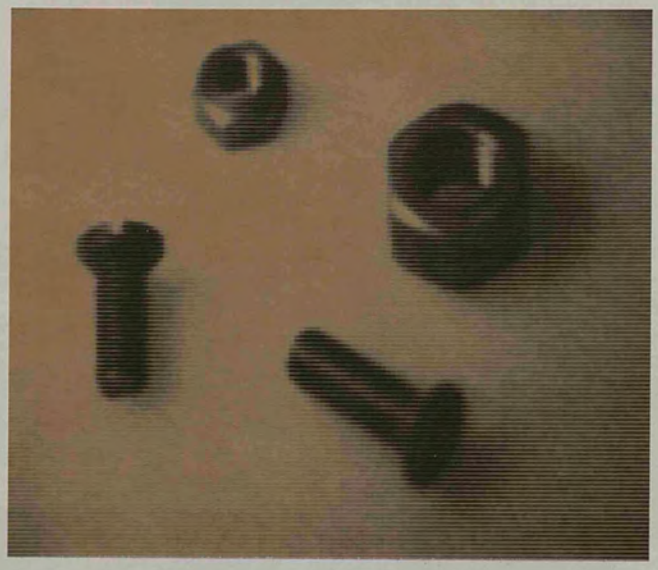


Figure 3.4: Nuts and Bolts

Figure 3.5: Pen

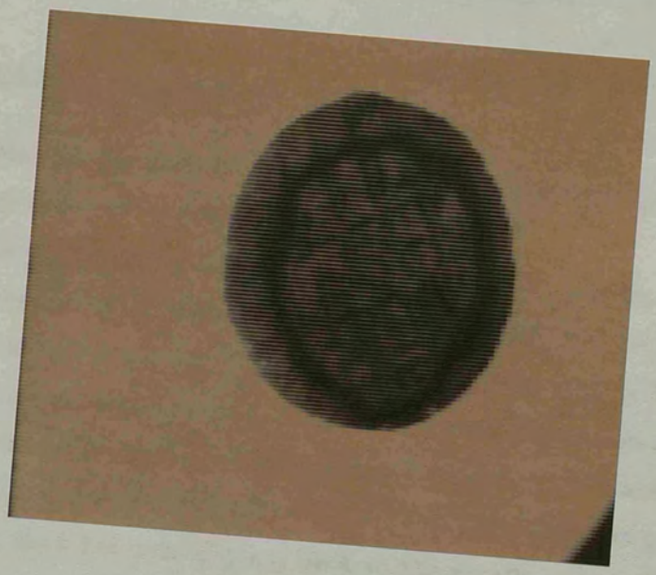


Figure 3.6: Biscuit

Chapter 4

Quadtree algorithms

"It is not enough to take steps which may some day lead to a goal; each step must be itself a goal and a step likewise" — Goethe

4.1 Introduction

This chapter presents a selection of algorithms for quadtree generation and quadtree controlled image processing.

The presentation of algorithms is a difficult process, since the reader requires a knowledge of overall purpose, strategy and low level details to fully comprehend a real program. Typically, high level strategy is highly dependant on the low level details, and so a simple 'top down' exposition is inadequate. In fact any sequential exposition of an algorithm's strategy and tactics is likely to be inadequate, and understanding may only come from an iterative process of looking at the details, then the overall plan, back to the details and so on. In an effort to simulate this process in a necessarily sequential text without inducing boredom, the algorithms in this chapter are introduced by a discussion of the problem and overall strategy, followed by a listing of a Pascal implementation and a detailed commentary. The performance of the algorithm is analysed and the strategy discussed in the wider context from which suggestions for improvements often arise.

4.2 Quadtree generation

Quadtrees were described in Chapter 2 and are an important data structure in image processing because (a) they may be rapidly calculated and (b) they can provide global information concerning the busy-ness of an image in a relatively explicit way. Quadtrees are also extensively used in Geographical Infor-

mation Systems (GIS) to represent maps which are sparse and therefore generate compact trees. Four algorithms for quadtree generation from an array representation are presented here: (1) a simple recursive decomposition algorithm that closely mirrors the description given in Chapter 2; (2) a bottom up leaf merging algorithm; (3) a fast bottom up algorithm that uses backtracking to save multiple scans; and (4) a sequential algorithm that avoids backtracking by directly calculating the leaf structure of a quadtree. Quadtree generation is examined in such detail here because of the lessons that may be drawn concerning the mapping of algorithms onto real architectures.

All four of these algorithms operate on grey scale images, though algorithm 4 produces approximate results with anything other than a binary image. The definition of a suitable *smoothness measure* may be dictated by practical constraints. Ideally, each subquadrant should be scanned to generate the mean and standard deviation of the intensity histogram. Speed considerations may require the use of cruder and less rigorous measures. The monochrome quadtree generation algorithms simply have to decide whether a subquadrant is all black or all white, which is equivalent to detecting the range of pixel values across a subquadrant. In the grey scale case, this may be generalised to checking the range of pixel values against some threshold. This can give good results, but is noise sensitive. A compromise might be to threshold against the 5% and 95% points in the histogram.

One important aspect of the smoothness measure is whether it may be calculated on the fly in a sequential manner, or whether the entire quadrant must be scanned before a meaningful result emerges — *i.e.* whether a parallel or serial smoothness measure is used. These three algorithms all use the absolute spread of brightness values to measure smoothness, and this may be calculated on the fly. Algorithm 1 is easily adapted to any other smoothness measure, algorithms 2 and 3 depend on using a sequential measure, and algorithm 4 only produces strictly accurate results for binary images.

4.2.1 A note on terminology

The total number of nodes in a quadtree is denoted by N. L and P are the total numbers of leaves and parents respectively. Clearly $N = L + P$. The number of nodes at a level i is n_i, likewise l_i and p_i are the number of i-level leaves and parents. The results in this chapter apply to $s \times s$ images where s is usually a power of 2. The results may easily be generalised to images of any other size by

Level	Size of quadrant	Area of quadrant	N
0	1 x 1	1	1
1	2 x 2	4	5
2	4 x 4	16	21
3	8 x 8	64	85
4	16 x 16	256	341
5	32 x 32	1024	1365
6	64 x 64	4096	5461
7	128 x 128	16384	21845

Table 4.1: Quadtree space requirements

adding a border of dummy pixels to side length up to the nearest power of 2.

4.2.2 Analysis of quadtree algorithms

Table 4.1 summarises the space requirements for various levels in a quadtree.

This maximum number of nodes is useful for worst case estimation of quadtree-based algorithm performance. Average performance (which will be linked to average size) is more difficult to derive. The actual size of a quadtree will depend on the smoothness threshold (for grey scale images) and the busy-ness of the image. A useful average measure would give the expected size of the quadtree for a large range of images and thresholds, and also perhaps a means of weighting the average for specific types of application.

In the long run, any given quadtree may arise. One way of expressing this is to assume that any potential node of the tree is equally likely to produce a leaf or a parent. This form of average has been used to analyse tree traversal algorithms [Sam82]. However, such an average leads to an immediate paradox. If there is a probability of 0.5 that any node in the tree may be a leaf, then on average half of the possible nodes will be leaves. Table 4.2 shows the area requirements that result from such an assumption. Naturally this situation cannot occur for any real image, because the total area of the quadrants corresponding to leaves is four times the area of the image.

A more meaningful average may be defined using the distribution of actual pixels amongst the levels in the tree. Assume that, in the long run, each pixel has an equal chance of mapping into a leaf at any level. Then on average,

Level	Area of quadrant	Average number present	Total area
0	1	8192	8192
1	4	2048	8192
2	16	512	8192
3	64	128	8192
4	256	32	8192
5	1024	8	8192
6	4096	2	8192
7	16384	0.5	8192

			65536

Table 4.2: Quadtree leaf totals

equal areas of the image will be occupied by leaves belonging to each level. There are $\log s + 1$ possible levels for a square image of s^2 pixels, hence the average area occupied by leaves belonging to level i will be $a_i = s^2/(\log s + 1)$. Since at level i the quadrant area is $a_{qi} = 2^{2i}$, the average number of level i leaves will be $l_i = s^2/(2^{2i} \log s + 1)$. Values for a 128×128 image are shown in Table 4.3. The number of parent nodes at level i, p_i, can be derived in general from the number of leaves l_i. Each group of four nodes at level i link to a single node at level $i+1$, *i.e.* $p_i = (p_{i-1} + l_{i-1})/4$ with $p_0 = 0$.

This gives for n_i, the number of nodes at level i ($n_i = p_i + l_i$):

$$n_i = \sum_{j=0}^{i} 2^{-2(i-j)} l_j$$

Hence

$$P = \sum_{k=0}^{\log s} \sum_{j=0}^{k} 2^{-2(k-j)} l_j$$

and

$$N = \sum_{k=0}^{\log s} \sum_{j=0}^{k} 2^{-2(k-j)} l_j + l_k$$

As can be seen from the above table $P_{av128} = 657.5625$ and $L_{av128} = 2666.75$ so $N_{av128} = 3324.3125$.

The definition of average used above assumes no prior knowledge of the type of application. In many cases it may be possible to characterise the application in such a way as to provide a weighted version of the average size that more fully reflects actual quadtree sizes. Section 4.8 examines data compression

Level	Quadrant area	Total area	Average leaves	Average parents
0	1	2048	2048	0
1	4	2048	512	512
2	16	2048	64	256
3	64	2048	32	80
4	256	2048	8	28
5	1024	2048	2	9
6	4096	2048	0.5	2.75
7	16384	2048	0.25	0.8125
			---	---
			2666.75	657.5625

Table 4.3: Quadtree mean leaf totals

using quadtrees. In this case a target is given for the total number of leaves required (say $(\log s)/10$ which will compress the data by an order of magnitude) and the threshold adjusted until a suitable tree has been generated.

Where the type of data being processed is relatively uniform, it is possible to build a real probability distribution for leaf and node occurrence by testing many images. This distribution may then be used in place of the square distribution.

4.3 Algorithm 1 — top down recursive decomposition

In Section refq:def the quadtree was defined in terms of the recursive subdivision of an image. Algorithm QUAD TOP RECURSE below is a straightforward implementation of this definition.

```
 1 PROCEDURE QUAD_TOP_RECURS(xstart,ystart,size: integer);
 2 VAR
 3  min, max, xfinish, yfinish, temp: integer;
 4 BEGIN
 5  xfinish:=(xstart+size); yfinish:=(ystart+size);
 6  min:=maxint; max:=0;
 7  x:=xstart;                    {scan quadrant}
 8  REPEAT
 9   y:=ystart;
10   REPEAT
11    IF p0>max THEN max:=p0; IF p0<min THEN min:=p0;
```

```
12    y:=y+1
13   UNTIL y=yfinish;
14   x:=x+1
15  UNTIL x=xfinish;          {scan quadrant}
16  IF max-min<thresh         {test}
17  THEN
18   BEGIN                    {fill quadrant}
19    temp:=(max+min) DIV 2;
20    x:=xstart;
21    REPEAT
22     y:=ystart;
23     REPEAT
24      q0:=temp; r0:=size;
25      y:=y+1
26     UNTIL y=yfinish;
27     x:=x+1
28    UNTIL x=xfinish;
29   END                      {fill quadrant}
30  ELSE
31   BEGIN                    {subdivide}
32    temp:=size DIV 2;
33    QUAD_TOP_RECURS(xstart,ystart,temp);
34    QUAD_TOP_RECURS(xstart+temp,ystart,temp);
35    QUAD_TOP_RECURS(xstart,ystart+temp,temp);
36    QUAD_TOP_RECURS(xstart+temp,ystart+temp,temp);
37   END                      {subdivide}
38 END;
```

4.3.1 Commentary

The procedure takes as parameters the coordinates of the top left corner and the size of the quadrant to be scanned. The smoothness threshold is accessed *via* global integer `thresh`.

Line 5: the endpoints of the quadrant (the coordinates of the bottom right corner plus 1) are calculated from the parameters.

Line 6: `max` and `min` are used during the scan of the quadrant to hold the current brightest and darkest pixel values. They are initialised here to worst case values of 0 and maxint respectively.

Lines 7–15: the quadrant is scanned to find the spread of brightnesses.

Line 16: if the brightness spread is within threshold, then a leaf is created, *i.e.* the corresponding quadrant in q-space is filled with the leaf colour, and the quadrant in r-space is filled with the leaf size (lines 18–29). If the spread of values is out of range, then QUAD_TOP_RECURS is recursively called for each of the four subquadrants (lines 31–36). At the lowest level (where nodes map

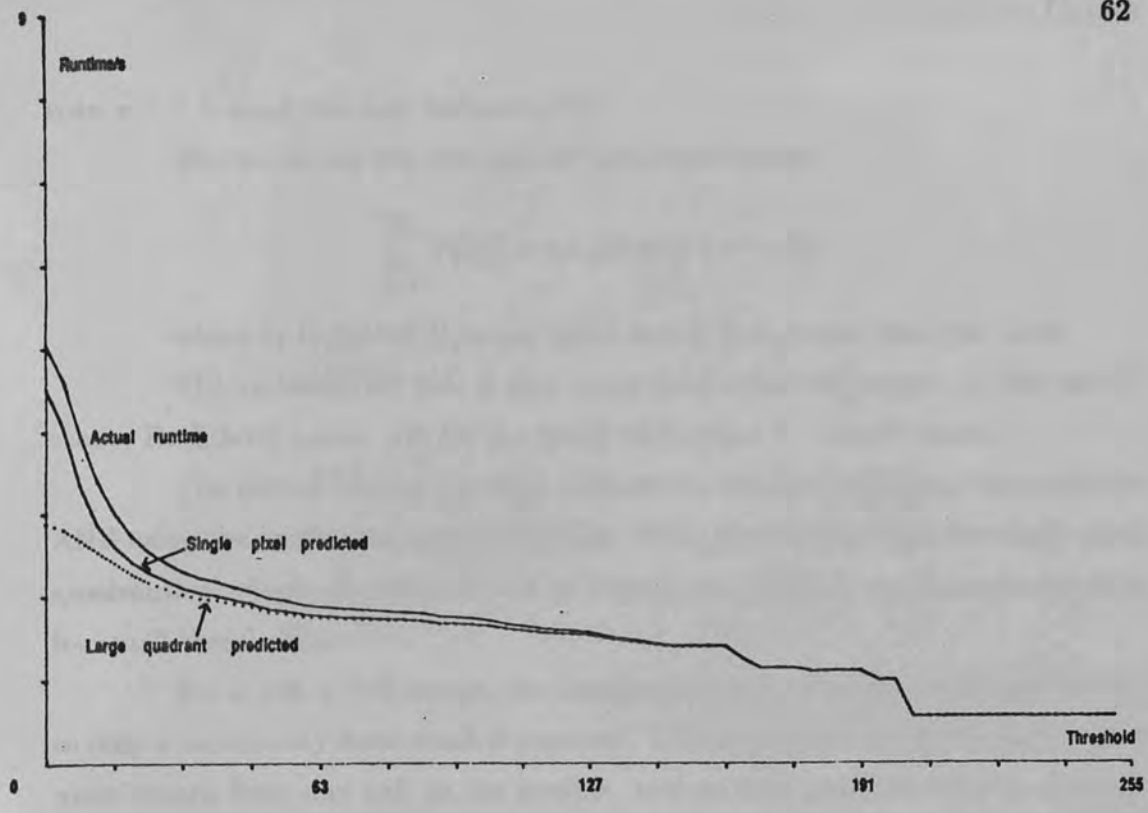


Figure 4.1: Performance of QUAD_TOP_RECURSE

to single pixels) max will be set to the same value as min, thus guaranteeing the creation of a leaf. This removes the need for a special condition to 'bottom out' the recursion.

4.3.2 Performance

The procedure is invoked once for each node in the tree. At each node the quadrant is scanned to check for max and min (lines 7–15). If the quadrant is a leaf then the quadrant is passed over again, and the colour and depth values filled in (lines 20–28). Figure 4.1 shows a graph of execution time for a spread of thresholds using ANB as the source data. Run time is dominated by the time taken to scan and fill quadrants. For very small thresholds, most of the image is tiled with small quadrants, necessitating deep recursion and multiple scans of the image. At the high end, the algorithm executes a single scan and fill across the whole image.

On this system (a PDP 11/23 with the IMP/V1 framestore) a frame store read takes 5.5μs, and a write 3.9μs. The procedure call overhead is 41.3μs and the overhead of the `REPEAT--UNTIL x=xfinish` requires around 12μs.

Large quadrants are scanned at the rate of 20.3μs per pixel and filled at the rate of 13.8μs per pixel. Overheads such as REPEAT count initialisation and fill colour calculation become significant for small quadrants. It takes 36.8μs to

scan a 1×1 quadrant and $45.2\mu s$ to fill.

We can model the run time of the algorithm as:

$$\sum_{i=0}^{\log s} 2^i(l_i t_f + n_i t_s((\log s) + 1 - i))$$

where t_f is the fill time per pixel and t_s is the scan time per pixel.

The rationale for this is that every level i leaf will require 2^i fills and 2^i scans. Each level i scan will be the result of $(logs) + 1 - i$ node scans.

The dotted line on Figure 4.1 shows the result of applying this model to ANB using the large quadrant pixel times. Using the correct times for single pixel quadrants produces the dashed line of Figure 4.1, which is much more accurate for small thresholds.

For a 128×128 image, the maximum depth of recursion is eight levels, so only a moderately sized stack is required. The parameters are not in fact reused upon return from any call to the routine, and so they could be held in external globals, reducing the maximum required stack space to eight procedure activation frames. This will also reduce the procedure overhead call.

4.3.3 Discussion

This algorithm is clearly inefficient for large quadtrees because it scans pixels repeatedly until a leaf is found. In the worst case (a maximum dynamic range checkerboard at the spatial resolution of the frame store, such as Figure 4.2) each pixel will be read eight times. The next algorithm attempts to reduce this overhead by constructing the quadtree bottom up. For small quadtrees the above algorithm is very efficient, and for the limiting case of $N = L = 1$ (*e.g.* threshold=256) should be optimal since all that is required is a single scan and a single fill of the entire image. Figure 4.1 shows that this minimal case requires some 0.5s to run, and this should be taken as the upper limit on the performance of a Pascal based algorithm on this processor.

4.4 Algorithm 2 — bottom up leaf merging

The run time for algorithm QUAD_TOP_RECURS is dominated by (a) the need for one scan per node, and (b) the need to scan a 2^{2i} sized region at each level i node. Algorithm QUAD_MERGE_RECURS, below, eliminates (b) by starting with the smallest possible leaves, and examining groups of four to see if they may be merged into a higher order leaf. The status values of the nodes and

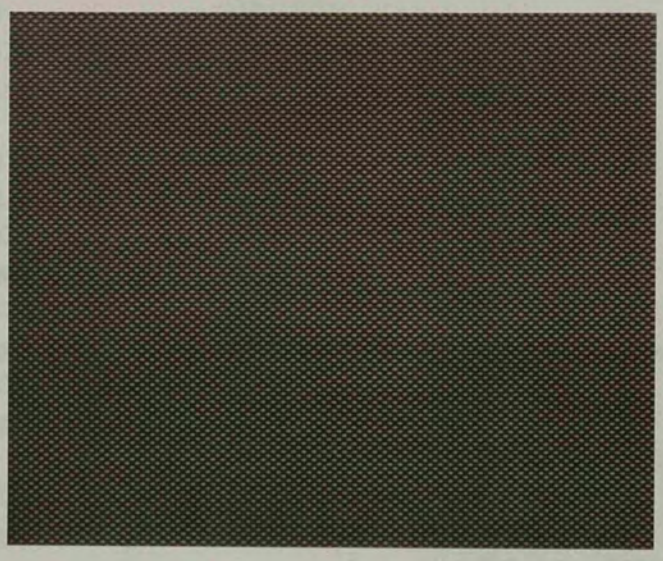


Figure 4.2: Q-image worst case input picture

their colours are propagated upwards until the complete tree is formed. When a true leaf is created, the quadrant in q-space is filled with the mean of the brightest and darkest points, and r-space is filled with the size of the leaf.

```
 1 VAR
 2 thresh,max,min: integer;
 3
 4 PROCEDURE fill(xstart,ystart,size,colour: integer);
 5 VAR
 6  xfinish,yfinish: integer;
 7 BEGIN
 8  xfinish:=(xstart+size); yfinish:=(ystart+size);
 9  x:=xstart;
10  REPEAT
11   y:=ystart;
12   REPEAT
13    q0:=colour; r0:=size;
14    y:=y+1
15   UNTIL y=yfinish;
16   x:=x+1;
17  UNTIL x=xfinish
18 END;
19
20 FUNCTION quad_merge_recurs
21  (xstart,ystart,size: integer; VAR max,min: integer): boolean;
22 VAR
23  newsize,temp: integer;
24  maxs, mins: ARRAY[0..3] OF integer;
25  filled,filled0,filled1,filled2,filled3: boolean;
26
27 BEGIN
28  IF size=1
```

```
29  THEN
30   BEGIN
31    x:=xstart; y:=ystart; max:=p0; min:=p0;
                    quad_merge_recurs:=false;
32   END
33  ELSE
34   BEGIN
35    newsize:=size DIV 2;
36    filled0:=
37     quad_merge_recurs(xstart,ystart,newsize,maxs[0],mins[0]);
38    filled1:=
39     quad_merge_recurs(xstart+newsize,ystart,
                          newsize,maxs[1],mins[1]);
40    filled2:=
41     quad_merge_recurs(xstart,ystart+newsize,
                          newsize,maxs[2],mins[2]);
42    filled3:=
43     quad_merge_recurs
44      (xstart+newsize,ystart+newsize,newsize,maxs[3],mins[3]);
45    filled:=filled0 OR filled1 OR filled2 OR filled3;
46    IF filled
47    THEN
48     BEGIN
49      IF NOT filled0
50      THEN
51       fill(xstart,ystart,newsize,(maxs[0]+mins[0]) DIV 2);
52      IF NOT filled1
53      THEN
54       fill(xstart+newsize,ystart,newsize,
              (maxs[1]+mins[1]) DIV 2);
55      IF NOT filled2
56      THEN
57       fill(xstart,ystart+newsize,newsize,
              (maxs[2]+mins[2]) DIV 2);
58      IF NOT filled3
59      THEN
60       fill(xstart+newsize,ystart+newsize,newsize,
61           (maxs[3]+mins[3]) DIV 2);
62     END
63    ELSE
64     BEGIN
65      max:=0; min:=maxint;
66      FOR temp:=0 TO 3 DO
67       BEGIN
68        IF maxs[temp]>max THEN max:=maxs[temp];
69        IF mins[temp]<min THEN min:=mins[temp];
70       END;
71      IF (max-min>=thresh)
72      THEN
73       BEGIN
```

```
74          fill(xstart,ystart,newsize,(maxs[0]+mins[0]) DIV 2);
75          fill(xstart+newsize,ystart,newsize,
                (maxs[1]+mins[1]) DIV 2);
76          fill(xstart,ystart+newsize,newsize,
                (maxs[2]+mins[2]) DIV 2);
77          fill(xstart+newsize,ystart+newsize,newsize,
78               (maxs[3]+mins[3]) DIV 2);
79          filled:=true;
80         END
81       ELSE filled:=false;
82      END;
83     quad_merge_recurs:=filled
84    END;
85 END;
86
87 BEGIN
88  thresh:=20;
89  IF NOT quad_merge_recurs(0,0,128,max,min)
90  THEN fill(0,0,128,(max-min) DIV 2);
91 END.
```

4.4.1 Commentary

In this case, an entire program has been presented rather than just the main procedure, because the mechanics of the implementation are a little more difficult to follow.

The algorithm is formulated as a recursive function. Line 88 sets the threshold to an arbitrary value. Line 89 calls the main function, which will recursively call itself until it reaches the lowest level of the tree (*i.e.* level 0). As the routines return, the brightness spreads across the quadrants propagate back, and if level 7 is reached without any of the q-image being filled in, then line 90 fills the entire image. Procedure `FILL` (lines 4–18) is called to paint in q and r spaces when a non-mergeable leaf is found.

Line 20: Function QUAD_MERGE_RECURS takes as parameters the coordinates of the top left hand corner, and the size of the quadrant to be examined. It returns *via* `max` and `min` the brightest and darkest points in the image. If the quadrant has already been painted in, (that is the spread of brightnesses in the quadrant is out of threshold) `QUAD_MERGE_RECURS` returns `TRUE` else `FALSE`.

Lines 22–25: `maxs` and `mins` are used to store the brightness values returned from the four leaves below the current level. Likewise `filled0` to `filled3` store the booleans returned from the function calls in lines 36–44. `filled` is the inclusive-OR of `filled0` to `filled3`.

Lines 28–32: at the lowest level of the tree where the leaf size is 1, the

function immediately returns FALSE with max and min set to the brightness of the current pixel. This initiates the merging process as the pixel values propagate back up the call tree.

Lines 35–44: at any other level in the tree, the function recursively calls itself for each of the four subquadrants. On return from these calls maxs, mins and filled0 to filled3 will hold the brightness spreads and leaf statuses for each of the four subquadrants.

Line 45: filled is set to the inclusive-OR of filled0 to filled3. filled is thus TRUE if any of the four subquadrants have already been filled in. The information returned from the lower leaves will cause one of four things to happen. (1) If all four subquadrants have already been filled in then the function simply returns TRUE. (2) If only some of the subquadrants have already been filled in, then the rest of the subquadrants must also be filled (lines 48–62). If none of the subquadrants have been filled, then the smoothness values across the quadrant must be calculated (lines 65–70). The whole algorithm depends upon the use of a smoothness measure that may be simply combined from the four subquadrants to provide a value for the whole. (3) If the brightness spread is out of threshold, then the subquadrants must be filled in (lines 74–78), otherwise (4) the function returns max, min and FALSE to continue the propagation of pixel values up the tree.

Line 46: filled is tested to see if any of the subquadrants have already been filled in.

Lines 48–62: the individual filled flags are tested to find any subquadrants that have not yet been filled in, and they are then painted over.

Lines 64–70: if none of the subquadrants have already been filled, then the individual max and min values are tested to find the spread of brightnesses across the quadrant. Note that this process is difficult to generalise to other smoothness measures.

Lines 71–82: If the brightness spread is out of threshold, then the individual subquadrants are filled in (lines 74–78) and filled is set TRUE (line 79). Otherwise the function will exit TRUE with max and min showing the brightness spread across the quadrant.

4.4.2 Performance

QUAD_MERGE_RECURS does perform far fewer pixel accesses than QUAD_TOP_RECURS. Rather than scanning the entire quadrant at each node

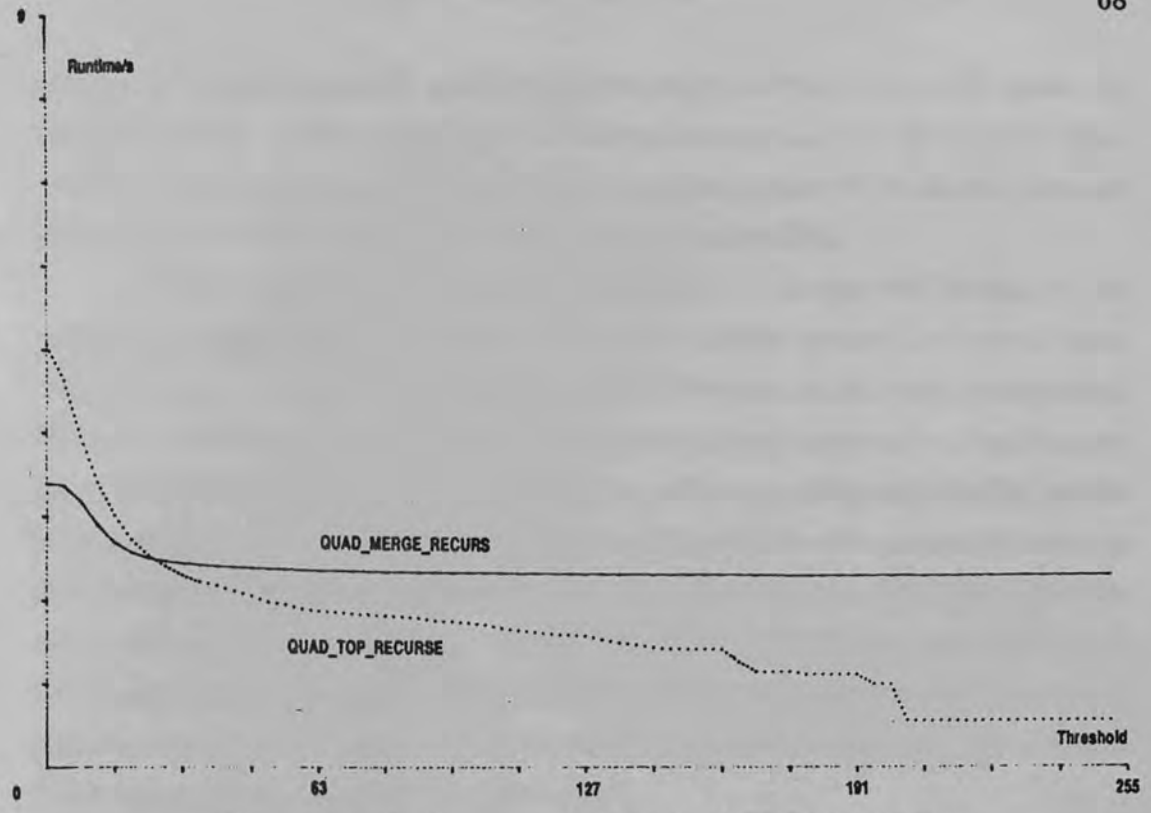


Figure 4.3: Performance of QUAD_MERGE_RECURS

in the tree, it looks at the entire picture once (at level 0 nodes) and then passes the values back up through the call tree. At each level four values are examined to decide whether a node is a leaf or a parent. This clear advantage is significantly offset by the need to examine every potential node of the tree, rather than every actual node of the tree as in QUAD_TOP_RECURS. For large trees, where QUAD_TOP_RECURS has to recurse deeply and therefore scan many pixels, QUAD_MERGE_RECURS should execute more rapidly. However for small trees where QUAD_TOP_RECURS will only call itself a few times QUAD_MERGE_RECURS will be at a severe disadvantage because it is always called N times. The actual crossover point will depend on the execution time of each call to function QUAD_MERGE_RECURS, and is likely to be heavily dependant on the Pascal function call overhead because of the large number of parameters required. Figure 4.3 shows that for ANB, crossover occurs at a threshold of around 30. However it also shows that QUAD_MERGE_RECURS is less 'pathological' than QUAD_TOP_RECURS, that is its performance is more uniform across the entire range of trees, whereas QUAD_TOP_RECURS varies by an order of magnitude in execution speed, showing a very steep increase in run time for large trees.

The run time of QUAD_MERGE_RECURSE is dominated by the procedure call overhead. This will be an advantage on fast computers with slow frame

stores. Framestores such as the CRS 4000 connect to PDP-11 or VAX hosts *via* DMA interfaces. Single pixel access to this frame store is very slow, so an algorithm that trades off procedure calls against pixel accesses will be faster than one such as QUAD_TOP_RECURS that repeatedly scans pixels.

The maximum depth of recursion for an $s \times s$ image will be $\log s + 1$ as before, *i.e.* eight levels for a 128 × 128 image. In fact rather more stack space will be required than for QUAD_TOP_RECURS because of the extra parameters. With the Pascal compiler in use in this laboratory, stack space is required to save internal registers which are in use at the time of the procedure call (including the return address of the routine), plus a one word pointer to the parameter area on the stack, plus the space required for the actual parameters, which includes the return parameter for functions. For QUAD_TOP_RECURS at least five words are required per call, and for QUAD_MERGE_RECURS eight words. Temporary register storage may require up to six further words in each case. In addition stack space will be required for call to FILL.

4.4.3 Discussion

The performance of recursive algorithms is heavily dependant on the procedure call overhead for the implementation language. Unfortunately, the present Pascal implementation is poor in this respect. When lines 29–32, 35 and 45–82 are removed (*i.e.* the algorithm is reduced to call tree generation and return), QUAD_MERGE_RECURS executes in 1.9s. If this overhead could be reduced or even removed, the run time of the algorithm would be substantially improved.

It is a basic property of the algorithm that the main function is called once for every potential node in the tree, *i.e.* 21845 times for a 128 × 128 image. Programming the algorithm in machine code could substantially reduce the time required to call the function because this compiler makes poor assumptions about which registers to save at entry to the routine.

As noted in Section 4.4.1, each call results in one of four basic actions. Type (1) calls are clearly redundant and serve simply to maintain the call tree as the recursion unwinds. The next two algorithms attempt to reduce the overall redundancy of algorithm 2 by removing the need for a call tree. Both algorithms scan the image in the same order as algorithm 2, but they make immediate decisions concerning leaf position without passing data back up to higher level processes. In principle, these algorithms should have run times linked to the number

of leaves in the tree rather than the number of nodes (actual or potential).

4.5 Algorithm 3 — bottom up backtracking

Simple image processing operators are usually applied in a raster scan across the image. Since they are emulating the operation of a parallel processor (where the operator is applied simultaneously to all points in the image) the scan order is in fact irrelevant. However, serial algorithms are often highly sensitive to scan order. Typically sequential algorithms maintain running results which differ if the operation is applied from different directions.

A sequential quadtree generation algorithm must track round the leaf structure within the image. The raster scan does not correspond to a 'natural' ordering of leaves. Close examination of algorithm 2 shows that the tree is visited in preorder and that this is equivalent to visiting the pixels in *z-order*:

```
 0   1   4   5   16  17
 2   3   6   7   18...
 8   9  12  13
10  11  14  15
```

The next algorithm implements the leaf merging idea of algorithm 2 by scanning the image in z-order, and backtracking when an out of threshold point is found. The max-min measure of brightness is maintained in a sequential fashion as the scan proceeds to larger and larger leaf sizes. When max-min goes out of threshold, the largest block so far completed is calculated, and it and its three siblings are re-examined by a recursive call.

```
 1 PROCEDURE quad_back_recurs(first,last,size: integer);
 2 VAR
 3  max,min,point,tempend,tempsize: integer;
 4  exit: boolean;
 5
 6 BEGIN
 7  max:=0; min:=255; point:=first; exit:=false;
 8  REPEAT           {scan}
 9   x:=scan_x[point]; y:=scan_y[point];
10   IF p0>max THEN max:=p0; IF p0<min THEN min:=p0;
11   IF (max-min)>=thresh
12    THEN
13    BEGIN
14     tempend:=last-first; point:=point-first; exit:=true;
15     tempsize:=size;
```

```
16      REPEAT
17       tempend:=tempend DIV 4;
18       tempsize:=tempsize DIV 2;
19       quad_back_recurs(first+1+tempend,first+1+2*tempend,
                          tempsize);
20       quad_back_recurs(first+2+2*tempend,first+2+3*tempend,
                          tempsize);
21       quad_back_recurs(first+3+3*tempend,first+3+4*tempend,
                          tempsize);
22      UNTIL tempend<point;
23      quad_back_recurs(first,tempend+first,tempsize);
24     END
25    ELSE
26     point:=point+1;
27   UNTIL exit OR (point>last);        {scan}
28   IF point>last
29   THEN
30    BEGIN
31     point:=first;
32     REPEAT                          {fill}
33      x:=scan_x[point]; y:=scan_y[point];
34      q0:=(max+min) DIV 2;
35      r0:=size;
36      point:=point+1;
37     UNTIL point>last;               {fill}
38    END
39 END;
```

4.5.1 Commentary

Procedure QUAD_BACK_RECURS takes as parameters the first and last points of the block to be scanned, and the size of the block sidelength. The size is only required when filling in the contour map of the quadtree, and could be omitted if only the q-image is required. The smoothness threshold is supplied *via* global integer `thresh` as above.

The scan order is defined by lookup tables `scan_x` and `scan_y` which are both of type `ARRAY[0..16383] OF 0..255`. They contain the x and y coordinates for every point in the scan. Scan coordinate generation will be discussed further in Section 4.7.1.

Line 3: `point` is a running variable pointing to the current pixel in the scan. It is used as an index into the lookup tables.

Line 7: `max` and `min` are initialised. `point` is initialised to the first point in the quadrant.

Line 9: the current pixel is addressed by loading the frame store coordi-

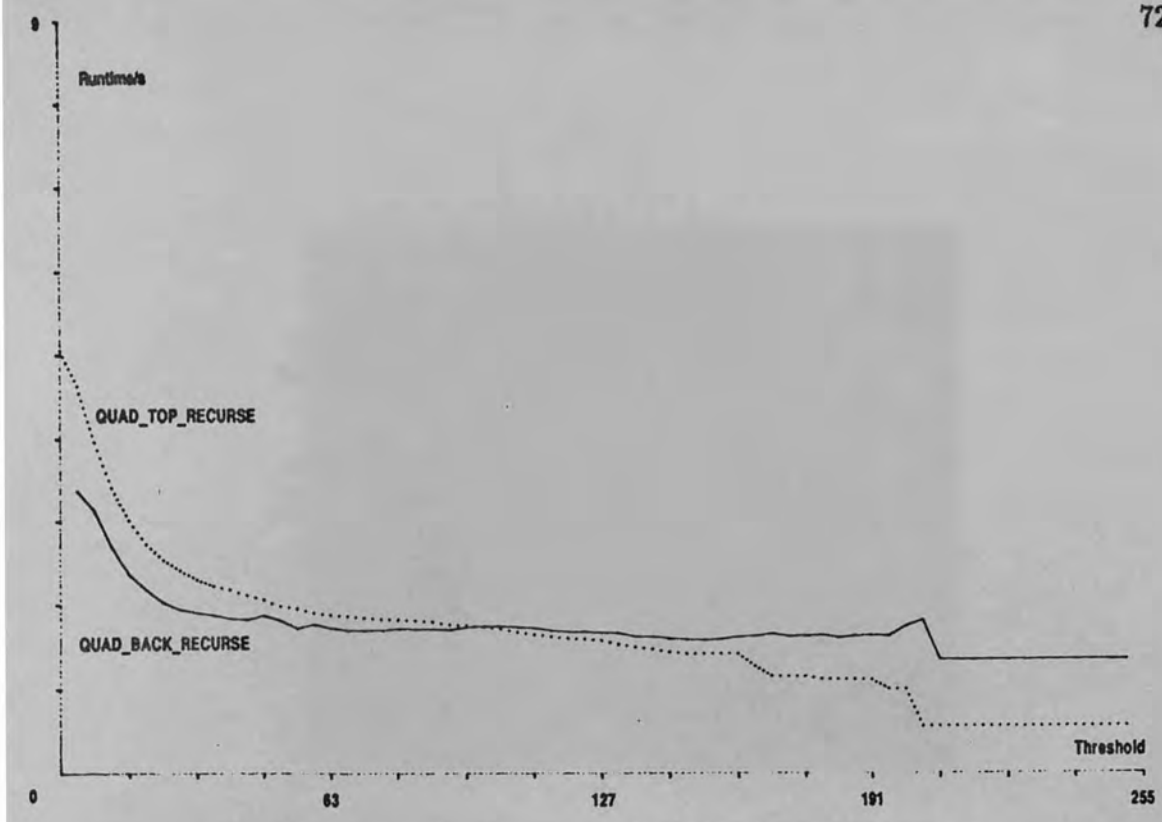


Figure 4.4: Performance of QUAD_BACK_RECURS

nate registers from the lookup table,

Line 10: max and min are updated with the new pixel.

Line 11: the brightness spread is tested against thresh.

Lines 13–24: if the spread is out of threshold, then block subdivision starts.

Lines 16–22: each pass of the loop recursively calls the main function for subquadrants 1, 2 and 3, and tests the size of quadrant 0. If quadrant 0 lies entirely within the region scanned then the loop exits, else the process repeats with subquadrant 0 as the new quadrant.

Line 26: if max-min is within threshold, point is simply incremented.

Line 27: when point reaches last, a block has been successfully scanned and may be filled in (lines 32–37).

4.5.2 Performance

Figure 4.4 shows the performance of algorithm 3 at various thresholds for ANB. The dotted line is the equivalent performance curve for algorithm 1.

As can be seen, algorithm 1 only becomes more efficient for thresholds greater than about 100, which results in quadtrees that hold too little information to be useful for image processing or representation, as shown in Figures 4.5 and 4.6.

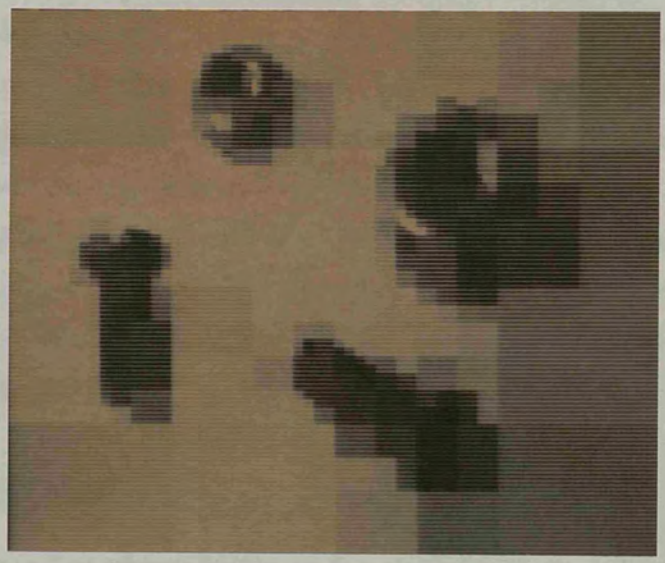


Figure 4.5: Q-image of ANB, threshold=100

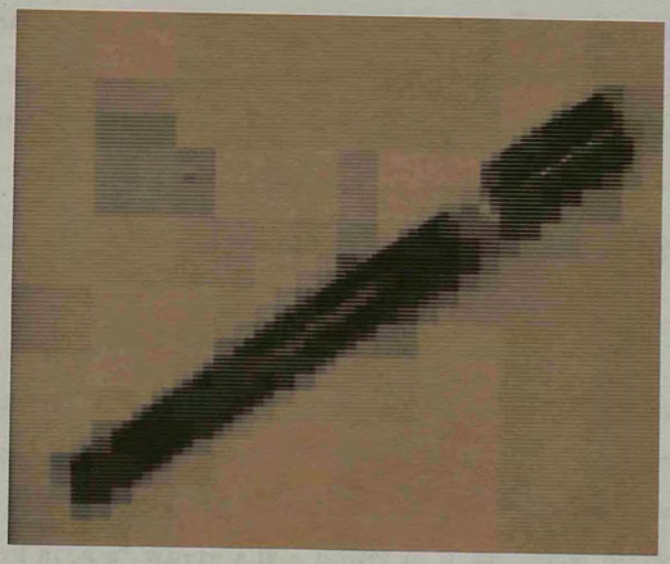


Figure 4.6: Q-image of PEN, threshold=100

For all useful cases, QUAD_BACK_RECURS is faster. The storage requirement depends, as ever, on the level of recursion. The worst case storage occurs when the second point in an image is out of threshold. In this case the loop in lines 16–22 descends through 6 leaf sizes, generating three recursive calls at each level until it reaches level 0, at which point line 23 generates a further call. With the initiating call from the main routine this gives a total of 20 stacked calls, which is considerably greater than algorithms 1 and 2.

The run time depends critically on the amount of backtracking performed. The worst case occurs for a smooth image with a single out-of-threshold point at the end of the z-scan. Maximal backtracking also generates the maximal number of pixel scans.

4.5.3 Discussion

Algorithm 3 produces better performance in all cases than algorithm 2 by dispensing with the large call tree and the associated movement of large amounts of data. If the amount of backtracking could be reduced the performance might be expected to improve. Algorithm 4 is able to reduce backtracking to zero, but incurs a heavy penalty in calculation overhead.

One significant inefficiency in this algorithm lies in the z-scan generation. Most computers have increment instructions which allow one to be added to a variable quickly. Algorithms 1 and 2 exploit this through the use of a raster scan which only requires simple increment and test instructions. The z-scan requires a shift and merge operation which is not easily implemented on a PDP-11, and so a lookup table is used instead. This will be examined in more detail in Section 4.7.1.

4.6 Algorithm 4 — an optimal quadtree generator

Algorithm 3 does not fully exploit the available information when it encounters an out-of-threshold point. There is a unique pattern of leaves joining any two points A and B in the z-scan. This pattern of leaves will consist of maximal blocks of area s^2 where s is a power of 2, and each block will be aligned so that the coordinates of its top left corner modulo s is 0. The area of the image traversed between points A and B must be completely tiled by blocks. The proof that there is a unique pattern lies in the requirement that the area be tiled with maximal blocks. Clearly, the region could otherwise be tiled with single pixel

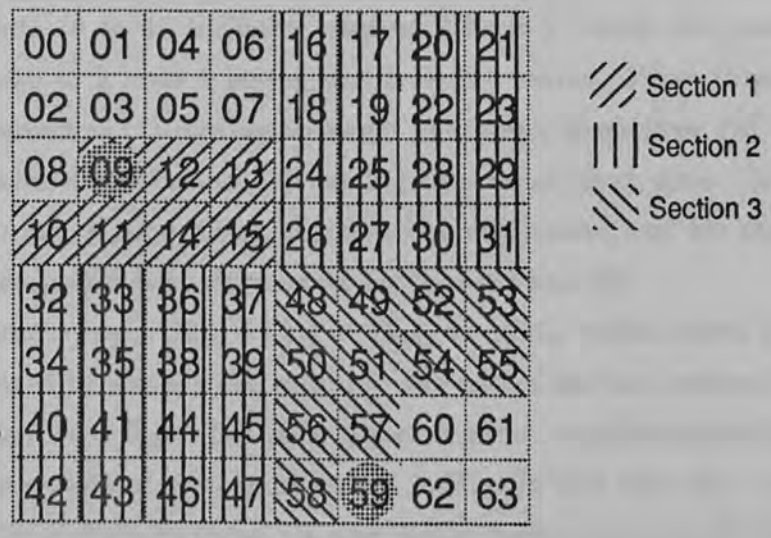


Figure 4.7: Calculation of leaves from discontinuities

sized blocks, but because of the maximal condition groups of four, it must merge and remerge until the largest possible block appears.

Given that the unique pattern exists, the values of A and B must contain all the necessary information to deduce the leaf structure of the quadtree between A and B. As noted in Chapter 2, in any data structure some information is explicit and cheap to retrieve, and some is implicit, either in the topology of the structure or in the coding of the data in the nodes of the structure. Clearly an extremely compact but impractical representation of the quadtree could be constructed merely by enumerating the discontinuities in the z-scan. Algorithm 4 effectively converts this representation to the explicit q-image/contour map representation used elsewhere in this chapter.

Consider the case where point A is the origin. Figure 4.7 shows that if a discontinuity occurs at the point $B = 59_{10}$ $(= 111011_2)$, then 3 level 2 leaves, 2 level 1 leaves and 3 level 0 leaves must be created. As will be seen this information may be extracted directly from the binary value of the discontinuity point number. By selecting ascending pairs of bits we have 11_2, 10_2 and 11_2 or (decimal 3, 2 and 3).

When point A is not the origin, the situation is more complex. In algorithm 3, the maximal block detection algorithm required the origin to be moved to the first point of the scan region (line 14). A similar procedure is used here, except that the process of normalisation is a little more subtle.

Figure 4.7 shows points $A = 9$ and $B = 59$. Here the leaf pattern falls into three sections. Points 9 to 15 inclusive map to 3 level 0 leaves and 1 level

1 leaf; points 16 to 47 inclusive map to 2 level 3 nodes; and points 48 to 58 inclusive map to 2 level 1 leaves and 3 level 0 leaves. These three regions can be characterised as (1) moving to larger and larger block sizes, (2) traversing up to four maximal blocks and (3) moving to smaller block sizes. In general it is possible for any of these three regions to be degenerate, and the case where A is the origin corresponds to degenerate sections (1) and (2).

Interestingly, $59_{10} - 48_{10} = 11_{10}$, or 1011_2, which shows that section 3 can be derived by moving the origin to the end of the last maximal block in the region. Also, $16_{10} - 9_{10} = 7_{10}$, or 111_2 showing that a similar normalisation can be used to derive the leaf pattern in section 1. Finally note that $59_{10} - 9_{10} = 50_{10}$, or 110010_2. The highest occupied bit pair here corresponds to level 3 leaves, and this is the size of the maximal block. The length of the section constitutes a *signature* describing the leaf pattern within that section.

These observations can be used to derive the entire leaf pattern between points A and B by direct calculation, reducing the scanning requirement to a single pass over the data, pausing at every discontinuity to calculate the next section of tree. This implementation colours leaves slightly differently to the other three algorithms except for the case of a binary image. However, the shape of the tree is always correct.

```
 1 PROCEDURE quad_opt;
 2 VAR
 3 leaves: ARRAY[0..7] OF integer;
 4  no_max_blocks,max_size,up_remainder,down_remainder,
 5  max_length,first,size,max,min,point,temp1,temp2: 0..65535;
 6
 7 BEGIN
 8  point:=1; first:=0;
 9  REPEAT                          {scan until out of threshold}
10   max:=0; min:=255; point:=point-1;
11   REPEAT
12    point:=point+1; x:=scan_x[point]; y:=scan_y[point];
13    IF p0>max THEN max:=p0; IF p0<min THEN min:=p0;
14   UNTIL (max-min>=thresh) OR (point=16383);
15   IF point=16383 THEN point:=16384;
16   max_size:=16384; temp1:=point-first; max_length:=128;
17   WHILE temp1 DIV max_size=0 DO  {find size of maximal block}
18    BEGIN
19     max_length:=max_length DIV 2;
20     max_size:=max_size DIV 4;
21    END;
22   up_remainder:=(NOT(first MOD max_size)+1) MOD max_size;
23   down_remainder:=point MOD max_size;
24   no_max_blocks:=(temp1-up_remainder-down_remainder)
```

```
                    DIV max_size;
25   size:=1;                        {fill section1}
26   FOR temp1:=0 TO 7 DO
27    BEGIN
28     temp2:=size*size*(up_remainder AND 3);
29     up_remainder:=up_remainder DIV 4;
30     WHILE temp2>0 DO
31      BEGIN
32       x:=scan_x[first]; y:=scan_y[first]; r0:=size;
         first:=first+1;
33       temp2:=temp2-1;
34      END;
35     size:=size*2;
36    END;
37   temp2:=max_size*no_max_blocks;      {fill section 2}
38   WHILE temp2>0 DO
39    BEGIN
40     x:=scan_x[first]; y:=scan_y[first]; r0:=max_length;
41     temp2:=temp2-1; first:=first+1;
42    END;
43   temp2:=down_remainder;                {fill section 3}
44   FOR temp1:=0 TO 7 DO
45    BEGIN leaves[temp1]:=temp2 AND 3; temp2:=temp2 DIV 4 END;
46   size:=128;
47   FOR temp1:=7 DOWNTO 0 DO
48    BEGIN
49     temp2:=size*size*leaves[temp1];
50     WHILE temp2>0 DO
51      BEGIN
52       x:=scan_x[first]; y:=scan_y[first]; r0:=size;
         first:=first+1;
53       temp2:=temp2-1;
54      END;
55     size:=size DIV 2;
56    END;
57  UNTIL point=16384
58 END;
```

4.6.1 Commentary

Procedure QUAD_OPT assumes the use of a 128×128 image, and therefore takes no parameters. The smoothness threshold is passed as usual *via* global integer `thresh`. QUAD_OPT uses the same mechanism as QUAD_BACK_RECURS to generate the z-scan, *i.e.* two globally declared lookup tables `scan_x` and `scan_y`.

The program consists of one large loop containing a smaller loop to actually scan the pixels (lines 11–14) and a series of loops to fill the three possible sections of the tree between two breakpoints.

Lines 2–5: the array `leaves[0..7]` is used to hold the bit pairs derived during decomposition of the section 1 and 3 signatures. `no_max_blocks`, `max_size` and `max_length` hold the number, area and sidelength respectively of the maximal blocks found in section 2. `up_remainder` and `down_remainder` hold the signatures for sections 1 and 3 respectively. `first`, `max`, `min` and `point` have the same meanings as in algorithm 3. Note that all simple variables are declared as subrange type `0..65535` rather than as type `integer`. This forces the present compiler to generate code for 16-bit unsigned arithmetic as opposed to the 15-bit signed arithmetic implied by type integer on a 16-bit computer. This is necessary to prevent sign bit propagation in the shift and merge sequences.

As an aside, note that this use of subranges to force 16-bit arithmetic is semantically unsound, since type `0..65535` is a subrange type of underlying type `integer` [Coo83b], yet type integer on a 16-bit machine ranges from -32768 to 32767. Clearly `0..65535` can hardly be called a subrange of `-32768..32767`!

Line 8: `point` and `first` are initialised.

Line 10: `max`, `min` and `point` are re-initialised after every fill operation.

Lines 11–14: as in algorithm 3, the image is scanned in z-order until a discontinuity (out-of-threshold point) is found, or the end of the image is reached.

Line 15: normally the scan loop (lines 11–14) exits with `point` at the first point of the next scan (*i.e.* the out of threshold point). A special case occurs when the end of the image is reached in that point is at the last point of the area to be filled. Line 15 corrects `point` under these circumstances so as to be compatible with normal processing.

Line 16: `temp1` is set to `point-first`, which corresponds to the distance between points A and B in the terminology of Section 4.6. This value will be searched by bit pairs until a maximal block is detected. `max_size` and `max_length` are initialised to their largest possible value before initiating the search for a maximal block.

Lines 17–21: `max_size` is stepped down through the possible sizes of a maximal block until `temp1 DIV max_size` returns non-zero. At this point the largest possible block lying between points A and B has been detected.

Lines 22–23: the signatures for sections 1 and 3 are derived.

Line 24: the number of maximal blocks (between 0 and 4) is derived by subtracting the length of the sections 1 and 3 (*i.e.* their signatures) from the distance between points A and B, and dividing the result by `max_size`.

Lines 25–36: section 1 is filled. The outer loop (lines 26–29, 35–36) selects a pair of bits from the signature, and the inner loop (lines 30–34) fills in

the corresponding blocks.

Lines 37–42: the maximal blocks (section 2) are filled in.

Lines 43–56: section 3 is filled using the same strategy as for section 1 (lines 25–36) except that the bit pairs must first be unpacked into the `leaves` array because the unpacking order is the reverse of the filling order.

4.6.2 Performance

Although the order relation governing algorithm 4 is near optimal, the constants of proportionality on the PDP-11 implementation yield actual runtimes that are longer than those of algorithm 1 over much of the range of possible thresholds. Figure 4.8 shows the usual run time curve against algorithm 1. Clearly, the scan time will be proportional to the number of pixels in the image since they are only scanned once. However, each discontinuity in the image will generate a sequence of complex (and slow) shift and select operations, and a series of leaf fill operations. The run time will therefore be roughly $s^2 t_s + d t_d + s^2 t_f$ where s^2 is the number of pixels in the image, t_s is the scan time for one pixel, d is the number of discontinuities in the image, t_d is the time taken to extract the leaf pattern using the shift and select loops and t_f is the fill time. The number of discontinuities in the image will be approximately the number of leaves in the quadtree, and since the calculations are so computationally intensive, the run time will be roughly proportional to the number of leaves. Figure 4.8 also shows the number of leaves in the tree for each threshold. Even on a PDP-11 without 'bit twiddling' instructions the algorithm is fastest over a useful range of values (72–164). In VAX machine code, the algorithm would speed up significantly. (See discussion below).

4.6.3 Accuracy

A grey scale quadtree is defined by the distribution of nodes and the colour of the leaves. QUAD_OPT correctly calculates the distribution of nodes (*i.e.* the shape of the tree) but cannot in general colour the leaves correctly, because the leaf colours cannot in general be calculated on the fly without back-tracking.

A `max-min` smoothness measure has been selected because it may be calculated and tested in a serial fashion, whereas more desirable smoothness measures such as the true standard deviation of the intensity histogram require a complete quadrant to be scanned before a test can be made. The first three algo-

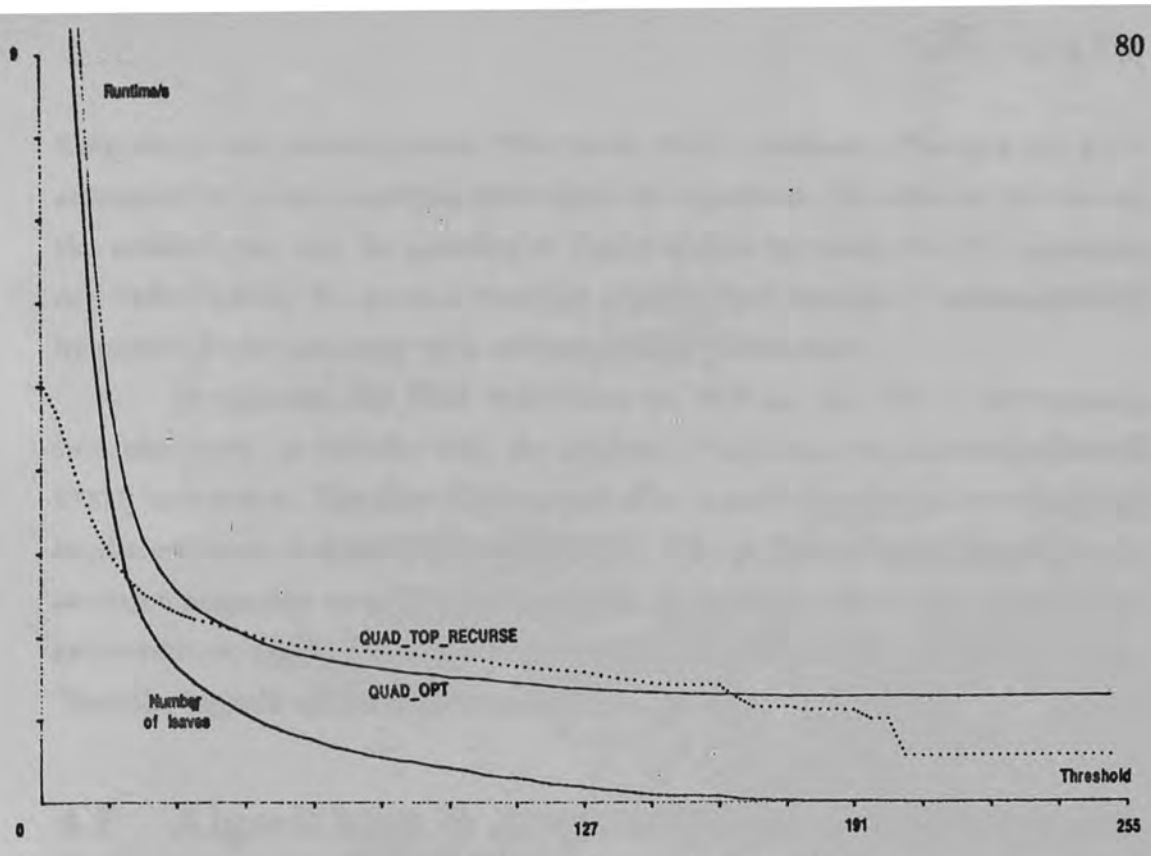


Figure 4.8: Performance of QUAD_OPT

rithms have defined the leaf colour as the mean of max and min. In QUAD_OPT, values of max and min may be accumulated across several leaves before a discontinuity is found so the individual max and min for each leaf are not available.

The max-min measure is itself approximate, and is used for speed reasons and because it is 'natural' for serial algorithms. By the same argument, colouring leaves according to the (max-min)/DIV 2 in QUAD_OPT is fast and natural. The maximum error with respect to QUAD_TOP_RECURS occurs when a scan section includes a pixel filled with pixel level i and quadrants containing pixels of level $i+t$ where t is the threshold. In such a case the leaves will be coloured with $(2i+t)/2$ which is an error of $t/2$. The Q-images are usually indistinguishable by eye.

4.6.4 Discussion

Algorithm 4 clearly illustrates a mismatch between an algorithms and the PDP-11 architecture. There is in fact scope for improving run time by removing some redundant operations and improving the loop design of QUAD_OPT. However, the fatal flaw of this implementation which leads to the disappointing run times is the lack of bit field manipulation instructions in the PDP-11. Most of the time associated with t_d is spent (a) finding the position of the highest set bit in temp1 (lines 17–21) and (b) extracting the bit pairs from the section 1 and

3 signatures and rotating them down to the bit0/1 positions. This has had to be accomplished using a multiple rotate and test algorithm. To make matters worse, the rotation can only be specified in Pascal syntax by using the DIV operator, and unfortunately the present compiler is not 'smart' enough to replace division by power-of-two constants with arithmetic shift instructions.

In contrast, the VAX instruction set includes the `FIND FIRST` instruction (also known as `PRIORITISE`), the `COMPARE FIELD` instruction and the `EXTRACT FIELD` instruction. Together, these would allow a very compact machine language implementation of algorithm 4 on the VAX. The problem of specifying these instruction sequences to a high level language compiler remains — it would require extraordinary depth of analysis in the compiler to recognise that lines 16–21 mean "find the highest set bit pair in temp1".

4.7 Algorithms + Architectures = Implementations

Order analysis of algorithms gives relations of proportionality between the *size* of a problem (in this case the number of nodes in the tree) and its execution time. The constants of proportionality are dependant on the particular implementation. This thesis is primarily concerned with real implementations and how architectural features perturb the run time behaviour of an algorithm. Given that the abstract algorithm is constant across implementations, computer architecture may be defined as all those properties of a system that can perturb the behaviour of an algorithm. This specifically includes such properties as processor-memory bandwidth and available addressing modes, but excludes engineering details such as transmission protocols and character coding schemes.

The mismatch between algorithm 4 and the PDP-11 is great, whereas with a VAX algorithm 4 would be the most efficient. Algorithm 3 would be more efficient on a system with a low bandwidth channel between the frame store and the processor. On the experimental system, which uses a frame store optimised for image processing to be described in Chapter 5, algorithm 1 performs surprisingly well.

4.7.1 Z-scan by bit twister

The calculations required to generate the z-scan impose a large overhead on algorithm 3 and 4. Look up tables have been used to speed access, but these

N	binary(N)	binary(X)	binary(Y)	X	Y
14	0000 0000 0000 1110	0000 0010	0000 0011	2	3
15	0000 0000 0000 1111	0000 0011	0000 0011	3	3
16	0000 0000 0001 0000	0000 0100	0000 0000	4	0
	yxyx yxyx yxyx yxyx	xxxx xxxx	yyyy yyyy		
	7766 5544 3322 1100	7654 3210	7654 3210		

Figure 4.9: Z-scan and cartesian coordinate relationship

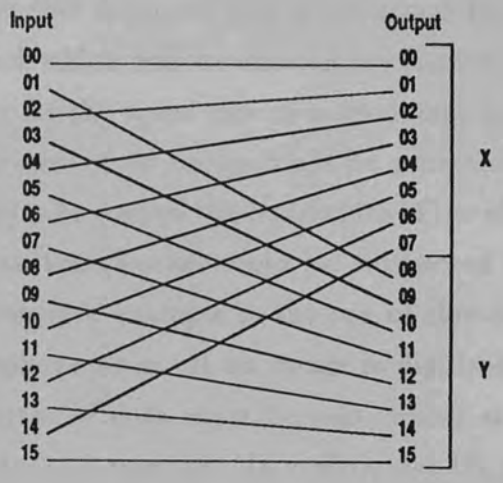


Figure 4.10: Bit twister

require 32K bytes of storage, which is half of the virtual address space of a PDP-11. A little analysis shows that the X,Y coordinates of Z-scan point are embedded in the binary representation of N as shown in Figure 4.9.

Thus by separating out alternate bits from the Z-scan number the X,Y coordinates may be directly obtained. The PDP-11 has to use shift and test operations to extract the coordinates.

If the computer is equipped with a parallel port, a trivial hardware add-on called a bit twister can be used to generate the coordinates in two machine instructions. The sixteen output bits are cross connected to the 16 input bits as shown in Figure 4.10.

The processor moves the z-scan coordinate to the output register and then reads the x,y coordinates back off the input.

4.8 Data compression using quadtrees

Traditional methods of signal transmission use fixed bandwidth sampling. Television signals require a full 5.5MHz bandwidth channel even though uniform areas in an image will not exercise the available bandwidth. The situation here is analogous to Dijkstra's multiplier [Dij76]:

> "...during its lifetime the multiplier will be asked to perform only a negligible fraction of the vast number of all possible multiplications it could do: practically none of them! Funnily enough, we still require that it should do any multiplication correctly when ordered to do so. The reason underlying this fantastic quality requirement is that we do not know in advance, which are the negligibly few multiplications it will be asked to perform."

The television engineer has assumed that he does not know which parts of an image will be busy and which will be smooth, so enough channel capacity has been allocated to allow for the worst case of a maximally busy image. If however a measure of the busy-ness of an image could be generated at the transmission end and used to dynamically control the bandwidth of the channel, say by varying the sampling rate, then the channel could be engineered to cope with average conditions. A more concrete example is the use of slow scan TV transmission down voice grade telephone lines. If an image is digitised to 128 $\times$ 128 eight-bit pixels then 16K bytes of data must be sent, which at a typical 1200 baud will take approximately 2.27 minutes. In reality, five-bit pixels would be quite acceptable, so transmission time reduces to 1.42 minutes (assuming no parity). Using a quadtree representation, each leaf could be sent as a five-bit colour code combined with a three-bit level code. If the leaves are sent in some predefined order (*e.g.* postorder as above) then the tree could be constructed unambiguously without any explicit coordinate information. The Q-image of ANB in Figure 4.11 contains only 1636 leaves, and could be transmitted in 13.6 seconds. The q-image has the normal blocky appearance which may be unacceptable in some images. Low pass filtering may be applied to produce a more aesthetically pleasing image. Figure 4.12 shows the result of applying a median filter to the Q-image, and for comparison, Figure 4.13 shows a median filter of the original image.

Figure 4.11: Q-image of ANB at threshold 36

Figure 4.12: Median filtered Q-image

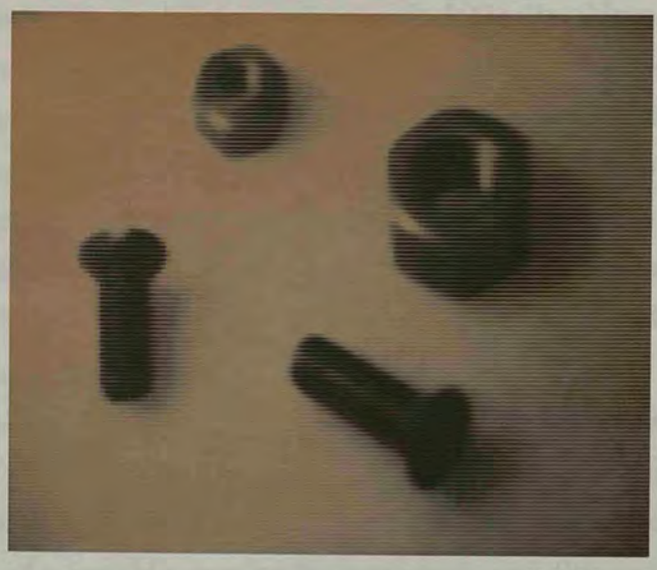


Figure 4.13: Median filtered original

4.9 Quadtree controlled image processing operators

Many image processing operators manipulate the image according to busy-ness. Apart from the simple edge detection and smoothing operators, more subtle segmentation and region merging algorithms may also be edge sensitive. Since quadtrees provide explicit information about the busy-ness of an image at a point they can be used to *skim* over smooth areas in the image, restricting full processing to points of special interest.

4.9.1 Edge detection

The well known Sobel filter [DH73] is probably the most common edge detection operator in use. It is empirically known to be accurate, and a paper by Davies [Dav84] has given a theoretical base on which to design families of Sobel like 'circular' operators. However the Sobel is expensive to implement in full since it requires a square root operation to derive the magnitude of the intensity gradient. Typically this square root is calculated using a lookup table or one of several single iteration approximations to the square root. Cheaper operators such as the Robert's Cross are also in use, but these do not provide accurate estimates of edge direction as opposed to magnitude, and are unsuitable for some algorithms (*e.g.* the Hough Transform based circle detector described in Chapter 9).

One approach to speeding up the Sobel which has been used for indus-

trial vision in this laboratory is to apply simple intensity skimmers before the actual Sobel. Assume the use of a high contrast image (*i.e.* the mean foreground intensity should be at least 30 grey levels away from the mean background intensity) and sharp edges where the transition from mean foreground to mean background intensities occurs over less than 10 pixels. If high and low thresholds *hi* and *lo* are applied to the image then only points with $lo < intensity < hi$ will be passed to the Sobel. For simple images such as BISC (Figure 3.6) this gives a massive reduction in processing.

The application described in Chapter 9 includes a circle detector that relies on the use of a good edge detector. It has been found that of the order of 100 points around the edge of a 30 pixel radius circle are sufficient to reliably locate the centre of a roughly circular food product, and this criterion has been used to provide values for *hi* and *lo* in an online inspection system.

The problems with simple skimming are (a) it only works for high contrast images, (b) it requires two thresholds to be derived using a single setup criterion and (c) intensity thresholding is vulnerable to changes in lighting level or appearance of the product. All of these factors combine to reduce the robustness of the technique, and can contribute to the well known problems of transferring laboratory techniques to the real world. These problems stem from the mismatch between the skimming process and the edge detector. Assumptions are being made about the relationship between absolute intensities and the busy-ness of the image. If a part of the image is darker then the *lo* threshold then it is assumed smooth, and likewise for bright points.

It would be much better to apply a busy-ness related skimmer to the image, such as preprocessing by a Robert's Cross to find the points of high gradient magnitude, followed by application of a Sobel to find the gradient directions. This involves the application of a single threshold (of gradient magnitude) that is closely related under all conditions with the quantity under inspection.

The lowest leaves of a quadtree correspond to busy parts of the image, ie the edges. Figure 4.14 shows the positions of all level 7 leaves in the Q-image of ANB for a threshold of 25. The results of applying a Sobel to all corresponding points in ANB is shown in Figure 4.15 , and for comparison a full Sobel of ANB is shown in Figure 4.16. The Sobel applied to the entire image executed in 1.19s, whilst with the help of the quadtree this decreased to 0.29s.

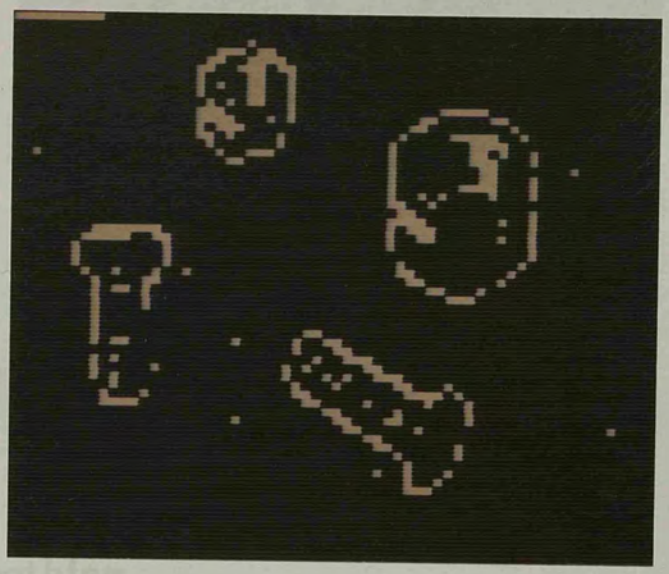


Figure 4.14: Level 7 leaves in nuts and bolts

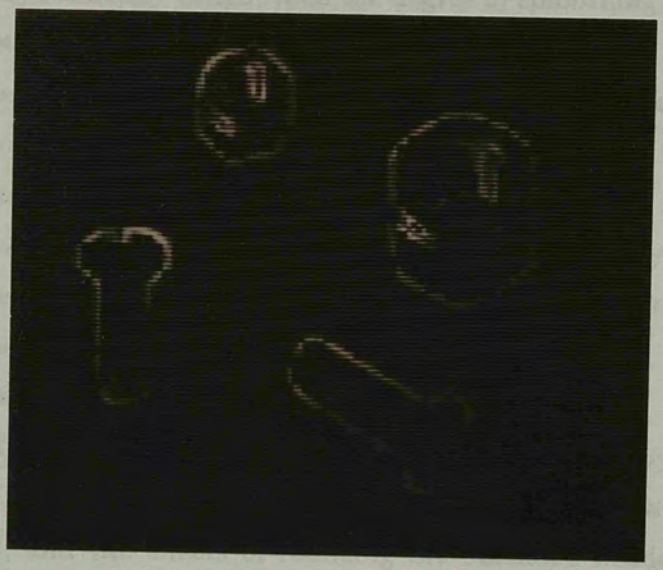


Figure 4.15: Level 7 leaves in nuts and bolts with Sobel data

Figure 4.16: Full Sobel of nuts and bolts

4.9.2 Smoothing

Smoothing is the process of low pass filtering to reduce the effect of high frequency noise. Smoothing will, of course, also degrade edges, and must therefore be applied with care.

In general, each pixel is replaced by the average intensity of its neighbours, typically the median of a 3 × 3 region. The degree of smoothing can be varied by adjusting the size of the region over which the average is calculated. If the region size is dynamically varied, then the degree of smoothing can be reduced near known edges.

It is possible to use the quadtree contour map (*i.e.* the contents of r-space as generated by algorithms 1–3 above) as a controller for the smoothing operator. However, the quadtree is essentially an edge preserving representation, and smooth regions in the image may be distorted (*i.e.* 'blocky'). Because of this, the quadtree controller should only modify the region size near known level 7 leaves, since they are guaranteed to be 'real' as opposed to an artefact of the quadtree conversion process.

Figure 4.17 shows the pen picture of Figure 3.5 after application of a 3 × 3 median filter. Figure 4.18 shows the position of all level 7 leaves in PEN for a threshold of 20, and the effects of reducing the median's window area to the four-connected neighbours at level 7 pixels is shown in Figure 4.19. The writing is more readable using the modified median, but background areas are still well smoothed.

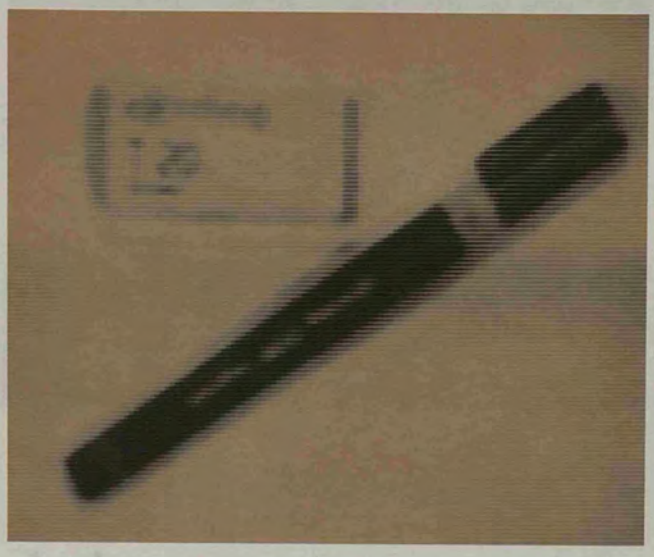


Figure 4.17: Median filter of PEN

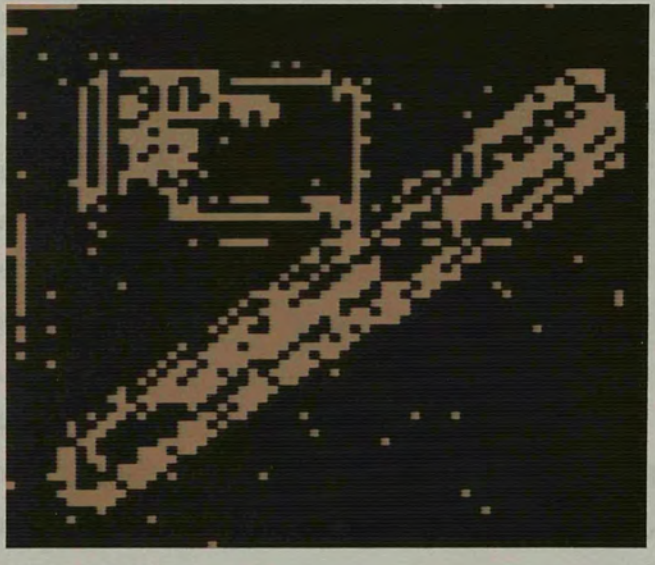


Figure 4.18: Position of level 7 leaves in PEN

Figure 4.19: Median filter of PEN

4.10 Conclusions

Four quadtree generation algorithms have been presented. The simple recursive algorithm QUAD_TOP_RECURS works well on the experimental system that includes a high performance frame store. A bottom up merging algorithm QUAD_MERGE_RECURS requires relatively few frame store accesses and would be preferable when a low bandwidth channel is used between the processor and the frame store. This is especially so on processors such as the VAX or 68000 that have efficient procedure call and stack frame generation instructions. Two sequential algorithms, QUAD_MERGE_RECURS and QUAD_OPT use backtracking and a serial scan of the image. QUAD_OPT, the best algorithm, is dominated by the calculations required to find the leaf positions after a discontinuity has been found. The PDP-11 lacks the necessary bit field instructions to perform the calculations efficiently, but even so QUAD_OPT performs better than QUAD_TOP_RECURS over a useful range of conditions on a system optimised for frame store intensive operations. In VAX machine code, QUAD_OPT is expected to show significantly better throughput.

The algorithms described in this chapter were all developed independently by the author. Bottom-up algorithms for quadtree generation are discussed in [Sam84].

Applications of quadtrees to the global control of local image processing operations have been considered. Such use can lead to a significant speed up, but the rather coarse 'globality' information they provide limits the applicability of the technique, and the time taken to generate the representation can be significant.

They can also be used to adapt the range of a filtering operator around edges so as to preserve sharpness, whilst removing noise from background areas.

Chapter 5

Framestore design

5.1 Introduction

Low level image processing requires high bandwidth transfers between the processor and its frame buffer. Modern computers such as the VAX or 68000 with large virtual address spaces may be able to store complete images using memory mapped buffers, although the multiuser operating systems that support such machines often require non-memory devices to be accessed *via* device drivers that impose substantial overheads. Older machines such as the PDP-11 rapidly run out of virtual address space, and therefore require some kind of memory management unit which may be integral with the frame store.

This chapter looks at the design of frame stores specifically for image processing, as opposed to graphics generation. Display generation, memory subsystems, digitisers and host frame store communications are considered, and four frame stores designed by the author are described. It is shown that direct memory mapping of the frame buffer may not be the best solution, even for those computers that support it.

Software support for these devices is described in Chapter 6.

5.2 Basic frame stores

A video frame store is a block of memory into which digitised video data may be written, retained and subsequently read out for display. The memory is sharable between the video circuitry and some processing circuitry, often a general purpose computer. For image processing purposes, the frame store parameters are:

1. the spatial resolution of the frame store,

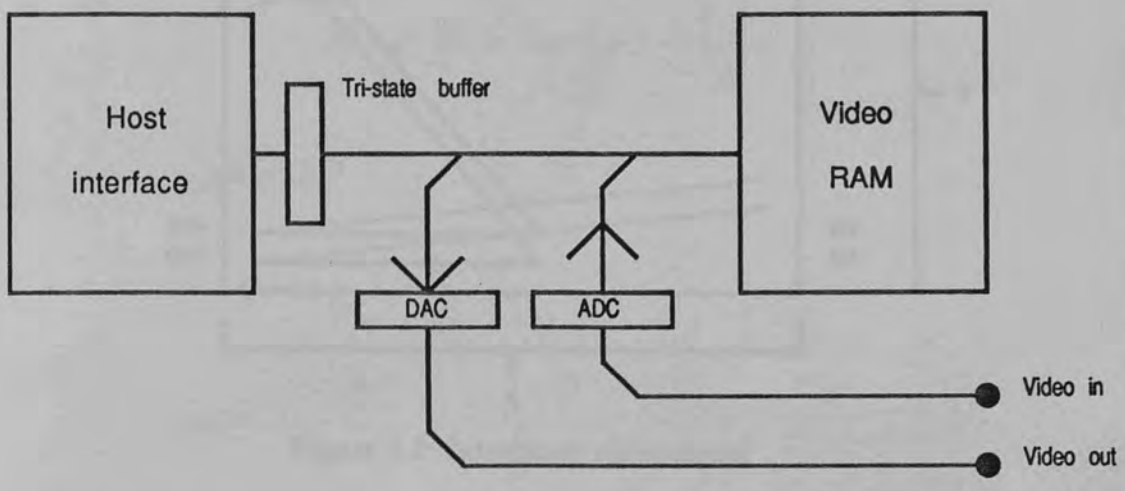


Figure 5.1: Frame store block diagram

2. the grey scale resolution of the digitiser and the memory planes,

3. the ease and speed with which data may be retrieved from the frame store for processing.

   Frame stores attached to computers are also used for graphics work and the display of user interfaces to complex programs. For such systems, the ability of the frame store to maintain multiple images, pan and zoom, and even draw graphics primitives autonomously, are important. Therefore, for completeness we include:

4. the display formatting capabilities,

5. the graphics generation commands.

5.3 Frame store blocks

Figure 5.1 shows a block diagram of a frame store.

The RAM may be addressed *via* the multiplexer from either the host or the video address generation circuitry. There are two primary data buses within the frame store — the Pixel Data (PD) bus which connects the video RAM to the analogue-to-digital and digital-to-analogue converters, and the Control Data (CD) bus which connects the host to the control and status registers. They are connected *via* the PD buffer. In operation, the host sets up control registers *via*

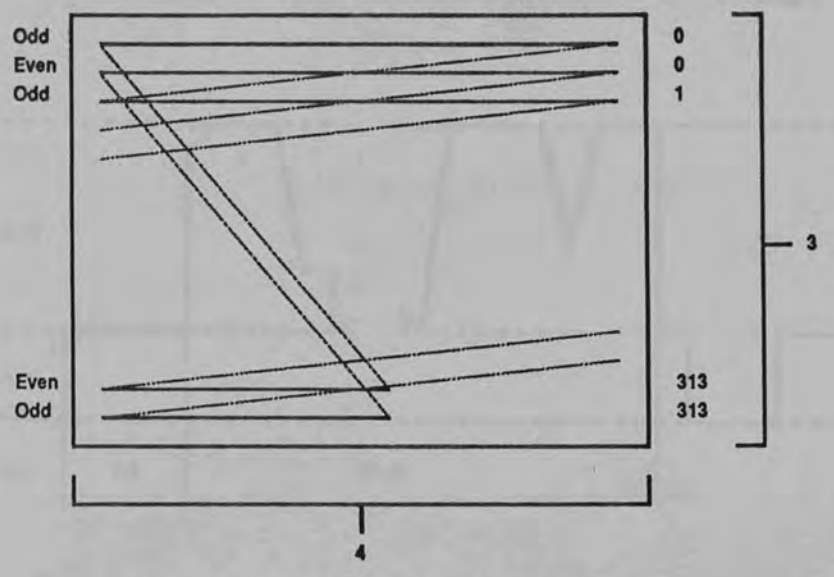


Figure 5.2: Interlaced video signal

the CD bus which define the form of the display and whether frame grabbing is active.

5.4 Video signal timing

The display generation hardware is responsible for supplying the addresses to the memory array which define the way in which pixel data appears on the screen. The simplest display merely maps a contiguous block of memory to a rectangle on the screen. More advanced features allow zooming into the image, scrolling and panning of the display area relative to the stored image, windowing of the data to allow both reduced aperture viewing and the presentation of several images on screen at once, the generation of cursors and cross hairs without disturbing the memory contents, and the mapping of actual pixel values to various on-screen colours.

5.5 Basic video timing

Apart from exceptional cases where a very high on-screen resolution is required, a standard video monitor will be used as the display device. This will be able to display a maximum of around 800 × 600 pixels. When higher resolution is required, special high scan rate monitors are available.

A standard TV signal generates a raster scan with a 2:1 interlace and a 4:3 aspect ratio [IB71] (Figure 5.2).

The line time is 64μs, made up of 1.55μs front porch, 4.7μs sync pulse, 5.8μs back porch and 51.95μs of active video. 'Black' is 0.3V above sync level and

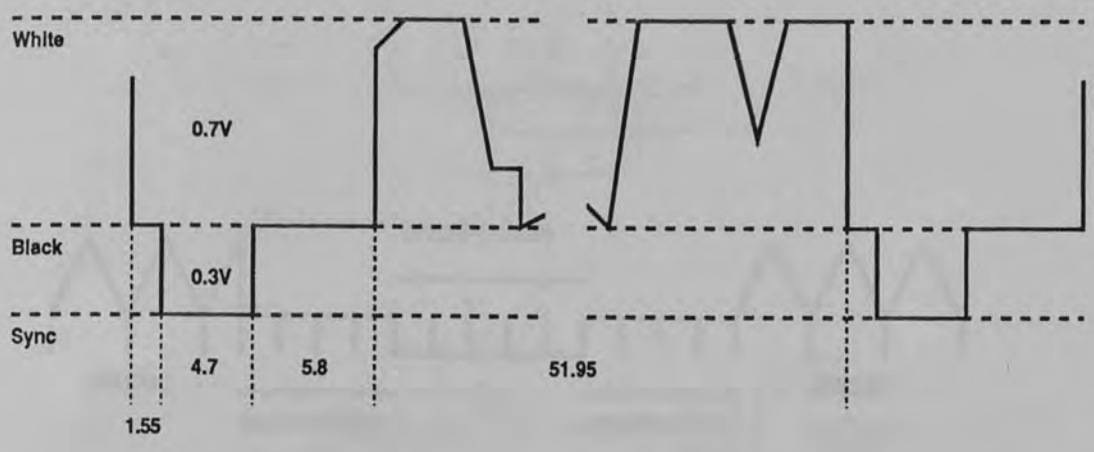


Figure 5.3: Horizontal video timing

'white' 0.7V above black (Figure 5.3).

Each of the two fields is made up of 312.5 lines. *Broad* and *equalising* sync pulses are used to trigger the vertical scan and differentiate between the odd and even fields (Figure 5.4).

5.6 Video timing generation

The video timing will generate horizontal and vertical sync pulses and blanking signals along with addresses for the video RAM.

If less than 300 vertical pixels are required, then the interlacing can be ignored, and two identical fields generated. If the signal is to contain Phase Alteration Line (PAL) encoded colour information, then a colour burst will be required. However, even for low resolution colour displays, PAL significantly degrades the displayed image, so discrete red, green and blue (RGB) drives are usually used. A video cassette recorder will require PAL for colour recording.

The simplest display is formed by mapping video RAM locations to a single rectangle on the screen as shown in Figure 5.5.

This may be achieved by using two counters, one for horizontal timing clocked off a master pixel clock, and another clocked off the HSYNC signal. Registers holding the display and sync start and end points in pixel coordinates are compared with the value of the counter at each cycle and sync and blanking signals generated accordingly (Figure 5.6).

The following pseudo code describes the X-logic function. Note that the

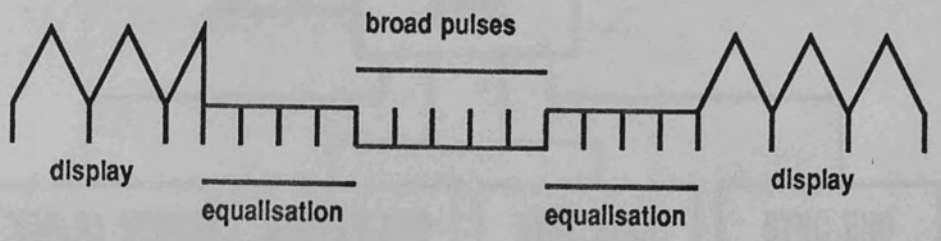


Figure 5.4: Vertical video timing

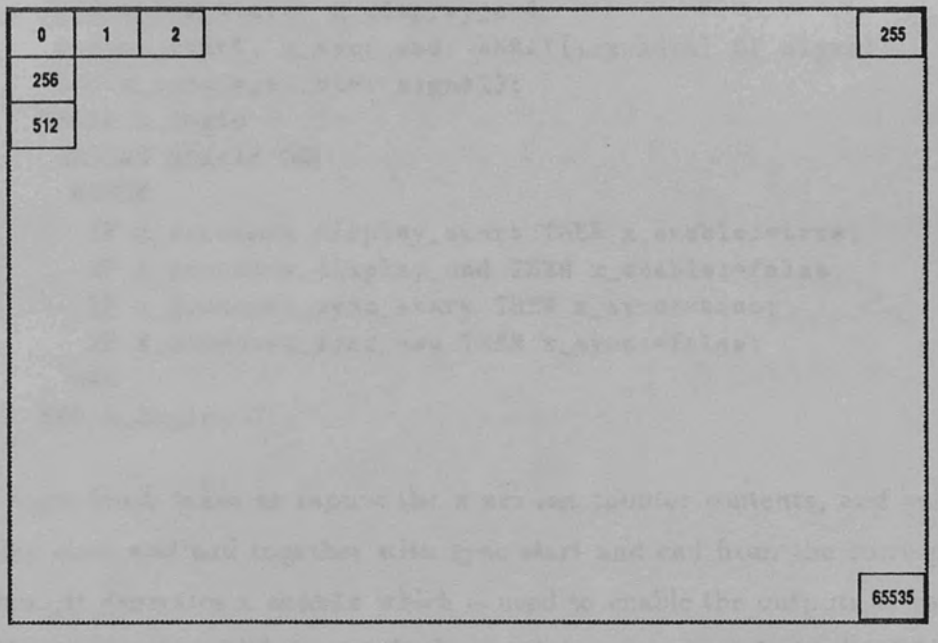


Figure 5.5: Video RAM addressing

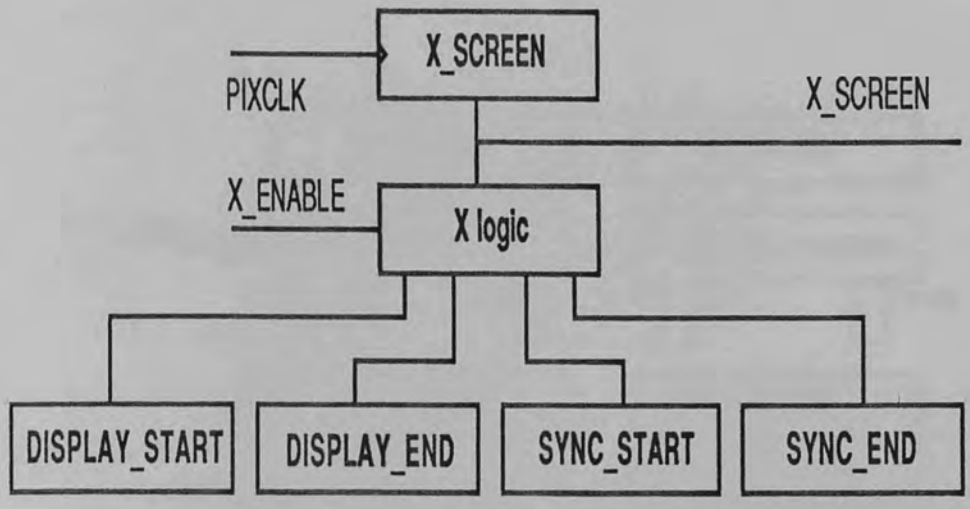


Figure 5.6: Simple display generation logic

`RISING pixclk DO` construction indicates that the enclosed clauses should all be evaluated in parallel on every rising edge of the signal PIXCLK.

```
MODULE x_logic
 (IN pixclk: signal;
  x_screen,
  x_display_start, x_display_end,
  x_sync_start, x_sync_end: ARRAY[1:xwidth] OF signal;
  OUT x_sync,x_enable: signal);
 BEGIN x_logic
  RISING pixclk DO
   BEGIN
    IF x_screen=x_display_start THEN x_enable:=true;
    IF x_screen=x_display_end THEN x_enable:=false;
    IF x_screen=x_sync_start THEN x_sync:=true;
    IF x_screen=x_sync_end THEN x_sync:=false;
   END
 END x_logic;
```

The logic block takes as inputs the `x_screen` counter contents, and values for display start and end together with sync start and end from the corresponding latches. It generates `x_enable` which is used to enable the outputs of the video RAM and `x_sync` which forms the horizontal sync pulse. A similar logic block would be used for the y axis. In a fully programmable system, the position and size of the display window could be modified by updating the latch contents.

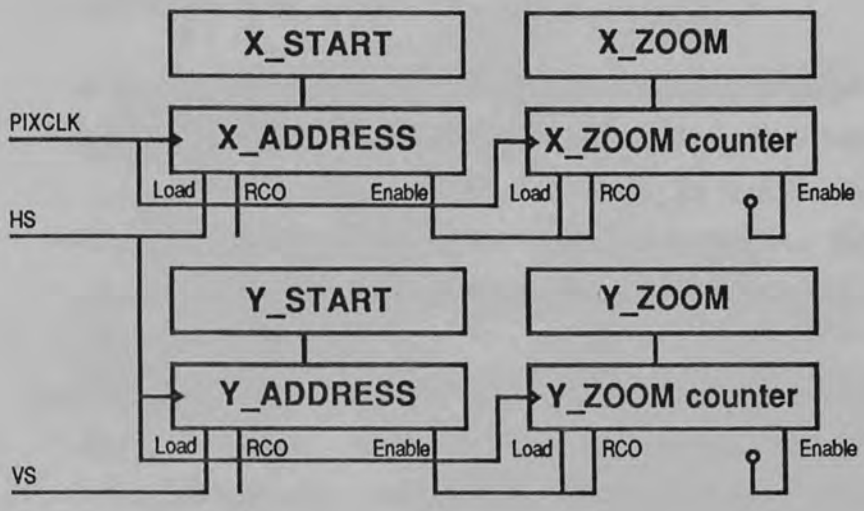


Figure 5.7: Advanced video timing

5.7 Advanced video effects

By changing the mapping of video RAM addresses to points on the screen, zoom, pan, scroll and multiple window displays become possible. In the basic scheme above, the outputs of the screen address counters are used directly to address the video RAM. The advanced scheme requires a separate `x-address` counter, a zoom counter and `x_start` and `x_zoom` registers as shown in Figure 5.7.

The `x_address` counter is loaded by the horizontal sync pulse. The `x_zoom` counter is clocked from the master pixel clock when `x_en` is active. The Ripple Carry Out (RCO) from `x_zoom` is used to clock `x_address` and to reload `x_zoom` from its associated register.

The displayed image can be zoomed by loading a zoom factor into `x_zoom`. This effectively divides down the clock rate to the `x_address` counter causing pixels to be repeated in adjacent display slots. Scrolling is achieved by changing the `x_start` address which will move the display window over the video RAM contents.

Multiple windows may be displayed by replicating the logic of Figure 5.7 and adding multiple `screen_x` registers to the logic of Figure 5.6 to enable the windows. A priority encoder will be required to resolve addressing conflicts between the multiple windows if more than one is active at any point on the screen

(*i.e.* when overlapping windows are present)

5.8 Memory

Large amounts of storage are required for high resolution images. Graphics oriented frame stores are available with on screen resolutions of 1024 × 1024 24-bit pixels, and several separate image stores. However, for real time image processing work it is preferable to use the lowest resolution compatible with solving a given problem, because of the increase in processing time associated with high resolution images.

Most of the work described in this thesis is based around 128 × 128 eight-bit monochrome images, and each of these requires 16K bytes of storage. It is useful to have between four and sixteen images available simultaneously, and so between 64K and 256K bytes of memory are required. The number of simultaneous images required is independent of display resolution, so that for a 1024 × 1024 24-bit colour system, between 12M and 48M byte (1M = 1024K) are needed.

The cheapest way of implementing memory systems is to use dynamic RAMs (DRAMs) which are usually a generation ahead of static RAMs (SRAMs), *i.e.* at any one time equivalently priced DRAMs and SRAMs will differ in their capacity by a factor 4. However, 512 × 512 pixel timing requires memory cycle times below 70ns, and no commercially available DRAMs have such a short cycle (as opposed to access) time.

The straightforward solution to this bandwidth problem is to use static RAMs which are available with cycle times down to 40ns and below. Another advantage of using static RAMs (apart from their cycle time) is the lack of critical timing needed for refreshing dynamic RAMs, although in a video system continual scanning of the memory array is already occurring, so refresh may not be much of an overhead.

Large static RAM arrays are uneconomic in terms of price and consumption of board real estate. Various modifications to the basic DRAM design have been made by manufacturers to overcome the bandwidth problems for video applications:

1. Page mode [Tex84]

   Internally, dynamic RAMs comprise square arrays of bit cells. Read-out occurs by precharging an entire set of column lines which are then condi-

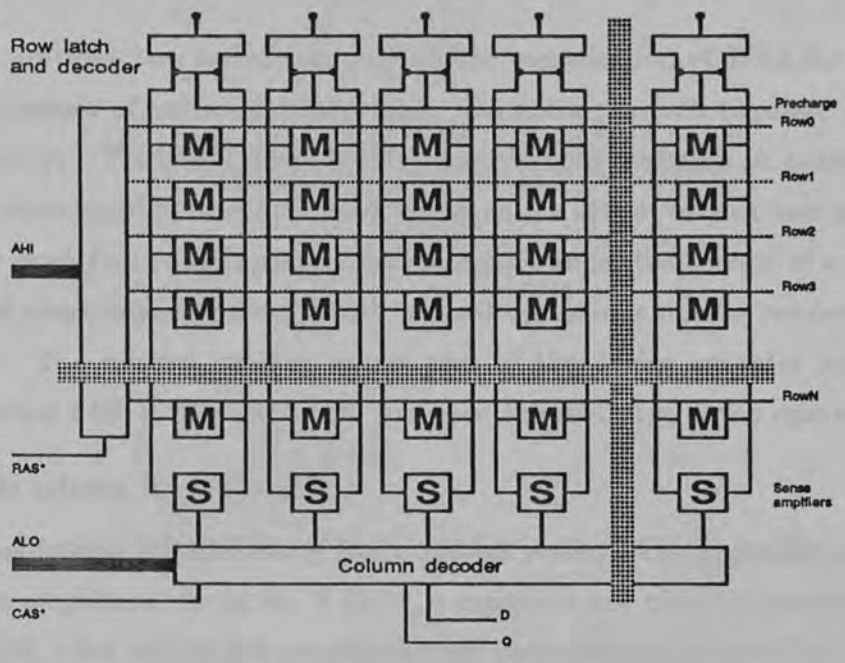


Figure 5.8: Dynamic RAM architecture

tionally discharged by a selected row of cells (see Figure 5.8). The resulting logic levels are sensed and latched in a row of sense amplifiers. The required bit is then steered to the output *via* the column decoder. This is an inherently two phase operation which gives rise to the familiar multiplexed row/column addressing mechanism of commercial DRAMs. However, once a row of bits has been latched in the sense amplifiers, fast single phase access to other bits in the row is possible by latching new column addresses. This can double the bandwidth of the device for the sequential type of access required for video display. Most new DRAMs support page mode.

2. Nibble mode [Inm84b]

   Many dynamic RAMs are internally organised as four separate arrays. Four bits are accessed on each cycle, and two bits of the column address are decoded to provide an apparent single bit access. An access to any other bit, including one in the same nibble, requires a complete memory cycle. INMOS market a device (IMS2600) [Inm84a] that has a two-bit presettable counter between the bottom two column address inputs and the nibble-bit decoder. This counter is loaded on the falling edge of $\overline{CAS}$, and is clocked on the rising edge. Normally this device behaves as a standard 64K $\times$ 1-bit DRAM, but by toggling $\overline{CAS}$ whilst $\overline{RAS}$ is low the counter will be updated and all four bits in the nibble may be sequentially accessed.

3. VRAM [Hit86]

The previous two techniques exploit the organisation of DRAMs and use a minimum of extra on board logic. No extra pins are required for their operation. VRAMs (video RAMs) contain shift registers in parallel with the sense amplifiers which can load an entire row of bits in one cycle and then clock them out through a separate port under the control of a separate serial clock asynchronously with normal operations on the random access port. The normal random access part of the device operates just like a standard 64K $\times$ 1 DRAM, and can even support page mode operations.

4. Static column RAM [Inm86]

   Inmos have a DRAM which has a 256-bit static RAM in parallel with the sense amplifiers. As in the VRAM, a complete row may be loaded into the SRAM, after which full random access operations can occur both on the DRAM and the SRAM. The disadvantage of this approach is the extra pins required to support independent addressing on two arrays.

In the general case where completely random access at high speed to all pixel data is required, none of these techniques are helpful, since the linear address scan is no longer guaranteed. Random access may be required by high speed processors including a graphics processor. Of course, it is possible to limit the cycle time of the processor to allow for the full access time of the RAMs, but if this is not satisfactory there is no recourse but to a true high bandwidth device. As a compromise it may be viable to maintain just one display plane of static memory, and build a bulk array of slower dynamic RAM. A simple Direct Memory Access (DMA) unit consisting of three counters would be sufficient to transfer blocks at high speed across this two tier system. Static column RAMs may be the forerunners of a whole family of devices that integrate useful amounts of static RAM and large amounts of dynamic RAM in a transparent way onto one chip.

5.9 Digitiser

For image processing, the digitiser is the most critical part of the system, and it will often be the most expensive too. A 128 pixel line requires samples every 400 ns (2.5 MHz) and a 768 pixel line every 66 ns (15MHz). At these frequencies a *flash converter* will be required. This comprises a set of matched comparators connected to a resistive divider network. The outputs of the comparators will usually be fed to an on-board encoder so that a binary (rather than thermometer code) output will be produced. These devices are expensive because of the

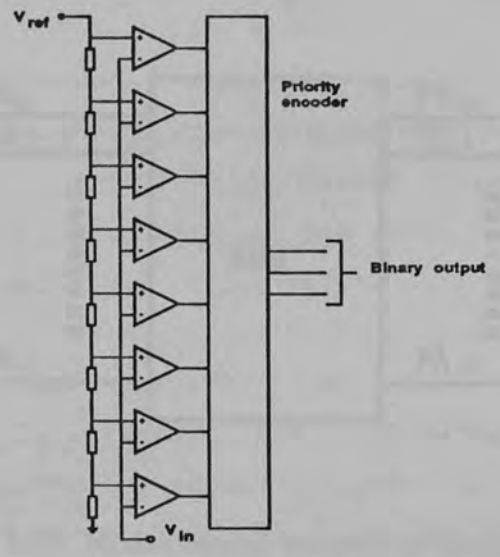


Figure 5.9: Flash analogue to digital converter

difficulty of matching the comparators and resistors to such an accuracy that the inherent non-linearity is less than the resolution of the converter. The resolution increases exponentially with the number of bits in the output, so an 8-bit converter will be much more expensive than the 7-bit version. Only the most expensive of cameras are capable of producing eight bits of information, and so it is common to find only seven or six-bit converters used. To resolve information in the eighth bit, the noise in the system must be less than 1/256 of the maximum amplitude. This corresponds to a signal to noise ratio of better than 48.2 dB. An expensive broadcast quality colour camera (JVC KY-2000B) in use in this laboratory has a S/N ratio of 52dB. Generally available monochrome cameras suitable for industrial inspection would have a worse intrinsic S/N ratio and be susceptible to noise pickup from nearby machinery.

5.10 Host interface

The most important attribute of a frame store for real time image processing is the data transfer speed between image memory and the processor. One of three strategies may be adopted:

1. Integrate the image memory directly into the processor's standard memory architecture so that images appear as arrays in main memory.

2. Attach the frame store as a high speed peripheral, either *via* special I/O channels or *via* DMA links transferring image data into main memory. In

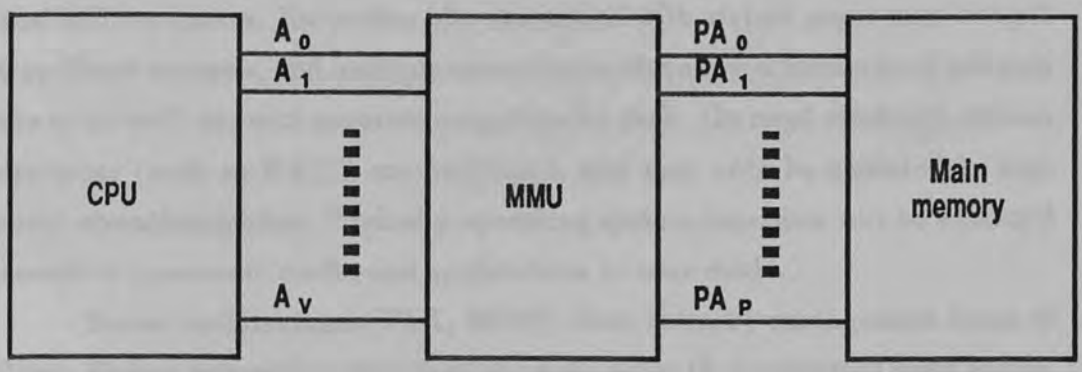


Figure 5.10: Minicomputer memory architecture

this case the images again appear as arrays in main memory, but access times will be long.

3. Use a special memory port that exploits the special behaviour of image processing operators to increase throughput.

5.11 Memory architectures

Many older generation processors have severe limits on their addressing range, and this may itself slow down frame store accesses. Figure 5.10 shows a typical minicomputer memory architecture.

The CPU has an address bus of a_v lines. Thus the maximum number of addresses that may be accessed directly by a program is 2^{a_v}. Typically a_v ranges between 16 and 32, and the address range is known as the virtual address space. The MMU provides two functions. For systems with a small virtual address space, access to larger physical arrays is possible *via* expansion bits stored in MMU registers. On the PDP-11/34a for instance, the 64K byte virtual address space is split into eight pages, each with an associated Page Address Register (PAR). The PAR contains a 12-bit offset specifying the physical address of the first location in the page. Offsets must be aligned with a 32 byte boundary, that is the 12-bit offset forms the high 12 bits of an 18-bit address. During execution, this 18-bit address is added to the offset into the page specified by the virtual address to form an 18-bit physical address. Extensions of this arrangement allow PDP-11s to access up to 4M byte of memory using a 22-bit address.

The second function of memory management is to protect segments of

code and data from each other, and this is useful even on processors with large virtual address spaces. Protection bits associated with virtual pages may be used to trap illegal accesses, and multiple execution modes allow a hierarchy of software layers to be built up, with separate mappings for each. On most machines, certain instructions (such as HALT) are privileged, and may only be executed in high priority execution modes. Typically operating system functions will be executed in *kernel* or *supervisor* mode, and applications in *user* mode.

Newer architectures (VAX, 68020) allow memory management traps to be taken during instruction execution, at which point the instruction itself aborts, rather than running to completion before entering the trap service routine. This allows the generation of *virtual machine* systems, in which a large virtual address space is mapped into a small physical address space (the reverse of the earlier memory management scheme). Only small parts of a program need to be memory resident simultaneously. If a reference to a non-resident page of code is made, a *page fault* occurs which suspends instruction execution whilst the relevant page is fetched from backing store. The offending instruction is then restarted without any disturbance to the program's context.

Indiscriminate use of virtual memory may produce poor results. Many studies have been made into the optimum size of page and the best algorithm to use for retaining pages in memory [Knu70]. The VAX 11/780 uses a page size of 512 bytes. Efficiency of accesses to large arrays can vary significantly depending on the order in which the array is accessed. Consider a 1024×1024 byte array which is to be accessed *via* one 512 byte page (an extreme example). FORTRAN stores arrays by column, rather than the more natural row first ordering. If the array is to be raster scanned in row order, then each pair of references will be separated by 1024 addresses, and so every single reference will cause a page fault. However, if the array is scanned in column order, a page fault will occur only when a page boundary is crossed. Row order traversal thus generates 512 times as many page faults in the worst case. For simple processing, page faulting may dominate computation thus increasing execution time by greater than two orders of magnitude. Section 5.14 describes a memory management approach that allows pixel windows to be accessed *via* absolute addressing.

5.11.1 Integration into main memory architecture

If the host has a large virtual address space, or if the virtual to physical relocation is efficient and the physical address space is large, it may be possible

to place image memory on the same bus as main memory. A contention strategy will be needed to resolve conflicts between video and host access. Usually this will mean either making the processor wait until video blanking time before allowing memory access, or allowing the processor to interrupt video operations, thus causing on-screen *hashing*. Once the image is available in memory, efficiency will be very dependent on the speed of indexed addressing on the host, and the algorithm used to calculate array offsets.

5.11.2 Array access techniques

Access to an element $a(i,j)$ of a two dimensional array will be mapped to an address:

$$base + j \times imax + i$$

where *base* is the address of the first element in the array and *imax* is the largest value taken by i. Calculation of the product term will dominate the access time, especially on processors lacking hardware multiply.

This multiplication overhead may be removed by the process of vectorising, *i.e.* using an auxiliary vector $v(j)$ that holds the start addresses of each of the j rows. In that case, access to element $a(i,j)$ maps to

$$v(j) + i$$

The space overhead is one column of pointers, which is usually the same as one column of integers.

Many high level language compilers automatically make use of vectored arrays for wide and high dimensional cases. The RT-11 FORTRAN IV compiler in use in this laboratory vectors arrays if the ratio of sizes between the array itself and the access vector is less than 25%. It always attempts to vector the higher dimensions, so it is good programming practice to declare a 5×100 array as `a(100,5)` not as `a(5,100)` since only 5 words of vector storage are required as opposed to 100.

5.11.3 Indirect and indexed addressing

Both of the above formulae for accessing array elements require at least one addition and one indirection. Nearly all processors provide an indexed addressing mode for this purpose. In general, a constant is added to the contents of a register, and the result used as the address of the element. Because of this,

indexed addressing is slower than using the contents of a register as the address directly, because the constant must be fetched from memory and the addition performed before the effective address is available. On an MOS memory PDP 11/34a, use of indexed addressing adds around 1.5μs (dependent on instruction) to execution time over simple indirection. Absolute addressing is equivalent to indexed addressing on the program counter, but if the address is known at compilation time, the dereferencing of coordinates is not required. Thus accessing of array elements, even using special addressing modes is slower than accessing absolute locations.

5.11.4 Integration into the peripheral system

If the available physical or virtual memory space of the processor is less than that required for image storage, then the image memory cannot be integrated with the main memory architecture. In this case, the frame store may be treated as a form of backing store, and integrated into the peripheral system. Some machines use specialised I/O processors (called 'channels' in IBM installations) to transfer data between peripherals and main memory. Other smaller scale machines use simple DMA controllers, which typically consist of a word counter, a main memory address counter and a peripheral controller. A transfer is initiated by loading the start address and the number of words to be transferred to the relevant registers, and proceeds autonomously to completion or an error abort, at which point the host is interrupted. In the case of image memory, the peripheral controller section can be as simple as a third counter addressing the memory planes.

This form of connection is popular with frame store manufacturers because it requires the minimum disturbance to already existing hardware and operating systems. The only hardware modification required is the addition of a small peripheral controller (rather than the rebuilding of the memory environment required above), and since all modern operating systems implement device independence through a device driver protocol, the only system software required to support the frame store is a simple device driver to read and write blocks of data.

Unfortunately, the peripheral type connection brings no operational advantages to the user, since the data, once transferred, merely appears in memory as an array just as in the memory integrated case. The transfer itself imposes an overhead, which may be significant since peripheral channels are only engineered to cope with disc speed accesses and may not be able to make use of the extra

bandwidth available from the frame store.

5.11.5 Special purpose memory architectures

As noted above, image memory requirements often outstrip available virtual or even physical addressing capabilities, and a common solution to this is to attach the frame store as a bulk storage peripheral. Another common solution is to use coordinate registers to specify the required pixel, and a pixel register to access the image planes. Four 128 $\times$ 128 byte wide planes could be accommodated on a PDP-11 with the use of only three 16-bit words: an X register, a Y register and a pixel register. This may be seen as a special purpose memory management scheme, whereby the X and Y registers hold the offset into image memory (equivalent to the PDP-11 PAR) and the pixel register corresponding to the virtual page. Naturally, this arrangement is not very efficient, since in general a random access to any pixel will require three memory accesses, although multiple accesses to the same pixel will in fact be more efficient than in the array mapped case because as noted above, compile time calculated absolute or relative addressing is quicker than run time calculated indexed addressing. Overheads can be reduced by providing multiple pixel registers, thus increasing the size of the window into image space. At this point a memory management scheme that parallels the virtual/physical mapping has been created, except that address translation occurs only for that block of addresses allocated to the pixel window.

Many of the image processing operators do not access images randomly, but in a well defined scanning sequence. At each point in the sequence, a small area of the image surrounding the central pixel is accessed. The memory management units on frame stores built for this project map the central pixel and its neighbours to a series of absolute locations in memory. This allows window operators to use absolute or relative addressing to access the window, rather than the complex indexed accesses used when the image data exists in main memory. Thus, although memory management is being used to circumvent the limitations of the processor, it actually provides significant improvements in efficiency. Extra speed can also be gained if a second set of window registers is provided that when accessed cause an automatic increment of the window position to the next location. For applications requiring a raster scan of the image this completely removes the coordinate update overhead.

This approach works well on a PDP-11. However, on the microVAX, the peripheral devices on the Qbus (including the frame store) have much longer access

times than the processor's main memory which is connected *via* a synchronous bus called the PMI (Private Memory Interconnect). The speedup is due to three factors:

1. the PMI runs synchronously with the processor removing de-skew overheads associated with the Q-bus,
2. the PMI data bus is 32 bits wide whereas the Qbus is only a 16-bit bus ,
3. the microVAX Q-bus interface is optimised for block mode DMA operations and is in fact slower than some PDP-11 implementations for programmed I/O.

As a result, it is almost always best to transfer data from the frame store to the microVAX main memory for processing rather than performing the processing 'in the framestore'. The exception is when main memory on the microVAX is too small to hold the process, in which case paging will begin to occur. In extreme cases when multiple large processes are attempting to run, swapping of entire processes will occur and a significant degradation of performance will result.

The memory architecture described in this section has been used to good effect on the IPOFS and V1 frame stores described below. V2 which was designed for VAX hosts would not gain from such circuitry and adopts a more conventional memory mapped approach. V3, which will also be used with PDP-11 hosts will have a memory management unit.

5.12 IPOFS — a compact high performance frame store

The Image Processing Oriented Frame Store (IPOFS) is designed to be compatible with a frame store in use in this laboratory described by Cook [Coo83a]. Cook's frame store occupies a large part of the address space of a PDP-11 but allows access to a 256K byte image address space using window mapping. IPOFS is constructed using more modern and compact technology and offers several new features. It is more closely optimised than Cook's framestore for use with PDP-11 systems in that it has a fully asynchronous interface to the host and it requires only 1K byte of address space. This is important because PDP-11 systems expect all peripherals to be located in an 8K byte section at the top of the address space, and accessing devices outside that area causes problems with

the operating system. Two software systems (PPL2 [Coo83a] and PIPE which are described in Chapter 6) have been developed to overcome this addressing difficulty, but neither would work on a basic PDP-11 lacking memory management hardware. The new frame store can operate with a minimal single board PDP-11 computer.

5.13 IPOFS specification

IPOFS is capable of holding and displaying up to eight 128 × 128 pixel images at 8-bit resolution. It can acquire real time video data from a camera digitised to 7 bits. Memory management allows a host processor direct access to points within a 5 × 5 window centred on any pixel. Access to other parts of the image requires the updating of a coordinate register.

The main controller and host interface reside on a single quad-width Q-bus card. Up to 128K bytes of static RAM may be used as the pixel memory. The prototype is based around 16K bit RAM chips and two boards the same size as the controller would be required to fully populate the address space. Not all of the controller card is occupied at present, and if a PCB were designed it would be possible to include all 128K of RAM on board using 64K × 1 static RAMs.

In addition, a small digitiser board is housed in a die cast box for improved noise immunity; this passes digitised video to the controller over a ribbon cable. The controller and memory is designed to reside inside a VT103 intelligent terminal along with a PDP-11/21, 11/23 or 11/73, disk interface, host memory and a real time clock card. This forms a very compact system, and when coupled with a small disc drive provides a portable version of the main research minicomputer. Algorithms developed in laboratory conditions can be tested in the factory, and only a small amount of equipment needs to be transported.

5.14 IPOFS theory of operation

A block diagram of IPOFS is shown in Figure 5.11. Since a 6845 CRTC is used to generate the video timings, some display effects are achievable including scrolling, windowing and the display of a non-destructive cursor. A light pen input is also available to the frame store. A ROM based window mapper is used in conjunction with X and Y coordinate registers to speed access to the image data. The size of the window is restricted to 32 elements so as to reduce the demands on address space. Individual windows are available for each image, but

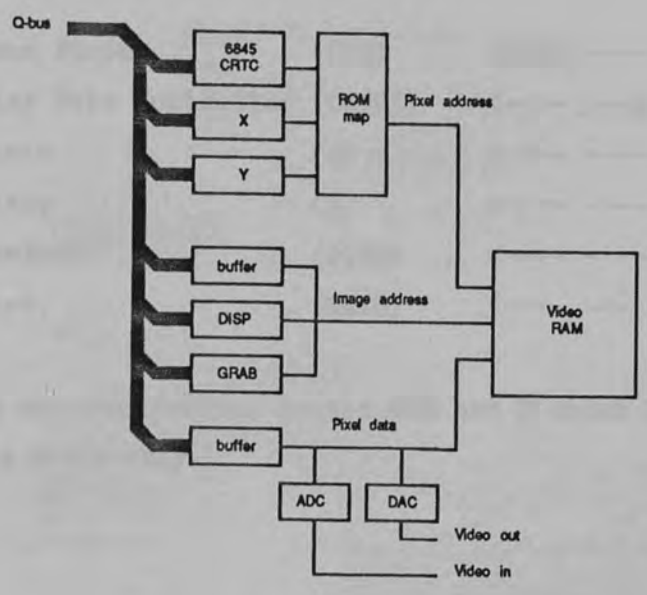


Figure 5.11: IPOFS block diagram

they are all linked to the same coordinates.

5.14.1 Register set

IPOFS occupies 1K bytes of the PDP-11 I/O page which is split into two 512 byte pages, one for control registers (CONBLK) and the other filled with window registers (REGBLK). There are 6 control registers as shown in Table 5.1.

The CSR has four read only bits that return the current status of the synchronisation and blanking signals. V is the vertical sync, H the horizontal sync and B the blanking signal. In addition the C bit returns the logical OR of the syncs, which supplies composite sync to the camera. The CSR also has one read/write bit (G) that governs the state of the video acquisition and display circuitry. If G is high then the ADC will write data into memory giving a continuous frame grab cycle. When G is reset, normal display of the stored image is resumed.

The CRTC register is used to access the 6845, which contains 18 internal registers. Rather than mapping these directly, which would require 18 words of address space, a separate address register is used to specify the active internal register, which is then accessed *via* a 'data' pseudo register. As a result, only two bytes are used in an eight bit system. In IPOFS, this is further reduced by using bit 8 of CRTCR to specify which pseudo register is to be opened, and accessing

Control and Status	(CSR)	<CVHB ---- ---- ---G>
Cathode Ray Tube Controller	(CRTCR)	<---- ---R cccc cccc>
X coordinate	(X)	<---- ---- xxxx xxxx>
Y coordinate	(Y)	<---- ---- yyyy yyyy>
DISPlay select	(DISP)	<---- ---- dddd dddd>
GRAB select	(GRAB)	<---- ---- gggg gggg>

All bits are read/write, except CVH and B which are read only, and R which is write only.

Table 5.1: IPOFS register set

both address and data registers through bits 0-7. To clear internal register 5, the following sequence is required:

```
MOV    #5,CRTC      ;bit 8=0 for address register; data = 5
MOV    #400,CRTC    ;bit 8=1 for data register; data = 0
```

By manipulating the internal registers, timing parameters can be set up and certain video effects can be generated such as scrolling and limited zoom and windowing. Because of the constrained nature of the 6845, programming is quite difficult, and for further details reference should be made to [Mot81]. Usually, the 6845 will be initialised at powerup using the supplied utility IPINIT, and not subsequently altered.

The X and Y registers together specify the centre point of the window. These registers are eight bits wide, with only the lower seven bits significant.

DISP specifies the number of the image to be displayed. It is an eight bit register with the lower three bits significant. If the G bit in the CSR is set, video acquisition is in progress, and the display will show the outputs of the digitiser.

GRAB is an eight-bit register — one bit for each of the eight available images. When the corresponding bit is set, that plane is enabled for frame grabbing, and will have video data written into it when the G bit in the CSR is set. In this way, multiple copies of an image may be acquired simultaneously. This is useful for serial algorithms that destroy the original image during processing but need to refer to the original in later stages.

The other 512 byte page (REGBLK) contains 8 sets of 32 window registers, one set per image plane. These are allocated in a spiral fashion:

```
27        31        26
   16 15 14 13 12
   17  4  3  2 11
   18  5  0  1 10 30
   19  6  7  8  9
   20 21 22 23 24
28        29        25
```

This provides a 5 × 5 window with some useful extra points. The numbering is unfortunate since the spiral is not positively increasing (the origin is in the top left of an image so as to conform to the conventional video raster scan) — this is to retain compatability with Cook's frame store.

5.15 Internal operation

IPOFS has a controller board, one or two memory boards and a digitiser board interconnected by ribbon cables.

5.15.1 Controller board

The controller has three main internal sections: the Q-bus interface, the video timing logic, and the registers and multiplexers. A diagram is shown in Figure 5.12.

Host access occurs *via* the Q-bus, which is an asynchronous multiplexed bus. Address/data lines, address and read/write strobes, and the RPLY handshake line are buffered from the Q-bus connector. Addresses are latched on the rising edge of the address strobe, and decoded to start host cycles. REGBLK addresses are decoded to give individual clocks for each of the onboard registers. A byte wide static RAM (called the shadow RAM or SHRAM) parallels all REGBLK addresses. On a write both the register and the RAM are updated. On a read, only the SHRAM is accessed. This allows read-back of all registers.

Master timing is generated from a 20MHz crystal which is divided down to give 2.5 and 3.3MHz pixel clocks for rectangular and square pixels respectively. In the prototype, only square pixel timing is used, but the addition of a multiplexor

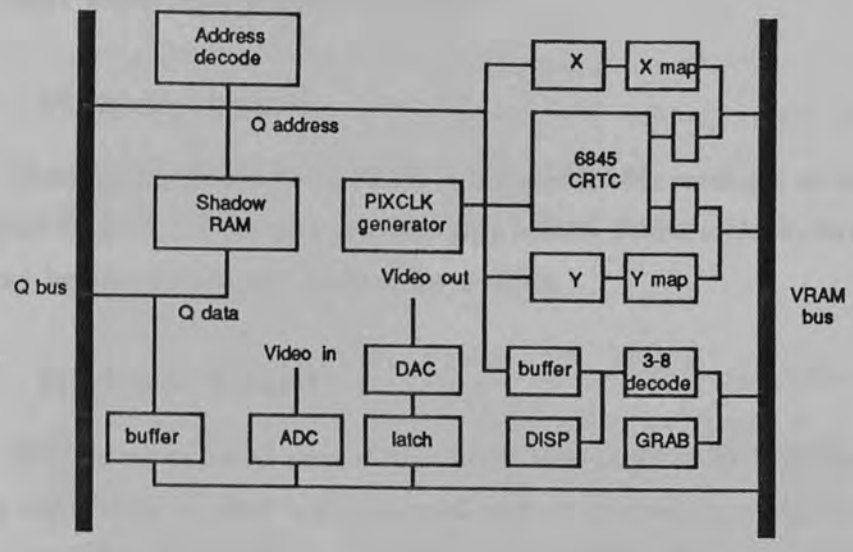


Figure 5.12: IPOFS internal operation

under control of one of the unused CSR bits would provide software selectable square or rectangular pixels.

Sync signals, blanking and physical address signals are provided by the 6845. At its maximum pixel frequency of 2.5MHz, the 6845 is not able to generate a 128 × 128 square pixel display, so the CRTC is programmed to provide a 64 × 128 pixel display, and the least significant PX line is driven directly from the 3.3MHz clock.

Video data from the digitiser or memory board is latched and passed to a fast DAC. The latch is cleared during blanking time. The output of the DAC is mixed with sync pulses in a high output gated-input differential amplifier, which drives 75Ω loads directly.

An eight-bit latch is provided for each of the registers. The outputs of the X and Y registers are combined with the low 5 address bits from the host in two mapping PROMS. These contain lookup tables for the offsets required to generate the spiral window. During host access to the memory board (*i.e.* CONBLK accesses) video addresses from the 6845 are disabled and the memory bus is driven from the mapping PROMs.

CSR bit 0 (G) governs the acquisition/display mode. During display mode, the low three bits of DISP are decoded to select one of eight image address lines. During acquisition, these lines are driven by the outputs of the GRAB

register, and write pulses are sent to the memory board. The selected images will therefore have video data written into them.

5.15.2 Memory board

The memory board holds 32 2K $\times$ 8 static RAMs arranged as four blocks of 16K bytes each. A 3-8 decoder provides chip selects within a block, but multiple blocks may be simultaneously enabled for writing.

5.15.3 Digitiser board

This board is based around the TRW 1047, a 7-bit 20 MSPS convertor. Incoming video data is black level clamped and dc coupled to a high bandwidth inverting amplifier. Conversions occur on command from the video timing logic, and are multiplexed onto the image data bus using tri-state buffers.

5.16 Programming differences with Cook's frame store

There are several differences between IPOFS and Cook's framestore that affect the portability of software:

1. square/rectangular pixels,
2. edge registers,
3. length of coordinate registers,
4. frame grabber sequencing,
5. no hexagonal tessellation.

The square pixels of IPOFS mean that frame store coordinates map directly to real world coordinates. With Cook's frame store, a 4/3 correction must be applied to all y coordinates to compensate for the rectangular mapping used. When IPOFS is in rectangular mode, the mappings correspond. However, IPOFS rectangular mode is disabled in the prototype.

Cook's frame store has *edge registers* which are used to catch window accesses mapping to out of bounds addresses as for instance when x=y=0 and an attempt is made to access window element P2. IPOFS was designed with edge

registers, but they are disabled on the prototype so that out of bounds accesses wrap round.

The coordinate registers on Cook's frame store are seven bits long, *i.e.* the Pascal fragment `x:=127; x:=x+1; writeln(x)` outputs zero. This is dangerous, because the host processor will assume that the registers are byte-wide. An x-scan on Cook's frame store is typically generated with:

```
x:=0; REPEAT {function}; x:=x+1 UNTIL x=0;
```

An optimising compiler will generate the following machine code:

```
        clr x
loop:
        ; function
        inc x
        bne loop
```

When x is 127 the `inc` instruction will yield 128 in the processor's internal ALU. When this result is written back to the frame store, the top bit will be discarded leaving a zero. The next increment instruction will therefore yield a one. As a result the ALU result will never be zero, so the loop will never terminate. The coordinate registers on IPOFS are eight bits long and behave exactly as normal memory locations.

Cook's frame store contains internal logic for sequencing the frame grab. IPOFS requires the host to monitor the vertical synchronisation bit and enable and disable grabbing directly.

Cook's frame store supports hexagonal mappings of the window registers. This is not available on IPOFS because it was a very rarely used feature of the earlier frame store. It could be added simply by putting new addresses into the lookup ROMs.

5.17 V1

The project described in Chapter 9 is based on a real time MIMD multiprocessor system called IMP (Imaging Multi-Processor). A new framestore called V1 was designed based on IPOFS but with a line-scan interface, a high speed memory mapped port, VME bus to the IMP backplane and using newer memories that provide a true single board frame store.

V1 is internally similar to IPOFS with the following differences:

1. The Qbus interface is replaced with a VMEbus interface. It is intended to be used with a separate Q-bus to VMEbus converter to be described in Chapter 9.

2. High speed 16K $\times$ 1 static RAMS are used to reduce the physical size of the memory array.

3. Access to a 7 $\times$ 7 window is supported (5 $\times$ 5 on IPOFS).

4. V1 has a novel line-scan interface.

5. V1 has a memory mapped port in addition to the ROM mapped window port.

6. V1 has a WIPE register which enables an image plane to be initialised to a constant value in one frame time without processor intervention.

7. Simultaneous grabbing to multiple image planes is not available as a result of buffer load problems experienced with V1.

5.17.1 Register set

V1 occupies 1024 bytes of VME bus address space, arranged as eight blocks of 64 word registers. Most of the registers are eight-bit only, with the top eight bits of the word unused.

IPOFS has two separate 512 byte pages, one for window registers and the other for control registers. However, V1 is accessed *via* the Qbus to VME bus protocol converter described in Chapter 9 which maps 256 byte regions of VMEbus space to Qbus space. To ensure that if only one page were available for mapping all the V1 control registers would be available, multiple copies of the control registers are available as shown in Figure 5.13. Each register block contains the elements shown in Table 5.2.

The window element, x,y and display registers work as for IPOFS. CRTCR is identical to the IPOFS CRTCR except that the R bit has been moved into the CSR. The GRAB register takes a three-bit number to select one of eight planes for frame grabbing.

5.17.2 Control and status register

The CSR on V1 has extra bits to control the extra frame store features:

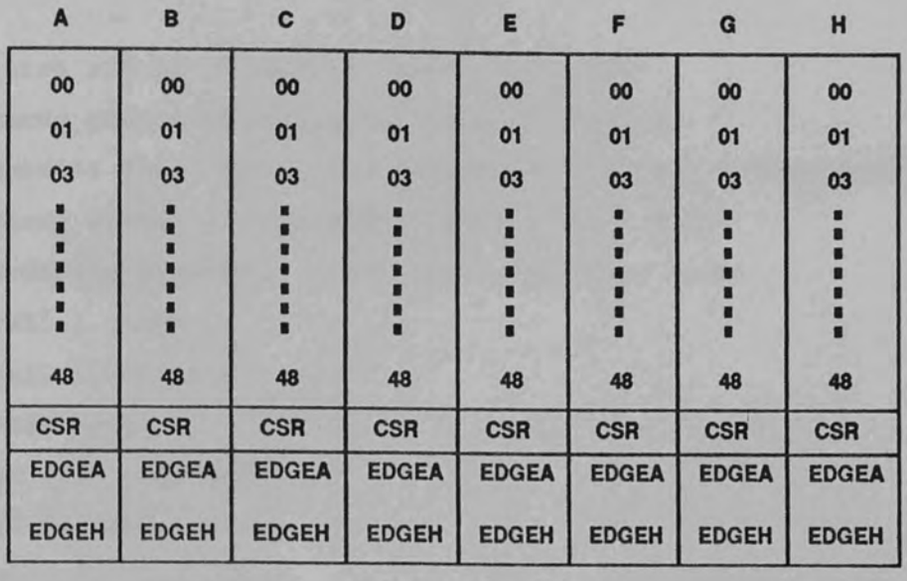


Figure 5.13: V1 register blocks

Offset	Name	Mnemonic	Bits
0	A window element 0	A0	<---- ---- aaaa aaaa>
2	A window element 1	A1	<---- ---- aaaa aaaa>
96	A window element 48	A48	<---- ---- aaaa aaaa>
98	Control and status	CSR	<vhcb x--- pqal rdgs>
100	CRTC control	CRTCR	<---- ---- cccc cccc>
100	Wipe	WIPE	<---- ---- dddd dddd>
102	Expose	EXPOSE	<---- ---- dddd dddd>
104	GRAB	GRAB	<---- ---- ---- -ggg>
106	display select	DISP	<---- ---- ---- -ddd>
108	y coordinate	Y	<---- ---- dddd dddd>
110	x coordinate	X	<---- ---- dddd dddd>
112	A edge register	EDGEA	<---- ---- dddd dddd>
114	B edge register	EDGEB	<---- ---- dddd dddd>
126	H edge register	EDGEH	<---- ---- dddd dddd>

Table 5.2: V1 registers

```
v - returns status of vertical sync (read only)
h - returns status of horizontal sync (read only)
c - composite sync, {\em i.e.} logical OR of h and v (read only)
b - returns status of composite blanking (read only)
x - low during line-scan active data time (read only)
p - parallel output p
q - parallel output q
a - access speed
l - line-scan mode enable
r - CRTC register select
d - digitiser/wipe select
g - grab enable
s - square pixel enable
```

Parallel outputs p and q are buffered and routed off board *via* the VME-bus P2 connector. They are used to control lights, reject mechanisms etc.

The a bit is used to synchronise frame store accesses with the onboard CRTC bus logic. Normally the frame store host interface runs asynchronously, providing very fast access to the video memory. The 6845 requires slow synchronous access.

5.17.3 Wipe circuitry

The wipe register on V2 is an eight-bit latch whose outputs are paralleled with the outputs of the ADC. The CSR d bit selects between the ADC and the wipe register. If a grab sequence is performed with the wipe register active, the selected image plane is initialised to the constant value in the wipe register.

5.17.4 Line-scan interface

Line-scan cameras consist of a line of photoreceptor sites paralleled by an analogue shift register. Typically two signals are required to drive the sensor: an exposure clock X and a transport clock T. When the sensor receives an active X edge, the charge accumulated in the photoreceptors is transferred to the analogue shift register and clocked out under control of the T clock.

The sensitivity of typical devices is low compared to some vidicon tubes, which means that they cannot be run at video speeds. The line-scan interface on V1 attempts to emulate a true video signal as closely as possible by (a) providing

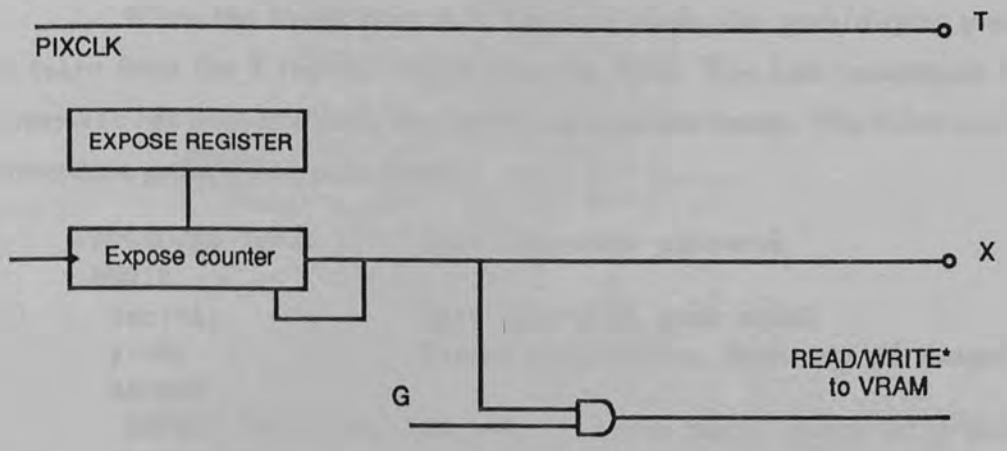


Figure 5.14: V1 line-scan interface

an exposure timer programmable in units of 64μs (*i.e.* one video line time) and (b) clocking data out of the sensor at video speeds, *i.e.* T is driven directly from the pixel master clock. The result of this is a 'video' signal composed of multiple blank lines followed by a single line of video speed data. There are no vertical sync pulses. Such a signal can be directly supplied to a video frame store for for digitisation. The host could then pack together the lines with valid data by copying within the frame buffer. However, V1 allows the host to modify the y address of the currently grabbed line in real time so that a packed line-scan image can be acquired with no time overhead.

Although the 6845 is designed to generate normal video timing, it was found that suitable programming of the internal registers could disable the vertical timing logic so that the device was providing continuous horizontal video lines with no vertical blanking or syncing. The grabber logic was modified so that during line-scan cycles data was only written to the image planes during the line after an X pulse was generated. During other lines, the same data is repeatedly read out and displayed. This gives a useful realtime representation on the monitor screen of the camera outputs.

In practice, the integration time of the sensor (*i.e.* the period of the X waveform) will be locked to the speed of the conveyer belt being viewed. If a line-scan pixel subtends p meters in the direction of belt travel, and the belt is

moving at q meters per second, then the X period must be p/q seconds. Typical exposure periods will be in the range 5–20ms for commercial speed conveyers.

When the frame store is in line-scan mode, the grab/display y address is taken from the Y register rather than the 6845. The host increments Y after every line has been grabbed, thus providing a packed image. The following Pascal procedure grabs a line-scan image:

```
PROCEDURE lgrab;      {get line-scan picture}
 BEGIN
  csr:=1;             {set line-scan grab mode}
  y:=0;               {start acquisition from top of image}
  REPEAT
   REPEAT UNTIL csr AND x=0;    {wait until start of X pulse}
   REPEAT UNTIL csr AND x<>0;   {wait until end of X pulse}
   y:=y+1              {increment line counter}
  UNTIL y=256
  csr:=f;             {set normal frozen video mode}
 END
```

The prototype V1 was used for the factory trial described in Chapter 9 and has since been in use for several years as a replacement for Cook's frame store which has now been decommissioned.

5.18 V2

The IMP system described in Chapter 9 is currently being developed to support high speed microcoded processors. The new system is downwards compatible with IMP and frame stores may be interchanged between the two systems. However, the philosophy of the system is different. IMP is a powerful system containing special purpose hardwired processors that are difficult to use. This is partly a result of its origins with low powered PDP-11s such as the single board PDP 11/21. This encouraged the use of hardware 'widgets' to make up for the low performance of the soft processors in the system.

The new system is intended mainly for use with VAX hosts and fast microcodable processors. The emphasis is on a 'soft' system comprising processors and large amounts of memory with as little special purpose hardware as possible. This is most noticable in the design of the framestore, where the display hardware has been cut to a minimum. Programmable processors on the bus will be able to transfer an entire image in less than one frame time, and so special displays (such as multi-windowed zoom displays) can be created in near real time using software.

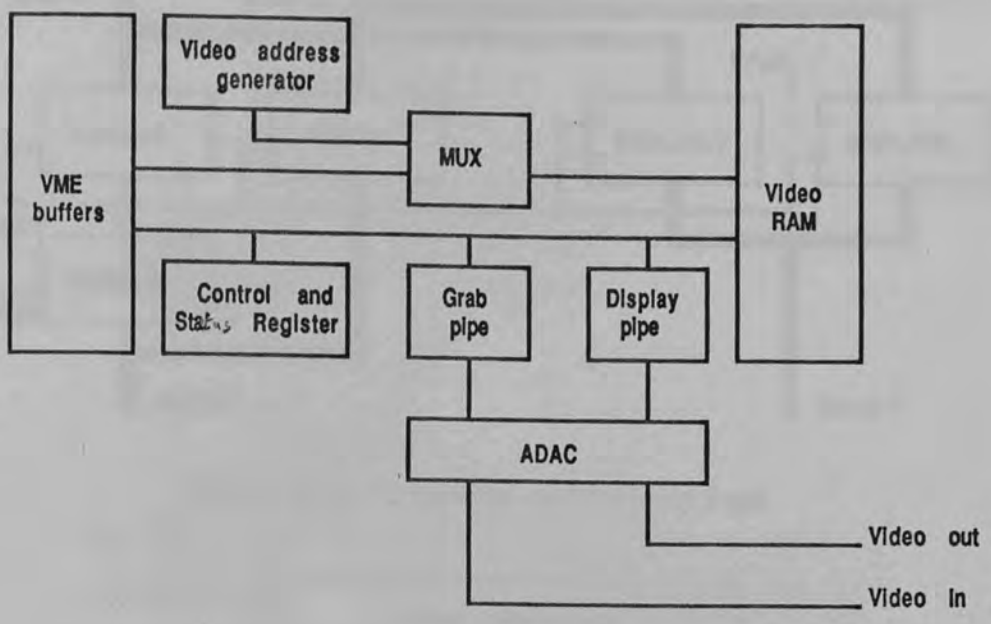


Figure 5.15: V2 block diagram

As a result, V2 is an extremely simple frame store. It provides up to eight 256 × 256 pixel image planes with square or rectangular pixels at eight-bit resolution. It will digitise RS170 video at eight-bit resolution. Host access is *via* a 0.5M byte memory mapped port. Image data is packed two pixels to the word, and so V2 appears to the host exactly like a 0.5M byte static RAM card. Frame store control is *via* a single byte control and status register. Multiple framestores are available in a single crate and the system software (see Chapter 6) allows dynamic allocation of resources in a multiuser environment.

5.18.1 V2 theory of operation

V2 has three main components as shown on Figure 5.15: the memory subsystem, the video timing generator and the host interface. Unlike IPOFS, the design is fully synchronous and makes extensive use of Field Programmable Logic Sequencers (FPLS), fuse programmable devices from Signetics [CD83] that allow powerful Mealy type state machines [PW87] to be implemented on one chip.

5.18.2 Memory subsystem

Although V2 generates 256 × 256 pixel displays, it was also intended as a testbed for the techniques required in a 512 × 512 frame store. As a result the memory subsystem has sufficient bandwidth to supply a new pixel to the display logic every 66ns. This is achieved by providing a 100ns cycle time memory array

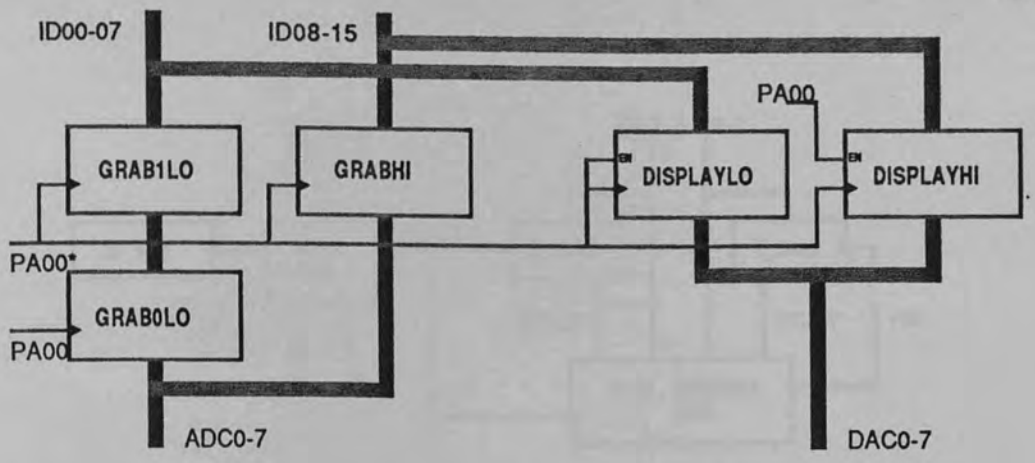


Figure 5.16: V2 pipeline and memory logic

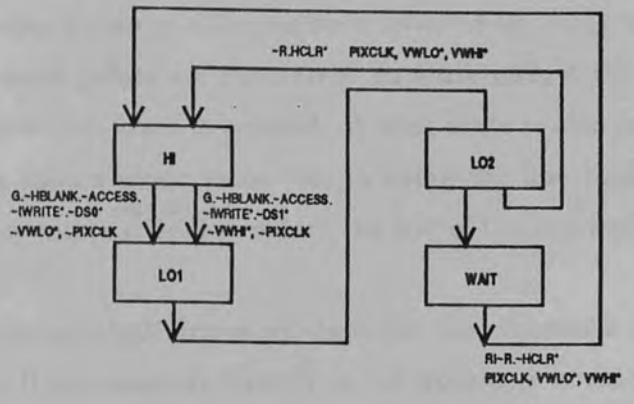


Figure 5.17: Pixel clock state machine

that is two pixels (*i.e.* 16 bits) wide, and merging the data streams using grab and display pipelines as shown in Figure 5.16.

The registers on the MD bus are updated every even pixel cycle, thus providing a double length memory cycle. In V2 square pixel mode the memory is cycled every 264ns. A full speed square pixel 512 × 512 system will cycle memory every 132ns.

5.18.3 Video timing subsystem

The master pixel clock is a selectable 176ns/132ns cycle clock generated from a 22.125MHz master oscillator using a simple state machine that monitors the state of the R and G bits in the CSR along with the status of the host interface and generates the pixel master clock and write pulses for the video RAM (see Figure 5.17).

This state machine is implemented in one third of a Signetics PLS105A FPLS, the rest of which contains the host interface controller. In state `HI`, if the

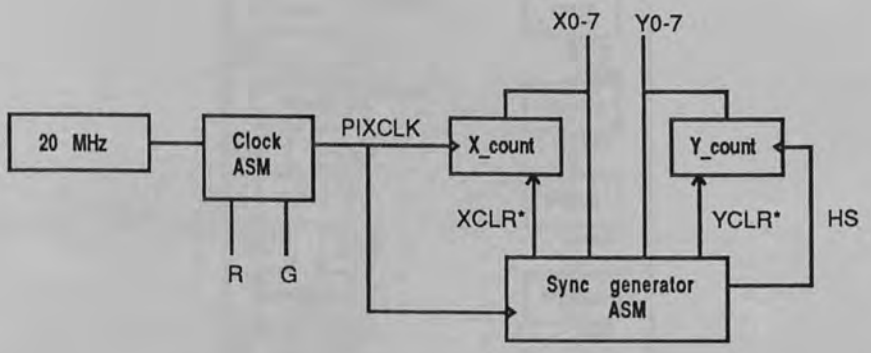


Figure 5.18: Video timing logic

host requires write access to either or both bytes of an image word or if grabbing is active then write pulses are generated. In state L02, if the R bit in the CSR is active an extra wait state is entered. A wait state is also inserted for the last two cycles of a square mode video line to bring the line time up to 64μs. The state machine monitors $\overline{\text{HCLR}}$ to detect the end of the line and inserts wait states accordingly.

The master clock drives an eight-bit counter which supplies the X display addresses. It corresponds directly to the `screen_x` counter described in Section 5.6. There is a second eight-bit counter driven by the horizontal sync pulse which corresponds to the `screen_y` counter. Sync pulses and clear pulses for the address counters are supplied by another FPLS which monitors the contents of the address counters.

Internally, the video FPLS contains two similar state machines, one for each axis. The display and sync start and end registers described in Section 5.6 are effectively embedded in the FPLS — the state machines wait for a preprogrammed number to appear on the address counter outputs and then transition accordingly causing a change in enable or sync outputs.

The X machine is complicated by the need for two sets of constants, one for square and one for rectangular pixel timing.

5.18.4 Host interface

The host interface consists of a set of buffers and an FPLS that performs address decoding and VMEbus protocol handling.

On V1, host accesses are handled completely asynchronously with respect to the video timing logic. This can cause a problem if the host attempts to access the image buffers during a frame grab sequence. The frame grab cycle

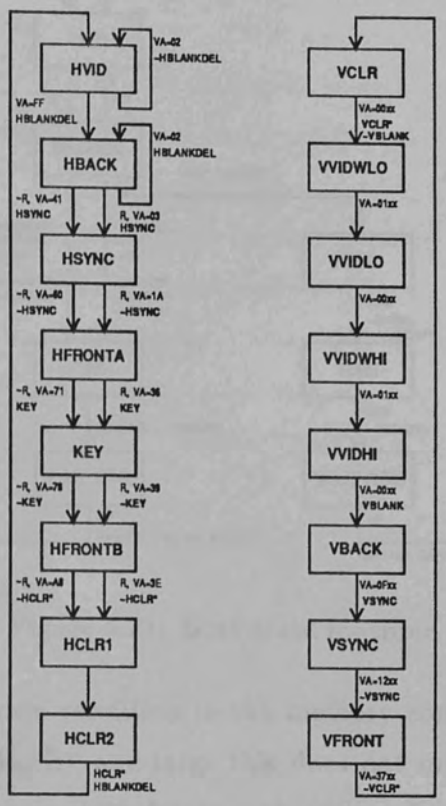


Figure 5.19: Video timing state machines

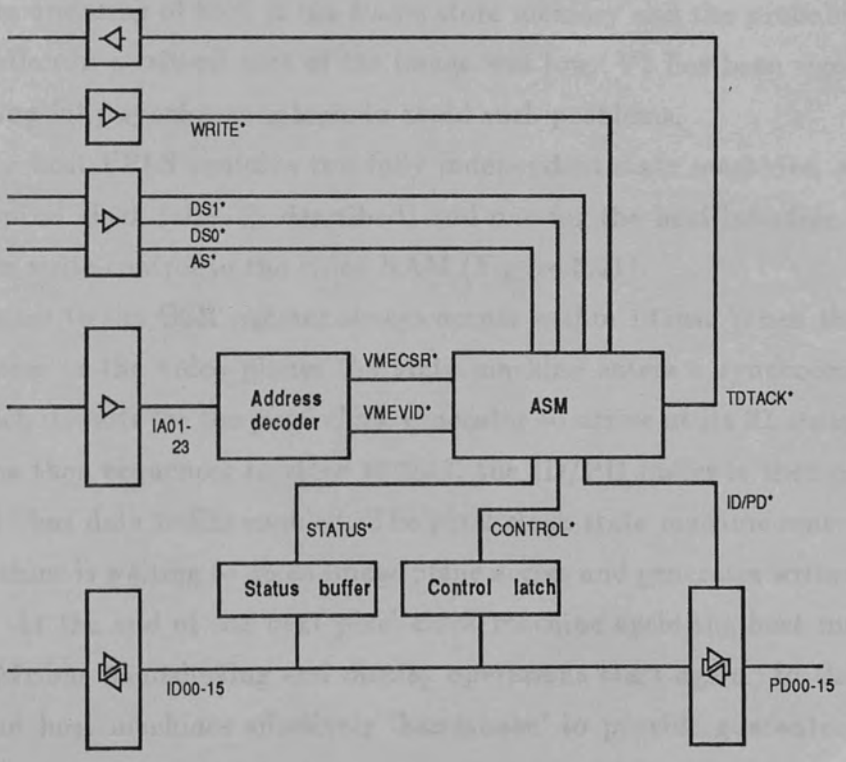


Figure 5.20: Host interface logic

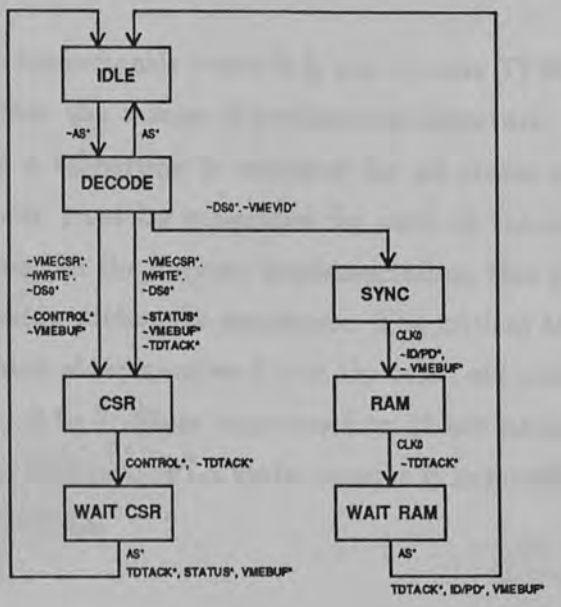


Figure 5.21: Host state machine

will be aborted, but a race condition in the memory controller could cause corruption of random pixels. By and large this does not matter since it is unusual to want to mix host accesses with frame grab cycles. However, the real time object detecting routine described in Chapter 9 does require such a facility. V1 was used successfully in spite of the race condition because the detection of an object triggered the updating of 85% of the frame store memory and the probability of pixel corruption in a critical part of the image was low. V2 has been rigorously designed using fully synchronous logic to avoid such problems.

The host FPLS contains two fully independent state machines, one for the master pixel clock (already described) and one for the host interface which also provides write control to the video RAM (Figure 5.21).

Access to the CSR register always occurs within 132ns. When the host requests access to the video planes the state machine enters a synchronisation state in which it waits for the pixel clock generator to arrive at its `HI` state. The host machine then sequences to state `ACCESS`, the ID/PD buffer is then opened and the VMEbus data buffer enabled. The pixel clock state machine senses that the host machine is waiting to do an image plane access and generates write pulses if required. At the end of the next pixel clock machine cycle the host machine performs VMEbus handshaking and display operations start again. In this way the pixel and host machines effectively 'handshake' to provide guaranteed safe timing.

The PLS105A was not an ideal device to implement these state machines. Specifically, the pixel clock machine has to sense when the host machine is in

state SYNC and more importantly when it is not in state SYNC. This is difficult because the PLS devices use a sum of products architecture. If a machine has n states $a_1 \ldots a_n$ and a transition is required for all states except (say) state a_1 then a product term must be generated for each of the other terms. This rapidly consumes terms. In the current implementation, this has been overcome by using don't care states within the sequencer. The critical ACCESS state of the host machine is allocated state number 8 and the other six states allocated to internal state numbers 2 to 7. State numbers 9 to 15 are unallocated, so simply by testing bit 2 of the internal FPLS state counter it is possible to see whether the host is in state ACCESS.

5.19 V3 enhancements to V2

V3, which is currently under development, is a stepwise refinement of V2. Most importantly it provides 512 × 512 timing. The video data pipelining necessary to support 512 × 512 timing has already been implemented on V2, so the primary difference lies in the sync pulse generation. In addition, V3 has an Inmos colour look up table to provide colour outputs, a ROM port of the type used on V1, a line-scan interface and a genlock input to allow the frame store to be sync locked to an external source.

Chapter 6

Architectural issues for sequential image processors

6.1 Introduction

In this chapter an attempt is made to isolate the particular characteristics of real time image processing that might influence the design of a processor system. The discussion proceeds at two levels — the hardware features that might improve raw throughput, and the software environment that allows non-specialists to extract the theoretical performance from the system. In many ways these two needs run counter to each other. Recent controversy in the Computer Science community concerning the merits of simplified processor design (the RISC philosophy) might be seen as a way both to improve performance and simplify programming, but it is shown that regardless of the merits or otherwise of RISC designs the key problem in image processing systems is processor-memory bandwidth. Many RISC designs have large internal register sets which may contribute more towards their performance than the high throughput of their simple instructions, but the data throughput requirements of image processing immediately remove the advantages of a large register set because it is not possible to hold a useful amount of an image in internal registers.

The most obvious architectural attribute of many image processing systems is their high degree of parallelism, both the SIMD parallelism of array processors and the MIMD parallelism of multiprocessor systems. This chapter concentrates solely on the attributes of sequential Von Neumann type machines, and discussion of parallelism along with non-Von Neumann architectures such as dataflow machines and systolic arrays is delayed until Chapter 7.

6.2 Hardware requirements for image processing

Traditional data processing can be characterised as either computationally bound or data bound. Typical scientific programming involves large amounts of real arithmetic calculation on relatively small data sets, sometimes smaller than the register set of the processor and almost always smaller than the virtual address space. Commercial programming typically involves collation and presentation of information from very large databases — well beyond the virtual address space of the processor. The database is however a relatively fixed dataset in that it is unlikely to change drastically within the execution time of a typical database operation. Calculation is more often integer or fixed point than real, and string operations may dominate arithmetic.

Although image processing is often characterised as computation intensive, its primary characteristics are the high volume and rapid turnover of the data sets. Individual images are much larger than the internal register sets of current computers but usually smaller than the virtual address space except in the case of outdated designs such as the PDP-11. In addition, images can be generated every 20ms, or in 20,000 instruction cycles of a typical 1 MIPS minicomputer. Since there will be 256K pixels in a 512×512 image there may be only 0.1 instruction cycles available in which to process each pixel, thus showing the futility of attempting real time implementations for high resolution processing on conventional machines.

It is important to note that the size of an image is far in excess of the internal storage of current processors, and that most low level vision operations are extremely pixel intensive, that is nearly all the data used is pixel data, either local to a particular point for window type operations, or globally for transform based operations. Transfer of pixels between frame buffer and processor will therefore dominate all processing and processor-memory bandwidth is likely to be the most accurate metric of system performance.

6.3 Processor design philosophy

Despite interest in dataflow architectures [GKW85] and distributed logic [PFP85] approaches to computing, the Von Neumann type architectures are still almost universally applied to real problems, either as straightforward sequential machines or in parallel configurations. It is possible to discern trends in Von

Neumann processor development, and in this section we examine their relevance to image processing.

6.4 Language directed machines

Most commercial mainframes, and latterly minis and micros, have evolved towards the use of more complex instruction sets in an attempt to narrow the 'semantic gap' between the high level languages and machine code, the intention being to implement high level language constructs directly in the instruction set. This trend has been most noticable in the area of procedure activation instructions. It is now generally accepted that modularisation and top down design of software increases programmer productivity, but increased use of procedures slows execution over inline code because of the need to pass parameters and store return addresses for subroutine linkage.

6.4.1 Procedure call instructions PDP-11, 68000 and VAX

Some PDP-11 processors implemented an instruction called MARK which was used instead of a ReTurn from Subroutine (RTS) to clean up the stack on exit from a procedure. The call convention required that R5 be stored on the stack, the stack pointer updated to leave space for parameters and the PC at the point of call be stored in R5. This was all achieved using standard instructions. The single instruction MARK N then restored the PC from R5, removed N parameters from the top of the stack and popped the old value of R5, thus restoring the calling procedure's context [Dig79b]. This instruction is only available on the PDP-11/34 and later machines, and has never been widely used. Even commercial compilers do not use the MARK instruction, and the T-11 microprocessor PDP-11 implementation does not recognise it.

The 68000 has a very similar instruction called UNLINK and a partner called LINK which can be used to initialise the stack frame as part of procedure entry. These instructions are used, no doubt because they were defined as part of the base processor's instruction set. Any commercial PDP-11 software that did use the MARK instruction would not run on early processors.

The VAX architecture continues this process by defining a single system-wide procedure calling convention. There are in fact two basic calling mechanisms implemented *via* the CALLS instruction for stack based parameter passing (*e.g.* for Pascal and Algol) and CALLG for use with global parameter blocks (*e.g.* for

FORTRAN). As well as the usual PC and stack pointer, the VAX designates two general purpose registers as the Frame pointer (FP) and the argument pointer (AP). Both the CALL instructions assume that the first word of a procedure is an entry mask which specifies which registers are used and therefore need to be saved on the stack. In addition both CALL instructions always save the old AP, FP and PC along with the current Processor Status Word, the entry mask itself (needed for stack clean up on exit) and a pointer to an exception handler that will be invoked if an error is detected during procedure execution. Finally the FP and AP are updated to reflect the position of the procedure's local data, and control is passed to the procedure's start address.

A particular feature of this system is the inclusion of the user specified exception handler allowing multiple exception handlers to exist at different levels of the program's execution tree. Exception handling on the VAX involves the 'unwinding' of successive stack frames until a handler is found that will service the current exception. Whilst this is an extremely powerful feature for real time failsafe systems, it might be considered overkill for a general purpose minicomputer.

6.4.2 Other complex VAX instructions

Non-primitive instructions on the VAX include (a) those for floating point and packed arithmetic, (b) direct calculation of array subscripts, (c) the extremely useful bit field manipulation instructions mentioned in the discussion of Algorithm 4 in Chapter 4, (d) case statement support, (e) queue instructions to directly support operating system structures and (f) character string instructions powerful enough to move a string in memory, translating on the fly and terminating if an out of range character is found. The increase in size of the instruction set has improved performance over that of the PDP -11: to quote from [Str78]:

> "First, despite the larger virtual address and instruction set support for more data types, compiler (and hand) generated code for VAX-11 is typically smaller than the equivalent PDP-11 code for algorithms operating on datatypes supported by the PDP-11. Second, of the 243 instructions in the instruction set, about 75 percent are generated by the VAX-11 FORTRAN compiler. Of the instructions not generated, most operate on data types not part of the FORTRAN language."

6.5 Big fast and simple machines

Large supercomputers such as the Cray series of machines are designed for extremely fast clock cycles. This requires the use of ECL and (in the future) Gallium Arsenide devices, which consume large amounts of chip area and power. These high speed technologies usually only offer SSI and MSI complexity parts. Seymour Cray, the principal designer of the Cray machines espouses a philosophy of 'big fast and simple' which rejects clever use of complex instructions in favour of simple high speed design and the application of large scale pipelining. Such *vector processors* are considered in more detail in Chapter 7.

6.6 Register and stack based machines

The traditional Von Neumann machine has a fixed set of internal registers each with a unique address, and a large main memory which is also arranged as a vector of uniquely addressed cells. Work on compilers, especially for the Algol descended languages emphasised the naturalness of the stack structure for expression evaluation and re-entrant procedure calling. Coupled with the need for reentrant and nestable interrupt service routines, this has forced almost every new commercial architecture since 1970 to have at least rudimentary hardware for stack implementation.

Some machines, notably the Burroughs mainframes, take the concept a step further and dispense with most visible internal registers, implementing a unified stack scheme that combines both internal registers and main memory into a single stack. No absolute addressing is required since all variables are stored relative to a Frame Pointer. Retrieval of non-local variables will require chaining back through a series of stack frames using the nested Frame Pointers. When the internal register based stack is full, the hardware automatically writes part of it to main memory. [Mye77]

Using a pure stack architecture in this way completely removes the register allocation problem which is possibly the major concern of compiler writers. Indeed, the low level language C even provides the user with a directive that may be used to warn that a particular variable will be frequently accessed and should therefore be put in a register if possible.

6.7 RISC machines

6.7.1 IBM

The first true Complex Instruction Set Computers (CISC) were probably the IBM 360 series of mainframes, that made extensive use of microcode to provide powerful machine code instructions. As hardware costs came down and software production became recognised as the real bottleneck to computer development, it was natural for more and more software functionality to be embedded directly in the hardware. The average instruction execution time may have risen, but throughput increased.

In 1975, a team at IBM was developing a fast controller for a large telephone exchange. Later the controller was developed into the IBM 801 machine (which was never sold commercially). During the development of the machine the design team had examined traces of executing programs on conventional machines and avoided implementing rarely used instructions. They used a fixed instruction format that allowed high speed decoding, and fast memory that allowed an instruction to be executed for every machine cycle. The lack of microcode allows straightforward pipelining techniques to be implemented and can significantly improve interrupt response.

6.7.2 Berkeley

The availability of VLSI CAD tools and fabrication facilities to academic researchers has given rise to most of the major RISC developments. The Berkeley RISC I and II machines were based on the IBM work. RISC I suffered from a design flaw and therefore did not meet design goals. The modified RISC II processor is a register based machine fabricated in 2 micron NMOS. It has 39 opcodes, a 32-bit virtual address space and supports 8, 16 and 32 bit datatypes.

A major feature of the processor is its overlapping window register architecture. RISC I has 138 registers of which only 32 are visible at any one time. The first ten registers are always visible and are used for global data storage. Ten other registers are available in each window for local storage. Another ten are referred to as 'overlapping' registers. Five of these contain parameters passed from the procedure above and five for parameters to be passed to the procedure below. These overlapping windows are automatically updated at procedure activation and return.

This arrangement has two important advantages over a typical processor

with 32 fixed registers. Firstly the large register set allows more data to be held local to the processor, significantly reducing main memory cycles. The register is faster than a local data cache. Secondly, the overlapping windows reduce the need to save registers in main memory at procedure entry, *i.e.* the calling procedure's context is saved using a register level type of memory management unit. This again has the effect of reducing main memory cycles. RISC I procedure activation typically requires $2\mu s$ against the $20\mu s$ required by a VAX 11/780. This is critical for RISC machines as the reduced instruction set implies more use of procedures to construct complex sequences out of primitive instructions.

Several workers have suggested that the excellent performance of the RISC I and II machines is directly attributable to this register arrangement.

> "The Berkeley group has responded by agreeing that a significant portion of the speed is due to the overlapped register window. However, the group notes that critics have ignored a key point in the design — that a drop in control logic due to the reduced set of instructions (from 50 percent to 6 percent) created space for the expanded number of registers in the first place." [Mar84]

In essence, their philosophy is to trade off microcode ROM against program available register storage. However this is clearly unnecessary since the RISC I and II chips are not high density designs. RISC I is a 44,500 transistor chip. Chips with 1,000,000 transistors on board are within the capabilities of current production technologies — the midrange T414 Transputer contains 150,000 devices — so there would be no problem in retaining the large control blocks needed for CISC designs and adding the novel large scale register sets.

The Berkeley team has also implemented a chip called SOAR (Smalltalk On A RISC) and a symbolic processor called SPUR (Symbolic Processing Using RISCS) designed to form the basis of a multiprocessor LISP workstation. SPUR comprises a three chip set (cache controller, CPU and FPU) fabricated in 2 micron CMOS. The CPU is similar to RISC II but with a 512 byte instruction cache, a four stage pipeline, a coprocessor interface and support for tagged data.

6.7.3 Stanford

A similar early project at Stanford resulted in the MIPS (Microprocessor without Interlocked Pipeline Stages) processor. The emphasis in MIPS is on pipelining and advanced compiler technology rather than large register banks. There is a five stage pipeline composed of instruction fetch, instruction decode,

operand decode, operand store/execute and operand fetch components. Each 32-bit instruction word may hold two instructions, and instructions typically require two cycles to execute. Pipelining complicates the handling of non-linear sections of code such as jumps, branches and procedure calls. A major effort in the Stanford project was to investigate the reordering of code generated by the compilers to improve instruction packing and to support 'delayed branching'.

6.7.4 Inmos Transputer

The Transputer is a stack based RISC. The T414 device contains 2K bytes of 50ns RAM which allows 10 MIPS operation and is also available for stack storage. The Transputer can operate at up to 10 MIPS from this internal RAM because of the 20MHz processor-memory bandwidth. This reinforces the assertion that availability of opcodes and data at high speed is the critical factor, and the existence of a simplified instruction set is merely a side effect of the need to reserve large parts of the chip for memory. Since the Transputer is stack based, the non-generalities of the Berkeley window register design, *i.e.* lack of overall storage space and inappropriate register partitioning, are not present.

6.7.5 Other commercial designs

The MIPS project has yielded a commercial machine (also called MIPS) which has recently been adopted by Digital Equipment Corp. for their work-station range. Other commercial RISC processors include the IBM RT/PC, the HP Spectrum range, the Inmos Transputer [Whi85], the Acorn Risc Machine and Sun Microsystems SPARC architecture. Texas Instruments and CDC are jointly involved in a DARPA project to produce a radiation-hard GaAs RISC processor with a clock speed of 200MHz.

6.8 RISC machine common features

The discussion of MIPS rate in Section 3.6 shows that a simple MIPS comparison across different architectures is dangerous, especially when the RISC machines are specifically designed with very high execution rates but instructions with very low semantic content.

It is difficult to be exact about what constitutes a RISC architecture but the following features seem typical:

1. No microcode — all instructions are hardware encoded

2. Load/store architecture — no memory to register or memory to memory operations other than data transfer

3. Large register sets — necessary to support efficient load/store architecture

4. Primitive addressing modes as a result of load/store architecture

5. Simple instruction decoding, often fixed field

There are exceptions to this list: the Inmos Transputer is a stack based architecture and the Stanford MIPS machines do not have especially large register sets but rely on large scale pipelining to improve throughput.

6.8.1 Single instruction per cycle execution

The overall aim of the RISC designer is to produce a machine that executes one instruction per main memory cycle. Removal of microcode and complex instructions enables one instruction per cycle operation to be obtained using simple design techniques, but pipelining, the use of large register sets and powerful code generation techniques in compilers can all be applied to complex machines so that they can also execute an instruction per cycle. The higher semantic content of a CISC instruction should then provide higher overall throughput.

6.8.2 Processor memory bandwidth

RISC machines can match the throughput of more complex processors at lower cost and with less design time required, so they may triumph for economic reasons. Ultimate high performance may be obtained by applying RISC type techniques (large register set, pipelining, reducing main memory accesses) to CISC machines. The limiting factor on performance will be the speed with which operands can be fetched from relatively slow main memory, *i.e.* the processor memory bandwidth. Although the large RISC I type register sets have been demonstrated as sufficient for a variety of applications [Rob87] in an image processing environment, data fetching will still dominate unless the processor has an internal register set two orders of magnitude larger than today's RISCs.

6.9 Software requirements for image processing

In her monumental work 'Programming Languages: History and Fundamentals' [Sam69] Jean Sammet mentions some 120 languages known to have been

implemented, and that was only at the end of 1967. She was aware of at least as many unimplemented proposals, and in the intervening period it is likely that the rate of production of new systems has significantly increased. There seems to be a new language inside every systems programer trying to get out. When, in 1975 the US Department of Defense conducted a survey of the programming languages in use for military applications, they counted over 400. At the time, they were spending over three billion dollars a year on software [Ich84]. It was this that spurred on the development of Ada.

There have been attempts to define 'doomsday' languages (eg. PL/1) which attempt to be all things to all people, not by implementing unified fundamental concepts, but by layering features from several language traditions with possible overlap of structures. In the case of PL/1 this resulted in a language so large and cumbersome that special subsets had to be defined so that the overworked programer could handle the complexity.

The major requirement for safe and efficient program design is discipline. This should be coupled with programming tools that are as simple as is commensurate with the job in hand.

Multiplication and overlap of concepts and their implementation within a programming language provides more possibilities for side effects, *i.e.* unexpected behaviour of one part of a program due to a breakdown in a completely different part. This gives symptoms that show no simple connection with their cause, and makes the program very difficult to debug.

6.10 Orthogonality

It is useful to distinguish between *feature oriented* and *unified* systems. Although the human is capable of storing prodigious amounts of information, programming systems that require access to many facts and concepts appear to overweigh the programmer so that even experts rapidly get to the point that they cannot remember the command for this or that operation. Although mnemonic or preferably English names help, it is desirable to minimise the amount of information required to use a system. The only way to minimise the load on the programmer without restricting the utility of the environment is to unify various features into a more general command with natural modifiers. [Hoa81] says:

> "(This) method is far more difficult. It demands the same skill, devotion, insight, and even inspiration as the discovery of the simple physical laws that underlie the complex phenomena of nature. It also

> requires a willingness to accept objectives which are limited by physical, logical, and technological constraints, and to accept a compromise when conflicting objectives cannot be met. No committee will ever do this until it is too late."

6.11 Languages and programming environments

In the not too distant past all programs were submitted in batch form and there was no direct interaction with the system. Any data had to be included after the program deck, and lineprinter output would be received back anything up to several days later. It is not hard to see why assembly language programming had such a long development cycle under such conditions. Much of the history of programming systems since then has concerned the improvement of the development cycle, both by speeding the edit-compile-run loop and by producing higher level languages in which to express the algorithm. This may be seen as the adaptation of the machine to the thinking patterns of the human, rather than the human having to accommodate to the machine. Such operating systems and high level languages exist to hide the inner workings of the processor from the human, preferably without introducing too many inefficiencies.

The second thrust of language development is the design of more powerful notations which actually help the programmer's thinking. The whole of mathematics may be viewed as the search for more elegant and compact notations for the description of problems, for without a notation there is no well defined problem statement.

> "By relieving the brain of all unnecessary work, a good notation sets it free to concentrate on more advanced problems, and in effect increases the mental power of the race. Before the introduction of the Arabic notation, multiplication was difficult, and the division even of integers called into play the highest mathematical faculties. Probably nothing in the modern world would have more astonished a Greek mathematician than to learn that ... a large proportion of the population of Western Europe could perform the operation of division for the largest numbers. This would have seemed to him a sheer impossibility. ... Our modern power of easy reckoning with decimal fractions is the almost miraculous result of the gradual discovery of a perfect notation." [Whi11]

6.12 Image processing with conventional languages

Image processing is extremely data intensive, and code often contains many small loops. Even if these characteristics were radically different from nor-

mal programming, the design of a new language would not be justified. Instead, standard techniques for solving those problems should be developed and built into existing languages, either in the form of a macro preprocessor or procedure calls. If already existing language features or philosophy block this 'stepwise refinement' approach then a new language may be called for, although this may only indicate that the base language was a bad choice.

6.13 PPL2

PPL2 is a special purpose image processing language developed in this laboratory [Coo83a] and syntactically based on Algol-68. Apart from predeclared variables representing the frame store registers, the only novel programming structure of the language was the use of a [[...]] construct to denote the application of a window operation to the image. In a conventional language this would be specified using two nested DO loops. The language was very small and did not contain the semantic richness of Algol-68. Integer only arithmetic was available, the procedure call mechanism was implemented *via* macro text substitution without parameter passing, variables were not predeclared with all the attendant insecurities this brings and source code was limited to 8K byte of text with no linking to other PPL2 programs or to routines written in other languages. There was no file handling, and screen I/O was rudimentary. The storage space available for variables was also limited but more seriously the register allocation algorithm used for arithmetic expression evaluation could cause run time failures due to lack of spare PDP-11 registers.

The language was interpreted apart from sections within a [[...]] construct which were compiled and then discarded at the completion of the scan. The [[...]] construct only supported top left to bottom right and bottom right to top left scans, which meant that routines requiring one of the eight possible scans (*e.g.* convex hull programs) had to be constructed out of DO loops anyway. Any programs written in the language had to be converted to a conventional high level language if a compiled speed implementation, or a portable one, was required.

One of the most serious problems with the language was that its implementation was dependent on Cook's frame store. Because of the unorthodox way in which this was interfaced to the host PDP-11, PPL2 had to directly manipulate the memory management registers, risking disastrous interaction with the operating system. As a result, a complete rewrite of the system would have been required even to move it on to a Q-bus based PDP-11 containing an IPOFS or

V1 frame store, which follow PDP-11 hardware configuration rules.

Of these deficiencies, the lack of a recognisable procedure calling mechanism was the most awkward. In spite of this, PPL2 was used extensively and successfully for rapid prototyping of image processing routines. A particular strength was the capability to experiment quickly on problems brought to the laboratory by industrial and other visitors. PPL2 was highly interactive, containing its own simple editor and run time system. In use it was far better than the conventional macro based languages available on commercial image processing systems at that time (1980–83), such as the BRSL Autoview Viking machine which is based on Batchelor's work. This is because in spite of its limitations PPL2 was a general programming language in which novel image processing algorithms could be expressed, whereas Autoview programs are constructed by concatenating predefined operations with consequent lack of generality and efficiency.

The case for a highly interactive programming system which has predefined knowledge of the available hardware is clear, even though PPL2 was an unsatisfactory implementation.

6.14 PIPE

PIPE (Pascal Image Processing Environment) is a programming environment designed and implemented by the author to replace the PPL system. Although it lacks the interactive features of PPL, it gains by making use of standard Pascal and conforming to operating system protocols. PIPE has been extensively used in this laboratory and was used to implement the project described in Chapter 9.

The non-standard hardware interface of Cook's frame store required programs written in conventional high level languages to directly manipulate the memory management unit. PIPE aims to hide these manipulations from the user, and to ease the transition from PPL programming to conventional large scale programming in Pascal. The Pascal compiler in use, Parallel Pascal [Uni81], includes the multitasking primitives of Modula [Wir77] which can be used in the programming of the IMP system described in Chapter 9.

PIPE comprises:

1. a modified Run Time Library which initialises the framestore and restores the operating system environment at the end of a run,

2. a set of Pascal source preludes containing definitions of hardware registers

and data types,

3. an object code library containing picture I/O routines and timer support,

4. a set of stand alone utility programs which may be run from the operating system command line,

5. a command file build utility called NEW. This is used with template files to generate an indirect command file that will edit, compile, link and run the current program.

The facilities of the Pascal compiler and operating system may be used to maintain object code libraries and to link with machine code and FORTRAN routines.

PIPE has been implemented for Cook's frame store, IPOFS and V1. The modified run time library is not required for IPOFS and V1 since these do not require memory management manipulations. Apart from the hardware programming differences noted in Chapter 5 PIPE provides full portability of source programs between each of these systems. All the algorithms in Chapter 4 were originally implemented on Cook's frame store and were ported to V1 simply by changing the coordinate register loop counters to take account of V1's eight-bit coordinate registers.

Although the NEW utility automates the various steps in the building of a PIPE program, the system lacks the interactive 'feel' of PPL because the compile-link stage can take several minutes for large programs. However, PIPE suffers none of the operational limitations of PPL. The [[. . .]] construct is available with the use of an optional preprocessor, but in practice this has not been used because users prefer to insert the nested loops directly rather than accept the compile time overhead of the preprocessor.

6.15 PIPE-32

The V2 and V3 framestores described in Chapter 5 are supported on the VAX with a software package called PIPE-32 [Joh88a]. This provides a full set of data types, framestore control routines, transfer routines and VMS utilities for the framestore user. Interlocked access to multiple framestores is supported, and programs are dynamically reconfigurable from the command line for different framestores.

6.16 A specification for interactive image processing

An interactive image processing language system based on a large subset of Pascal is under development. The PIPE system provides a useful environment for eventual implementation of algorithms but is slow and unwieldy for simple experiments and algorithm development. This is partly because the particular Pascal compiler in use is very slow (although it does produce efficient code) and partly because of the batch oriented edit-compile-run cycle.

Interactive programming has been available using the PPL2 system described above, but the lack of file handling, procedures, local variables, library facilities and real arithmetic coupled with the limited program and variable space and the fact that the entire PPL system must be resident for a program to run meant that serious applications were difficult or impossible to develop. Conversion of programs to a more normal system is not easy because of the Algol-68 derived syntax and especially the macro oriented calling convention.

What is required is an interactive system that is compatible with a full blown language compiler, so that if a problem becomes too large for the system, the programmer can transfer to the main compiler. A problem could be broken down into subproblems which may be individually programmed interactively and then gathered together by the main compiler. This is an attempt to provide the 'best of both worlds' without the need to develop a completely new language.

The system is called Pascal-I (for Pascal-Image processing) or PI and is conceived as a direct replacement for PPL. Syntactically it is a strict subset of ISO Pascal [Coo83b]. However the compiler recognises several system procedures and reserved variable names that are not in the standard Pascal symbol table. These extra variables and procedures may be thought of as being defined in a system prelude compiled in front of the user's program. PPL2 system utilities will be implemented as PI procedures, available at program level and interactively from the keyboard.

Interaction will be provided through the use of an incremental compiler and a screen editor outwardly similar to EDT, the VMS system editor. Programs will be compiled one procedure at a time and stored for later execution. When not running a program, PI will present the user with a list of known procedure names. Any procedure declared at the global level is displayable, along with the prewritten routines. The user may run any of these procedures by placing the cursor over the name and hitting a non cursor key, or by explicitly entering the

name in the system prompt area.

The present prototype is based on Wirth's Pascal-S [Wir81] with the code generator modified to produce PDP-11 and subset VAX-11 machine code. The system is being written in Pascal and is being developed on a MicroVax II. In future it is intended to add features to support parallel programming using the Modula-1 model, and features to support array processing for the NPL LAP machines [McC85] based on the Actus array processor language [Per79].

Chapter 7

Parallelism in hardware

> "... recollect that the multiplication of two numbers, consisting each of 20 figures, requires at the very utmost three minutes.
>
> ... when a long series of identical computations is to be performed, such as those required for the formation of numerical tables, the machine can be brought into play so as to give several results at the same time, which will greatly abridge the whole amount of the processes."
>
> General Menabrea on Babbage's Analytical Engine, 1842

7.1 Introduction

This chapter first reviews parallelism from a structural viewpoint, that is from a consideration of the underlying hardware. The next chapter looks at software constructs for the control of parallel machines.

7.2 Spatial parallelism within the CPU

All real computers exhibit some degree of parallelism. Usually this is in the form of multiple bit Arithmetic and Logic Units (ALUs) with associated multiple bit memories. The width of the ALU, memory and communication buses define the word size of the processor. However, parallelism exists (at least conceptually) at an even lower level within the ALU itself.

7.2.1 Bit level parallelism

In the limit the ALU may be only one bit wide, *i.e.* the processor is bit-serial. In the past some conventional commercial machines have been bit-serial for cost reasons such as the PDP-8S and its close cousin the Digico M16-S. Some specialised machines such as the PDP-14 [BM78], an industrial controller designed

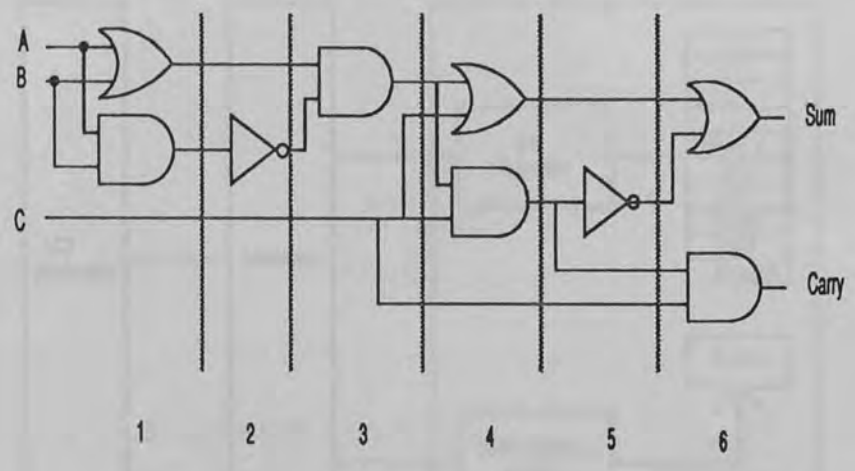


Figure 7.1: Parallelism within the full adder

to replace hardwired relay logic controllers in an industrial environment, also used bit-serial processors.

Within the one bit wide ALU a variety of configurations are possible. Typically a single full adder will be used, but it is clear that a single NAND gate and a controlling state machine would provide a general purpose processor since all logic equations may be implemented by a set of NAND gates or by feeding the terms through a single NAND gate with appropriate sequencing and storage of intermediate results. The use of a full adder provides parallelism (Figure 7.1).

The complexity of the ALU could be described in terms of NAND (or NOR) gate equivalents. This does not usually translate directly into a gate count of the actual device since most integrated circuit technologies provide logic primitives more complex than a single NAND gate. In TTL the multiple emitter transistors provide an n-input NAND primitive. ECL logic provides NOR and OR complementary outputs at the most basic level, and CMOS technology provides arbitrary logic functions using $2n$ transistors for a simple n-input gate, as well as 'trick' circuits such as the transmission XOR gate [SOA73] described in Chapter 10.

Most array processors (discussed below) use bit serial processing elements to reduce the hardware cost of the machine.

7.2.2 Word level parallelism

As noted previously, the word size of the processor's ALU provides some parallelism. The actual implementation of the ALU can also provide parallelism. In a simple ripple carry adder the bits are in a sense evaluated sequentially because

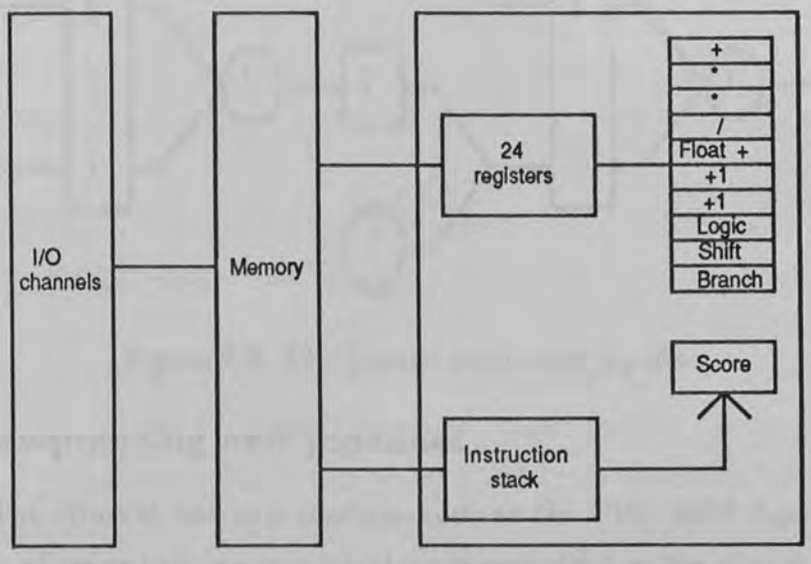


Figure 7.2: The CDC 6600 processor

a high order bit cannot settle until the carry has been produced from a low order bit. The carry path can be speeded up using various adder designs such as the carry lookahead adder [FP87], the carry select adder [FP87] and the two stage adder [WE85]. For VLSI implementation the dynamic Manchester carry adder [WE85] provides a fast MOS adder, and the binary lookahead carry adder uses a binary tree of adders to provide carry propagation times proportional to $\log_2$ of the adder size.

7.2.3 Multiple functional units

The processor may well contain other logic blocks to augment the typical ALU operations of integer add and subtract and bitwise logical operations. Typically these would include combinatorial multipliers, barrel shifters, and floating point units. Simple machines do not usually allow concurrent operation of these blocks due to problems associated with synchronising the available resources and saving the machine state efficiently in the event of an interrupt or exception.

In some computers, notably the CDC 6600 and 7600, multiple functional units are employed within the processor and may operate concurrently. The 6000 series machines used a *status checkboard* to indicate the availability of resources. If an instruction required a block that was already in use, the conflict was recorded in the checkboard and the instruction queued for deferred execution. Instructions could be deferred several times. This mechanism provided a hardware interlock system. The CDC 6600 includes ten functional units and 24 registers (Figure 7.2).

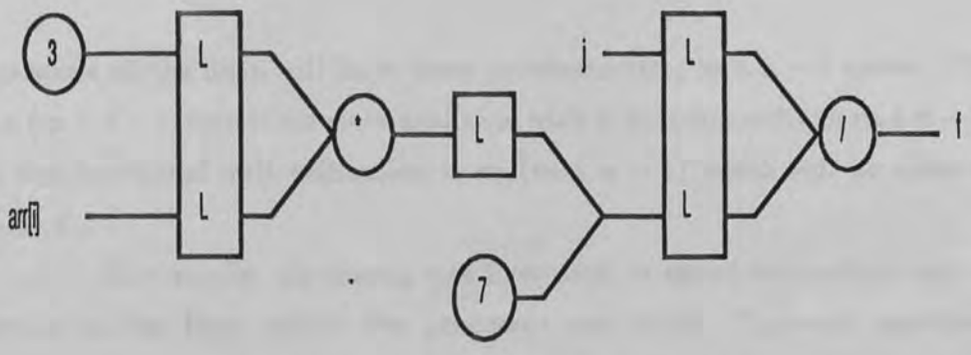


Figure 7.3: Three stage arithmetic pipeline

7.2.4 Sequencing and pipelines

The efficient use of a machine such as the CDC 6600 depends on the availability of many independent calculations embedded in the stream of machine code. A calculation obviously cannot be started until its operands are available. In the worst case, a sequence might require each functional unit in the processor in turn, with the operands at each stage of the calculation depending on the results of the previous one. In this case only one functional unit can be used at a time.

If however, such a sequence is to be repeated many times using different data then the functional units could be sequenced to act as a pipeline. Consider the following program fragment:

```
{1} FOR i:=1 TO max_arr DO
{2}  f:=(3*arr[i]
{3}      +7)
{4}      /j;
```

This could be implemented using a pipeline of 3 units with data latches between each stage:

With such an arrangement, lines 2–4 would execute in three cycles for a single datum. When `i` is 1, `arr[1]` is supplied to the inputs of the multiplier during cycle 1. The result `3*arr[1]` is supplied to the adder input during cycle 2, and without pipelining the multiplier would be idle. If, however, `arr[2]` is supplied to the multiplier during cycle 2 then multiplication of the second array element is overlapped with addition.

Clearly, if `max_arr` is greater than or equal than the number of stages in the pipe then there will be some time during the execution of the loop when all units are doing useful work. It is also clear that for an n-unit pipe there will be $n-1$ cycles at the beginning and $n-1$ at the end when the pipe is filling or draining during which some units will be idle. For an n stage pipe and m

operands all the data will have been processed after $m+n-1$ cycles. There are $n \times (m+n-1)$ operation slots available with n function units in $m+n-1$ cycles, so the functional unit utilisation is $m/(m+n-1)$ which will be close to 1 for $m \gg n$.

Historically, pipelining was first used to speed instruction and operand fetches rather than within the processor core itself. Typically pipeline stages implement instruction fetch, instruction decode (including address mode calculation), operand fetch, execution and storage of results. On multiple address architectures extra address calculation stages might be used. On linear sequences of code, the instruction pipeline can be kept filled, but any branching in the control flow will require the pipeline to be flushed and refilled. This results in loss of throughput. Modern compilers for heavily pipelined machines such as the Stanford MIPS RISC processor [Mar84] attempt to reorder the execution of loops so that calculations that do not modify the loop decision operands are brought to the end of the loop, and the branch decision is taken n cycles early where n is the number of cycles in the pipe. In this way what would have been wasted cycles during the pipe flush can perform useful work. This allows instruction pipes to approach 100% utilisation even in the presence of branching code.

7.3 Vector processors

Machines with arithmetic pipelines such as the CDC 7600 operate at peak efficiency when supplied with highly repetetive sequences of instructions that map well onto the pipeline. Vector machines such as the CDC STAR 100, its descendants the CYBER 203 and CYBER 205, the TIASC, the Cray series and the Fujitsu VP-200 provide explicit vector instructions with associated vector registers. A single vector instruction replaces a whole sequence of scalar instructions that need to be repetitively fetched from program store on the CDC 7600. The vector registers allow intermediate results to be held in the central processor rather than requiring main memory transfers.

The performance of vector processors is heavily dependent on the match between the vector and pipeline lengths. The performance of the Cray-1 can range from 2.5 to 153 Mflops [HJ81]. Arithmetic utilisation rises as the vector length approaches that of the pipeline and then halves as the vector length becomes 1+(*length of pipeline*) due to the load and flush phases. As a result throughput rates show discontinuities where the vector length is around a multiple of the pipeline length.

7.4 Parallel processor classification

7.4.1 Stream classification

The previous levels of parallelism have all been applicable to the execution of a single program on a single data stream. Flynn defined four categories of computers in terms of their instruction and data streams:

1. SISD — Single Instruction, Single Data stream computers such as the VAX,
2. SIMD — Single Instruction, Multiple Data stream computers such as the ICL DAP [Red73],
3. MIMD — Multiple Instruction, Multiple Data stream machines such as the IMP system to be described in Chapter 9,
4. MISD — Multiple Instruction, Single Data stream of which no practical examples exist.

Fountain [Fou87] claims that Flynn designated pipelined systems as MISD because multiple instructions are being applied to the same data items, but this seems no more valid than claiming that a non-pipelined system is MISD — each datum is only being operated by one functional unit at a time whereas true MISD implies that multiple instructions are operating on the same datum simultaneously. A dual ported cache memory might be interpreted as a true MISD 'processor' in that multiple comparators might be testing the same tag simultaneously, but the author is not aware of any real implementations of such a device. Indeed Hockney and Jesshope [HJ81] say that the class seems to be void, and Hwang and Briggs [HB85] say that "No real embodiment of this class exists". [HJ81] notes that the original paper by Flynn [Fly72] "states that it includes specialised streaming organisations using multiple instruction streams on a single sequence of data. However no examples are given". Most authors prefer to consider pipelining as being different in nature to the organisations described by Flynn's taxonomy. In this work the term *spatial parallelism* will be used to describe the topological layout of processors as described by Flynn's classification, and the term *temporal parallelism* to describe pipelining.

7.4.2 Functional unit classification

Flynn's elegant classification into streams which yields three useful classes is insufficiently detailed to do other than sketch the architecture of a real system.

Shore [Sho73] described six classes of machine based on their constituent parts and interconnection. Machine I corresponds to the conventional Von Neumann architecture. Machine II reads a bit slice from all words in memory and operates bit-serially rather than fetching data a word at a time. Array processors such as the ICL DAP and CLIP-4 fall into this category. Machine III is a combination of I and II with separate processors for words and bit slices. Machine IV has a single control unit but multiple processing units. However no inter-processor communication is provided except *via* the controller. A machine such as the ILLIAC-IV would be type IV if all inter-PE communication paths were disabled. Machine V is the same as machine IV but with inter-PE communication added. Machine VI corresponds to the associative processors that will be described below. They are characterised by having processor logic distributed throughout the memory.

Shore's classification is not much more detailed than Flynn's and lacks the elegance or mnemonic value of the class names. It does not address the issue of pipelining and does not really address the MIMD class. Indeed classes II to V are merely subdivisions of Flynn's SIMD class.

Hockney and Jesshope [HJ81] developed a rather baroque notation for writing detailed structural descriptions of the number of instruction, execution and memory units and the manner of their interconnection and control. Interconnections are represented using a notation analogous to that of a chemical formula. The Backus Naur Form [BBG*60] definition of their notation requires about two pages, and can be used to construct extremely complex expressions. The simplest example is that of a von-Neumann serial machine:

```
C=I[E-M]
```

This defines the computer `C` to be a single instruction processor `I` controlling the unit in the brackets, being a single execution unit `E` connected by a single data path to a single memory bank. The notation is like a combination of the structural notation used by chemists and the processor-memory-switch (PMS) notation of Bell and Newell [BN71]. The advantage of this notation over the other classifications described here is that pipelining is explicitly represented.

7.5 Array processors

An array processor is usually taken to be a system of the strict SIMD type, *i.e.* a single control unit (CU) broadcasting instructions to a set of processing elements (PEs) which operate in lockstep. The advantage of this arrangement is

that by definition all processors are synchronised so that there can be no memory contention or deadlock problems. An array processor could be thought of as a simple sequential machine that includes unusual data types, such as the bit slice vector or bit slice plane, amongst its base operands.

Some recent designs such as PASM [KS86] and CLIP-7 allow the array to be partitioned into a number of autonomous SIMD arrays. This is important, because in a conventional SIMD machine the only way of conditionally processing parts of the data set is to disable the processing elements corresponding to the unprocessed part. SIMD PEs are usually very simple bit-serial devices, and the throughput of the array is sustained only through the high degree of parallelism.

Image processing is one of the main applications for such machines. For an algorithm to execute efficiently on a SIMD processor it must (a) operate on data which is highly structured in a topology similar to that of the array itself, and (b) require only short range communication between processing elements. Small window operators such as the Sobel edge detector working on square arrays of pixels fit these requirements exactly.

Fountain identifies three active SIMD projects prior to 1975 — SOLOMON, the ILLIAC machines and the early CLIP machines. The basis for these was Unger's paper [Ung58] which described a computer oriented towards spatial problems. His abstract machine showed the classical SIMD features, namely broadcast of instructions from a single Control Unit to multiple lockstepped PEs, each of which had its own local memory and was connected in a two dimensional array.

The second generation of systems is characterised by machines such as the ICL DAP [Red73], CLIP 4 [FG80] and the Goodyear MPP [Bat80]. These are all large arrays of bit-serial PEs and have all been used extensively for image processing. They use mature technology and have been commercially successful. Other interesting systems include the GAPP chip from NCR [NCR84], GEC's GRID processor [RM82] and the MIT Connection Machine [Hil85].

A third generation of arrays that provide some local autonomy are currently being developed. These include CLIP 7 [Fou85] and the Purdue University machine PASM [KS86].

7.5.1 Solomon

In the early 1960's a machine called SOLOMON (Simultaneous Operation Linked Ordinal MOdular Network) was described by workers from the Westinghouse Corporation [SBM62,GM63]. Bit serial processing elements were

arranged in a four connected mesh with duplication of routing functions so that each input of the ALU could be connected independently to any of the available data sources. The maximum array size was to be 32 × 64, and each PE had 8K bits of storage (*i.e.* a total of 16M bits for a maximally configured array). The control unit would accept variable length bit-parallel instructions such as multiply and divide and generate the necessary bit-serial operations for broadcast to the array. It is not clear that SOLOMON was ever completed, and the machine does not appear to have been used for image processing. The design was clearly very ambitious, and this may have contributed to its demise.

7.5.2 ILLIAC III and IV

The ILLIAC III was a special purpose machine for analyzing bubble chamber photographs to aid the search for particle collisions. The machine described in [McC63] had an array of 32 × 32 special purpose PE's that could be eight or six-connected to their nearest neighbours. The eight neighbour inputs were routed *via* an OR gate to an eight-bit shift register. The shift register outputs were ANDed together and passed to the PE output for connection to neighbours. One end of the shift register was used for data I/O to the local memory which could act associatively. The prototype was damaged in a fire and the project abandoned in 1967.

ILLIAC IV was the first machine in the world capable of sustaining average execution rates in excess of 20 Megaflops. It was designed and built by the University of Illinois and the Burroughs corporation. Between 1972 and 1982 it formed one of the most powerful service nodes on Arpanet. The machine was an ambitious 'one off' which never reached its full specification but which nevertheless has influenced all later supercomputer designs and perhaps more importantly provided the test bed for several new ideas in programming languages and algorithm design.

The machine provided an array of 64 PEs arranged as an 8 × 8 matrix. The last PE in each row could be connected to the first in the next row so as to form a 64 × 1 array suitable for vector operations. Remarkably, the PEs were powerful 64-bit processors capable of 8, 32 and 64-bit arithmetic with on-board barrel shifters and floating point units, and 2K 64-bit words of local memory. This is in marked contrast to the simple bit-serial PEs used in most other array processors. When used in 32-bit mode, each PE could operate on two independent data elements, providing a 128 element array. Basic local memory access time

was 188ns, but contention between PEs for memory could raise this to 350ns. A Burroughs B6500 was used as a front end for compilation, loading and data transfer.

One of the strengths of the ILLIAC-IV was its backing store. A fixed multi-head 20M byte disk was used with an average access time of 20ms and a 500MHz transmission bandwidth. A good programmer could maintain a throughput of more than 7 million words per second, allowing the disk to be used as part of the primary store.

ILLIAC-IV was a very expensive machine. Development and construction cost around $40 million. Running costs were around $2 million per annum. One of the reasons for these high running costs was the absence of Error Detecting and Correcting (ECDC) circuitry. The entire system contained around 6 million components, and the MTBF was measured in hours, not days. Extensive diagnostics were run frequently and upon detection of an error the relevant PE or control module was unplugged and subjected to detailed test offline on a separate diagnostic machine. New modules were available to maintain operations. This lack of ECDC and the use of the (then) leading edge technologies such as 256-bit semiconductor memories must be seen as design weaknesses. Compare the long development time with the four year development cycle of the Cray-1 vector processor [HJ81] which used conservative technologies throughout and which has been a major commercial success.

As well as being a SIMD array processor the machine could demonstrate some concurrency. The master control unit was more like a small computer in its own right, and the order code for ILLIAC-IV comprised two sets of instructions, one for execution in the control unit and one for the array. Processing of these two types of instructions could proceed concurrently.

Several programming languages were developed for use on ILLIAC-IV such as Actus [Per79], Glypnir and CFD [Ste75] (an array processor FORTRAN).

7.5.3 CLIP 1–4

University College, London were also interested in detecting particle collisions in bubble chamber photographs and demonstrated a 20 × 20 array called UCPR1 [DJT67]. This summed and thresholded 3 × 3 and 5 × 5 neighbourhoods and could find sharp changes in line orientation.

CLIP 1 [Wat74] was a 10 × 10 four-connected mesh of very simple PEs with no local memory. The PE function set was limited to extraction of closed

loops of ones, extraction of ones connected to the image edge and extraction of the outer edge of blocks of solid ones. Input was from a flying spot scanner *via* a shift register. The outputs were displayed on an oscilloscope.

CLIP 2 [Fou87] which was completed in 1972 was a much more general system comprising a 12 × 16 six-connected array of PEs containing two one bit ALUs each capable of generating all 16 boolean functions of two variables. One processor provided the neighbourhood output which was fed to the six neighbours and the other provided an output which could be displayed or fed to one of two image memories for use by later processing. Although the processors were more general, there was no way of selecting data input from a particular direction since all neighbourhood inputs were routed *via* an OR gate.

CLIP 3 [DWFS73] replaced the OR gate input selector with gated direction inputs, so that the data source direction could be explicitly defined in the machine code. A 12 × 16 array was built which supported both six and eight-connected meshes. Each PE had 16 bits of local memory.

The early CLIPs were really only demonstration machines. Their application was limited due to the small array size. CLIP 4 [FG80] was designed as a 96 × 96 array of processors with functionality similar to that of CLIP 3. A full custom integrated circuit containing eight PEs using about 3000 gates was specified in 1974, but a full speed prototype array was not available until 1983 due to a series of problems with the chip fabrication. The CLIP 4 PE is similar to that of CLIP 3 with the addition of a carry bit to assist in grey scale arithmetic. The dual boolean processors were retained, and the local memory increased from 16 to 32 bits. The CLIP 3 analogue threshold gate was replaced by a simple OR gate since a digital threshold gate would have consumed as much silicon area as the rest of the processor.

CLIP 4 was available commercially from Stonefield Electronics and there have been sales to the US military.

7.5.4 CLIP 7

The CLIP 7 chip is designed to support varying degrees of local autonomy allowing some freedom from the confines of SIMD processing [Fou87]. It is a word parallel device containing a 16-bit ALU capable of performing addition, subtraction and all 16 bitwise functions of two variables, a 16 bit shift register to support logical and arithmetic shifts and logical rotates, four 16-bit registers, a 16-bit C register that may be used for local function control or as a data register

and an 8-bit D register for data I/O. Neighbourhood registers store data from the eight connected neighbours and communication is *via* bit serial connections so as to reduce the pinout of the device.

Apart from its word parallel structure, the main novelty of the CLIP 7 chip is its ability to locally modify the broadcast instruction stream using the contents of the C register. An external pin is used to set the operation mode as either global (in which case the C register is available for data storage and no control modification occurs) or local. In local mode the contents of the C register are used to control the following modifications:

1. disable PE — an activity bit can be set on carry, overflow, zero or sign and used to disable the PE.

2. ALU operation — the carry input can be either the previous carry output or the output of bit 2 (this is used for modulo 8 arithmetic which is required for direction calculations).

3. register address — two bits of the C register may be used to address one of the four data registers. This provides locally modifiable indexed addressing, although with such a small register set its usefulness must be limited in a single PE array. The prototype system uses two CLIP 7 chips per PE with one dedicated to local address generation.

4. connectivity control — eight bits of the C register are used to enable the eight-bit serial neighbourhood inputs. The bottom three lines can be used in grey scale mode to address one of the eight on-chip neighbourhood registers.

The chips are being used to construct CLIP 7A, a 256 × 1 array of PEs. Each PE uses two CLIP 7 chips, one to generate addresses for the 64K byte of local memory and the other to perform the pixel calculations. It is intended that CLIP 7A will be the first of a series of machines that will be used to investigate locally autonomous arrays and PE interconnectivity.

7.5.5 The ICL DAP

The DAP is an array processor add-on to the ICL 2900 range of processors. The production system, first delivered in 1980, consists of a 64 × 64 four-connected array. Each PE contains a full adder, a carry bit, an activity bit that may be used to disable PEs for conditional processing, a Q bit that latches the sum output of the full adder and 4k bit of local memory. The first description

of the system was in 1973 [Red73] and a prototype 32 × 32 array built from MSI TTL was available by 1976.

As well as the nearest neighbour connection network, the DAP has row and column buses which may be used to broadcast data and instructions to large numbers of PE's simultaneously. The host 2900 sees the DAP as a normal 2M byte memory array, and the row bus is used to access the local memory of 64 PEs at a time, appearing as a single 64-bit word fetch to the 2900. This facility helps overcome the inherent I/O problem of SIMD processor arrays.

The cost of a DAP in 1980 was about £500,000 on top of the cost of the host 2900, but the extra memory provided by the DAP would have cost a significant part of this figure anyway [HJ81]. A VLSI implementation of the DAP is now available from Active Memory Technology Ltd in the form of the MiniDAP add-on to Sun and Vax hosts. It uses a CMOS full custom chip. There is a bipolar version of the design (MILDAP) for military applications. The chips are intended for use in 32 × 32 arrays. There has also been a proposal for a bit slice word parallel processing element, but it is not clear if this design is going ahead.

7.5.6 The Goodyear MPP

The Massively Parallel Processor is designed to process LANDSAT-D satellite images at real time speeds. This requires a processing rate in excess of 10^9 operations per second [HJ81]. A clustering benchmark demonstrated by Goodyear has achieved processing rates about 1400 times faster than a VAX 11/780. The machine is a 128 × 128 four connected array of DAP like processing elements [Bat80]. There are no broadcast buses and data I/O is from the left edge to the right. During I/O the MPP PEs act as 128, 128-bit shift registers. There is a global broadcast facility. The major enhancement over the DAP PE is the addition of a programmable shift register (analogous to the Q-register in the 2900 bit slice element) which is used to improve multiplication time using Booth's algorithm. The execution frequency of the array is 10MHz (the DAP runs at 5MHz, CLIP 4 at 2.5MHz) and it can perform a 32-bit floating point multiply in 60μs.

MPP is implemented using full custom VLSI with eight processors per chip. Two external 4k bit RAM chips provide the 1k bit per PE storage. Since the RAM is off chip, improvements in commercial memory technology will allow increases in local storage. An interesting feature of the MPP is the provision of four additional redundant columns that can be switched in after a failure in one of

the normal columns. This should significantly increase the MTBF of the system.

7.5.7 The NCR GAPP chip

NCR has produced the first in a series of array PE chips, the Geometrical Array Parallel Processor (GAPP) [DT84,NCR84]. The processor is a simple full adder four-connected to its neighbours *via* a comprehensive switching network and to 128 bits of local on-chip memory. Connection to external memory is possible, but external cycles will be at least an order of magnitude slower than local accesses. The first GAPP chip integrates 72 of these processors in a 6×12 matrix. This very high level of integration is expected to increase to a 2048 PE chip by the early 1990's. 5MHz and 10MHz parts are available.

7.5.8 The GEC GRID

The GEC Rectangular Image and Data processor (GRID) resulted from collaboration between Southampton University and the GEC Hirst Research Centre [RM82]. It is intended to integrate 64 PEs (requiring some 50,000 transistors) onto a single chip which will run at 10MHz. The PE is essentially conventional with a single bit ALU and a block of local memory. The novel aspects of the GRID concern its connectivity features. Firstly the local RAM is dual ported, thus speeding two operand calculations. Secondly the individual PEs may be addressed using X-Y addressing to access a single PE and row/column broadcast as in the DAP. Thirdly, although the array is basically four connected, the carry bit can be routed to diagonal neighbours thus providing a limited eight connected pathway. Finally a histogram bit H is present in each PE wired across the chip as a shift register.

7.5.9 The MIT connection machine

The Connection Machine [Hil85] prototype contains 65536 PEs arranged as a 12 dimensional n-cube. The attractive property of the n-cube connection is that for a hypercube of dimensionality N each node is connected to N others and the maximum number of links between two nodes is N. Each chip contains 16 PEs, and strictly speaking the hypercube connectivity is only present at the chip level, *i.e.* between the 4096 chips (note that 4096 is 2^{12}). Within each chip 4×4 array of single bit PEs is connected *via* a message passing router chip. The router can accept four messages per cycle, and will direct the messages off chip into the

hypercube network if necessary. The machine is being commercially developed by the Thinking Machines Corporation.

7.6 Systolic arrays

The term systolic array was coined by Kung and associates at Carnegie-Mellon University [KL79]. In its purest and most generally understood form a systolic array comprises a two dimensional array of non-programmable processors acting like a two dimensional pipeline. The only control signal required is a clock. Data is fed in at the edges of the array and passes through being transformed *en route*. The technique is especially applicable to algorithms that require long range calculations in the result such as matrix multiplication. The array needs to be filled and flushed like any other pipeline, and as a result the technique is most useful when high speed repetitive calculations are required. The term systolic refers to the 'pumped' nature of the data pathways and is a reference to the pumping action of the heart.

Recently, workers have produced more and more complex systems with locally autonomous processors, inter-processor queues and local data loops. These systems, typified by WARP [Kun84], are not easily distinguishable from more general pipeline processors and it would perhaps be best to retain the term systolic for the simpler non-programmable devices.

Simple systolic arrays are very amenable to VLSI implementation. The primary constraints on any VLSI project are (a) transistor count, (b) pinout and (c) routing requirements. In a macroscopic design active circuit elements (transistors) are expensive and interconnect (wire) low cost. At the silicon level the two dimensional nature of the medium along with the large relative size of interlayer contacts means that wiring can consume a considerable area (typically 60% in a modern microprocessor [WE85]). Long distance wiring on the chip also slows the cycle time of the device. To propagate a signal across a 10^2mm die such as that used for the device described in Chapter 10 can take more than 10 transistor switch times. The simple systolic array can be constructed from small cells that butt together with no global wiring — a so-called tiling architecture. Systolic arrays are also prime candidates for implementation using Wafer Scale Integration (WSI) when that becomes technically feasible.

GEC at the Hirst Research Centre in association with the RSRE are producing a set of bit serial systolic arrays including a correlator and a convolver [MM82]. Workers at Purdue and Carnegie Mellon [Kun84] have shown how various

classes of algorithms may be automatically mapped onto systolic array architectures.

7.6.1 Warp and PSC

Recent systolic work at Carnegie Mellon has concentrated on two more general pipelined systems — the WARP pipeline processor and Programmable Systolic Chip (PSC).

Warp consists of a linear pipeline of 32-bit processors buffered by FIFO queues. Each PE contains a Weitek WTL1032 multiplier and a WTL1033 ALU [Wei84b,Wei83b,Wei83a]. These are 32-bit IEEE standard floating point chips which use a five stage internal pipeline to produce a new result every two clock cycles. External pipeline registers are connected *via* a crossbar switch to a 4k $\times$ 32-bit data RAM which can store intermediate results and lookup table coefficients.

The ten cell prototype is capable of 100MFLOPS. A 5 $\times$ 5 convolution kernel can be pipelined at the rate of one every microsecond, and a 512 $\times$ 512 image can be processed in about 250ms. The use of the Weitek units and the FIFOs to smooth data transfer make Warp a very high performance scientific processor which can perform pipelined complex FFTs at the rate of one every 615μs. However, the excellent (and expensive) floating point capability is probably overkill for most image processing applications.

The PSC device is intended to be used in two dimensional arrays that are much closer to the original systolic concept than in the case of Warp. Each chip contains an 8-bit ALU, an 8-bit input/16-bit output multiplier/accumulator, three 8-bit data input ports and three 8-bit data output ports [FKM*83]. The most intriguing feature of the chip is the provision of a 64 word writable microcode store and an associated stack based sequencer with a 64 word register block for data storage. This allows the processor to locally execute its own program, and therefore is much more general than the CLIP 7 chip which only effectively allows for local address modification and carry input control. In CLIP 7A all control flow is still centralised in the classical SIMD fashion, although future versions of the system may be partitionable in a manner similar to that of PASM (see below).

7.7 MIMD multiprocessors

It is difficult to present a systematic survey of MIMD systems because the available degrees of freedom provide a very large space of possible configurations.

In particular:

1. the processing elements are not usually limited to the simple bit-serial devices that array processors favour,

2. more general interconnection schemes are used,

3. varying degrees of coupling exist between the processing elements.

In an array the PEs are in lockstep, *i.e.* they all share the same instruction stream. This is the case even in the case of CLIP 7A where the instruction stream is only locally modifiable by address mode and ALU function. In a close coupled multiprocessor the processing elements are executing at similar rates and share some memory so that data may be exchanged synchronously and at high speed. Loose coupled multiprocessors provide (possibly asynchronous) communication channels between processors, and data interchange is by message passing modelled on the traditional I/O read and write operations.

In this section the early Carnegie Mellon multiprocessors C.mmp and CM* are described, followed by the image processing computers PICAP and PASM and finally a brief review of cone and pyramid architectures.

7.7.1 Communication networks for multiprocessors

The multiprocessor systems described so far have used provided local connections to four, eight or six neighbours. In some cases row joining is available (ILLIAC IV, GRID histogram network), other machines provide broadcast facilities at the row or column level (DAP and GRID) and in one case (the Connection Machine) a long range hypercube network is present to allow global interchange of data. Low level window based image processing operations fit such machines well. However, real vision problems require considerably more complex processing than that provided by window operators, and that this often takes the form of gathering together information from disparate parts of an image, in other words long range communication is required.

One way of providing such capability is to provide a conventional sequential host which can access the local memory of the PEs. The ICL DAP and NPL LAP 2 [McC85] adopt this approach. This imposes a hierarchy on the system and the programmer has two systems (supporting two completely different programming paradigms) to control. Some high level language compilers (such as ACTUS and DAP-FORTRAN which will be discussed below) provide a unified notation, but it is attractive to consider the design of a system without hierarchy where the

PEs can communicate over long distances and may be highly autonomous, *i.e.* MIMD systems in Flynn's taxonomy.

Considerable research effort has been expended on the investigation of communication networks for multiprocessors. For small numbers of processors it may be possible to provide complete interconnection such that every processor is connected to each of its neighbours. The cost of the interconnect rises as n^2 and the size of the processing element is also likely to rise as n for large n because the multiple interfaces will dominate over the processing circuitry itself. At the other extreme all processors may be connected to a single bus. This provides total interconnectivity with no routing overhead but will suffer from bus contention since only one processor-processor communication may be in progress at a time. A modern bus specification will however have a high bandwidth relative to the instruction execution frequency of conventional serial processors, so for algorithms requiring limited intercommunication the bus may be just as fast as a maximally connected network. The commercial ELXSI multiprocessor [Ste87] adopts this approach by interconnecting up to 12 fast processors over a 320M byte per second 64-bit databus.

In particular cases, either of the above extremes may provide the best compromise between performance and cost. However, for systems containing large numbers of processors (say > 10) neither is likely to be very satisfactory — on the one hand a bus will probably be reaching saturation, and on the other the cost of (say) 90 interprocessor buses may well be greater than the cost of the ten processors. A cost effective solution is likely to lie within the space of partially connected networks.

7.7.2 Logical and physical networks

The bus is logically equivalent to the maximally connected net in that there is a direct connection between all processors in the system. Segmenting the bus architecture into multiple buses implies the need for routers to connect processors on independent segments. Given that routers exist to connect all sub-buses together then the segmented bus is logically equivalent to a physical maximally connected net, but with a time overhead. If some parts of the processor matrix were isolated then some processors would be unable to intercommunicate.

If two bus structures are logically equivalent then they are able to simulate each other, but usually there will be a time penalty. Given a network comprising nodes that may process and route, the degree of a network is defined

as the number of links connected to each node and the diameter of the network is defined as the maximum number of links separating two nodes. For a given number of nodes, increasing the degree will reduce the number of steps required to route a message between processors on opposite sides of the network. Routing will absorb resources at the routing node and delay the propagation of messages. The bus is degree one, diameter one. A maximally connected net of n processors is degree $n-1$, diameter one. A hypercube of 2^n processors has degree n and diameter n and provides a good compromise between cost and routing overhead.

7.7.3 C.mmp

The C.mmp system [WC72], comprised 16 PDP 11/40E processors connected to 16 shared memory modules *via* a crossbar switch. The total physical address space was 32M bytes. The 11/40E processors were modified to make user execution of instructions such as HALT, RESET and RTI (return from interrupt) illegal, to allow bounds checking on the stack pointer, and to provide an extended writable control store. Backing store, in the form of four 40M byte disk drives, was attached to the Unibus of specific processors. A processor could not initiate I/O on a drive that was not connected to its own Unibus, *i.e.* peripherals were not shared.

As well as the processor-memory crossbar, an interprocessor bus was used to provide a common clock and common interprocessor control. The clock lines provided a 60-bit clock counter updated at 250kHz which was multiplexed onto a 16-bit datapathway and read into four sixteen bit registers on each processor. The operating system (called Hydra) made extensive use of this clock counter to generate unique names within the system. The top four bits of the most significant local clock register were set to the address of the processor. The interprocessor bus also provided interprocessor interrupts and control. Each processor could halt, interrupt, continue or start any processor, including itself.

The virtual address range of a PDP-11 is limited to 64K bytes, and this was one of the major limitations of the C.mmp. The already rather baroque memory management scheme of the 11/40 was made even more complex with the provision of a 25-bit shared address space accessed *via* the crossbar which automatically queued memory requests when contention occurred. It was planned to add caches to each of the processors, but this was not in fact implemented.

The basic intercommunication mechanism in Hydra was the channel — an I/O like link between two processes analogous to the mailbox in VMS and

the message queue (MQ) device in RT-11. This was implemented using the same protocols as for I/O processing, providing a unified programming environment for interprocessor and peripheral communication. Since this message passing protocol was inefficient for transferring large blocks of data, locks and semaphores were also provided to control access to shared memory.

7.7.4 CM*

CM* [FOR*78] is also constructed out of PDP-11's — in this case the LSI-11 chip set. Unlike C.mmp, the processor-memory structure is hierarchical, and as a result a large system with many processing elements could be practically considered. However, each processor can directly address all of the available main memory but not at uniformly fast speeds. The basic PE (called a Computer Module or CM) comprised an LSI-11 with local memory and I/O devices on a local Q-bus. Essentially the only difference between a CM and normal LSI-11 processor was the insertion of a switch between the processor and the Q-bus that could either route addresses to the Q-bus with relocation, or route to a map bus. Each map bus connected up to 14 CM's to a K map. This ensemble was referred to as a cluster, and multiple clusters could be used in a single system. The K maps were rather complex microcoded processors each comprising some 750 MSI chips on six cards. The reason for all this complexity was to enable CM* to simulate many architectures easily, and in particular to research the virtual addressing and memory protection requirements of multiprocessor systems.

CM* has been used for speech recognition [JCD*78], solution of partial differential equations by finite differences [Bau76] and to implement a subset of Algol 68 with extensions to allow concurrent execution of tasks and synchronisation. The Algol 68 system has been used to investigate the automatic decomposition of tasks.

7.7.5 PICAP I and II

PICAP-I and II [KDG82] are processors specifically designed for image processing at Linköping University.

A preliminary PICAP-I prototype was completed in 1973 but the final system was not ready until 1975. It was fundamentally a SIMD machine operating on 64 × 64 arrays of four-bit pixels. The 64 × 64 array was mappable within a TV frame containing 512 × 640 square pixels. Grey scale range and sampling density were programmable, so a full screen 64 × 64 image could be obtained or

a high resolution sub-image. Nine image planes were available.

The PICAP-I instruction set breaks down into four categories:

1. 3×3 convolution,
2. pointwise arithmetic between separate image planes,
3. template match,
4. I/O.

The convolution multiplies each element in a 3×3 window by a coefficient, sums the results and normalises by dividing by an appropriate power of 2 (*i.e.* by shifting the sum right).

The pointwise arithmetic instruction replaces the nine window element inputs of the convolver with pixels from each of the nine image planes and then performs the transformation as for convolution.

The template match employs an associative match unit to search through up to eight supplied templates. The templates are 3×3 arrays of integers, and during the associative search each template element is compared against its corresponding picture element. The less than, greater than, equal to and don't care relations are all available.

The above array operations may also be applied sequential to the image in a top-left to bottom-right scan.

Finally, a set of hardware counters are provided that (a) collect grey-level histogram data and give the maximum, minimum and average of the distribution, (b) count the number of hits on each of the eight templates and (c) track the maximum and minimum coordinates of template hits, thus providing the minimum rectangle enclosing all hits in an image.

PICAP-I has been used to inspect printed circuit boards, fingerprints and to detect malaria parasites.

PICAP-II is a bus based multiprocessor with a 40M byte/s bandwidth synchronous bus. Multiple memory modules are used with interleaving, *i.e.* memory words with consecutive addresses are located in different modules. A maximum of 4M byte of error-correcting memory is available. Four main processor blocks are available along with video I/O and graphics display:

1. filter processor, a generalisation of the convolution operator available in PICAP-I to neighbourhoods of any size,

2. logical processor, a generalisation of the PICAP-I template search instructions to allow eight-bit pixels and allow virtually any logical combination of neighbourhood pixels,

3. binary logical processor, a fast processor dedicated to logical operations on packed binary images,

4. measurement and region processor, which looks for connected regions and measures perimeter, extent and area for each of them. It can label each region uniquely and produce chain code of their outlines. Nested regions are handled correctly and connectivity graphs can be produced of prelabelled regions.

An Algol like language called PPL has been designed to program the system. It has an editor, incremental compiler to intermediate code and an interpreter which together provide an interactive program development system.

7.7.6 PASM

PASM (Partitionable SIMD/MIMD) [KS86], is a proposal for a large scale SIMD machine in which parts of the SIMD array (referred to as partitions) may conditionally execute from different control units. This is more general than the CLIP 7A design which allows conditional local modification of the instruction stream from a single control unit.

A PASM system comprises a Parallel Computation Unit (PCU), a set of microcontrollers which provide the instructions streams for the PCU, a set of backing store devices controlled by a memory management system and a host computer.

The PCU contains $N = 2^n$ processors, N memory modules and an interconnection network. The N PCU processors are controlled by $Q = 2^q$ microcontrollers (MC). Each MC controls N/Q processors which defines the size of a processor partition. Each partition has an independent control unit, so PASM may be considered a MIMD system where the processors are small SIMD arrays. PASM can simulate larger SIMD arrays by loading the same code into multiple partitions, *i.e.* into multiple control units. Possible values for N and Q are 1024 and 16 respectively.

Communication between processors is achieved *via* the switching network which allows memory to be shared amongst various processors. The memory modules are double buffered to allow simultaneous processor and file system

access.

7.8 Cone and pyramid architectures

It is clear that a (possibly ill-defined) hierarchy exists in the classical pattern recognition process. At the lowest level we have simple pixel level operations such as edge enhancements and smoothing that are applied right across an image. The results of such operations which do not differentiate between the parts of the image are then processed to detect local features such as corners and parts of edges, at which point large parts of the original image are discarded in favour of a higher level representation. Local features may be grouped into more global features and so on until the representation of the image is in terms of a specific classification.

Simple SIMD machines are useful at the lower levels, but become increasingly inefficient at the higher levels. A SIMD machine in which one processor handles each pixel will become progressively idle as more and more of the picture is discarded, and eventually a single SIMD PE will be operational. Since SIMD PE's are usually slow and simple devices, the machine will run slowly compared to a conventional sequential processor.

Several machines have been proposed and built which incorporate explicit hierarchy in an effort to smooth the transition from space parallel to sequential processing.

[Uhr78] described a recognition cone structure which comprised a series of 2-D data memories interconnected by transform layers. A transform layer took the data from the memory below, processed it in some way and passed it up to the memory above. The transform layer was conceptually a SIMD processor operating on all of the points in the layer below. At each stage resolution could be reduced, giving a cone of layers converging at the output terminal. This cone could be thought of as a pipeline of SIMD processors. The throughput of the cone would be limited by the slowest processing layer, as normal for a pipeline.

Hanson and Riseman [HR74] described a generalisation of Uhr's recognition cone called a processing cone. In this structure, the transform layers permit lateral data movement and data flow down the cone, in addition to movement up the cone.

Cone architectures call for multiple connections to each processing element in the transform layer because the resolution reduction properties require the use of 5×5 or larger input windows. A more restricted class of machines

called pyramid machines [TK80], [Dye82] restrict the local interconnections to either four or eight lateral neighbours, four sons and a father. They are believed to be as general as the processing cone, and are manifestly more economically feasible.

Pyramid machine may be used to map various data structure representations of n image onto hardware such as the quadtree, the roped quadtree [HS79b], overlapped quadtree, and colour, edge and texture pyramids [Lev80]. These representations have been used for region analysis, hierarchical searches and image compression for transmission.

A particular feature of pyramid machines is that they are able (due to their very regular architecture) to exploit VLSI technology, but they overcome the limitations of traditional SIMD arrays, *i.e.* lack of long distance connections and high degree of parallelism which is unsuitable for more sequential algorithms. Long range communication in a pyramid machine may be simulated by propagating up the tree and then back down to the working layer.

7.8.1 PCLIP

The PCLIP machine [Tan83] is a simulation of a 32×32 base pyramid running on a VAX processor. The simulated bit-serial PE's accept inputs from thirteen neighbours and process them using boolean pattern matching. There are three 1-bit registers per cell and a local memory of 128 bits. The Propagation Register is accessible to neighbours, the Condition Register is used as an activity flag, causing the PE to ignore all instructions except those causing a transfer of data into the condition register. The Target Register receives the result of a match instruction.

Match instructions compare a vector of 15 bits from the controller to the 13 neighbourhood inputs plus the local T and P bits. A second 15-bit vector (called the mask) is used to indicate don't care positions.

Algorithms have been described for pyramid formation, selection and segmentation and region colouring. In particular the region colouring algorithm requires $O[\log D]$ steps on the PCLIP and $O[D]$ steps on a conventional SIMD array such as CLIP-4.

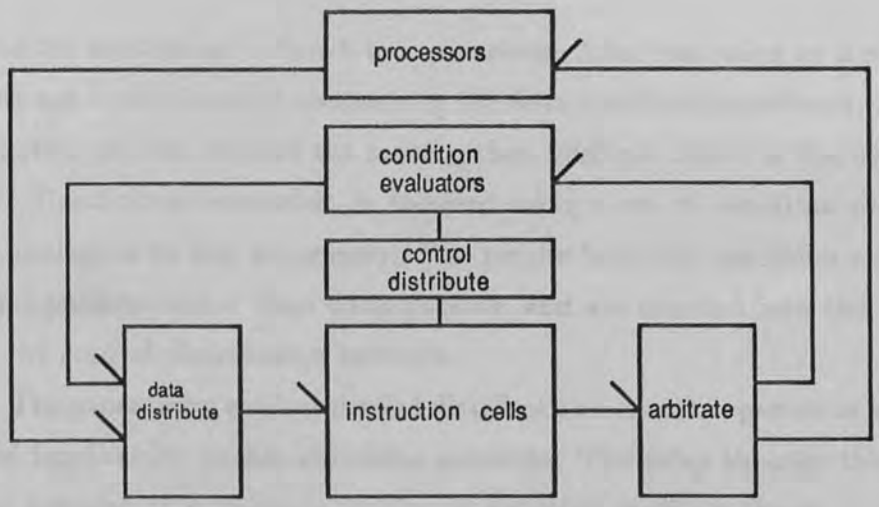


Figure 7.4: A dataflow machine

7.9 Data flow processors

Von Neumann architectures and the Non-Von Neumann machines described so far all belong to the class of control flow computers, in which the program has complete control over instruction sequencing. Data flow computers do not have explicit instruction streams. Instead, instruction packets are constructed containing an op-code and the 'address' of operands, and a central controller examines these packets. When the operands for an instruction become available, then the packet is routed to a functional unit which performs the computation. There is no shared memory, which mean that side effects resulting from assignment to global variables are eliminated. Data flow computation is purely functional. Operands are passed directly as *tokens* instead of addressed variables and dataflow computations have no far reaching effects, they are inherently local. Operation is inherently asynchronous, and this coupled with local computation and the underlying functional programming model make such machines very amenable to parallel implementation. In principle, the addition of extra functional units and routing paths will automatically allow the system to make use of the available parallelism within the algorithm.

A data flow computer comprises a ring connecting memory, processors, routers and token matching sections. The processors are fixed functional units performing the traditional arithmetic and logical functions. In this respect the processing section is reminiscent of the CDC 6600 CPU where multiple functional units are allocated to the instruction stream using a hardware interlock called the status checkboard.

When an instruction is enabled by the availability of its operands, it is

routed *via* the arbitration network to a processor. After execution by a processor the results are routed back to memory by the data distribution network. If multiple instruction packets require the results, then multiple copies of the data must be made. Conditional execution is handled using a set of condition evaluators that are analagous to the processors. The results from the condition evaluators are control packets rather than data packets, and are inserted into the memory units by the control distribution network.

The processors, evaluators and distribution networks operate as a pipeline connected together by packet switching networks. The delay through this packet switching network is a primary source of potential inefficiencies in a dataflow computer.

The critics of dataflow machines point out that normal programs generally have very low levels of implicit parallelism and that there is little real speed advantage to be gained on general applications. It may well be that the highly structured nature of low level image processing does lend itself to dataflow machines.

Another real problem with functional implementations in general is that the lack of shared memory requires replication of operands. In the case of large arrays (which are typical in image processing) this can cause very large amounts of processor time to be absorbed in array copying.

7.10 Graph reduction processors

Data-flow machines are data driven, and operation packets can be executed as soon as their operands become available. In a graph reduction machine, an operation is only evaluated when the result is required as input for some other operation. Graph reduction machines also have an underlying functional representation which requires the use of languages without side effects (*i.e.* functional languages).

Proponents of graph reduction machines claim that complex data structures are more easily supported and that replication of large data structures is reduced.

A graph reduction machine comprises a pool (memory unit) which holds the operation packets, and interconnection unit and a series of processors which individually are conventional Von Neumann processors. In operation the processors run asynchronously in parallel taking packets from the pool, processing them and returning the results to the pool. The overall architecture is demand driven

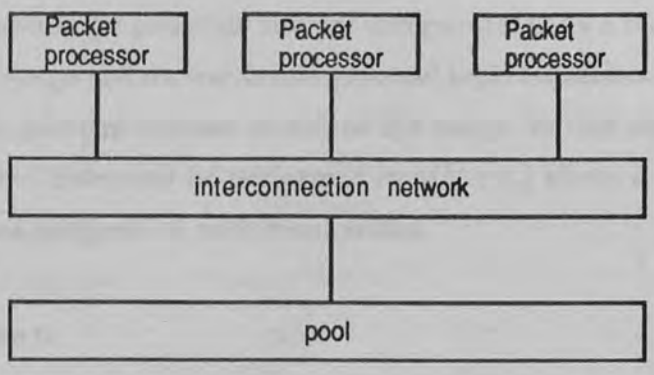


Figure 7.5: Graph reduction processor

by the ready status of the packets in the pool.

The overall block diagram is very similar to that of a general MIMD processor. It is the way in which the processors interact with each other (*via* the status fields of the packets, rather than *via* explicitly programmed shared variables) that characterise the machine as a graph reduction processor.

7.11 Associative processors

The ILLIAC III and PICAP processors incorporated associative capabilities. The term 'associative' is applied to memory systems with some inbuilt pattern matching capability. Conventional memories provide an array of unique locations that may be addressed by location, or coordinate. An associative, or content addressable memory provides an array of locations that may be addressed by content, that is a data word can be presented to the associative memory and it will return a list of locations containing that data word.

Conceptually each word in an associative memory chip comprises a conventional memory location n bits wide and an n-bit comparator which checks the contents of the word against the test data word. The outputs of all the comparators are available at the output of the chip, and a full search of the contents of the chip may be performed in time $t_r + t_c$, where t_r is the read access time of the memory and t_c is the propagation time of the comparator. Their most common application is in cache memories for conventional processors, and direct support in the form of special cache-tag RAMs is now available from some manufacturers [IDT86].

Large associative memories are prohibitively expensive because of the silicon area consumed by the comparators and the number of pins required to carry

the match information (in principle one per comparator). As a result comparators are assigned to a range of data words and external logic sequences the conventional address inputs to perform a linear search of the range. In this case an exhaustive search of the stored data may be performed in $n(t_r + t_c)$ where n is the size of the range of addresses assigned to each comparator.

7.11.1 Staran

STARAN [Pot82] is a SIMD array of bit serial PE's each of which contains an X register, a Y register and a 16 function boolean processor. An activity bit called the M register allows conditional execution of code sections. The local memory is 256 bits wide.

STARAN can have up to 32 arrays of 256 processors each. The arrays and PE's are assigned a unique number, so that every processor is location addressable.

In operation, the PE's perform local searching (*i.e.* bit-serial comparisons) in their local data space. The results are loaded into the Y register. An external resolver then associatively returns the address of the first PE by ordinal number of array and PE to be generated in an internal register of the common control unit, and this information can be used to access the PE for further processing. Essentially this performs an associative parallel to serial conversion.

7.12 Hardware summary

Image processing architectures have been dominated by the use of geometric arrays and even MIMD type systems show SIMD features. PASM is an array of SIMD machines. The pyramid machines are pipelines of SIMD processors running at successively lower resolution. The use of associative processors and techniques is of great importance in database applications and in pattern recognition proper, but is of marginal interest in image processing. More general MIMD systems are difficult to program and may be unable to exploit the available geometric parallelism implicit in low level image processing.

The particular problem in image processing is that the simple nature of the computation coupled with the very high data rates and tight coupling between adjacent parts of the image yield a computational problem that is data, not computation, dominated. Pixel bandwidth rates dominate over compute operations. A 3×3 convolution requires nine pixel fetches and only simple accumulation

within the ALU since no intermediate terms are generated. Even the Sobel only requires one intermediate result to be held. This heavy dependence on global memory accesses rather than internal temporary register accesses means that the processor cycle will be limited to the pixel read cycle. MIMD machines usually rely on loose coupling between processors (*i.e.* low communication rates relative to computation) and this makes them unsuitable for low level image processing.

On the other hand, it is clear that the simple processors used in arrays are unsuitable for higher level algorithms, and that the performance of simple processor arrays drops sharply as the degree of geometric parallelism is reduced. Machines such as PASM and CLIP 7A attempt to remedy this by partitioning by providing sophisticated conditional execution capabilities within the array but a more useful solution might be a MIMD system, one or two processors of which were themselves SIMD image processors.

Chapter 8

Parallelism in software

> "If we wish to succeed in building large concurrent programs which are reliable, we must use programming languages that are so well structured that a compiler can catch most time-dependent errors (because nobody else can)." [BH77]

8.1 Introduction

It is not currently (and may never be) practical to design programming systems capable of catching all parallelism related errors, but a variety of programming constructs have been proposed and implemented with a view to reducing the error rate in concurrent programs. This is analagous to the development of structured programming constructs in sequential languages.

This section reviews the fundamental problems of concurrency, and then describes the *semaphore*, *monitor* and *rendezvous* constructs. Much of the discussion involves generalised MIMD type processors. Array processors are by definition synchronised and therefore many interesting concurrency problems do not arise. Of course, the penalty for this is the loss of generality leading to low efficiency (in terms of actual, as a fraction of potential throughput) when array processors are applied to problems that do not exhibit a matching space parallelism. The chapter concludes with a discussion of Very Long Instruction Word (VLIW) machines that present a compromise between array and full MIMD systems. A special compilation technique useful on VLIW and pipelined RISCs known as trace scheduling holds the promise of being able to extract high levels of parallelism from ordinary sequential scientific programs.

8.2 The concurrency problem

On a multiprocessor system, or a timesliced uniprocessor, multiple tasks or processes may be running simultaneously. Strictly speaking, the uniprocessor runs processes sequentially, switching contexts under the control of a central scheduler or by programmed transfer between jobs. Since the scheduler is driven by an external clock, the time slicing behaviour of the system is effectively indeterminate, and it is convenient to think of the system as a set of tasks running on their own processors.

If two tasks share no data (or other resources such as tape drives or frame stores) they are said to be *disjoint*. Execution of the tasks may progress with no synchronisation between them and at no point is the correctness of the system reliant on concurrent properties of the programs. The correctness of the system may be shown by showing the correctness of the two tasks in isolation.

In general, however, tasks need to cooperate. This may involve disciplined access to a peripheral such as a frame store, updating of a common data base, or transmission of information between processes. All of these activities require synchronisation, and optionally data transfer. If two tasks are in a producer-consumer relationship then it is necessary for the consumer to wait until the producer has generated the next data frame. In a shared database environment, it is reasonable for multiple processes to be reading the database simultaneously, but only one process may write at a time. In addition, whilst a writer is active, all readers must be blocked. Failure to ensure this may result in a reader reading data which is being updated.

8.2.1 Mutual exclusion

The most fundamental problem in concurrency is that of mutual exclusion to shared data. This includes access to other shared resources such as peripherals.

Two processes **P1** and **P2** access shared data. Only one process at a time is allowed to access the data. If the other process attempts access, then it is *blocked*, that is execution is suspended pending release of the data.

The part of the process in which the shared data is accessed is called the *critical section* of the data. A protocol is required that allows a process to request entry to its critical section and to signal successful completion of the critical section.

A solution to the mutual exclusion problem must not only ensure that

the two processes do not clash, it must ensure that *deadlock* cannot occur (that is both processes wait endlessly for the other before continuing. It is also desirable that the solution be *fair*, that is a process wishing to enter its critical section will eventually be allowed to do so. A synchronisation scheme is fair if no process is delayed indefinitely waiting on a condition that happens infinitely often.

Fairness is especially difficult to ensure when several processes are cooperating. Two processes may 'conspire' together to lock a third process out of a critical section by passing control between themselves. In reality, it is often necessary to use protocols that are deliberately unfair, *e.g.* prioritisation of interrupt requests. In this case, a rapidly interrupting process at a high priority will block out low priority tasks.

8.3 Historical overview

Concurrent programming problems first arose in the design of multitasking operating systems with batch and spooling facilities. Interactive multitasking and user written interrupt service routines for real time programming were supported using operating system calls and a monolithic executive that performs all time critical operations. The FORK and JOIN programming constructs [Ben82] are implemented in many operating systems such as RT-11 [Dig82] and Unix.

The need for structure in sequential programming gave rise to a family of *structured* programming constructs, typified by the D-structures that are single entry, multiple exit control structures. In concurrent programming, the time dimension requires special structuring constructs. FORK and JOIN are analagous to the GOTO in that they allow completely random creation and destruction of parallel control flows. The COBEGIN-COEND construct allows controlled execution and destruction of processes. Synchronisation of processes may be achieved using shared memory variables, although the solution to the mutual exclusion problem using shared variables is non-trivial. As a result, a variety of synchronisation primitives have been designed. These fall into two classes: shared memory constructs which rely on the existence of high speed memory shared between cooperating processes, and message passing constructs which reflect a distributed view of multiprocessor systems where common memory may not be available.

The semaphore [Dij68] is both powerful and elementary, and has become the standard by which other synchronisation primitives are judged. The solution to the mutual exclusion problem is trivial with semaphores. Many languages including Algol-68 , Modula and Parallel Pascal implement semaphores.

The operating system concept of a centralised executive controlling all time critical operations has been generalised by Hoare into the monitor programming construct. A monitor is a package of sharable data and the routines required to manipulate it. Mutual exclusion is provided by ensuring that only one process may enter the monitor (*i.e.* be executing a monitor procedure) at a time. Monitors may use semaphores internally and may have synchronisation code distributed within them. Monitors have been implemented in many languages including Concurrent Pascal [BH77], Pascal-Plus and Modula-2. Further structuring of the monitor concept yields the path structure [CH74], a monitor like construct in which all concurrency constraints are defined in one place. Paths have been implemented in Path Pascal.

Recently the message passing paradigm has become increasingly important both theoretically and practically due to the arrival of networked asynchronous computer systems such as workstation networks and transputer systems. The most important message passing construct is the rendezvous which provides synchronisation and (optional) transfer of information between processes. The rendezvous forms the basis of Hoare's Communicating Sequential Processes (CSP) [Hoa85] and has been implemented in its most pure form in Occam, and in an extended form using remote procedure call semantics in Ada.

8.4 Coroutines

On a uniprocessor control is traditionally transferred using programmed jumps, subroutine calls and interrupts, which may be viewed as externally triggered prioritised subroutines. The subroutine has an implied hierarchy built in which does not match the semantics of concurrent execution even though interrupt service routines are a prime example of concurrent programming. In operation, the contents of the program counter at the point of subroutine call are stored and control is transferred to the subroutine. At completion, the old value of the PC is restored. This allows a subroutine to be coded without knowledge of the return jump address. However, the subroutine must run to completion before control is returned to the main line routine and in that sense the subroutine is a slave to the mainline routine.

The coroutine generalises this mechanism to provide a non-hierarchical call-and-return mechanism. Each coroutine has a local Program Counter. The statement TRANSFER(ROUTINE) suspends execution of the current routine and restarts execution of ROUTINE at the point of last suspension.

Coroutines automatically provide mutual exclusion between routines. There is no preemptive suspension of routines, so each routine is guaranteed sole access to shared data structures until it explicitly executes a TRANSFER() to a named routine.

However, this deterministic switch in control severely limits the usefulness of a pure coroutine implementation. Consider the classical bounded buffer problem. A buffer is required between two processes — the consumer and the producer. The producer inserts items into the buffer, and the consumer removes them for processing. The whole point of the buffer is that the processes should be decoupled — the only time a consumer should block is if the buffer is empty, and the only time a producer should block is if the buffer is full. With coroutines, the processes are constrained to taking strict turns to access the buffer and so the processes are not decoupled.

It is possible to use coroutines to construct queueing and ring scheduling mechanisms [Wir83] so that for a uniprocessor coroutines can simulate semaphores. However, the semantics of coroutines do not adequately describe the operation of a true multiprocessor since by definition only one coroutine can be executing at a time.

8.5 The common memory arbiter and busy waiting

Processes may synchronize by setting and unsetting flags in common memory. Inherent in such mechanisms is some idea of a hardware interlock called a *common memory arbiter* that ensures that the results of two processes writing to the same location simultaneously are consistent with the result of them writing sequentially. This means that if **P1** attempts to write a 1 to a location and **P2** attempts to write a 0 then the result should be either a 1 or a 0, not some corrupted value such as -1. This is ensured by the underlying hardware. The execution order of the processes access is unknown.

It turns out that correct solutions to the mutual exclusion problem using simple access to shared variables is non-trivial. The most well known solution is Dekker's algorithm [Ben82] which requires a flag variable for each process and an arbitration variable. It has been generalised for N processes by Dijkstra.

A simpler solution is provided by Peterson's algorithm [Ben82]. Lamport's algorithms [Lam74] have the useful property that the flag variables need

only be written by one process which may reduce memory contention time.

The common feature of all these algorithms is that a blocked process cycles waiting for a shared variable to change state. This is referred to as *busy-waiting*, or *spinning*, and naturally consumes processor resources. In an environment where each process has its own processor this still implies a loss of throughput because busy waiting will consume shared memory bandwidth. The semaphore and monitor constructs implement queueing of blocked processes which does not consume processor cycles unnecessarily.

As well as consuming processor cycles by busy-waiting, these solutions are difficult to use in practice because of the complexity of their working and because there is not necessarily a clear distinction between process variables for synchronisation and normal data storage. Hence busy-waiting algorithms are unstructured both in terms of control and data structures.

8.6 Semaphores

A semaphore is a variable that can take positive integer values and zero. The only operations allowed on a semaphore are initialisation, wait(s) and signal(s) [Dij68]

The operations are defined as follows:

wait(s) `IF s>0 THEN s:=s-1 ELSE (suspend calling process)`

signal(s) `IF (a process P has been suspended by a previous wait on this semaphore) THEN (resume(P)) ELSE s:=s+1`

In use a semaphore is associated with each synchronisation condition. Typically one semaphore is required per critical section. Mutual exclusion with semaphores is trivial. The following example demonstrates mutual exclusion between two processes A and B with critical sections critical_A and critical_B:

```
s: SEMAPHORE;
PROCEDURE A;
 BEGIN
  REPEAT
   wait(s);
   critical_A;
   signal(s);
   { other non-critical code }
  UNTIL false;
 END;

PROCEDURE B;
 BEGIN
```

```
    REPEAT
     wait(s);
     critical_B;
     signal(s);
     { other non-critical code }
    UNTIL false;
   END;
BEGIN
 s:=1; {initialisation of semaphore}
 COBEGIN
  A; B;
 COEND
END.
```

The synchronisation details are hidden by the implementation, freeing the programmer to think about the higher level aspects of the problem.

wait and **signal** are primitive (uninterruptible) operations and therefore exclude each other. Many computers such as the IBM 360/370 and the 68000 have uninterruptible *test-and-set* instructions that can perform semaphore implementations and directly ensure mutual exclusion of semaphore variable manipulations. More advanced machines such as the VAX have uninterruptible *enqueue* and *dequeue* instructions. Test and set instructions can handle the mechanics of manipulating the semaphore variable correctly but offer little assistance in the implementation of suspend/resume mechanisms. The description of signal and wait above has deliberately been couched in terms of process suspension. A more traditional definition of the semaphore operations is

wait(s) `REPEAT UNTIL s>0; s:=s-1;`

signal(s) `s:=s+1;`

The behaviour of these primitives is equivalent to the earlier definition but implies busy-waiting which is undesirable. Real implementations of semaphores associate a queue of processes with each semaphore. A process performing a **wait(s)** puts itself on the queue for **s** if **s** is zero and relinquishes control of the processor. A process performing a **signal(s)** activates a process if the queue is non-empty, otherwise it simply increments the semaphore variable. This ensures that a suspended process consumes no processor time. The only overhead is the memory required to hold one process control block per blocked process. The VAX enqueue and dequeue instructions ensure mutual exclusion of semaphore queue updates.

8.7 Monitors

A monitor [Hoa74] is a module that packages the definition of a shareable resource and the operations that manipulate it. Condition synchronisation within the monitor can be achieved using semaphores or one of a variety of other synchronisation primitives. The synchronisation primitives are not available outside monitor blocks. A monitor contains a set of static variables which are only accessible to routines within the monitor, a set of callable procedures for manipulating them and a mainline routine that is executed as initialisation code by the run time system at startup. All monitors are initialised before the user's main line program starts executing. The callable procedures may have parameters that are instantiated at call time. Monitor routines are mutually exclusive by definition, hence interleaved execution of monitor routines need not be considered when establishing the correctness of a concurrent system using monitors.

Monitors are derived from the operating system executive concept wherein all time dependent activities are performed by the executive on receipt of a system call. Many systems use synchronous interrupts or traps to activate these services allowing a centralised despatcher to field both systems calls and external interrupts. Such an executive is termed a *monolithic monitor*. Since there is only one monitor controlling all resources, two processes vying for use of, say, a disk drive may block a process requiring memory allocation. The monitor itself becomes a bottleneck since unrelated synchronisation activities are coupled together. This is a practical problem in uniprocessors, and becomes absolutely unacceptable in multiprocessor systems.

Hoare's monitors are distributed, and in general one monitor will be used to control each independent resource. This allows decoupling between concurrent processes. The monitor concept lends itself well to implementation *via* the software packaging mechanisms present in many modern languages such as Mesa [MMS79] and Modula-2. Indeed Wirth's standard library [Wir83] for Modula-2 includes a implementation of monitors and semaphores that requires almost no extensions to the base language. In Modula-2 a monitor is simply a module with a priority, giving a very elegant and sparse syntax. Monitors were first implemented in Concurrent Pascal [BH77], a Pascal like language that included extensions that included **process**, **monitor** and **class** types. Synchronisation within monitors was performed using primitives called **delay** and **continue** operating on **queue** types. The **class** type allowed encapsulation of data and routines in a way analagous to the **class** construct of Simula-67 and **module** construct of

Modula-2.

8.8 Distributed systems and mailboxes

A monitor encapsulates a single copy of the synchronisation control data as well as the shared resource. It relies on semaphores (or similar primitives) and these rely on common memory arbitration. This becomes more and more expensive as the degree of physical parallelism in system increases. On systems with multiple buses, shared memory techniques rapidly become prohibitive. The limiting case is that of a network where self contained computers are connected over relatively slow communications links.

Such systems require synchronisation primitives based on messages passed around the system. This gives rise to an input/output model of synchronisation rather than the shared variable model used previously. Several authors (notably [Hoa85]) have argued forcefully that this model is easier to program with correctly, as well as being necessary for distributed systems. Implementation of message passing on a uniprocessor, or on a tightly coupled multiprocessor with shared memory, is less efficient than the use of shared variables.

It is important to note that no timing can be guaranteed between sending and receiving messages. Messages may pass each other in transit and there is no central arbiter. Protocols must be self enforcing, to allow individual processes to decide on their own initiative when to proceed. This is analagous to the busy waiting solutions because there is no central agent that can wake a process up. This contrasts with the semaphore and monitor solutions which are designed to wake up other processes. Message passing protocols assume that there is no way of remotely manipulating the execution path of another process by aborting it, suspending it or restarting it. In general, messages are sent and received asynchronously. After a **send** the transmitting process usually continues. The receiving process will not usually be ready to immediately process the message which implies that it must be temporarily stored in a system area known as a mailbox.

A significant problem with mailboxes is deciding on their size. Small mailboxes may require messages to be split up into many sub messages, and long mailboxes will be under-utilised by short messages. Variable length mailboxes require extra protocol overhead to define their size and imply the use of variable-sized memory allocation routines that will generate the usual heap management problems and require periodic garbage collection.

Message passing has been used in operating systems applications for many years. Most operating systems support the use of mailboxes which allow producer-consumer interaction between processes. Unix uses an elegant notation to allow interconnection of producers and consumers using pipes which act as mailboxes between programs. This feature is very powerful because the Unix I/O system treats pipes and I/O devices as files in a completely unified way. This means that a user program can be written using simple file I/O and can then be connected to files, output devices such as printers or piped to other programs by supplying the correct commands at run time. RT-11 provides similar unified I/O and mailbox system requests but does not allow the flexible command line redirection of Unix, so mailbox transfers tend to be 'hardwired' into user processes. VMS also provides explicit mailbox services.

8.9 Rendezvous

Mailbox communication, even with fixed length mailboxes, is inefficient. The rendezvous is an attempt to improve the efficiency of mailbox schemes by removing generality. A rendezvous may be considered as a mailbox transfer with a zero sized mailbox. This implies that there is no buffering, and therefore that the first process arriving at the rendezvous must wait for the other process to catch up. Hence the rendezvous is inherently self synchronising. At the point of rendezvous, data may be transferred between processes. Since each datum requires a separate rendezvous and each rendezvous implies a context switch on a uniprocessor (since the first process arriving will suspend itself) efficient uniprocessor implementation of the rendezvous relies on a low context switch overhead. This approach is exemplified by the Inmos Transputer which is designed to support rendezvous communication and which can context switch by updating only two registers which can potentially be performed in only four instruction cycles.

It turns out that as well as being more efficient, the simplicity of the rendezvous allows rigorous mathematical treatment. Hoare's Communicating Sequential Processes (CSP) [Hoa85] is a mathematical notation for analyzing concurrency problems. The Occam language is heavily based on CSP.

8.9.1 The Occam rendezvous

The Occam rendezvous is programmed as an explicit I/O transfer using named channels between processes. Channels are unidirectional and only one

writer and one reader process is allowed per channel. Processes in Occam are unnamed because there are potentially a very large number of them. In contrast to conventional concurrent languages such as Concurrent Pascal and Modula-2 which support concurrency at the procedure level, Occam allows individual statements to be evaluated concurrently. Use of named channels rather than named processes is unfortunate from the point of view of correct program structure. The one writer and one reader per channel restriction has to be checked by the compiler. If channels were declared with named source and sink processes then the syntax of the declaration would automatically enforce the restriction.

In use, the syntax `chan1 ! variable` transmits the contents of variable down the named channel `chan1`. The syntax `chan1 ? temp` reads a datum off channel `chan1` into variable `temp`. If the writing process arrives at the send command before the reading process arrives at the receive command then the writer is suspended. When rendezvous is achieved the writer is awakened and the data transfer takes place. This simple syntax is elegant and easy to use. Potentially each statement in Occam is a process in its own right and the `!` and `?` operators form two of the five basic processes, the others being `:=` (assignment), `STOP` (a process that never terminates and performs no useful work) and `SKIP` (a process that terminates immediately and performs no useful work). Read-only shared variables are allowed between processes, although real implementations will usually forbid this when the processes are executing on different processors.

8.9.2 The Ada rendezvous

Ada uses an extended rendezvous in which two way communication of data is allowed in a single rendezvous. The programmer's model of the rendezvous uses the concept of remote procedure activation rather than explicit I/O transfer as in Occam. As a result the syntax is rather unwieldy, and difficult to use.

An Ada process is called a *task*. Each task is a package containing routines that may be remotely called by other processes as well as internally called routines and static data. Ada packages have a specification part and an implementation part called the *body*. The specification part names the identifiers that may be accessed from outside the package. In a task specification, procedure names indicate potential entry points. Shared variables are allowed by naming in the specification part. The actual entry points are defined with **accept** statements in the body of the task. The following example shows an Ada task that takes an integer parameter from the calling process and returns a running average, along

with two calling processes:

```
PROCEDURE taskdemo IS
 numberofprocesses: CONSTANT:=2;
TASK TYPE process;
TASK runningaverage IS
 ENTRY compute(x: IN integer; average: OUT integer);
END runningaverage;
TASK BODY process IS
 i, returnedaverage: integer;

BEGIN
 LOOP
  i:=randomnumberfunction;  -- externally declared
  compute(i,returnedaverage);
 END LOOP
END process;

TASK BODY runningaverage IS
 n,total: integer;

BEGIN
 n:=0; total:=0;
 LOOP
  ACCEPT compute(x: IN integer; average: OUT integer) DO
   n:=n+1;
   total:=total+x;
   average:=average/total
  END compute;
 END LOOP;
END runningaverage;

P: ARRAY(1..numberofprocesses) OF process;

 BEGIN
  NULL;
 END taskdemo;
```

8.9.3 Non-determinacy in rendezvous systems

Simple use of the rendezvous implies a tight coupling between consumer and producer processes which causes problems similar to those of the coroutine implementations above. A process can not proceed beyond a rendezvous point until synchronisation is achieved with its partner. Buffering between processing is impossible: even with a separate process controlling the buffer contents it will still be constrained to giving alternate access to the producer and consumer. This occam fragment outlines a buffer process with channels called put and get for use

by the producer and consumer:

```
CHAN OF INT put, get:
SEQ
  INT hi, lo, items:
  VAL size IS 100:
  [size]INT buffer    -- circular buffer
  SEQ
    items:=0; hi:=0; lo:=0
    WHILE TRUE
      put?buffer[hi]; items:=items+1; hi:=(hi+1) REM size
      get!buffer[lo]; items:=items-1; lo:=(lo+1) REM size
```

What is required is some way of allowing the buffer process to examine the state of the two channels and rendezvous with whichever is ready. This is supported in occam by the use of the **alt** constructor and in Ada with the **select** statement:

```
CHAN OF INT put, get:
SEQ
  INT hi, lo, items:
  VAL size IS 100:
  [size]INT buffer    -- circular buffer
  SEQ
    items:=0; hi:=0; lo:=0
    WHILE TRUE
      ALT
        items<size & put ? buffer[hi]  -- guard 1
          SEQ
            items:=items+1; hi:=(hi+1) REM size
        items>0 & get ! buffer[lo]     -- illegal guard 2
          SEQ
            items:=items-1; lo:=(lo+1) REM size
```

When the buffer process reaches the **alt** statement the guards are evaluated. In general a guard can comprise a boolean expression and a channel read (?). If the boolean evaluates **true** and there is a datum waiting to be read on the input channel then the guard is said to be open. If several guards in an **alt** are open, then the program decides arbitrarily which guarded process to execute. If no guards are open then a run time error results.

The above example is not legal occam because output operations are not allowed in guards (as in guard 2). The reason for this is that if both input and output operations on the same channel were part of guarded commands in separate processes, the choice of alternative made in one process would have to result on the choice of alternative made in the other process and vice versa. Resolution could only be achieved by sequentially evaluating the ALT statements. This would be

extremely difficult to implement especially in a distributed system. The problem may be overcome by constructing another process that fields the GET channel and makes a request to the main buffer process using an auxiliary channel. This effectively implements handshaking between the buffer processes and allows the logic of the GET channel to be inverted.

8.10 Array processor language features

The above sections have dealt with language extensions for the programming of explicit parallelism on MIMD type systems. There exists another class of language extensions targeted at vector and array processors. DAP-FORTRAN [ICL79], ILLIAC IV CFD [Ste75] and ACTUS [Per87] fall into this class. Typically, extensions are provided that modify array declarations so as to declare operands that will be operated on using spatial parallelism, as well as parallel operators for element selection, loop control and parallel decision evaluation. [PZ86] provides a useful review of these languages.

8.11 Automatic detection of parallelism and VLIW architectures

The design of correct parallel algorithms is non-trivial. Many attempts have been made to produce compilers that will allow the programmer to use a sequential notation and automatically detect parallelism and map it on to hardware. The most successful of these have been the vectorising compilers such as Cray CFT [Res82] and the CDC Cyber 200 FORTRAN [Cor82] for the Cyber 205. The process of vectorisation essentially involves examining the contents of DO loops to find sequences of statements that may be executed by a pipeline of vector functional units. Use of GOTOs, IF statements, subroutines and I/O within the DO loop naturally preclude vectorisation.

A completely different approach to the automatic parallelisation of algorithms is provided by the use of functional programming languages running on graph reduction and data-flow architectures. In principle the absence of side effects in a functional language coupled with the data driven evaluation of these non-Von Neumann processors should allow any parallelism existing in the problem to be automatically exploited. Opponents of this approach claim that sequentially constructed code simply does not contain enough parallelism to provide a signif-

icant speed up. Of course, functional languages may be used in an explicitly parallel way to program the underlying hardware in much the same way that the imperative languages described above may, but that is beyond the scope of this thesis.

An interesting recent development known as Very Long Instruction Word processing combines elements of RISC technology, array processing and MIMD procedural programming. A VLIW machine is a collection of RISC processors with a large number of parallel, pipelined functional units but a single control stream [Ell86]. It is similar to an array processor in that there is a single instruction stream but each 'processing element' is a general processor with its own control fields in the instruction word. A VLIW machine might more usefully be thought of as a microcoded machine with a very large flat instruction word controlling many functional units. A practical machine would have an instruction word of the order of 1000 bits or more, hence the term Very Long Instruction Word. The instruction word also controls a switching matrix that connects functional units to banks of memory, allowing swapping of operands between functional units.

Manual programming of such a system would be all but impossible, and the power of the system can only be exploited using high level languages and sophisticated compilers. VLIW technology has its origins in the ELI project at Yale University [Fis82], [Jos83] and a compilation technique known as trace scheduling. Parallelising compilers for machines such as the CDC 6600 and the scalar part of the Crays operate by compiling basic blocks with associated register storage and then attempting to allocate such blocks with different resource requirements on to a fixed set of processors. Experiments have shown that one could expect at most a two or three times speed-up by parallelising basic blocks [FR72]. However later experiments showed that many scientific programs contained fine grained parallelism averaging a factor 90 [NF81].

Trace scheduling operates by tracing execution paths through blocks and attempting to predict the most likely execution path. Ellis's compiler (called Bulldog) uses programmer supplied hints to aid in this process. The compiler also analyses operand references to allow instruction streams to be allocated to processors and operands to be allocated to memory in such a way as to minimise horizontal interdependencies. Sometimes an operand would be copied into several memory banks.

Bulldog has been tested on matrix multiply, FFT and other scientific calculations. It is claimed that Bulldog can find significant parallelism and generate order of magnitude speed-ups using conventional technology. The system is

being commercially exploited by Multiflow Inc.

The combination of detection-of-parallelism compilers and massively parallel hardware of a synchronised array type (rather than an unsynchronised MIMD type) suggests that the techniques may yield great returns in image processing much of which resembles scientific processing with its emphasis on repetetive calculation on large data sets.

8.12 Conclusions

The wide variety of available constructs for concurrent programming indicates that a concensus has not yet been reached concerning the ideal programming paradigm. There is a trend towards rendezvous based synchronisation primitives as evidenced by Ada and Occam, but these do not address the requirements of shared memory systems. The emphasis on message passing is a result of the greater security offered to the programmer, and their amenability to formal analysis. Image processing MIMD systems are likely to continue to be dominated by shared memory processors because of the very high data rates that would be required in a message passing system where images are the primary data.

Chapter 9

The IMP system

"imp [*imp*] *n.* minor demon; mischievous goblin, sprite etc."

9.1 Introduction

This chapter describes the author's Image-handling MultiProcessor (IMP) and its application to a realtime grey-scale industrial inspection problem.

IMP is a MIMD system which allows normal computers, hardwired processor boards, microcoded CPUs and frame stores to be connected together over a 16-bit data bus capable of sustaining 6.6MHz transfers. This allows a complete addressing cycle to be completed in one 256 × 256 square pixel time. Since the data bus is sixteen bits wide, a byte packed framestore like the V2 board described in Chapter 5 can provide pixels at 512 × 512 pixel rates. The communications standard is an upwards compatible implementation of the commercially available VMEbus [VIT85]. This means that the critical electrical components (backplane, drivers etc.) are commercially available and that the various protocols involved are supported by custom integrated circuits.

9.2 Derivation of IMP architecture

The original intention was to upgrade IPOFS to 256 × 256 pixel images and add bus multiplexing to allow another port into the image memory. This extra port would be used as a coprocessor bus. Extra lines on the coprocessor bus would control the operation of the processors themselves, and these lines would be controlled by the host PDP-11 *via* extra control registers in the IPOFS CONBLK area. Since the control signals for 256 × 256 video are already available within the IPOFS controller, only the addressing scheme and the window mapper circuit

would need to be expanded.

Although this new system would be comparatively simple to implement, it would have several disadvantages and problems:

1. the window mapper hardware which is the major feature of IPOFS would only be available to the host processor,
2. all synchronisation and control of coprocessors would have to be performed by software in the host and this could become the major bottleneck,
3. all data transfers between coprocessors would have to be routed through the host, since there is no inter-processor bus,
4. the backplane would have to be very well engineered to cope with the high frequency signals generated during a processing pass.

In a system comprising the host PDP-11 and a series of small hardware modules implementing simple parts of an algorithm, these restrictions would not be too severe. For instance a median filter, Sobel operator, threshold sequence might be implemented with each part of the sequence a separate hardware module started up in turn by the host. Assuming that a pixel could be read from memory in 150ns then the processing time for a module would be approximately n frame times, where n is the number of memory accesses required per pixel. For a threshold this would be 2 (one read and one write), and 10 for a straightforward implementation of a 3×3 Sobel operator. It is possible to speed up the Sobel by retaining some parts of a window for the next operation [Lee83,Pic84]. In this case an average of only one read and one write are required per pixel, and therefore the total execution time would again be 2 frame times. The address generation circuitry required to do this is quite complex, and it would need to be replicated on each module. Ideally resources such as this would be centralised and available to all processors.

9.2.1 Coprocessor bus

Since all control registers would reside on the main controller card, all control information would have to be provided by the extra lines on the coprocessor bus. The best way to do this is to provide an extra low speed address and data bus from which the host can download data into the coprocessors. Thus two separate buses are provided, a pixel access bus and a processor bus.

The system as described is highly hierarchical, in that the host has absolute control, and all data transfers between coprocessors must proceed through it.

Ideally, coprocessors would be allowed to communicate directly with each other. A hybrid approach might be for the message sending coprocessor to leave the data to be communicated in an image space, relinquish control of the pixel bus, and rely on the host to initiate receipt of the data by another processor. In this case, only arbitration transactions are required on the processor bus, and data transfer occurs at hardwired speeds, rather than host speeds.

A better solution is to allow the coprocessors themselves to become masters on the coprocessor bus, and to directly address the other coprocessors. No interaction is required by the host, but some central arbitration scheme is required. Given that the coprocessor bus is to be arbitrated centrally, it makes sense to arbitrate the pixel bus at the same point. If these arbitration functions are performed in hardware, then coprocessor operations will all proceed very quickly, and the role of the host can be reduced to monitoring the buses, and initial preloading of the system. Since it will not have to waste time transferring data and arbitrating buses, it will be free to perform background tasks such as trend analysis and statistical reporting of the line under inspection.

9.2.2 Unification of multiple bus scheme

It is clear that there is significant redundancy in the two bus scheme. In general, a coprocessor will be using either the pixel bus or the coprocessor bus. The two buses will require a large number of backplane lines each, and it is likely that the addressing and data range of the buses will be restrictive for an economically viable scheme. It is useful to have between four and sixteen image spaces available, and so the pixel bus will need to access between 256K and 1M locations, which requires 20 address lines. A 16-bit data bus, read/write and valid address strobes will require another 18 lines. All of these lines will require buffers on each coprocessor board, and if two buses are to be implemented, then these will have to be duplicated. 10 twenty pin IC packs will be required for this alone, and when two arbitrator interfaces are required along with control logic it is clear that the real-estate overhead on each board is significant.

If the coprocessor and pixel buses are not to be used concurrently, then it would be much more sensible to combine the two into one large address range data bus. Pixel memory and coprocessor registers can be allocated addresses throughout the address map, and a much more unified (and economic) design results.

The final modification to be made to the simple expanded IPOFS model

is to provide centralised memory management resources, so that address generation for, say, 3 × 3 window operations can be done by one piece of hardware for all processors. The bus must provide a mechanism for a bus master to relinquish the addressing section and allow the memory management unit to fetch the data.

Several design exercises were conducted, mainly centering on the arbitration structure and the physical design of the backplane. It was quickly recognised that an electrically robust bus requires very high engineering standards, and that the development of a dedicated backplane for IMP might be a lengthy job. Electrical problems become more important as the number of loads in a system and the operating frequency become greater. Clearly, the backplane is the potential Achilles heel of a system, since it will probably have the greatest loading, and is subject to variable loading depending on the card configuration, so 'single case' specification is not possible.

Real time industrial control is a growing market, and several manufacturers have defined bus standards for multiprocessor systems. The IMP bus requires very high throughput and memory management facilities considerably in excess of those normally found in industrial systems, but a survey was made of existing buses to see if one could be adapted for use in IMP. Adaptation could be at one of three levels:

1. Use of manufacturer's backplane as the physical substrate for IMP, but with an IMP-specific allocation of lines and protocol,

2. Use of manufacturer's bus standard without modification,

3. Use of manufacturer's bus standard with IMP-specific enhancements, preferably not conflicting with manufacturer's standard.

Level 2 (no adaptation) would obviously be ideal, allowing work to commence at once on the functional parts of IMP.

9.2.3 Commercial multiprocessor buses

Any real computer bus will be a collection of several logical buses supporting the distribution of power and different kinds of signals such as:

1. data transfer between processors and memories (which may reside within other processors),

2. arbitration,

3. interrupt information,

4. system wide control, such as reset signals and master clocks.

These logical buses may be time multiplexed onto one set of wires, in which case there is only one physical bus, or they may be implemented as separate physical groups of lines. There may even be multiple physical buses for one logical function — for instance two data transfer buses to speed memory access. The use of a separate I/O bus falls into this category, where transfers between processors and memory may occur concurrently with transfers between processors and backing store over two sets of wires. Within each physical bus further multiplexing may be used, for instance between addresses and data. Each operation on a bus is called a transaction, and may be a complete arbitration cycle, an interrupt-interrupt acknowledge sequence or the fetching of one word of data.

A bus may be evaluated in terms of economics and performance measurements. For instance a 16 line multiplexed address/data bus running at a maximum square wave frequency of 10MHz can support up to 128K bytes of memory which can be accessed at a maximum frequency of 20M byte per second. It will require 17 lines and buffers (one extra line is needed for demultiplexing) and for many applications at least 16 latches will be required. An arbitration bus will be able to support varying numbers of prioritised requests depending on the number of lines dedicated, and similarly for the interrupt bus. The arbiter may prioritise requests using a preset priority scheme or a round robin scheme which will ensure that all masters get a fair share of the bus.

Bus transactions fall naturally into two classes — asynchronous and synchronous. These describe the nature of the transfer protocol. In a synchronous bus, all transactions occur at fixed times governed by a centrally generated clock. Most small computers use this kind of bus, and with the addition of some arbitration logic, a multiprocessor bus can be constructed. In an asynchronous bus, all transactions are *handshaked.* A bus master writing to the data bus asserts address, data and cycle type information on the bus, waits for the bus to settle and then asserts a *strobe* that tells the slave to begin address decoding and data accessing. When the slave has acquired (write cycle) or fetched (read cycle) the data, an acknowledge signal is returned to the master, which then removes the strobe and other data. Arbitration and interrupt transactions may be similarly handshaked.

Asynchronous transactions have the advantage that they automatically adapt to different speed masters and slaves. In a synchronous system some de-

cision has to be made concerning the base frequency. Devices which run faster must have some means of idling while the bus responds. Slower slaves will be unusable for reads, and will require front end synchronisation latches for writes. Any future improvements in technology allowing faster operation will be difficult to incorporate. In a mixed hardwired/conventional computer environment such as IMP there are bound to be at least two basic operating speeds — the hardwired accesses will probably operate at near video pixel rates (say 150ns per access), and the computer will operate at around 1μs per access. An asynchronous system will be able to cope with this without requiring any hardware overhead on the individual boards.

The review was restricted to widely distributed buses, on the basis that the economic advantages of adopting a manufacturer's bus would be lost if that bus were an obscure one. The systems considered included CAMAC, the DEC omnibus, Unibus, Q-bus family, the synchronous SBI used in the VAX 11/780 memory subsystem, the STD and STE buses, the Motorola Versabus and VME-bus, and the Intel Multibus and Multibus II.

9.2.4 Conclusions

Of the buses considered, only VMEbus and Multibus II are capable of providing pixel rate throughputs at small system cost. It is clear that VMEbus is more applicable to IMP because it can adapt gracefully to the different access rates required, and most importantly of all it provides a means of specifying special cycles and protocols in an upwards compatible way through the use of address modifiers. Since slaves must only respond to address modifiers that they understand, making use of the user defined codes will automatically disable commercial and non-IMP slaves. This allows IMP-specific and commercial boards to be mixed in a single system with no contention. As a bonus, the protocol is similar to the Unibus and Q-bus protocols and this eases the design of a DEC to VME interface. This makes up for the impossibility of building the IMP system directly onto the Q-bus host. Finally, a series of VLSI bus protocol chips to handle functions such as bus arbitration and interrupter/interrupt servicing are appearing, and 68000 family peripheral chips will interface directly.

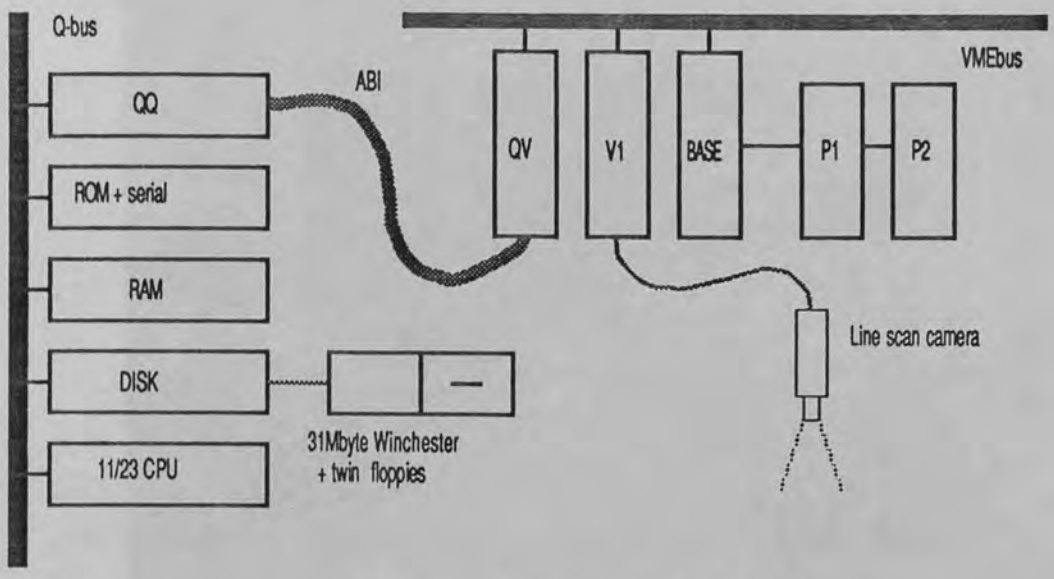


Figure 9.1: IMP block diagram

9.3 Architectural overview

Figure 9.1 shows a block diagram of IMP at the backplane/board level. Figure 9.2 is a photograph of the prototype. This is based around a 6U Eurocard crate with a 9 slot VME backplane mounted in the upper connector position. A 250W power supply is mounted at the rear for the main logic supply. A smaller power supply for the analogue circuitry is bolted to one end, along with a die cast box containing the digitiser logic. Each circuit board is of double extended Eurocard size with the lower connector available for I/O specific to that board.

The system controller card provides bus arbitration and memory management, power up and manual resets, and a bus watchdog that will time out any bus transactions that are not acknowledged within 16 microseconds. A 16 MHz clock is also provided for compatability with other VME systems, although this is of limited use in IMP.

The Q-bus link is a two board set providing transmission between Q-bus and VMEbus backplanes over a 20-way twisted pair cable driven to the RS-422 electrical standard. The protocol used is a slightly modified version of the Q-bus protocol which may be generated with some further multiplexing of the Q-bus lines, and a single flip-flop for interrupt latching. The electrical and protocol standard form another bus specification called the Asynchronous Bus Interconnect (ABI). The smaller Q-bus card carries RS-422 transceivers, Q-bus buffers and a small amount of multiplexing logic. The main part of the protocol conversion is performed on the larger VME card.

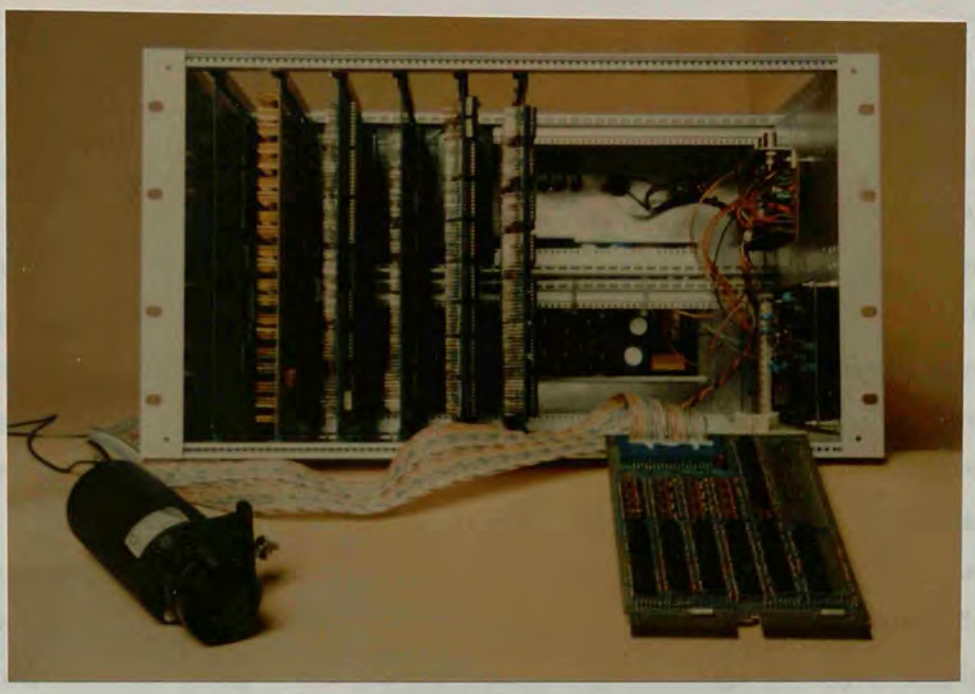


Figure 9.2: IMP prototype

The industrial inspection system described in Section 9.10 used the V1 framestore described in Chapter 5 and a PDP 11/23 host. Current versions of the system use the V2 framestore and PDP 11/73 or MicroVax II hosts. Three systems have been constructed and are in daily use in the Machine Vision Group's Laboratories.

9.4 Project management

IMP is a potentially large system, and the prototype implementation described here involves some 250 integrated circuits distributed over four boards. Large projects can rapidly become unmanageable, but significant efforts have been made to maintain a disciplined documentation system from the start. A complete description of all component positions and interconnections is maintained in machine readable form using a suite of programs written by the author. These programs generate wiring lists, automatically flagging some simple errors. Board maps can be produced on a normal printer. There is also an interface to the Racal Redboard PCB layout system and the combined system has been used for the implementation of a commercial high-density peripheral card and the prototype of the NPL LAP2 array processor.

When the paper design of a board has been completed, a prototyping card is selected and the components laid out as desired. The type of the board, the components and the pin 1 coordinates are typed into a component listing file. Program CADEXP then expands this listing using two databases containing

descriptions of the various prototyping boards and IC types. The result is a listing of all IC pins on the board with their coordinates and electrical types (*e.g.* totem pole output, open collector output, bidirectional tri-state etc.), as well as the function of each chip and pin. Signal names are then taken from the paper design and added beside each pin name. Another program called CADSIG then goes through the file collecting references to signals and collecting them together in alphabetical order so as to form a wiring list. Errors such as the connection of two outputs together are caught at this stage. The board is wired directly from this list on a coordinate to coordinate basis. Wiring up is thus reduced to a mechanical task which does not require constant reference to the component side of the board to check that the correct IC has been located. This has reduced the incidence of wiring errors (as opposed to logical design errors) to negligible proportions. Wiring errors were extremely common whilst building IPOFS, and this was exacerbated by the 'design it on the board' approach.

The machine readable wiring lists can be used to generate tapes for automatic wire wrapping machines which may provide cheaper fabrication for small production runs (say less than 10) than using a PCB, especially since a complex board like V1 would require a multi plane board, or segmentation onto two separate PCBs.

Within each generation, a board may go through many changes as it is debugged or enhanced. These small Engineering Change Orders (ECOs) are accumulated on a hard copy listing of the board's last dumped state, or marked in a disk file. At intervals, a block of ECOs will be applied to the board's source file, and new wiring lists will be generated. This constitutes a new revision of the board. Therefore under this system a board's state at any time will be given in a file tagged by PROJECT/BOARD-REVISION-ECO, *e.g.* IMP/V1-E-B.

Since this software is running on the IMP host computer, it is available for use on the bench, and has proved an extremely powerful debugging aid. If a signal is found to be showing unexpected behaviour, the system editor can very rapidly find all pins attached to that signal, and which is the output. Since all changes are recorded, unexpected side effects are avoided. During the development of IPOFS the situation often arose where a modification to one part of the circuit generated unexpected behaviour in a completely unrelated part, because a common interconnection had been overlooked, or because a signal line had not been completely restored after the removal of one link. This kind of bug, where the symptom does not have a clear causal connection with the real problem is the most difficult to track down, being highly analogous to the bugs found in

parallel software systems. Having a machine-checked list of every pin, wire and connection on the board has removed this problem, and allows debugging to proceed in a far more linear and structured way. Although originally conceived as a documentation aid to reduce the costs of PCB production, the system has turned out to be vital during development. The current documentation for the prototype IMP occupies some 300K bytes of disk space and it would clearly be impossible to carry even a small amount of that type of information in the head. Since the software automates working practices that had evolved during the development of IPOFS, little or no adaptation has been required to make use of it: however it has imposed more discipline, which ultimately reduces design time and costs.

9.5 Use of the VMEbus in IMP

IMP adheres closely to the VMEbus standard. Within the limits of the standard two kinds of 'customisation' are possible. Firstly there are various options concerning the width of data and address buses; the kind of bus arbitration provided; the distribution of interrupt fielding processors; and the kinds of bus cycles supported. Secondly, user definable address modifier codes are available for the specification of unusual data transfer cycles. In IMP, a centrally controlled memory management protocol has been designed and synchronous transfer modes defined through the use of address modifiers. These additions to the basic VMEbus cycle types will be implemented on a future bus controller card. The VMEbus specification allows boards that are addressed with cycle types that they do not recognise to return an error signal, thus maintaining the integrity of the system.

A technical overview of VME protocols is presented here to provide the reader with enough information to understand the detailed operation of the IMP boards. A fuller reference is [VIT85].

9.5.1 VMEbus lines

The 82 signal (*i.e.* non-power) lines on the upper (P1) Eurocard connector divide into four separate buses. These are: a 16-bit data/24-bit address Data Transfer Bus (DTB); a seven level interrupt bus (INT); a four level DTB arbitrator with selectable priority or round robin scheduler algorithms (ARB); and a miscellaneous system utility bus (UTL) carrying a 16MHz clock signal, reset and power monitor lines, an error line for boards to indicate failure of self-tests, and a low speed serial link for interboard communication and synchronisation. The

signals are summarised below:

```
DTB

A01-A23          address bus         Primary address
D00-D15          data bus            Primary data
AM0-AM5          address modifiers   DTB cycle control
LWORD*           long word           Used with 32-bit extension
WRITE*           write
AS*              address strobe
DS0*             data strobe low     low byte access
DS1*             data strobe high    high byte access
DTACK*           data acknowledge    Handskake from slave
BERR*            bus error           Bus protocol error

ARB

BR0*-BR3*        bus request         four level request bus
BG0IN*-BG3IN*    bus grant in        bus grant daisy chains
BG0OUT*-BG3OUT*  bus grant out
BBSY*            bus busy            master has bus
BCLR*            bus clear           master wants bus

INT

IRQ1*-IRQ7*      interrupt request   interrupt request bus
IACK*            acknowledge
IACKIN*          acknowledge in      daisychained acknowledge
IACKOUT*         acknowledge out

UTL

ACFAIL*          AC failure          power fail imminent
SYSRESET*        system reset
SYSFAIL*         system fail         failed self test
SYSCLK           system clock        16 MHz clock
SERDAT           serial data         low speed serial line
SERCLK           serial clock
```

9.5.2 Arbitration bus

IMP uses a four level priority arbitration scheme, allocated as follows:

```
3 - Executive processor                (highest priority)
2 - Hardwired coprocessors
1 - Software based coprocessors
0 - Video acquisition and display      (lowest priority)
```

Only one video coprocessor is allowed in the system. Note that the framestores described in this thesis are video slaves which do not qualify as a coprocessors. A graphics processor, or sync locked DMA transfer processor would be examples of true coprocessors. Normally there will be only one executive processor, although it is conceivable that a second level 3 processor might be used in some applications.

The executive processor is in overall charge of the system. It is able to initialise, start and stop all other coprocessors, and in the prototype implementation is able to monitor all bus activity above the level of individual data transfers, and so can be used as a sort of 'software logic analyzer' in the debugging of target coprocessors. Naturally it has the highest priority so that it can usurp an errant coprocessor which is tying up the bus.

The next highest priority is reserved for hardwired coprocessors. It is assumed that such machines will not in general be able to save their system state easily (*i.e.* they are uninterruptible in that they will not be able to recover after an interrupt, except *via* a general reset). Below these are the level 1 software controlled coprocessors based around conventional microprocessors and computers.

When there is no other activity on the bus, and when no requests are outstanding, the arbitrator gives control to the video subsystem. This allows the video board to display images from VMEbus memory, or to write digitised video or other data. When a coprocessor requests the bus, video display from the bus will be suspended. This will cause 'hashing' on the screen. However, all planned video boards also have onboard local memory from which displays may be maintained even when the video board is locked off the bus. V1 uses local memory exclusively, and therefore never needs to become a VMEbus master. The V1 onboard memory is accessible through two blocks of VME addresses — as a mapped set of registers forming a superset of the IPOFS registers, and as a simple block of RAM filling 128K contiguous locations.

Within each level, bus grants are daisychained so as to provide an extra level of prioritisation based on physical proximity to the arbitrator.

9.5.3 Arbitration protocol

In IMP, there is always a bus master. If no coprocessors or executives are active, control defaults to the video system. A simple bus request and grant sequence is illustrated in Figure 9.3.

1. A level 2 hardwired coprocessor needs the bus and asserts BR2*. The bus request lines are open collector, so several requests may be pending simul-

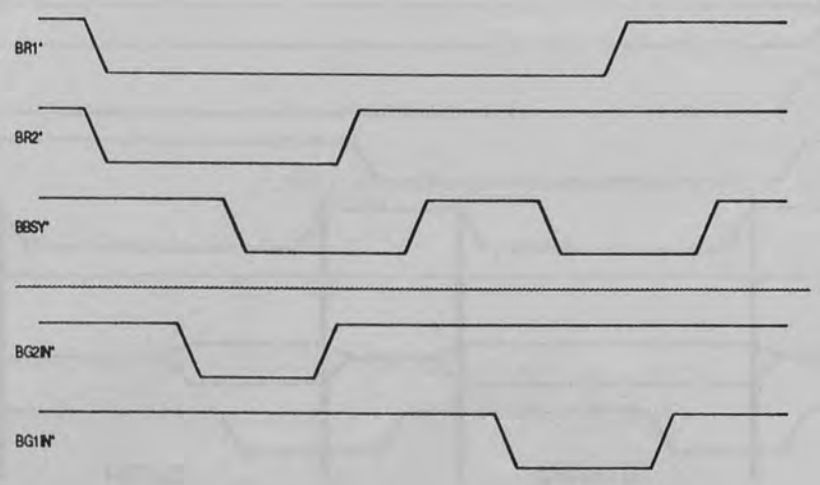


Figure 9.3: IMP bus arbitration

taneously.

2. When the arbitrator receives a request of higher priority than the last posted grant, it asserts BCLR* to order the current bus master to release the bus.

3. In this case, the video board will tri-state its bus drivers and release the BBSY* line.

4. The positive going edge of BBSY* is a signal to the arbitrator to issue a grant to the next eligible master, in this case on level 2.

5. The grant will propagate down the backplane being examined and be passed on by all boards until it reaches the requesting master where the grant is blocked.

6. The new master asserts BBSY* and the arbitrator responds by removing the grant (this implies that grants must be latched on the master).

This completes the arbitration transaction, and the new master will retain bus ownership until either it has finished, or the executive processor requests the bus, in which case a BCLR* signal will be generated.

It is possible for arbitration to be overlapped with DTB transactions. A master may release BBSY* after the start of its last DTB cycle. The new master will then receive a grant before the address and data strobes have been released. Naturally, the new master must wait until no strobes are present before activating its own bus drivers. This early release of BBSY is optional. Boards providing this feature are called type Pre RElease (PRE) masters.

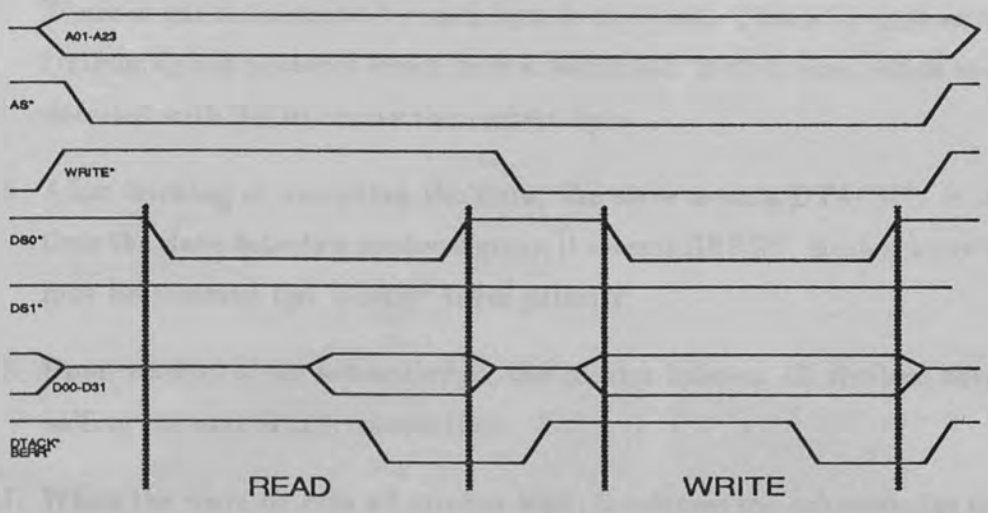


Figure 9.4: IMP data transfer

9.5.4 Data Transfer Bus

The basic DTB comprises 16 data lines, 23 address lines, one address and two data strobes, a write line, a normal and an error acknowledge; and six address modifiers which may be used to specify multiple address spaces (*e.g.* kernel, supervisor and user space) or to specify special address cycle types. If the second connector (P2) and backplane are fitted then the DTB is extended to 32 data lines with another 8 address lines. A line LWORD* is used on the main backplane to specify a 32 bit access. This extension is not used in the prototype IMP, so it is limited to an address space of 8M word of 16 bits.

9.5.5 Data transfer protocol

The IMP data transfer cycle is shown in Figure 9.4.

1. After having taken possession of the bus (see above), the master asserts A01-A23, AM0-AM5 and WRITE* with their required levels for this transaction.

2. The master waits for 35ns to allow for 25ns deskew and 10ns setup time at the slave, and then asserts AS*.

3. For a write cycle, either concurrently with the assertion of the addresses, or afterwards, D00-D15 are asserted as required.

4. The data strobes are asserted as required either concurrently with or after the assertion of the address strobe. On a write cycle, 35ns deskew and setup time must elapse after data assertion before data strobes are asserted.

There is one data strobe for each byte in the word. This is in contrast to the Unibus/Q-bus protocol which uses a dedicated 'BYTE' line, which must be decoded with A0 to access the correct byte.

5. After fetching or accepting the data, the slave asserts DTACK*. If at any time the slave detects a protocol error, it asserts BERR*. Both acknowledges may be asserted but BERR* takes priority.

6. Upon receipt of an acknowledge, the master releases all strobes, thus signalling the end of the transaction.

7. When the slave detects all strobes high, it releases the acknowledge lines.

The IMP controller carries an 8-bit counter driven by the 16MHz SYSCLK signal that is reset to 0 when the data strobes are high. The counter overflow also drives BERR*, so that if 256 16MHz cycles (*i.e.* 16μs) into a DTB data transfer no acknowledge has been received, the transaction will be 'timed out' and a protocol error reported. Without this feature, the addressing of non-existent memory would cause a system hang.

9.5.6 Use of address modifiers

The six address modifier lines allow one of 64 modes to be specified along with the main address. These modifiers are decoded along with the address itself, and slaves will respond only to the modes they recognise. (Typical commercial systems decode the address modifiers in a 64 word PROM which may be modified by the customer in line with any special needs.) There are 16 modes reserved for user application (numbers 10–1F hexadecimal). Although this feature will probably be rarely used in conventional VME computers, it provides the key to upgrading the bus in a safe way. As long as the strobe and interlock protocol is adhered to, any special protocol can be designed, because assertion of any of the 16 user modes will disable all 'normal' slaves in the system.

9.5.7 Interrupt bus

During the early design of this system an attempt was made to unify and condense the bus as much as possible — hence there is only one data bus over which all transfers whether DMA, I/O or programmed occur. However an interrupt mechanism is also required. If the executive is to manage the system it must be informed when a coprocessor has completed an operation, or requires

attention of some kind. It is not possible for the executive to *poll* the coprocessor by reading its status over the DTB because as mentioned above, hardwired processors especially are unlikely to tolerate interruption, and the executive will need to gain bus mastership to perform the poll.

One way to avoid using a separate communication channel for this status information would be to get the coprocessor to write to a special location which generates an interrupt in the executive. (This is just how the interrupt protocol on the Unibus works, except that a special IRQ line is used to address the interrupt fielder rather than a particular location. As a result no separate interrupt bus is required.) However, since the single executive fields all interrupts, and since there may be many interrupters, it makes sense to minimise the logic required on the interrupter at the expense of a more complex interrupt fielder. A single interrupt line on the backplane activated at the end of a process is much simpler to implement than the logic required to address and write to a special bus location. Another problem with Unibus style interrupt protocols is that masters that are not in possession of the bus have no way of asking the executive for service. Specifically the video circuitry in IMP can be used to supply timing information to the executive based on its frame and line timing. However, the video circuitry can never cause a coprocessor to relinquish the bus because it has a lower priority, and the arbitrator will simply ignore any requests.

As a result all 7 interrupt levels on the VMEbus are used for parallel communications. All interrupt lines are open collector, and so multiple requests on each line are possible. The 7 levels are defined as:

7 — Executive processor (reserved)

6 — Hardwired coprocessor error

5 — Hardwired coprocessor completion

4 — Software coprocessor error

3 — Software coprocessor completion

2 — Video coprocessor error

1 — Video coprocessor completion

Level 7 is reserved for any future implementation that may have more than one executive level processor. In such a case, one will still have to be designated the interrupt fielder and will therefore be the main system controller. The subsidiary executives, which may be controlling complete IMP systems of their own will communicate with the executive using level 7 interrupts. One application would be to wide production lines, which due to optical and throughput limitations could not be monitored with one system.

The other 6 levels are allocated equally to the other 3 processor levels. For each, a completion and an error interrupt are provided to signal correct termination and error aborts of an operation.

9.5.8 Utility bus

The utility bus provides power monitoring, reset and clock facilities. In the prototype the power monitoring lines are not used. The reset line is asserted on power up and under software control from the host. The clock provides a 16MHz 50% duty square wave in line with the VME specification. In fact a 20MHz clock would be more useful since the pixel timing for both square and rectangular images can be derived simply from this. The SYSFAIL* line provides a way of alerting the executive that a coprocessor has failed its self test. The serial lines SERCLK and SERDAT are not used in IMP.

9.6 Q-bus to VMEbus link

The prototype executive processor is a Q-bus based PDP-11. PDP 11/03, 11/21 (Falcon), 11/23, 11/73 and MicroVAX II hosts have been used successfully. The interface link was designed to be electrically robust so that operation in an electrically noisy environment such as a factory would not be impaired. The executive is also provided with a comprehensive bus monitoring capability.

Transmission between buses is over a 20-way twisted pair cable driven to the RS-422 electrical standard. This allows operation at frequencies of up to 10MHz over distances of up to 10m. The cable may be extended up to 1km, but with greatly downgraded frequency performance (50kHz maximum at 1000m) [Dev81a,Dev81b]. The receivers can discriminate against up to 25V of common mode noise, and this allows complete decoupling of the chassis grounds between the Q-bus machine and IMP. The protocol is a modified Q-bus scheme using additional multiplexing to reduce the number of signal lines to 20. The electrical and protocol standards together constitute another bus standard called the Asynchronous Bus Interconnect (ABI).

9.6.1 QQ card

The QQ card is essentially just a buffer board between the Q-bus and the ABI. A block diagram and a photograph of QQ are shown in Figure 9.5 and

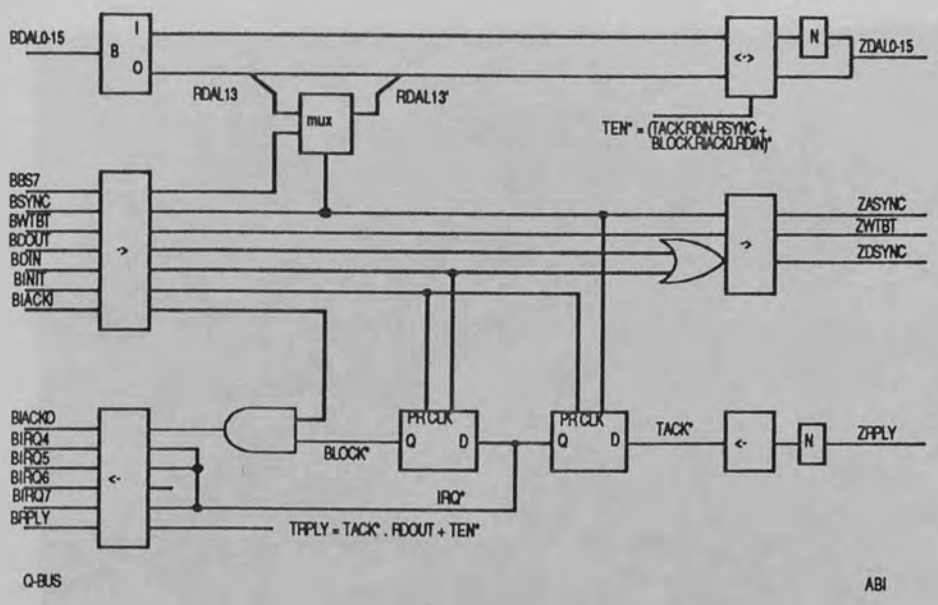


Figure 9.5: QQ block diagram

9.6 respectively.

The following lines are buffered from the Q-bus:

DAL0-15	16	Data Address Line	Multiplexed data/address
BS7	1	Bank Select 7	I/O page decode
WTBT	1	WRite/ByTe	Multiplexed write/byte
SYNC	1	SYNChronise	Bus cycle in progress
DIN	1	Data IN	Read strobe
DOUT	1	Data OUT	Write strobe
RPLY	1	RePLY	Slave handshake
IACKI	1	interrupt ACKnowledge In	Acknowledge daisy chain
IACKO	1	interrupt ACKnowledge Out	Acknowledge daisy chain
IRQ4	1	interrupt ReQuest	Request (lo priority)
IRQ5	1	interrupt ReQuest	
IRQ6	1	interrupt ReQuest	
IRQ7	1	interrupt ReQuest	Request (hi priority)

These lines are used in two ways:

1. Data transfer cycles between the PDP-11 host and the ABI.

2. Interrupt requests and acknowledges.

The Q-bus protocol for these transactions will now be described. Note that IMP does not use the DMA capability of the Q-bus, therefore the description below is not a complete description of Q-bus operation, but only an overview of those parts relevant to IMP. Fuller details may be found in [Dig79a].

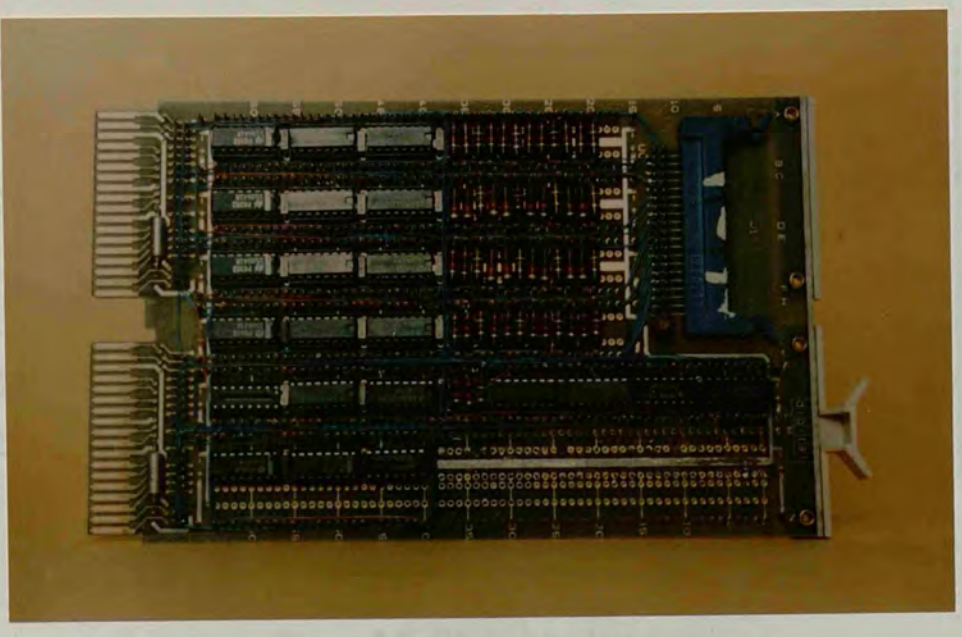


Figure 9.6: QQ prototype

9.7 Q-bus protocols

9.7.1 Data transfer cycles

The DTB transaction is shown in Figure 9.7

1. The master asserts DAL0-15 with the required address. If the address is within the I/O page (the top 8K byte of the memory map), it asserts BS7. If the cycle is a write, it asserts WTBT. After waiting 150ns minimum, the master asserts SYNC. This allows 75ns de-skew and 75ns setup time at the slave.

2. On the rising edge of SYNC, the slave latches the address, and if necessary WTBT.

3. After waiting a further 100ns minimum, the master removes the address from DAL0-15 and negates WTBT. This allows at least 25ns of hold time at the slave.

4. For a write, the master asserts DAL0-15 with the required data. If the cycle is a byte write, it asserts WTBT. After waiting at least 100ns, it asserts DOUT. This allows at least 25ns of setup time at the slave.

5. For a read, the master asserts DIN.

6. When ready, the slave asserts RPLY. At least 150ns will elapse at the slave between the assertion of RPLY and the negation of the strobe (DIN or

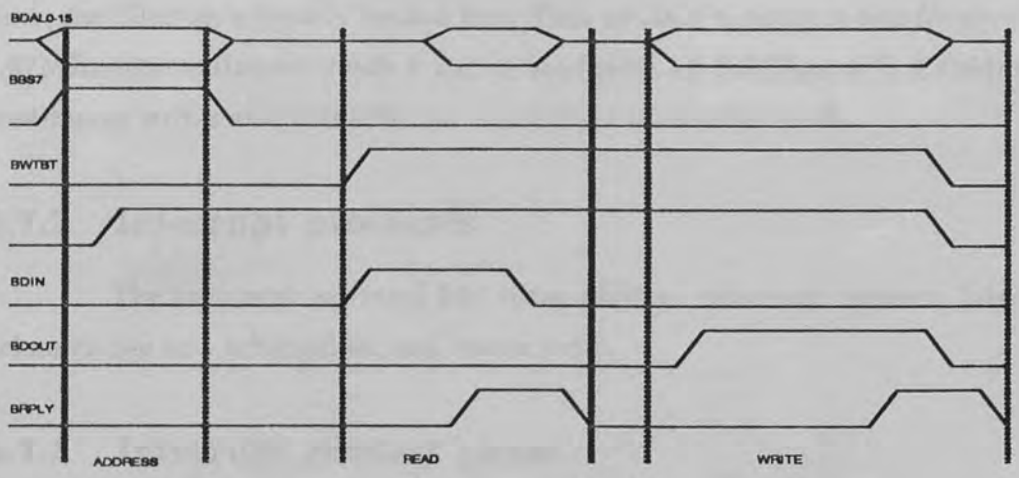


Figure 9.7: Qbus data transfer

DOUT).

7. For a read, data is latched at the master on the falling edge of the strobe. Because of setup time requirements, data must be presented by the slave not more than 125ns after the assertion of RPLY (*i.e.* this is an early acknowledge protocol). WTBT will remain valid at the slave for at least 25ns after the negation of the strobe.

8. For a write, data is guaranteed to be present at the slave buffers for at least 25ns after the negation of the strobe. WTBT also maintains its value for at least 25ns.

9. If this is a write-after-read cycle, SYNC will remain asserted and the data part of a write cycle will be performed. Otherwise SYNC will be negated.

10. SYNC must remain negated for at least 200ns. This means that at least 50ns dead time must occur before DAL0-15 can be asserted with the next address.

The minimum read time for Q-bus is 550ns + T_r, and for writes 650ns + T_r, where T_r is the response time of the slave. With current fast memories, it is quite possible to produce slaves that do all internal accesses during the allowed setup times of the protocol, and therefore appear to have zero response time. In this case Tr will consist of the time required for the strobe (DIN or DOUT) to propagate down the bus, be turned around onto RPLY and propagate back up. At a minimum, this would be two bus propagation times plus one gate propagation

time, say 50ns on a heavily loaded bus. This yields a maximum bus frequency of 1.67MHz for continuous reads (*i.e. a bandwidth of 3.3Mbyte* s^{-1}, 1.43MHz for continuous writes and 0.84MHz for continuous write-after-reads.

9.7.2 Interrupt protocols

The interrupt protocol has three phases: interrupt request; interrupt acknowledge and arbitration; and vector read.

9.7.3 Interrupt request phase

A device may assert an interrupt request at any time. Several devices may be requesting service at once because the request lines are open collector. The request codes are somewhat complicated by the need to maintain compatability across two versions of the Q-bus. Originally only one level of interrupt was provided, but with the release of the LSI 11/23 processor this was upgraded to 4. The earlier scheme supported only the lowest level (IRQ4) and so all higher requests must also assert IRQ4 to warn off earlier single level devices. To ease in decoding during interrupt acknowledgement, level 7 IRQs must also assert IRQ6. The full list of codes is:

4	IRQ4
5	IRQ4,IRQ5
6	IRQ4,IRQ6
7	IRQ4,IRQ6,IRQ7

These lines remain asserted until the request is acknowledged.

9.7.4 Interrupt acknowledge phase

At the end of each instruction, the master examines the state of the IRQ lines and compares the priority of any pending requests with its own execution priority as defined by the Processor Status Word (PSW). A request at a higher priority than the processor's will initiate an interrupt acknowledge transaction.

1. The master asserts DIN and at least 225ns later asserts IACKO. Note that SYNC is not asserted, and this may be used to differentiate between DTB read and interrupt acknowledge cycles.

2. The device electrically closest to the master receives the acknowledge on its IACKI receiver.

3. If not requesting an interrupt, the device asserts its IACKO line and thus propagates the acknowledge to the next device in the daisychain.

4. If the device is requesting an interrupt, it checks the IRQ lines to see if a higher level device is also requesting.

5. If no higher level request is present, the device blocks the acknowledge. (This is done by using the leading edge of DIN to clock a flip flop that disables the IACKO transmitter.) Arbitration is won, and the vector transfer phase starts.

6. If a higher level request exists, the device disqualifies itself (by clearing the blocking flip flop) and the acknowledge propagates to the next device.

9.7.5 Vector read phase

When arbitration has been successfully won, the device asserts RPLY, and within 150ns supplies an interrupt vector on DAL0-15. The master then reads the vector, and negates DIN and IACKO. The device then negates RPLY and within 100ns removes the vector. The master then uses the vector as the address of a two word area in memory containing the address of the device's service routine which is loaded into the program counter, and a new status word which is loaded to the PSW.

9.7.6 Q-bus interrupt protocol hazard

The Q-bus interrupt protocol is interesting because it effectively requires the device itself to perform priority arbitration. This is a sort of halfway house between the Unibus and VMEbus schemes. The Unibus requires the interrupter to become a bus master, and therefore all arbitration is done by the DTB arbitrator. The Q-bus requires the interrupter to decide for itself whether a received acknowledge is for it or another interrupter. The VMEbus interrupt fielder tells the interrupter which priority it is responding to by putting out a three-bit code on the address lines, so all the interrupter has to do is wait for an IACKI with a matching code on A2-A0. In fact the Q-bus protocol could also do this because not all 16 DAL lines are used for the vector (vectors are only allowed in the 512 bytes of memory, so only 9 DAL lines are needed for vector specification).

Because of the serial nature of the arbitration on the Q-bus, a potentially fatal race condition exists wherein a low priority interrupter near to the interrupt fielding processor may 'steal' a vector fetch cycle from a high priority interrupter

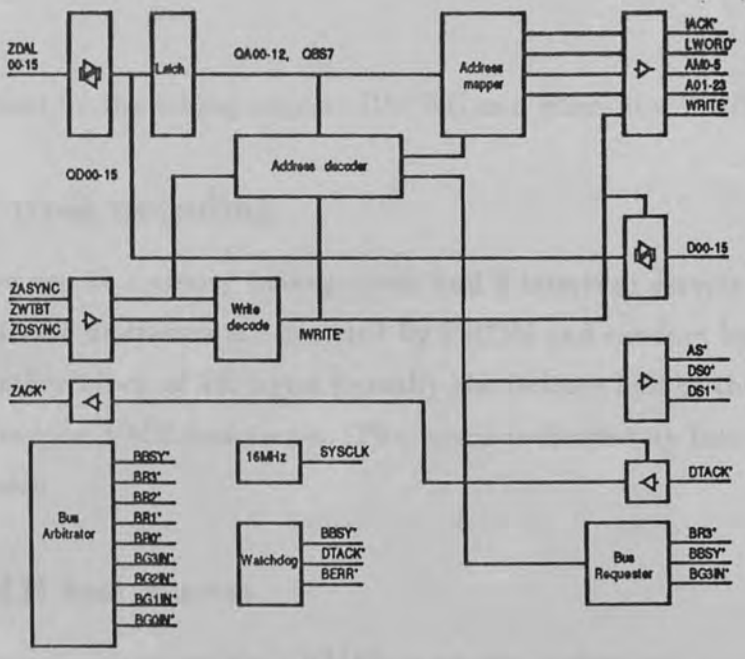


Figure 9.8: QV block diagram

further away. This is resolved by ensuring that high priority interrupters are geographically close to the interrupter.

9.8 QV card

The QV card forms the VMEbus end of the ABI link. The full design has been modified in later versions to remove the interrupt generation circuitry which turned out to be overkill for the current project.

QV provides controller functions for the VMEbus, such as clock generation and bus time-out watchdog, memory management hardware to convert Q-bus addresses to full 24-bit VMEbus addresses, and interrupt fielding on the VMEbus. The Q-bus data transfer and interrupt service protocols are converted to VMEbus protocols. A block diagram of the QV board is shown in Figure 9.8.

9.8.1 QV operation

9.8.2 Address and WTBT latches

The ABI DAL signals are buffered and passed to the address latches where the address/data information is demultiplexed under the control of ZASYNC. The ZWTBT line is buffered and latched in one half of a dual D-type flip-flop. A simple latch (such as those used for the DAL lines) is not sufficient because the latched read will need to be changed to a write signal at the end of the data strobe (DSYNC) during read-modify-write cycles. The other half of the

D-type is clocked by the falling edge of DSYNC and clears the WRITE flip-flop.

9.8.3 Address decoding

There are 32 memory management and 8 interrupt service registers on the board. Internal addresses are decoded by PROM and random logic.

A further block of 4K bytes (usually the bottom half of the Q-bus I/O page) is reserved for VME bus access. This space is divided up into 16 windows of 256 bytes each.

9.8.4 VME bus access

When an address within a VMEbus window is detected, a request is sent to the on-board VMEbus requester and when a grant is received the VMEbus buffers are enabled. Bits 1 to seven inclusive of the Q-bus address (corresponding to the address within the window) are connected directly to the low seven bits of the VME bus buffers. The WTBT line is decoded with bit zero of the Q-bus address to generate the correct set of data strobes for the VMEbus.

Bits eight through eleven are applied to the pair of 74LS621 address mappers which access internal registers to supply 24 bits of extra addressing information. This is connected to VME lines A8-23, AM0, LWORD* and IACK*.

9.8.5 Interrupt subsystem

Interrupts are latched by two AMD 9519A Universal Interrupt Controller chips. These sophisticated devices allow programmable edge detection on eight independent prioritised inputs and can be preloaded with vector information for the host. When an active edge is sensed by one of the interrupt controllers it sets an internal flag bit and asserts an open collector Interrupt Request line. This generates an interrupt cycle on the ABI. The host processor will read the vector information from the controllers automatically. Hence sixteen independently vectored interrupts are available. These are connected to (in decreasing priority order) SYSRESET*, SYSFAIL*, BCLR* BG3*-BG0*, IRQ7*-IRQ1* and BERR*. Using these interrupts, the host may monitor all transactions on the VMEbus above the level of individual data transfers (*i.e.* interrupts, system failures and bus master transfers).

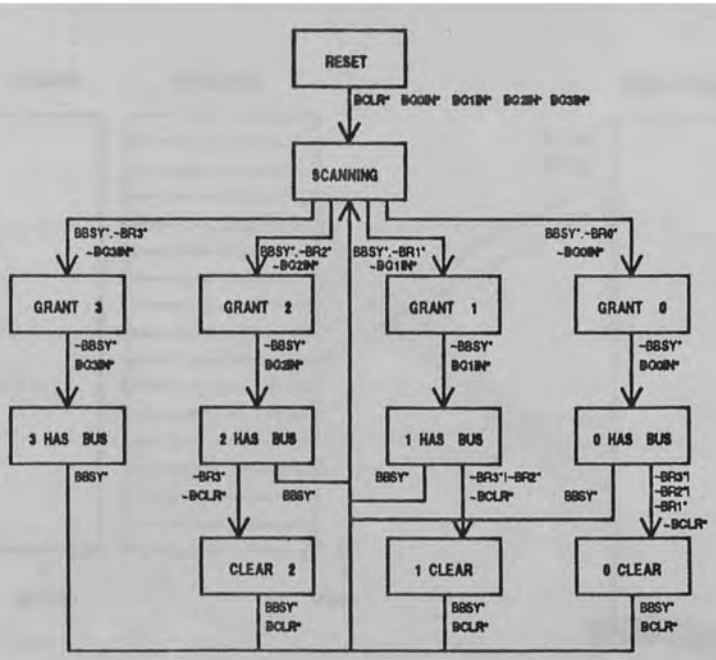


Figure 9.9: Arbitrator state diagram

9.8.6 Bus services

QV provides an arbitrator, a bus time-out watchdog and a system clock. These subsystems are completely independent of the other functions on the board and could be disabled if the Q-bus machine was to be used with a proprietary bus controller card.

The 16MHz clock is generated from a crystal oscillator and buffered directly onto the VMEbus. The clock is divided down by an eight-bit counter to provide a 16μs timer that is enabled at the start of each VMEbus address strobe. If a DTACK has not been received within that time, the counter asserts BERR* to indicate a bus time out.

The 16MHz clock drives the arbitrator which is a finite state machine implemented in a 85S105A Field Programmable Logic Sequencer. The state diagram is shown in Figure 9.9.

9.8.7 Software access to the VMEbus

The programmer's model of the Q-bus to VMEbus memory management is shown in Figure 9.10.

The sixteen VMEbus windows each have an associated VME Address Register (VAR), labelled VAR0–15. Each VAR is 24 bits long and is accessed *via* two Q-bus words. VARnLO holds bits 0–11 of the extension word, corresponding to VME bits A8–19. VARnHI holds bits 12–23 of the extension word corresponding to VME bits A20–23, AM0–5, LWORD* and IACK*.

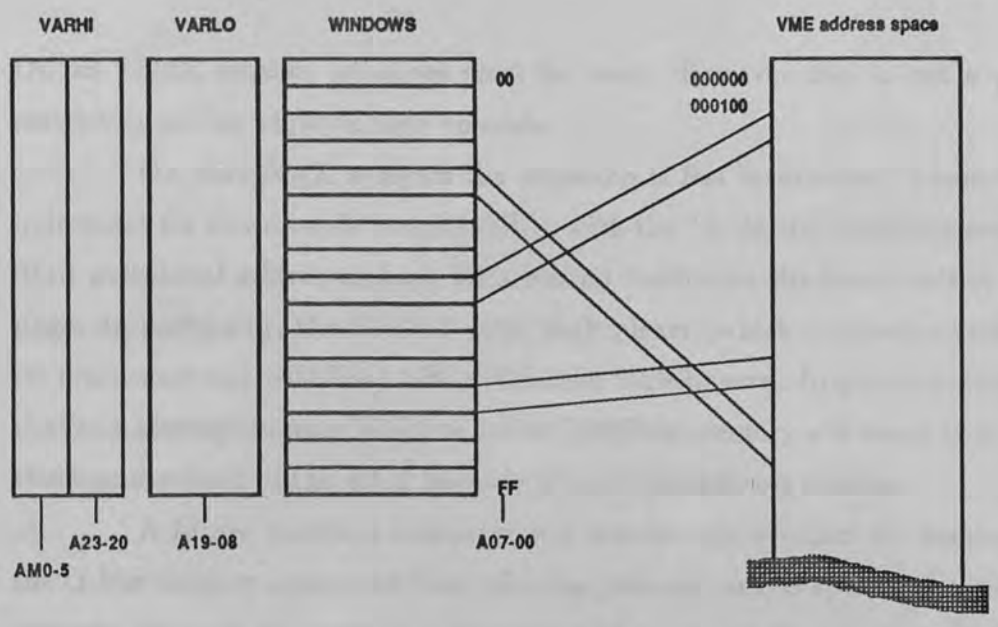


Figure 9.10: Memory management programmer's model

The sixteen windows may therefore be independently mapped to any 256 byte area on the VMEbus starting a 256 byte boundary. Any address modifier may be associated with the window.

As an example, suppose the Q-bus host needs to access a sixteen-bit word at VMEbus address $A9B115_{16}$ with address modifier $3C_{16}$, and that the first two VMEbus windows are already mapped.

In this case LWORD* and IACK* will be low, so the full 32-bit address will be $3CA9B115_{16}$. This address is divided into three fields — the top twelve bits, the middle twelve bits and the bottom eight bits, *i.e.* $3CA_{16}$, $9B1_{16}$ and 15_{16}. The twelve bit fields are loaded into VAR2HI and VAR2LO respectively. The Q-bus address is then `base + n × 256 + offset`, where `base` is the first location of VAR0, `n` is the number of the VAR and `offset` is the eight-bit field from the full VMEbus address.

9.8.8 Q-bus addressing conflicts

The QQ-QV board set occupies a large part of the Q-bus I/O page, and some care is required when configuring the system to avoid conflicts with other installed peripherals. On PDP-11 systems this is not usually a problem because when the LSI11/03 system was designed DEC allocated the bottom 4K bytes of the I/O page for extra memory (*i.e.* the LSI 11/03 had 60K of memory rather than the usual 56K). As a result all standard DEC peripherals are allocated space in the top half of the Q-bus to ensure compatability with the 11/03. Assuming the host processor is not an 11/03 therefore QV can safely use the 4K I/O space.

On an 11/03, smaller windows must be used. However this is not a practical restriction as the 11/03 is now obsolete.

On microVAX systems the situation is less favourable. There is no requirement for downwards compatability with the 11/03 and therefore some standard peripheral addresses have been moved down into the lower half of the I/O page. In particular, the DHV11 octal multiplexer (which is standard equipment on multi-user microVAXes) falls within the VAR2 space. In practice, this means that any attempt to map VAR2 to active VMEbus memory will result in a machine check and reboot of the VAX because of multiple address clashes.

A future protocol converter will resolve this problem by making use of the Q-bus memory space and thus relieving pressure on I/O space. The microVAX accesses its main memory in a separate address space that does not require Q-bus allocation, so there is some 4M bytes of free address space for mapping to the VMEbus. This will significantly increase throughput as remapping of registers will not be required unless more than 4M bytes of VMEbus space must be accessed. The current design works with a standard microVAX II as long as VAR2 is not mapped to populated VME locations.

9.9 BASE card

To ease interfacing to the VMEbus, a standard bus foundation module was designed which provides full buffering, a bus requester, an interrupt requester, and a simple synchronous bus slave protocol handler. Fuller details may be found in [Joh85]. Most of the functionality of the board is implemented in two 85S105A Field Programmable Logic Sequencers.

As well as being used to interface the hardwired coprocessors used in the factory application described below, the BASE circuitry has been incorporated into the SIPP [Edm88], SP1 and SP2 [Joh88b] microcoded processors. The slave protocol handler is also used in the V2 and V3 frame stores.

9.10 An industrial inspection application

A UK based food manufacturer approached the research group with a view to improving quality control on their production lines. A vision system was required that would gather statistics on product variability and remove bad products from the line. Central to the project was the requirement for 100% inspection of the line.

The resulting collaborative project between RHC, United Biscuits Ltd and Unilever Central Research Ltd provided much of the funding for the work described in this chapter. It was decided early on that a difficult problem be selected and a demonstrator system constructed and run on a real production line for a realistic period. The target processing rate was three products per second. In the event the system operated successfully for two weeks at a throughput of four per second using a PDP11/23 host. This was using a single frame store so that processing and image acquisition were not overlapped. Pipelining the acquisition and processing stages by using a second framestore and replacing the 11/23 with an 11/73 host would raise the throughput to over ten per second.

9.10.1 The problem

The product to be inspected was a circular chocolate coated sponge with a circular jam insert. An example is shown in Figure 3.6. The equipment was required to check for overall circularity, radius of the product, radius and concentricity of the jam insert, goodness of chocolate cover and gross failure, such as upside-down or broken products. Inspection of the jam distribution was particularly difficult because the chocolate coating was already in place.

9.10.2 Who did what

Dr E. R. Davies designed the algorithms to perform the inspection on static products which will be described below. He also designed two hardwired coprocessors at the block diagram level that implemented key parts of the algorithm at high speed. These processors were built by Dr M. Arain, and tested and debugged using routines written by E. R. Davies. The author designed and constructed the multiprocessor system that housed these coprocessors, the frame stores and camera hardware, the host protocol link and the interface for the hardware coprocessors to his system. He developed the image acquisition software that allowed rapidly moving objects to be captured and supplied to the rest of the system at a normalised position within the frame buffer in real-time. He also designed and implemented the system software, integrated the coprocessors and the software algorithms, and wrote the software to control the system in the factory which provided a front panel display and graphics to help demonstrate the system to management. The design decisions that resulted in this particular mix of software and hardware techniques are described in [DJ86] (a copy of which is bound in at the end of this thesis) and [DJ89]. As part of the evaluation process

the author wrote software simulators for the hardware coprocessors that may be dropped into the factory package as a direct replacement for the hardware control code. The author's factory package comprises some 3500 lines of code overall.

9.11 The algorithm

The algorithm comprises six phases: object detection and acquisition, thresholded edge detection, circle centre detection using a Hough Transform, *showthrough* (holes in the chocolate) inspection, jam inspection and pass/fail decision making.

9.11.1 Object detection

The V1 framestore allows host accesses during image acquisition. When used with a line scan camera, the line update counter is held at line 20 during object detection. The host monitors the horizontal flyback bit and reads the central part of the line during flyback. If it detects a dark area then the host begins the line counter until a whole object has been grabbed. The top edge of the object will therefore be fixed at line 20.

9.11.2 Edge detection

A Sobel operator is applied to the image. The results are thresholded so that (typically) about 100 points are marked as being strong edge points.

9.11.3 Centre detection

Starting at each edge point from the phase two list, the end point of a vector **r** (the radius of the circle) pixels along the normal to the direction of the edge is marked in an alternate edge space. At the end of this process, there will be a peak corresponding to the centre of the circle in Hough space, as shown in Figure 9.12.

9.11.4 Showthrough inspection

The calculated centre is used to position a circular mask over the product. All points falling within this mask are thresholded against a low and a high value. The high values show light patches which can be interpreted as holes in the chocolate (although there will be some contribution from specular reflection of the

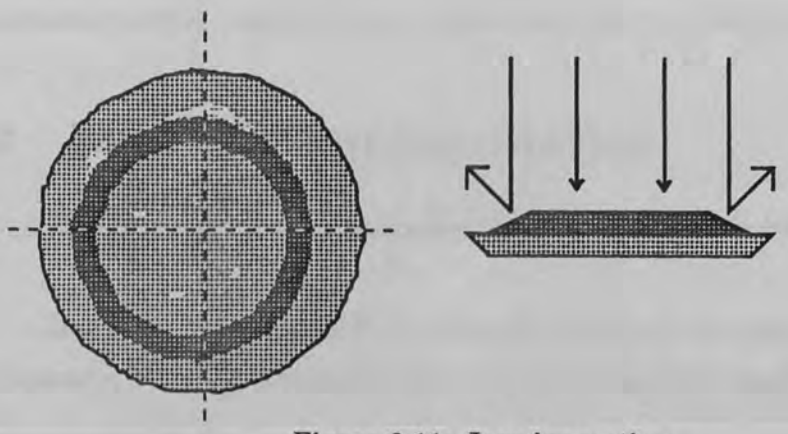


Figure 9.11: Jam inspection

lighting on the shiny chocolate). The dark area is used to control interpretation of the radial histogram.

9.11.5 Jam inspection

Inspection of the jam layer under the chocolate is difficult even for humans. However, Dr Davies observed that if the product is lit from above with a parallel light beam then the flat area of the product reflects light straight back to the camera. However the bevelled edges of the jam layer reflect light away from the camera. The result of this is that the machine 'sees' a dark ring on the product corresponding to the edge of the jam layer.

This ring is indistinct, so the Hough transform technique used to locate the product itself is not useful for the jam disc. Instead the radial grey scale histogram is collected. This will show a dip at the radius of a well formed jam layer. If the jam is not concentric with the product, or if some of the jam is missing then the dip will be smoothed out.

The histogram is correlated against a stored template for the 'ideal' product to provide an overall figure of merit for the jam position and size.

Correlation is performed by summing the pointwise products of the histogram bins over the range of the actual histogram, after the actual histogram has been normalised about the mean intensity.

9.11.6 Decision making

Management-supplied thresholds are supplied for circularity tolerance, radius, chocolate cover, jam figure of merit and dark ring area. Products are required to pass all these tests to proceed to the packing station. In a real sys-

tem, bad product would be blown off the line with a compressed air jet. In the demonstrator system, red and green lights were used to indicate pass/fail.

9.12 Real time implementation

The frame store provided some hardware assistance for the object locator.

Hardware coprocessor P1 performed the Sobel calculation on the image. Unfortunately, P1 only returned a list of coordinates that marked strong edges. The Hough transform requires the X and Y edge vectors so as to calculate the normal direction to the edge and these had to be recalculated by the host since P1 discarded the information. This incurred significant overhead ($\approx$24ms). A VLSI chip that can calculate Sobel components will be described in the next chapter.

Hardware coprocessor P2 provided a software controlled circular template used to specify the area of the image covered by the product. Within this area, light area, dark area and radial histogram information was collected in a single pass over the image. The template was also correlated on the fly during this scan.

The rest of the program was implemented in Pascal on the host, along with user interface functions.

9.13 Factory trial

The system was used on-line for two weeks. During this time there were visits by senior management and Unilever Central Research Ltd commissioned a short video showing the system in operation. The performance exceeded specification and we were able to gather useful statistics on product variability that had not been available using the existing batch sampling techniques.

9.14 Conclusions

This chapter has described a video-speed MIMD system and an application in the food processing industry. The system hardware has been in regular use for three years with modified versions running on VAX processors. A report commissioned by United Biscuits Ltd indicated that the commercial cost of producing the system would be in the range of £10,000 per unit. This would allow real time processing of grey scale images at the rate of around ten a second (depending

on the application). This is an advance on commercially available systems that are typically restricted to binary real-time processing after thresholding of the original grey scale image.

As well as improving quality control, systems such as IMP can control costs where expensive coatings such as chocolate are in use. Given that there will be variability of chocolate thickness due to environmental factors, the manufacturer must ensure that the worst case chocolate cover is still acceptable by the consumer. To do this, the mean of the distribution must be increased until the tail lies above the minimum acceptable level. Therefore, on average, chocolate cover will be better than required and this can be very expensive (at the time of the factory trial, processed chocolate cost around £2000 per ton). However, if system like IMP can guarantee to inspect every product and remove those that have thin chocolate coating then the mean of the distribution can be moved down to the point where chocolate wastage due to product removal matches the increased costs of thickening the chocolate layer.

9.15 Conclusions

A system capable of supporting real-time grey scale image processing has been designed and successfully demonstrated in a real factory environment. The system is easily expandable using commercial and in-house boards, and has been used as the foundation for other projects not described in this thesis.

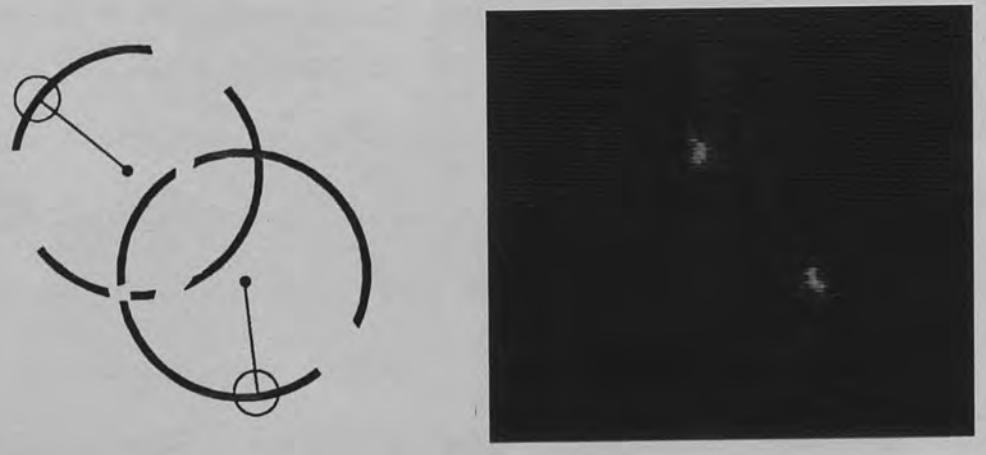


Figure 9.12: Centre detection

Chapter 10

A full custom VLSI SOBEL filter

10.1 Introduction

Edge enhancement is one of the most common operations in image processing. This chapter concerns the design and implementation of a VLSI Sobel filter (called SOBS-1) which can calculate differential gradient components with an internal propagation time of less than 10ns. It is designed to be used with an external lookup table in ROM which generates edge magnitude data.

10.2 Edge measurement operators

Edge measurement requires the calculation of both the magnitude and the gradient of the edge. The work described in the previous chapter recognised circles in the image using a Hough transform based algorithm. A fundamental requirement of such algorithms is a realtime edge detector with high angular accuracy. Several classes of edge enhancement operator have been developed:

1. template matching [Pre70,Rob77,NB80],
2. differential gradient calculation, [DH73,Rob65],
3. use of orthonormal basis functions [Hue71,Heu73],
4. difference of gaussians [MH80].

The template matching operators use a set of predetermined prototype edge masks and approximate the edge direction to that of the mask with the best match. To accurately approximate the edge direction, a large number of slightly different masks would be needed, however this implies elongated processing times. Here, speed militates against accuracy.

Hueckel's operators and the Marr-Hildreth edge detectors are computationaly intensive. [Hue71,Heu73] quotes runtimes of the order of 1.5 minutes on a DEC-10 for a 297 $\times$ 231 pixel array. The Marr-Hildreth operators require neighbourhoods of at least 35 $\times$ 35 pixels. Both classes of operator are currently uneconomic for realtime industrial applications.

The differential gradient operators attempt to calculate the x and y components of the gradient directly by convolving with X and Y masks and then taking the normalised root of the sum of the squares to find the edge magnitude. Essentially the operators attempt to fit a plane to the pixel intensities in the neighbourhood. This method is especially attractive for direction related applications as it directly provides the direction components. [Har80] suggests that the masks equivalent to the Prewitt operator are optimal for a three by three window, and that similar masks apply for larger areas, *e.g.* for a 5 $\times$ 5 neighbourhood:

$$M_x: \begin{pmatrix} -2 & -1 & 0 & 1 & 2 \\ -2 & -1 & 0 & 1 & 2 \\ -2 & -1 & 0 & 1 & 2 \\ -2 & -1 & 0 & 1 & 2 \\ -2 & -1 & 0 & 1 & 2 \end{pmatrix} \quad M_y: \begin{pmatrix} 2 & 2 & 2 & 2 & 2 \\ 1 & 1 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 \\ -1 & -1 & -1 & -1 & -1 \\ -2 & -2 & -2 & -2 & -2 \end{pmatrix}$$

In practice the Sobel operator seems to be preferred. [Dav84] suggests *circularity* as a criterion for testing the angular accuracy of differential gradient operators. Optimal (real number) mask coefficients are obtained by weighting according to the area of each pixel included within a circle enclosing the neighbourhood. According to this analysis, the angular response of the Sobel operator is optimal for masks with integer coefficients. The theory is attractive both because it offers a theoretical basis for the popularity of the Sobel operator and because it provides a rationale for the design of optimal edge operators using larger neighbourhoods which provide an increase in accuracy over the basic Sobel.

10.3 The Plessey edge detector

Plessey Semiconductors Ltd market an edge detection device (PDSP16401 [Sem86]) which uses template matching against the following four templates:

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

The first two masks are normalised by a factor 1.5 in an attempt to take account of the fact that the four-connected pixels are a factor $\sqrt{2}$ closer to the centre of the neighbourhood than the eight-connected pixels. Normalisation by 1.5 is of course only an approximation which eases the arithmetic implementation.

In operation, three ten-bit values representing the three lines of the current window are presented to the chip. Presumably the ten bits are intended to represent three three-bit pixels with one spare bit. Four separate FIR filters pass video data associated with the orientation of each mask. The outputs are then sorted to produce a three-bit word giving a one-of-eight approximation to the edge direction. The 13-bit output of the filter generating the strongest output is available at the chip outputs, and the most significant ten bits are compared against an externally provided threshold. If the output exceeds the threshold level, a chip output goes high. The chip can cycle at 15MHz, fast enough to process 1024 $\times$ 1024 pixel images in real time, but 20 cycles are required to process each set of results. This twenty cycle pipeline delay will complicate image buffer addressing.

Although the chip is fast and provides an on-chip comparator (thus saving one external package), it is based on theoretically unsound principles, and only calculates approximate normalised convolutions. Even assuming that the resulting responses are accurate, the one-of-eight direction indicator introduces an angular error of up to 22.5°. Convolution errors when the actual edge direction is midway between masks will add to this error. The chip processes pixels to only eight grey-level accuracy and does not provide internal pipelining of pixel fetches (as opposed to the processing pipeline), so considerable external support circuitry will be required.

10.4 SOBS-1 design derivation

The architecture of the SOBS-1 chip is derived from that of the P2 hardware processor designed by E R Davies for the IMP system application described in the last chapter. The chip implements a large array of adders along with a three by three stage pixel pipeline. Unlike the P2 implementation, SOBS-1 maintains full arithmetic precision throughout and generates 10-bit component outputs. Of itself this would be accurate enough for Hough transform circle detection to be performed for circles up to 2048 pixels in diameter in ideal conditions of zero noise. The P2 implementation is limited to 128 pixel diameter circles. To fully exploit the available precision a square root lookup table is required with a address lines where a is the number of significant bits taken from the adders. To fully exploit

the available precision a 1M × 10-bit lookup table is required, but in general a $2^{(\log n)+1} \times n$ bit table will be required for circles of diameter $2n$ pixels.

SOBS-1 also provides a prefetch stage on the input pipeline allowing a column of pixels to be preloaded before cycling the pipeline.

The full P2 system includes a set of comparators for detecting points in the image with large edge magnitudes and a coordinate RAM which is loaded during a picture scan with a list of the coordinates of such out-of-threshold points. These functions are not included in the present design, mainly due to chip pinout restrictions.

10.5 SOBS-1 arithmetic section

The Sobel convolution masks are:

$$S_x : \begin{pmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{pmatrix} \quad S_y : \begin{pmatrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{pmatrix}$$

If we number the pixels thus:

```
P4 P3 P2
P5 P0 P1
P6 P7 P8
```

then the mask equations are

$$S_x = P2+2\times P1+P8-(P4+2\times P5+P6) \quad S_y = P4+2\times P3+P2-(P6+2\times P7+P8)$$

These may be generated using two trees of adders and subtracters. The multiply by two is simply a shift left. In a combinatorial implementation, a shift left is obtained simply by connecting the adders with a one bit offset.

10.6 Pixel pipelining

A note by Lee [Lee83] with later expansion by Picton [Pic84] describes elimination of redundant arithmetic operations in the Sobel filter. If the simple Sobel convolutions are applied across an entire image then some terms are calculated more than once. An obvious example is the lower (negative) partial sum in S_y for line $y = n$ which will be the same as the upper (positive) partial sum in S_y for line $y = n + 2$. Throughput on a conventional sequential processor would

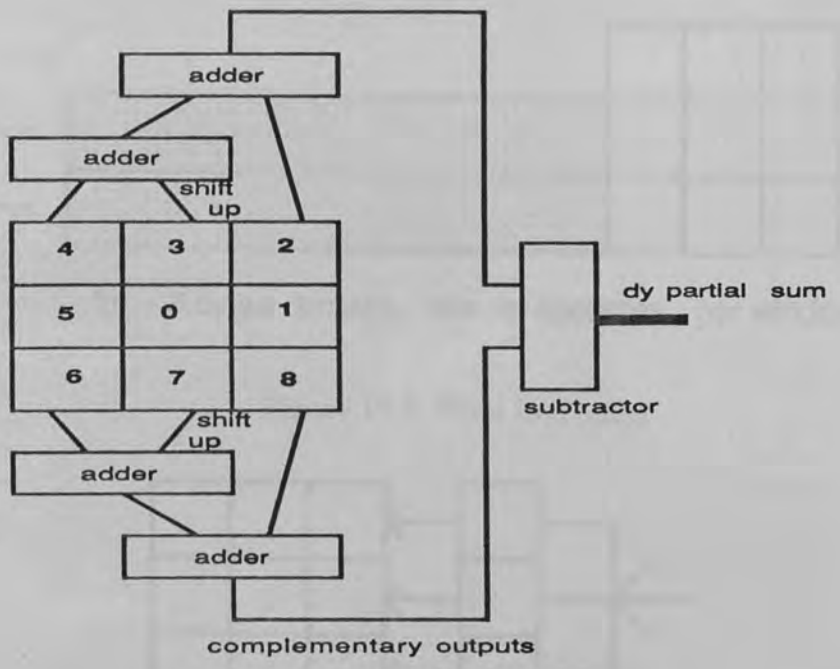


Figure 10.1: SOBS-1 arithmetic trees

be increased if these terms were stored and retrieved when necessary rather than being recalculated. However, the arithmetic structure in SOBS-1 calculates all terms combinatorially in parallel, and there is very little speed to be gained by storing intermediate results for later operations. The overhead in storage and sequencing would be high.

Given that SOBS-1 calculates results when needed, a bottleneck exists at the inputs where (potentially) nine pixel fetches are required from the framestore for each operation. This is unacceptable because the framestore access time is likely to be of the order of 50–100ns, and so a complete filter operation (with subsequent writing of results to an output frame buffer) will require 0.5–1μs. This is a gross mismatch with the speed of the arithmetic trees (~10ns). However, it should be remembered that Sobel magnitudes must be fetched from a ROM lookup table which will also limit the actual system speed. Large EPROMs and ROMs are widely available at speeds of around 250ns, and this provides the target speed for the pixel fetch circuitry.

Pixel I/O may be reduced by buffering of image lines. If two complete lines + 3 pixels are stored in shift registers internal to the chip then only one pixel read from the frame store is required per filter operation. This scheme requires a great deal of on-chip storage and produces a design that requires expensive high speed ROMs to make use of the extra speed. A better match to the ROM speed is obtained using a simple window-column buffer comprising three separate three

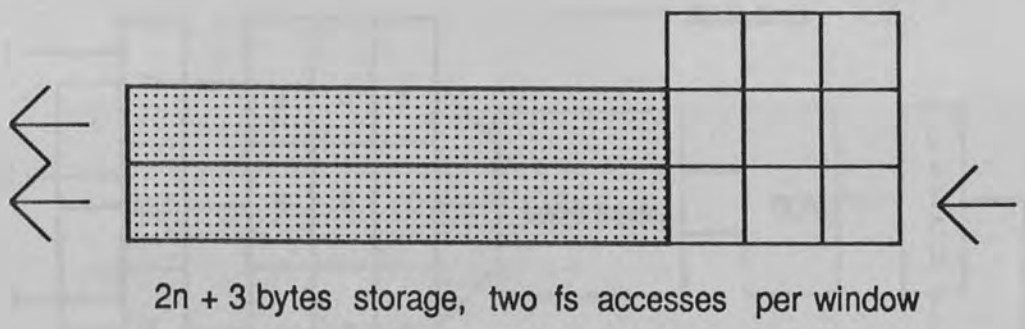


Figure 10.2: Pixel line buffer

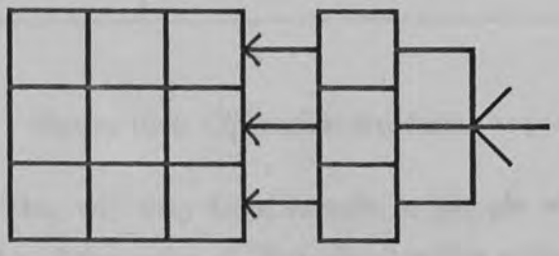


Figure 10.3: Window-column buffer

pixel-pipelines.

Three pixel reads are now required for each operation. Together with the output write this requires four frame store cycles, *i.e.* 268ns for a 67ns access time frame store (*i.e.* 512 × 512 square pixel speed). This matches the 250ns access time of the ROM well, and provides compact VLSI implementation. This scheme is used in SOBS-1. In a system with two frame buffers or one where the results are being written to a different space, the write cycle may be further overlapped with one of the read cycles giving a 201ns overall cycle. Naturally a faster ROM would be required in this case.

10.7 Operation pipelining

The propagation delay of the adder circuitry is negligible compared with the cycle time of the system, so no internal pipelining is required. However, prefetch buffering of the next column of pixels is required so that frame store accesses may be fully overlapped with the ROM lookup. One way of implementing this would be to latch the two twenty-bit differential gradient outputs at the inputs to the ROM. However, to maintain maximum throughput the output latch would need to be activated exactly when the outputs from the arithmetic tree

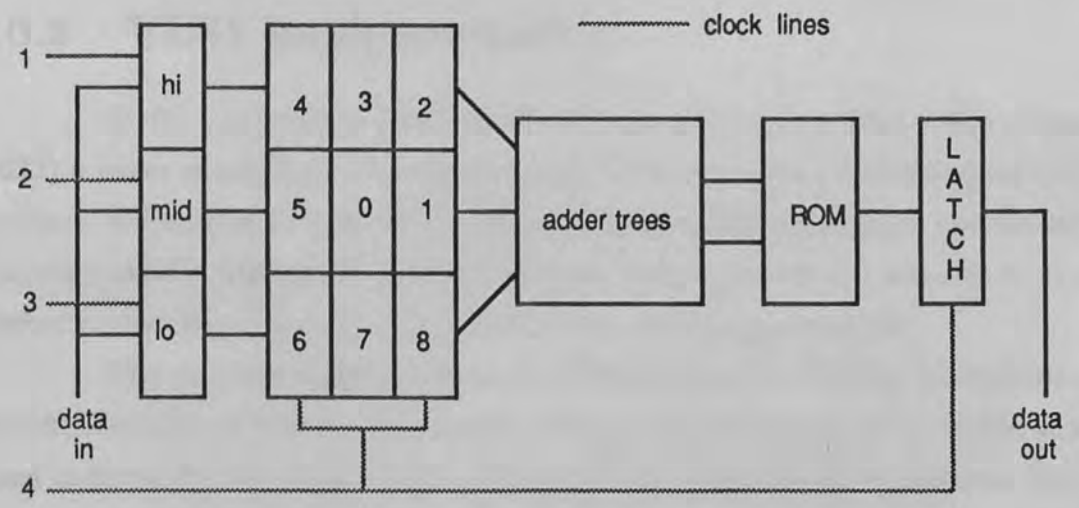


Figure 10.4: Operation pipelining sequence

first stabilised. This will vary from sample to sample of the chip, and in any case is likely to be of the order of 10ns after loading of the last pixel in the new column. Accurate generation of 10ns delays on-chip would be difficult and in practice the whole system would have to be slowed down to allow for variations due to temperature and processing factors.

The solution is to put the operation pipeline latch at the inputs to the device and an output latch after the ROM. The filter chip works on a four phase cycle. During phases one, two and three the pixels that will form P2, P1 and P8 of the next calculation are loaded into the prefetch latches. During phase four the prefetch latch and the three level pipelines are connected as four-pixel shift registers and data shifted across one column. At the beginning of phase four, the ROM output latch is clocked to load the result of the current operation and during phase four this value is written back to the frame store.

Many bus interfaces include output data latches in the output buffers, and this provides the SOBS-1 output latch 'for free'.

10.8 TTL equivalent chip count

SOBS-1 contains twelve 8-bit latches and ten 11-bit adders along with a small amount of clock driver circuitry. Allowing for the fact that two of the 11-bit adders could be implemented as 8-bit adders without loss of arithmetic accuracy, SOBS-1 is roughly equivalent to twelve 74LS374 octal latches, sixteen 74LS83A four-bit adders and a buffer (*e.g.* 74LS245) to provide clock drive.

10.9 VLSI implementation

SOBS-1 is built in 2 micron CMOS using European Silicon Structures (ES2) e-beam processing. It requires about 4,200 transistors (not counting I/O buffers). It fits on a 10^2mm die, which is small by modern standards and should therefore exhibit high yield. (Yield decreases with increasing die size due to the higher probability of including a crystal defect within a given chip).

The chip was designed using the ISIS software marketed by Racal-Redac [Rac87] which is a commercial version of the software developed by Inmos and used in-house for the design of the Transputer and other advanced devices. ISIS was selected by the Alvey directorate as one of the preferred tools on the Alvey VLSI projects and is regarded as one of the most advanced toolsets available for full custom VLSI design.

ISIS comprises a Hardware Description Language (HDL) used to describe the interconnectivity of transistors and higher level modules, a simulator called Hylas (HYbrid Logic and Analogue Simulator) which supports mixed mode simulation, and a layout package called HED (Hierarchical EDitor). The system imposes a strict hierarchy on the design process and in general attempts to apply the techniques of large scale software engineering to the process of hardware design.

10.10 ISIS concepts

ISIS HDL is a BCPL like language used to describe the network of elements and wires that makes up the chip. It is effectively a data description language, not an executable programming language. For example, an HDL DO-loop specifies the instantiation of multiple copies of a piece of hardware, not the repeated execution of one step in an algorithm.

Blocks of hardware in HDL are represented using a MODULE which roughly corresponds to a procedure in a programming language. Some modules are primitives supplied by the system such as transistors and resistors. Primitive modules are not decomposable in much the same way that FORTRAN intrinsics or Pascal predefined routines are monolithic. Other modules are formed by connecting modules together in a hierarchical fashion.

Each HDL module has a parameter list which describes the connections to the module. HDL modules map onto rectangular, non-overlapping areas of silicon at layout time. All the components (and their interconnects) listed in the

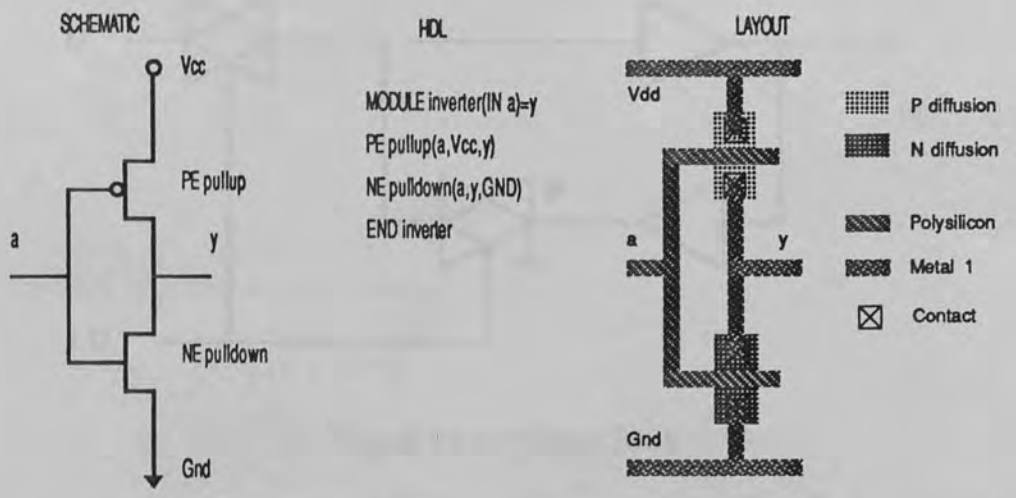


Figure 10.5: Inverter representations

HDL module must be contained within the layout module. Connections to the module correspond to wires crossing the boundaries of a layout module and these are often referred to as 'bristles'.

Interconnection is established in HDL by associating actual signal variables with the formal parameters in a module's list of bristles. The global (system) signals GND and Vdd are available throughout the circuit for power supply connections.

This example shows the schematic, HDL and symbolic layout representations for a simple CMOS inverter. The HDL calls two modules NE (N-channel Enhancement) and PE (P-channel enhancement) which are primitives supplied by the system:

10.11 SOBS-1 HDL implementation

10.12 Leaf cells

An ISIS design comprises a hierarchy of modules and routing buffers which provide interconnection at the different levels of the hierarchy. At the bottom of the hierarchy are cells containing only primitives with no calls to non-primitive modules. These are called leaf cells because they correspond to the leaves of the hierarchy tree. SOBS-1 contains only three types of leaf cell: a D-type latch, a full adder and an inverter. The basic inverter has been described above. The inverters used in SOBS-1 are more complex than that described above because they are essentially acting as buffers and contain an array of scaled

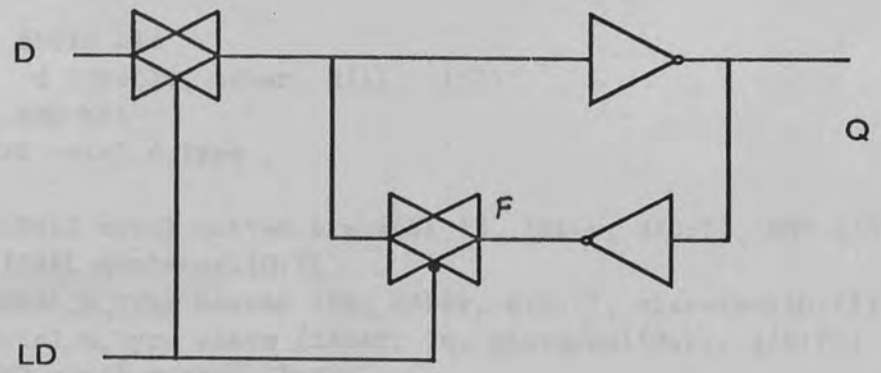


Figure 10.6: D-type latch

transistors: space precludes a detailed description of the design principles. In this section the modules containing logic transistors only will be described, and analogue engineering details will be omitted.

10.12.1 D-type latch

CMOS permits the design of compact latches using analogue switches and inverters rather than the conventional cross-coupled NAND gates. The d-type used in this design is a standard eight transistor circuit [WE85].

When ld is true then transmission gate **D** is open and transmission gate **F** is closed. Data flows through inverter **D** and the output follows the input. When ld is false, transmission gate **D** closes and **F** opens forming a feedback loop through the two inverters which latches the data.

The HDL representation of this circuit is:

```
// Transmission gate D latch
MODULE d_type(IN ld, ldbar, d, OUT q)
 SIGNAL id, qbar
id=UPE fp (ld, q)
id=UNE fn (ldbar, q)
id=UPE dp (ldbar, d)
id=UNE dn (ld, d)
PE i1p (id, Vdd, qbar)
NE i1n (id, qbar, Gnd)
PE i2p (qbar, Vdd, q)
NE i2n (qbar, q,Gnd)
END d_type
```

The D-types are combined into an octal d-type module. The prefetch and pipeline blocks are built from master-slave sections comprising two octal d-types.

```
MODULE octal_d_type(IN ld, ldbar, d[0:7], OUT q[0:7])
 FOR i=[0:7] DO
```

```
  BEGIN bit
   d_type(ld, ldbar, d[i], q[i])
  END bit
END octal_d_type

MODULE octal_master_slave(IN ld, ldbar, d[0:7], OUT q[0:7])
SIGNAL qinternal[0:7]
octal_d_type master (ld, ldbar, d[0:7], qinternal[0:7])
octal_d_type slave (ldbar, ld, qinternal[0:7], q[0:7])
END octal_master_slave

// Three octal ms D-types to hold incoming data

MODULE prefetch_latch
        (IN ld[0:2], ldbar[0:2], d[0:7], OUT q[0:23])
 octal_master_slave low (ld[0], ldbar[0], d[0:7], q[0:7])
 octal_master_slave mid (ld[1], ldbar[1], d[0:7], q[8:15])
 octal_master_slave hi (ld[2], ldbar[2], d[0:7], q[16:23])
END prefetch_latch

// Three by three window pipeline

MODULE pipeline(IN ld, ldbar, d[0:23],
                OUT p1[0:7], p2[0:7], p3[0:7], p4[0:7],
                    p5[0:7], p6[0:7], p7[0:7], p8[0:7])
SIGNAL p0[0:7]
 octal_master_slave p4latch(ld,ldbar,p3[0:7],p4[0:7])
 octal_master_slave p3latch(ld,ldbar,p2[0:7],p3[0:7])
 octal_master_slave p2latch(ld,ldbar,d[16:23],p2[0:7])
 octal_master_slave p5latch(ld,ldbar,p0[0:7],p5[0:7])
 octal_master_slave p0latch(ld,ldbar,p1[0:7],p0[0:7])
 octal_master_slave p1latch(ld,ldbar,d[8:15],p1[0:7])
 octal_master_slave p6latch(ld,ldbar,p7[0:7],p6[0:7])
 octal_master_slave p7latch(ld,ldbar,p8[0:7],p7[0:7])
 octal_master_slave p8latch(ld,ldbar,d[0:7],p8[0:7])
END pipeline

// Central register block with prefetch and window

MODULE registers (IN ld[0:3], ldbar[0:3], d[0:7],
                  OUT p1[0:7], p2[0:7], p3[0:7], p4[0:7],
                      p5[0:7], p6[0:7], p7[0:7], p8[0:7])
SIGNAL id[0:23]

prefetch_latch(ld[0:2], ldbar[0:2], d[0:7], id[0:23])
pipeline(ld[3], ldbar[3], id[0:23],
         p1[0:7], p2[0:7], p3[0:7], p4[0:7],
         p5[0:7], p6[0:7], p7[0:7], p8[0:7])
END registers
```

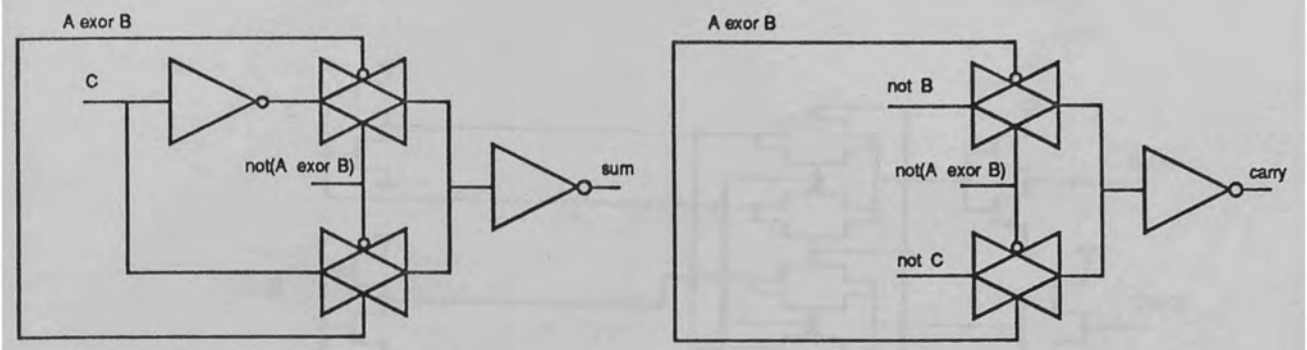


Figure 10.7: Transmission gate adder

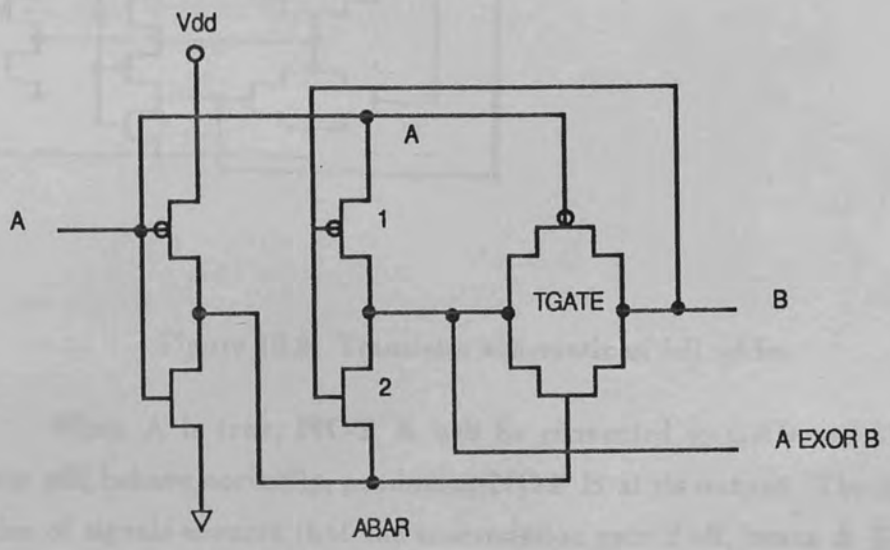


Figure 10.8: Transmission XOR gate

10.12.2 Full adder

The full adder is based on a transmission gate adder reported in [SOA73]. It uses a novel transmission XOR gate and a pair of multiplexers connected as shown in Figure 10.7

In a full adder, the following relations are true:

When A XOR B is true: sum = NOT C, CARRY = C

When A XOR B is false: sum = C, CARRY = B

The XOR gate is also based on transmission gates but uses an unusual form of pseudo-inverter.

The circuit comprises two inverter structures followed by a transmission gate. The second inverter structure is connected between the signals **A** and **NOT A** instead of between the supply rails as is normal. The input to this inverter is **B**.

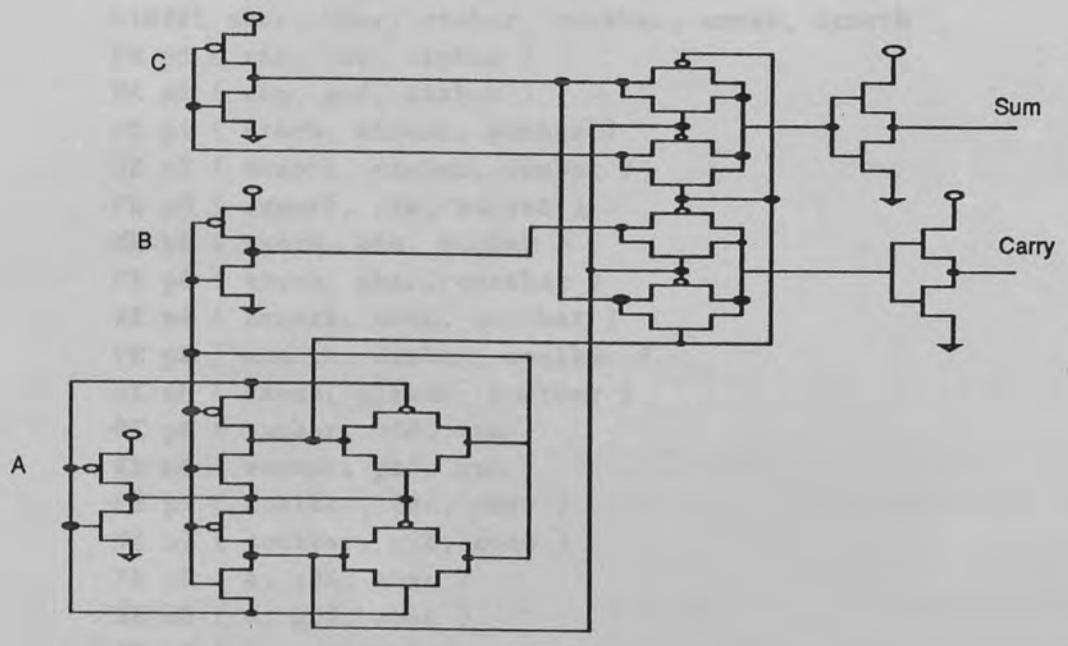


Figure 10.9: Transistor schematic of full adder

When **A** is true, **NOT A** will be connected to GND and the second inverter will behave normally, producing **NOT B** at its output. The same combination of signals ensures that the transmission gate if off, hence **A XOR B = NOT B** when **A** is true.

When **A** is false, **ABAR** will be true and the 'supply' connections to the second inverter structure will be reversed. This effectively disables the inverter and no output is produced. The transmission gate switches on, hence **A XOR B = B** when **A** is false.

An XNOR gate may be constructed by reversing the connections of **A** and **NOT A** to the second 'inverter'.

The complete full adder requires 24 transistors.

A 'TTL' style CMOS adder (*i.e.* direct implementation of Figure 7.1) with active low outputs may be constructed in CMOS using 24 transistors but the present circuit provides true outputs and has a balanced propagation delay through the carry and sum paths. The conventional adder has a longer delay through the carry path.

The HDL for the adder is simply a block of 24 transistors:

```
// Monolithic transmission gate adder
```

```
MODULE full_add ( IN a, b, cin, OUT sum, sumbar, cout )
 SIGNAL abar, bbar, cinbar, coutbar, axorb, axnorb
 PE p1 ( cin, vdd, cinbar )
 NE n1 ( cin, gnd, cinbar )
 PE p2 ( axorb, cinbar, sumbar )
 NE n2 ( axnorb, cinbar, sumbar )
 PE p3 ( axnorb, cin, sumbar )
 NE n3 ( axorb, cin, sumbar )
 PE p4 ( axorb, abar, coutbar )
 NE n4 ( axnorb, abar, coutbar )
 PE p5 ( axnorb, cinbar, coutbar )
 NE n5 ( axorb, cinbar, coutbar )
 PE p6 ( sumbar, vdd, sum )
 NE n6 ( sumbar, gnd, sum )
 PE p7 ( coutbar, vdd, cout )
 NE n7 ( coutbar, gnd, cout )
 PE p8 ( a, vdd, abar )
 NE n8 ( a, gnd, abar )
 PE p9 ( b, a, axorb )
 NE n9 ( b, axorb, abar )
 PE p10 ( b, abar, axnorb )
 NE n10 ( b, axnorb, a )
 PE p11 ( a, axorb, b )
 NE n11 ( abar, axorb, b )
 PE p12 ( abar, axnorb, b )
 NE n12 ( a, axnorb, b )
 END full_add
```

The adders are combined into a universal 11-bit adder. This may be used as an 11-bit subtracter by inverting one set of inputs and setting the least significant carry-in to 1. Complementary outputs are available from the adder to assist in this function. The next higher module (MODULE one_two_one) implements the $X + 2 \times Y + Z$ function needed for each Sobel partial sum.

```
// Standard eleven bit adder

MODULE add_11(IN a[0:10], b[0:10], cin, OUT sum[0:10],
              sumbar[0:10])
 SIGNAL ic[-1:10]

 ic[-1]=cin
 FOR i=[10:0] DO
  BEGIN adder
   full_add(a[i],b[i],ic[i-1],sum[i],sumbar[i],ic[i])
  END adder
END add_11

// a+2*b+c function
```

```
MODULE one_two_one(IN a[0:7],b[0:7],c[0:7], OUT sum[0:10],
                   sumbar[0:10])
 SIGNAL isum[0:10]
 add_11 one_two ([a[0:7],GND,GND,GND],[GND,b[0:7],GND,GND],
                 GND,isum[0:10],[?,?,?,?,?,?,?,?,?,?,?])
 add_11 one     (isum[0:10],[c[0:7],GND,GND,GND],GND,
                 sum[0:10],sumbar[0:10])
END one_two_one
```

10.13 Global interconnection

The top level of the chip hierarchy implements the block diagram of Figures 10.1 and 10.3. The module partitioning is not intuitive in the HDL. The register block is grouped with the D_x partial sum blocks (*i.e.* instances of MODULE one_two_one). The next level up includes the D_y partial sum blocks and the top-most level (MODULE sobs) incorporates the subtracters that provide the final D_x and D_y results. The hierarchy is arranged this way to ease the global routing on the silicon. The ISIS routing tools work best when connecting small numbers of blocks in a very hierarchical fashion and the present arrangement reflects the spatial relationship of the layout modules.

```
MODULE regs_and_dx(ld[0:3], ldbar[0:3], d[0:7],
                   p4[0:7], p3[0:7], p2[0:7],
                   p6[0:7], p7[0:7], p8[0:7],
                   dx_hi_bus[0:10], dx_lo_bus_bar[0:10])

 SIGNAL p5[0:7], p1[0:7]

 registers (ld[0:3], ldbar[0:3], d[0:7],
            p1[0:7], p2[0:7], p3[0:7], p4[0:7],
            p5[0:7], p6[0:7], p7[0:7], p8[0:7])
 one_two_one dx_hi (p2[0:7],p1[0:7],p8[0:7],
         dx_hi_bus[0:10],[?,?,?,?,?,?,?,?,?,?,?])
 one_two_one dx_lo (p6[0:7],p5[0:7],p4[0:7],
         [?,?,?,?,?,?,?,?,?,?,?],dx_lo_bus_bar[0:10])
END regs_and_dx

MODULE regs_and_dy(ld[0:3], ldbar[0:3], d[0:7],
                   dy_hi_bus[0:10], dy_lo_bus_bar[0:10],
                   dx_hi_bus[0:10], dx_lo_bus_bar[0:10])

SIGNAL
  p4[0:7], p3[0:7], p2[0:7],
  p6[0:7], p7[0:7], p8[0:7]

 regs_and_dx(ld[0:3], ldbar[0:3], d[0:7],
```

```
            p4[0:7], p3[0:7], p2[0:7],
            p6[0:7], p7[0:7], p8[0:7],
            dx_hi_bus[0:10], dx_lo_bus_bar[0:10])
 one_two_one dy_hi (p4[0:7],p3[0:7],p2[0:7],
            dy_hi_bus[0:10],[?,?,?,?,?,?,?,?,?,?,?])
 one_two_one dy_lo (p8[0:7],p7[0:7],p6[0:7],
            [?,?,?,?,?,?,?,?,?,?,?], dy_lo_bus_bar[0:10])
END regs_and_dy

//Main function block

MODULE sobs(IN d[0:7], ld[0:3], OUT dx[0:10], dy[0:10])
 SIGNAL
  ldbar[0:3],
  dx_hi_bus[0:10], dx_lo_bus_bar[0:10],
  dy_hi_bus[0:10], dy_lo_bus_bar[0:10]
// FOR i=[0:3] DO ldbar[i]=invert(ld[i])
 regs_and_dy(ld[0:3], ldbar[0:3], d[0:7],
            dy_hi_bus[0:10], dy_lo_bus_bar[0:10],
            dx_hi_bus[0:10], dx_lo_bus_bar[0:10])
 add_11 dx_sub (dx_hi_bus[0:10], dx_lo_bus_bar[0:10], Vdd,
               dx[0:10], [?,?,?,?,?,?,?,?,?,?,?])
 add_11 dy_sub (dy_hi_bus[0:10], dy_lo_bus_bar[0:10], Vdd,
               dy[0:10], [?,?,?,?,?,?,?,?,?,?,?])
END sobs
```

10.14 Simulation results

SOBS-1 has been extensively simulated at full circuit level. Switch level simulators have difficulty handling transmission gate intensive circuits where the direction of current flow through the gate may change. The Suzuki adder is particularly troublesome. As a result, full analogue simulation was required to get realistic results. MODULE sobs (*i.e.* the whole chip less the clock drivers) requires approximately 9 cpu minutes on a microVAX II to simulate 1ns of realtime operation. It turns out that the propagation time of the chip is about 12.5ns and therefore $128 \times 128 \times 12.5 = 204\mu$s of real-time simulation would be needed to process a complete image. This would require about 3.5 CPU years.

The simulator output below shows a complete simulation for the four adjacent pixels B1, B2, B3 and B4 shown in Table 10.1.

Simulator output for SOBS-1 follows table 10.1

			column				
	1	2	3	4			
0	0	0	27	140	240	A	
0	0	0	127	250	229	B	row
0	0	0	255	135	5	C	

Table 10.1: SOBS-1 test data

d[7:0]	sobs.d[7:0]
ld[3:0]	sobs.ld[3:0]
ldbar[3:	sobs.ldbar[3:0]
id[7:0]	sobs.regs_and_dy.regs_and_dx.registers.id[7:0]
id[15:8]	sobs.regs_and_dy.regs_and_dx.registers.id[15:8]
id[23:16	sobs.regs_and_dy.regs_and_dx.registers.id[23:16]
p8[7:0]	sobs.regs_and_dy.regs_and_dx.registers.pipeline.p8[7:0]
p1[7:0]	sobs.regs_and_dy.regs_and_dx.registers.pipeline.p1[7:0]
p2[7:0]	sobs.regs_and_dy.regs_and_dx.registers.pipeline.p2[7:0]
p7[7:0]	sobs.regs_and_dy.regs_and_dx.registers.pipeline.p7[7:0]
p0[7:0]	sobs.regs_and_dy.regs_and_dx.registers.pipeline.p0[7:0]
p3[7:0]	sobs.regs_and_dy.regs_and_dx.registers.pipeline.p3[7:0]
p6[7:0]	sobs.regs_and_dy.regs_and_dx.registers.pipeline.p6[7:0]
p5[7:0]	sobs.regs_and_dy.regs_and_dx.registers.pipeline.p5[7:0]
p4[7:0]	sobs.regs_and_dy.regs_and_dx.registers.pipeline.p4[7:0]
isum[10:	sobs.regs_and_dy.regs_and_dx.dx_hi.isum[10:0]
isum[10:	sobs.regs_and_dy.regs_and_dx.dx_lo.isum[10:0]
isum[10:	sobs.regs_and_dy.dy_hi.isum[10:0]
isum[10:	sobs.regs_and_dy.dy_lo.isum[10:0]
dx_hi_bu	sobs.dx_hi_bus[10:0]
dx_lo_bu	sobs.dx_lo_bus_bar[10:0]
dy_hi_bu	sobs.dy_hi_bus[10:0]
dy_lo_bu	sobs.dy_lo_bus_bar[10:0]
dx[10:0]	sobs.dx[10:0]
dy[10:0]	sobs.dy[10:0]

	d	l	l	i	i	i	p	p	p	p	p	p	p	p	p	i	i	i	i	d	d	d	d	d	d
	[	d	d	d	d	d	d	8	1	2	7	0	3	6	5	4	s	s	s	s	x	x	y	y	x	y
	7	[	b	[	[	[	[	[	[	[	[	[	[	[	[	[	u	u	u	u	-	-	-	-	[	[
	:	3	a	7	1	2	7	7	7	7	7	7	7	7	7	7	m	m	m	m	h	l	h	l	1	1
	0	:	r	:	5	3	:	:	:	:	:	:	:	:	:	:	[	[	[	[	i	o	i	o	0	0
	]	0	[	0	:	:	0	0	0	0	0	0	0	0	0	0	1	1	1	1	_	_	_	_	:	:
	]	3	]	8	1	]	]	]	]	]	]	]	]	]	]	0	0	0	0	b	b	b	b	0	0	
	:		]	6													:	:	:	:	u	u	u	u	]	]
0.000	0	LLLL	HHHH	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
500.0000p	0	LLLL	HHHH	0	0	0	0	0	0	0	0	0	0	0	0	0	2047	2047	2047	2047	2047	0	2047	0	2046	2046
1.000000n	0	LLLL	HHHH	0	0	0	0	0	0	0	0	0	0	0	0	0	2046	2046	2046	2046	????	2044	????	2044	0	0
1.500000n	0	LLLL	HHHH	0	0	0	0	0	0	0	0	0	0	0	0	0	2046	2046	2046	2046	3	2044	3	2044	0	0
2.000000n	0	LLLL	HHHH	0	0	0	0	0	0	0	0	0	0	0	0	0	????	????	????	????	????	????	????	????	0	0
2.500000n	0	LLLL	HHHH	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	2	????	2	????	0	0
3.000000n	0	LLLL	HHHH	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	6	????	6	????	0	0
3.500000n	0	LLLL	HHHH	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	????	0	????	0	0
4.000000n	0	LLLL	HHHH	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	2032	????	2032	????	0	0
4.500000n	0	LLLL	HHHH	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	????	2047	????	2047	0	0
5.000000n	0	LLLL	HHHH	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	2047	0	2047	0	0
5.500000n	0	LLLL	HHHH	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	2047	0	2047	0	0
6.000000n	0	LLLL	HHHH	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	2047	0	2047	0	0
6.500000n	0	LLLL	HHHH	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	2047	0	2047	0	0
7.000000n	0	LLLL	HHHH	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	2047	0	2047	0	0
7.500000n	255	LLLL	HHHH	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	2047	0	2047	0	0
8.000000n	255	HLLH	LHHL	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	2047	0	2047	0	0
8.500001n	255	HLLH	LHHL	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	2047	0	2047	0	0
9.000000n	255	HLLH	LHHL	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	2047	0	2047	0	0
9.500000n	255	HLLH	LHHL	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	2047	0	2047	0	0
10.00000n	255	HLLH	LHHL	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	2047	0	2047	0	0
10.50000n	255	LLLL	HHHH	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	2047	0	2047	0	0
11.00000n	255	LLLL	HHHH	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	2047	0	2047	0	0
11.50000n	255	LLLL	HHHH	255	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	2047	0	2047	0	0
12.00000n	255	LLLL	HHHH	255	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	2047	0	2047	0	0
12.50000n	127	LLLL	HHHH	255	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	2047	0	2047	0	0
13.00000n	127	LLHL	HHLH	255	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	2047	0	2047	0	0
13.50000n	127	LLHL	HHLH	255	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	2047	0	2047	0	0
14.00000n	127	LLHL	HHLH	255	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	2047	0	2047	0	0
14.50000n	127	LLHL	HHLH	255	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	2047	0	2047	0	0
15.00000n	127	LLHL	HHLH	255	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	2047	0	2047	0	0
15.50000n	127	LLLL	HHHH	255	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	2047	0	2047	0	0
16.00000n	127	LLLL	HHHH	255	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	2047	0	2047	0	0
16.50000n	127	LLLL	HHHH	255	127	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	2047	0	2047	0	0
17.00000n	127	LLLL	HHHH	255	127	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	2047	0	2047	0	0
17.50000n	27	LLLL	HHHH	255	127	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	2047	0	2047	0	0
18.00000n	27	LHLL	HLHH	255	127	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	2047	0	2047	0	0
18.50000n	27	LHLL	HLHH	255	127	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	2047	0	2047	0	0
19.00000n	27	LHLL	HLHH	255	127	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	2047	0	2047	0	0
19.50000n	27	LHLL	HLHH	255	127	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	2047	0	2047	0	0

```
20.00000n 27 LHLL HLHH 255 127  0   0   0   0   0   0  0 0 0  0   0   0    0   0 2047  0 2047  0   0
20.50000n 27 LLLL HHHH 255 127  0   0   0   0   0   0  0 0 0  0   0   0    0   0 2047  0 2047  0   0
21.00000n 27 LLLL HHHH 255 127  0   0   0   0   0   0  0 0 0  0   0   0    0   0 2047  0 2047  0   0
21.50000n 27 LLLL HHHH 255 127 27   0   0   0   0   0  0 0 0  0   0   0    0   0 2047  0 2047  0   0
22.00000n 27 LLLL HHHH 255 127 27   0   0   0   0   0  0 0 0  0   0   0    0   0 2047  0 2047  0   0
22.50000n 135 LLLL HHHH 255 127 27  0   0   0   0   0  0 0 0  0   0   0    0   0 2047  0 2047  0   0
23.00000n 135 HLLH LHHL 255 127 27  0   0   0   0   0  0 0 0  0   0   0    0   0 2047  0 2047  0   0
23.50000n 135 HLLH LHHL 255 127 27  0   0   0   0   0  0 0 0  0   0   0    0   0 2047  0 2047  0   0
24.00000n 135 HLLH LHHL 255 127 27  0   0   0   0   0  0 0 0  0   0   0    0   0 2047  0 2047  0   0
24.50000n 135 HLLH LHHL 255 127 27  0   0   0   0   0  0 0 0  0   0   0    0   0 2047  0 2047  0   0
25.00000n 135 HLLH LHHL 255 127 27  0   0   0   0   0  0 0 0  0   0   0    0   0 2047  0 2047  0   0
25.50000n 135 LLLL HHHH 255 127 27  0   0   0   0   0  0 0 0  0   0   0    0   0 2047  0 2047  0   0
26.00000n 135 LLLL HHHH 255 127 27  0   0   0   0   0  0 0 0  0   0   0    0   0 2047  0 2047  0   0
26.50000n 135 LLLL HHHH 135 127 27 ??? 127 ???  0   0  0 0 0  0   0   0    0   0 2047  0 2047  0   0
27.00001n 135 LLLL HHHH 135 127 27 255 127  27  0   0  0 0 0  0   0   0    0   0 2047  0 2047  0   0
27.50000n 250 LLLL HHHH 135 127 27 255 127  27  0   0  0 0 0 228  0   0    0   0 2047  0 2047  0   0
28.00000n 250 LLHL HHLH 135 127 27 255 127  27  0   0  0 0 0 229  0   0 ???? ???? 2047 27 2047  0   0
28.50000n 250 LLHL HHLH 135 127 27 255 127  27  0   0  0 0 0 229  0   0  255  255 2047 27 2047  0   0
29.00000n 250 LLHL HHLH 135 127 27 255 127  27  0   0  0 0 0 193  0   0  255 ???? 2047 27 2047 ???? 27
29.50001n 250 LLHL HHLH 135 127 27 255 127  27  0   0  0 0 0 209  0   0  255   63 2047 27 ???? 255  27
30.00000n 250 LLHL HHLH 135 127 27 255 127  27  0   0  0 0 0 ????  0   0  255  190 2047 27 ???? ???? 26
30.50000n 250 LLLL HHHH 135 127 27 255 127  27  0   0  0 0 0 ????  0   0  255  444 2047 27 1792 191  26
31.00000n 250 LLLL HHHH 135 ??? 27 255 127  27  0   0  0 0 0 153  0   0  255 ???? 2047 27 1792 190   0
31.50000n 250 LLLL HHHH 135 250 27 255 127  27  0   0  0 0 0  25  0   0  255  492 2047 27 1792 446 ????
31.99999n 250 LLLL HHHH 135 250 27 255 127  27  0   0  0 0 0  25  0   0  255  456 2047 27 1792 444 300
32.50000n 140 LLLL HHHH 135 250 27 255 127  27  0   0  0 0 0  25  0   0  255 ???? 2047 27 1792 492 300
32.99999n 140 LHLL HLHH 135 250 27 255 127  27  0   0  0 0 0 ????  0   0  255  392 2047 27 1792 456 ????
33.49999n 140 LHLL HLHH 135 250 27 255 127  27  0   0  0 0 0 281  0   0  255  408 2047 27 1792 456 828
33.99998n 140 LHLL HLHH 135 250 27 255 127  27  0   0  0 0 0 281  0   0  255  408 2047 27 1792 456 828
34.49999n 140 LHLL HLHH 135 250 27 255 127  27  0   0  0 0 0 281  0   0  255  280 2047 27 1792 408 1820
34.99999n 140 LHLL HLHH 135 250 27 255 127  27  0   0  0 0 0 281  0   0  255  280 2047 27 1792 408 1820
35.49998n 140 LLLL HHHH 135 250 27 255 127  27  0   0  0 0 0 281  0   0  255   24 2047 27 1792 ???? 1820
35.99998n 140 LLLL HHHH 135 250 ??? 255 127 27  0   0  0 0 0 281  0   0  255   24 2047 27 1792 280 1820
36.49998n 140 LLLL HHHH 135 250 140 255 127 27  0   0  0 0 0 281  0   0  255   24 2047 27 1792 280 1820
36.99998n 140 LLLL HHHH 135 250 140 255 127 27  0   0  0 0 0 281  0   0  255 ???? 2047 27 1792  24 1820
37.49998n  5  LLLL HHHH 135 250 140 255 127 27  0   0  0 0 0 281  0   0  255  536 2047 27 1792  24 1820
37.99997n  5  HLLH LHHL 135 250 140 255 127 27  0   0  0 0 0 281  0   0  255  536 2047 27 1792  24 1820
38.49997n  5  HLLH LHHL 135 250 140 255 127 27  0   0  0 0 0 281  0   0  255  536 2047 27 1792 536 1820
38.99997n  5  HLLH LHHL 135 250 140 255 127 27  0   0  0 0 0 281  0   0  255  536 2047 27 1792 536 1820
39.49996n  5  HLLH LHHL 135 250 140 255 127 27  0   0  0 0 0 281  0   0  255  536 2047 27 1792 536 1820
39.99996n  5  HLLH LHHL 135 250 140 255 127 27  0   0  0 0 0 281  0   0  255  536 2047 27 1792 536 1820
40.49996n  5  LLLL HHHH 135 250 140 255 127 27  0   0  0 0 0 281  0   0  255  536 2047 27 1792 536 1820
40.99997n  5  LLLL HHHH 135 250 140 255 127 27  0   0  0 0 0 281  0   0  255  536 2047 27 1792 536 1820
41.49996n  5  LLLL HHHH  5  250 140 255 ??? ??? 255 127 27 0 0 0 281  0   0  255  536 2047 27 1792 536 1820
41.99996n  5  LLLL HHHH  5  250 140 ??? 250 ??? 255 127 27 0 0 0 281  0   0  255  536 2047 27 1792 536 1820
42.49996n 119 LLLL HHHH  5  250 140 135 250 140 255 127 27 0 0 0 281  0 ???? ???? 536 2047 27 1792 536 1820
42.99996n 119 LLHL HHLH  5  250 140 135 250 140 255 127 27 0 0 0 ???? 0  54 ???? 536 2047 ???? 1792 536 1820
43.49996n 119 LLHL HHLH  5  250 140 135 250 140 255 127 27 0 0 0 ???? 0  54  125 ???? 2047 158 1792 536 1820
43.99995n 119 LLHL HHLH  5  250 140 135 250 140 255 127 27 0 0 0 640  0  54  117  552 2047 ???? 1920 536 1820
```

```
44.49996n 119 LLHL HHLH 5 250 140 135 250 140 255 127 27 0 0 0 640 0 54 117  809 2047 186 1922 536 1948
44.99995n 119 LLHL HHLH 5 250 140 135 250 140 255 127 27 0 0 0 640 0 54 ???? 809 2047 178 1930 552 1821
45.49995n 119 LLLL HHHH 5 250 140 135 250 140 255 127 27 0 0 0 640 0 54 613  811 2047 178 1930 ???? 1853
45.99995n 119 LLLL HHHH 5 ??? 140 135 250 140 255 127 27 0 0 0 640 0 54 ???? 811 2047 178 ???? 809 1597
46.49995n 119 LLLL HHHH 5 119 140 135 250 140 255 127 27 0 0 0 640 0 54 581  779 2047 162 ???? 811 1597
46.99994n 119 LLLL HHHH 5 119 140 135 250 140 255 127 27 0 0 0 640 0 54 581  783 2047 162 ???? 811 1597
47.49994n 240 LLLL HHHH 5 119 140 135 250 140 255 127 27 0 0 0 640 0 54 517  783 2047 130 1466 ???? 1085
47.99994n 240 LHLL HLHH 5 119 140 135 250 140 255 127 27 0 0 0 640 0 54 517  ???? 2047 130 1466 783 1597
48.49994n 240 LHLL HLHH 5 119 140 135 250 140 255 127 27 0 0 0 640 0 54 517  775 2047 130 ???? 783 1597
48.99993n 240 LHLL HLHH 5 119 140 135 250 140 255 127 27 0 0 0 640 0 54 645  775 2047 194 1530 783 1597
49.49994n 240 LHLL HLHH 5 119 140 135 250 140 255 127 27 0 0 0 640 0 54 645  775 2047 194 1530 775 1597
49.99993n 240 LHLL HLHH 5 119 140 135 250 140 255 127 27 0 0 0 640 0 54 645  775 2047 194 1530 775 1597
```

It will be seen that the overall propagation time of the circuit is around 12.5ns from receipt of a `ld[3]` going low (which initiates data transfer into the register bank) to results appearing on the outputs of the `dx` and `dy` adders. This does not imply that the overall chip propagation time will be so fast because delays through the I/O pads are liable to be of the order of tens of nanoseconds. However an overall performance of 20MHz (*i.e.* 50ns propagation time) would seem attainable.

10.15 Leaf cell layout

There are only two leaf cells in this design — the d-type latch and the full adder. These modules are connected hierarchically to form the full system. The hierarchical connections will be considered in the next section.

The cells are arranged with power connections in 8μ metal 1 running horizontally and signal connections (usually in metal 2 and polysilicon) running vertically. This allows cells to be abutted for power connection with horizontal signal routing buffers between cell blocks.

10.15.1 D-type latch

The d-type is $70\mu \times 50\mu$. The `ld` and `ldbar` connections are in polysilicon on the left of the cell. They swap over internally so that two d-types stacked vertically form a master-slave pair with the control lines correctly wired by abutment. Similarly, the `D` and `Q` outputs are aligned so that the master will feed the slave directly.

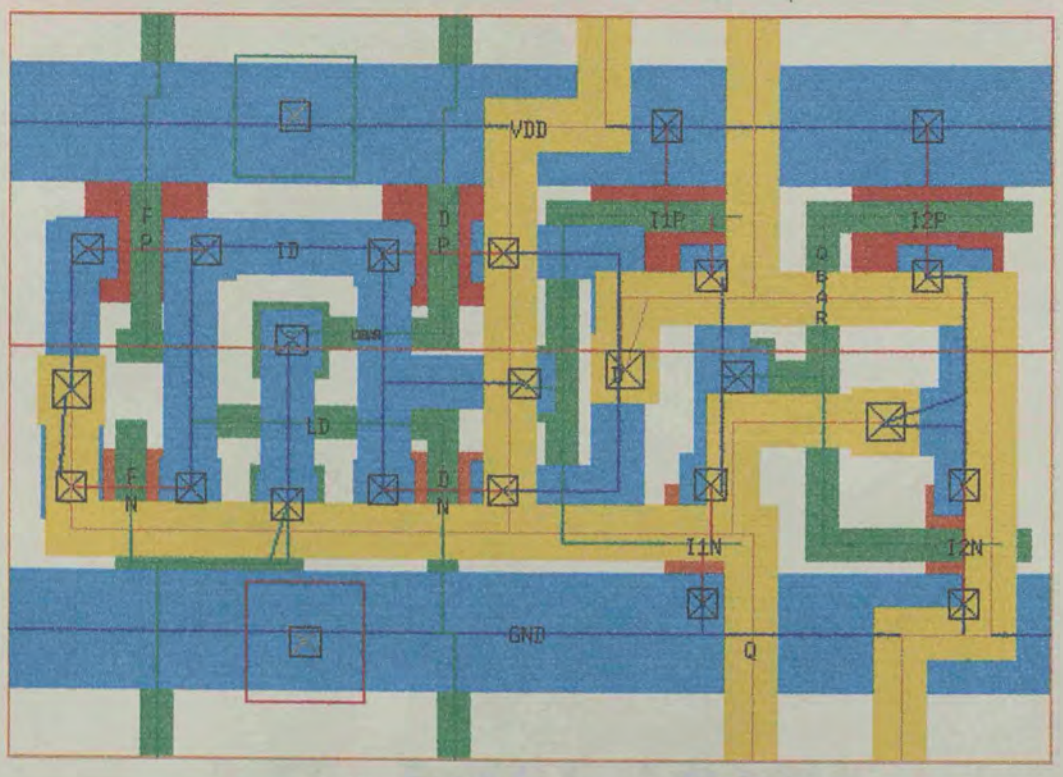


Figure 10.10: D-type latch layout

10.15.2 Full adder

The full adder cell is $115\mu \times 127.5\mu$. The `Cin` and `Cout` connections are available at the sides and at the top of the cell so that ripple carry adders may be constructed by horizontal abutment. The `a`, `b`, `sum` and `sumbar` connections are available at the top and the bottom of the cell to ease global routing.

10.16 Chip floorplan

The chip floorplan is shown with global routing removed in Figure 10.12

The d-bus runs from the I/O pads at the top of the chip to the tops of the registers block. The first layer of the registers block contains the prefetch latch and data flows down through the register block on successive cycles. The four Sobel partial sums are calculated using two 11-bit adders each in the one_two_one modules distributed around the register block. The final dx and dy components are calculated in outermost blocks of the chip and routed to the I/O pads.

10.17 Test results

The first full SOBS-1 prototype will be fabricated in Autumn 1988 with silicon expected back at about the beginning of March 1989. However, the leaf cell

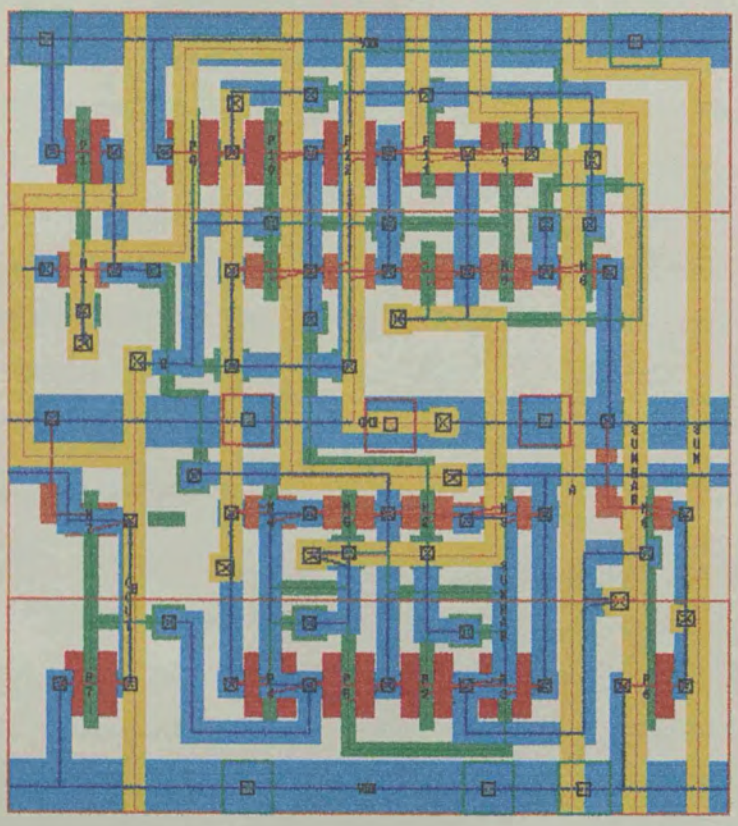


Figure 10.11: Full adder layout

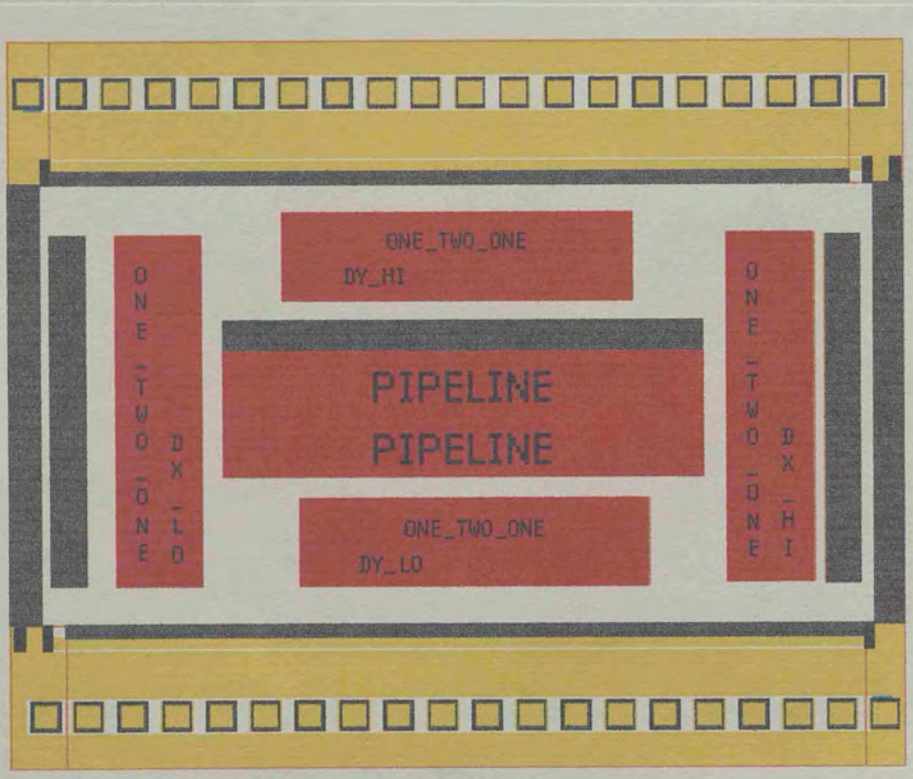


Figure 10.12: SOBS-1 floorplan

samples have been individually fabricated by borrowing spare area on undergraduate student designs fabricated as part of the third year VLSI course at RHBNC which is taught by the author. The full adder and a master-slave flip-flop were functionally perfect and performed as expected. In the absence of sophisticated test equipment it is impossible to accurately measure the speed of the devices which is in the nanosecond range. However, these results give confidence that the full chip will operate correctly.

10.18 Conclusions

A Sobel edge detector has been simulated and laid out for fabrication in 2 micron CMOS. Examples of the leaf cells have been fabricated and shown to work, and this gives confidence that the full chip will operate correctly. The design is compact and could be produced in medium volumes at low cost. This work indicates the major changes in implementation technology that would be expected were the inspection project described in the previous chapter started in 1989 rather than 1982. Availability of sophisticated CAD tools allows very high performance designs to be implemented with simulation replacing the traditional breadboarding techniques.

Chapter 11

Conclusions

11.1 Introduction

This thesis has presented novel techniques in algorithms, systems and components for real-time image processing.

11.1.1 Algorithms

Four algorithms for quadtree generation from array representations were presented in Chapter 4. The match between algorithms and architectures was explored especially in terms of the 'bit-twiddling' instructions required for algorithm four. The quadtrees were used to control simple image processing operators such as edge detectors and smoothers. It was shown that a quadtree controlled skimmer applied as a preprocessor to an edge detector provided better results than a simple threshold skimmer.

11.1.2 Systems

Three framestore designs (IPOFS, V1 and V2) were described in Chapter 5. These compact designs provided a high level of performance with special features (such as the ROM mapper and the wipe register) aimed at PDP-11 based systems. The differing constraints of the memory subsystems for VAX and PDP systems were shown to favour in-store processing on PDP-11's and main memory processing on VAXes, especially Q-bus based machines such as the MicroVAX II.

These framestores have been used in the IMP system described in Chapter 9 — a VME bus based MIMD multiprocessor with hardwired co-processors running at near-video speeds. A microVAX or PDP-11 may participate in the MIMD system *via* a Q-bus to VME bus protocol converter, and acts as over-

all controller in the system, as well as providing software development and user interface functions. The IMP system has been used with commercial 68000 processor and memory boards as well as several microcoded processor designs. A demonstrator system was constructed that performed automatic inspection of six products per second on a real industrial production line. The system performed well during a two week trial.

11.1.3 Components

Extensive use has been made of programmable logic devices in the systems described here. Standard sync pulse generator, VMEbus arbitrator, requester, interrupter and slave protocol handlers have been implemented.

The last chapter described a VLSI Sobel filter capable of analysing 20 million windows per second, or about 76 full 512×512 pixel images per second. The leaf cells for this device have been fabricated and shown to conform to simulated performance. The full version of the chip is going forward for fabrication in Autumn 1988.

11.2 Review

This thesis includes review of basic results concerning image representations, algorithm analysis, sequential processor design, asynchronous multiprocessor design, synchronous processor design and the design of programming language features to exploit such machines.

11.3 Further work

Work in the areas described in this thesis has continued. A microcoded processor called SP1 was developed and has now been superceded by SP2, a compact 20MHz processor. This is designed to work in both pipelined and VLIW configurations. High level language support for this processor is currently under development with the assistance of postraduate students supervised by the author. An array processor designed at the NPL has been integrated into the IMP architecture with a compiler for a high level array processor language. A separate project under the direction of E R Davies has been investigating theoretical results from the earlier collaboration with United Biscuits and Unilever, and the microcoded processors will be used as the implementation vehicle for algorithms

produced by that project.

Acknowledgements

I should especially like to thank Dr. E. R. Davies for providing guidance and encouragement throughout this project and during the writing of this thesis.

I also thank Messrs. United Biscuits and Unilever Central Research for providing some of the finance and impetus for this work.

Finally, I am indebted to the Science and Engineering Research Council for providing me with a student research grant, and for funding and loaning equipment used in this project.

Bibliography

[AHU75] A.V. Aho, J.E. Hopcroft, and J.D. Ullman. *The Design and Analysis of Computer Algorithms.* Addison Wesley, 1975.

[Bat80] K.E. Batcher. Design of a massively parallel processor. *IEEE Trans.*, C-29:836–840, 1980.

[Bau76] G. Baudet. *Asynchronous Iterative Methods for Multiprocessors.* Technical Report, Carnegie-Mellon University, November 1976.

[BB59] W.W. Bledsoe and I. Browning. Pattern recognition and reading by machine. In *Proceedings of the Eastern Joint Computer Conference*, pages 225–232, 1959.

[BBG*60] J.W. Backus, F.L. Bauer, J. Green, C. Katz, J. McCarthy, P. Naur, A.J. Perlis, H. Rutishauser, K. Samelson, B. Vauquois, J.H. Wegstein, A. van Wijngaarden, and M. Woodger. Report on the algorithmic lanuage ALGOL60. *Numer. Math.*, 2, 1960.

[Bel78] C Gordon Bell. Seven views of computer systems. In *Computer Engineering*, chapter one, pages 1–26, Digital Press, 1978.

[Ben82] M. Ben-Ari. *Principles of Concurrent Programming.* Prentice-Hall International, 1982.

[Ber76] M. Berry. *Principles of Cosmology and Gravitation.* Cambridge University Press, 1976.

[BH77] Per Brinch Hansen. *The architecture of concurrent programs.* Prentice Hall, 1977.

[Blu67] H. Blum. A transformation for extracting new descriptors of shape. In W Wathen-Dunn, editor, *Models for the Perception of Speech and Form*, MIT press, Cambridge, Mass., 1967.

[BM78] C. Gordon Bell and John E. McNamara. The pdp-8 and other 12-bit computers. In C. Gordon Bell, editor, *Computer Engineering*, chapter 1, pages 1–26, Digital Press, 1978.

[BN71] C.G. Bell and A. Newell. *Computer Structures: Readings and Examples.* McGraw-Hill, 1971.

[CD83] Napoleone Cavlan and Stephen J. Durham. Sequencers and arrays transform truth tables into working tables. In

Integrated circuits Book 4 — Integrated Fuse Logic, chapter 7, pages 187–194, Mullard Ltd, 1983.

[CH74] R.H. Campbell and A.N. Habermann. The specification of process synchronisation by path expressions. In Goos and Hartmanis, editors, *Lecture Notes in Computer Science*, pages 89–102, Springer-Verlag, 1974.

[Coo83a] B. M. Cook. *Construction and use of research tools for image processing.* PhD thesis, Royal Holloway and Bedford New College, University of London, 1983.

[Coo83b] Doug Cooper. *Standard Pascal.* Norton, 1983.

[Cor82] Control Data Corporation. *Cyber 200 Fortran, Version 1, Reference Manual.* 1982.

[CR61] H.M. Cundy and A.P. Rollet. *Mathematical Models.* Oxford University Press, 1961.

[Dav84] E. R. Davies. Circularity - a new principle underlying the design of accurate edge orientation operators. *Image and Vision Computing*, 2(3), August 1984.

[Dev81a] Advanced Micro Devices. *Am26LS31 quad high speed differential line driver.* 1981.

[Dev81b] Advanced Micro Devices. *Am26LS32/Am26LS33 quad differential line receiver.* 1981.

[DH73] R. O. Duda and P. E. Hart. *Pattern classification and scene analysis.* Wiley, New York, 1973.

[Dig79a] Digital. *PDP11 bus handbook.* 1979.

[Dig79b] Digital. *PDP11 processor handbook.* 1979.

[Dig82] Digital. *RT-11 Technical Summary.* 1982.

[Dij68] E. W. Dijkstra. The structure of the THE multiprogramming system. *Comm. ACM*, 11(5), May 1968.

[Dij76] E. W. Dijkstra. *A Discipline of Programming.* Prentice Hall, 1976.

[DJ86] E.R. Davies and A.I.C. Johnstone. Engineering trade-offs in the design of a real-time system for the visual inspection of small products. In *Proc. IMechE*, pages 15–22, 1986.

[DJ89] E.R. Davies and A.I.C. Johnstone. A methodology fpr optimising cost-speed tradeoffs in real- time inspection hardware. *IEE Proceedings Part E*, 136:in press, 1989.

[DJT67] M.J.B. Duff, B.M. Jones, and L.J. Townsend. Parallel processing pattern recognition system UCPR1. *Nucl. Instr. Meth.*, 52:284–288, 1967.

[DP81] E. R. Davies and A. P. N. Plummer. Thinning algorithms: a critique and a new methodology. *Pattern Recognition*, 14, 1981.

[DT84] R. Davis and D. Thomas. Systolic array chip matches the pace of high-speed processing. *Electronic Design*, 207–218, October 1984.

[DWFS73] M.J.B. Duff, D.M. Watson, T.J. Fountain, and G.K. Shaw. A cellular logic array for image processing. *Patt. Recogn.*, 5:229–247, 1973.

[Dye82] C.R. Dyer. Pyramid algorithms and machines. In K. Preston and L. Uhr, editors, *Multicomputers and image processing*, chapter , pages 409–420, Academic Press, New York, 1982.

[Edm88] M. Edmonds. *Studies of inspection algorithms and associated microprogrammable implementations.* PhD thesis, Royal Holloway and Bedford New College, University of London, 1988.

[Ell86] John R. Ellis. *Bulldog: A compiler for VLIW Architectures.* The MIT Press, 1986.

[FG80] T.J. Fountain and V. Goetcherian. Clip4 parallel processing system. *IEE Proc.*, 127E:219–224, 1980.

[Fis82] Joseph A. Fisher. *Computer Systems architecture at Yale.* Technical Report Research Report 241, Yale University, Dept of Computer Science, July 1982.

[FKM*83] A.L. Fisher, H.T. Kung, L.M. Monier, H. Walker, and Y. Dohi. Design of the PSC: a programmable systolic chip. In *Proc. 3rd CALTECH Conf. on VLSI*, pages 287–302, 1983.

[Fly72] M. J. Flynn. Some computer organisations and their effectiveness. *IEEE Trans. Computers*, C-21(9):948–960, 1972.

[FOR*78] Samual H. Fuller, John K. Ousterhout, Levy Raskin, Paul I. Rubinfeld, Pradeep S. Sindhu, and Richard Swan. Multi-microprocessors: an overview and working example. In C. Gordon Bell, editor, *Computer Engineering*, chapter 20, pages 463–484, Digital Press, 1978.

[Fou85] T.J. Fountain. Plans for the CLIP7 chip. In S. Levialdi, editor, *Integrated Technology for Parallel Image Processing*, pages 199–214, Academic Press, 1985.

[Fou87] T. Fountain. *Processor Arrays Architecture and Applications.* Academic Press, 1987.

[FP87] J. Hill Fredrick and Gerald R. Peterson. *Digital Systems, Hardware, Organisation and Design.* John Wiley and Sons, third edition, 1987.

[FR72] C. C. Foster and E. M. Riseman. Percolation of code to enhance parallel dispatching and execution. *IEEE Transactions on Computers*, 21(12):1411–1415, December 1972.

[Fre61] H. Freeman. On the encoding of arbitrary geometric configurations. *IRE transactions on Electronic Computing*, EC-10, 1961.

[GH77] S.E. Goodman and S.T. Hedetniemi. *Introduction to the design and analysis of algoritms.* McGraw-Hill, 1977.

[GKW85] J.R. Gurd, C.C. Kirkham, and I. Watson. The Manchester prototype dataflow computer. *Comm. ACM*, 28:34–51, 1985.

[GM63] J. Gregory and R. McReynolds. The SOLOMON computer. *IEEE Trans.*, EC-12:774–781, 1963.

[God62] Kurt Godel. *On formally undecidable propositions.* Basic books, New York, 1962. A translation of the original 1931 paper.

[Gre84] D. Green. Conveyer speeds. 1984. Private communication.

[GW87] Rafael C. Gonzalez and Paul Wintz. *Digital Image Processing.* Addison Wesley, second edition, 1987.

[Har80] R.M. Haralick. Edge and region analysis for digital image data. *Comput. Graph. Image Proc.*, 12:60–73, 1980.

[HB85] Kai Hwang and Faye A. Briggs. *Computer Architecture and Parallel Processing.* McGraw-Hill, 1985.

[Heu73] M.F. Heuckel. A local visual operator which recognises edges and lines. *J. ACM*, 20(4):634–647, 1973.

[Hil85] W.D. Hillis. *The Connection Machine.* MIT Press, Cambridge, Mass., 1985.

[Hit86] Hitachi. *HM53461 64K × 4 Video RAM.* 1986.

[HJ81] R.W. Hockney and C.R. Jesshope. *Parallel Computers.* Adam Hilger, 1981.

[Hoa74] C.A.R. Hoare. Monitors: an operating system structuring concept. *Comm. ACM*, 17:549–557, 1974.

[Hoa81] C. A. R. Hoare. The emperor's old clothes. *Communications of the ACM*, 24(2):75–83, February 1981.

[Hoa85] C.A.R. Hoare. *Communicating Sequential Processes.* Prentice-Hall, 1985.

[HR74] A.R. Hanson and E.M. Riseman. Pre-processing cones: a computational structure for scene analysis. *COINS Tech. Rep. 74C-7, Univ. Massachussetts*, 1974.

[HS79a] G. M. Hunter and K. Steiglitz. Linear transformation of pictures represented by quadtrees. *Comput. Gr. Image Process.*, 10(3), July 1979.

[HS79b] G.M. Hunter and K. Steiglitz. Operations on images using quad trees. *IEEE Trans. Pattern Analysis and Machine Intelligence*, PAM-I(2):145–153, April 1979.

[Hue71] M.F. Hueckel. An operator which locates edges in digitised pictures. *J. ACM*, 18(1):113–125, 1971.

[IB71] ITA and BBC. *Specification of television standards for 625 line System-I transmission.* 1971.

[Ich84] Jean Ichbiah. Ada: past, present and future. *Comm. ACM*, 27(10):990–997, October 1984.

[ICL79] ICL. *DAP: Introduction to Fortran programming.* 1979.

[IDT86] IDT. *IDT49C402A 16-bit CMOS microprocessor slice.* 1986.

[Inm84a] Inmos. *High Performance 64K × 1 Dynamic RAM.* November 1984.

[Inm84b] Inmos. *Nibble Mode Operation Simplifies High-Bandwidth Memory Applications.* March 1984.

[Inm86] Inmos. *IMS2800 High Performance 256k × 1 CMOS Dynamic RAM.* 1986.

[JCD*78] A.K. Jones, R.J. Chansler, I. Durham, P. Feiler, and K. Schwans. Programming issues raised by a multiprocessor. *Proc. IEEE*, 66(2):229–237, February 1978.

[Joh85] Adrian Johnstone. *Hardware verification system documentation set.* 1985.

[Joh88a] Adrian Johnstone. *PIPE-32 V2.00 user guide.* October 1988.

[Joh88b] Adrian Johnstone. *SP2.* November 1988.

[Jos83] A. Fisher Joseph. Very long instruction word architectures and the ELI-512. *The 10th Annual International Symposium on Computer Architecture*, June 1983.

[KDG82] Bjorn Kruse, Per-Erik Danielsson, and Bjorn Gudmundsson. From PICAP I to PICAP II. *Special Computer Architectures for Pattern Processing*, 127–156, 1982.

[KL79] H.T. Kung and C.E. Leiserson. Systolic arrays for vlsi. In I.S. Duff and G.W. Stewar, editors, *Sparse Matrix Proceedings 78*, pages 256–282, 1979.

[Knu70] D. E. Knuth. *The art of computer programming, Volume 3.* Addison Wesley, 1970.

[Knu79] D. E. Knuth. *The art of computer programming, Volume 1.* Addison Wesley, second edition, 1979.

[KS86] J.T. Kuehn and H.J. Seigel. Multifunction processing with PASM. In M.J.B. Duff, editor, *Intermediate-Level Image Processing*, pages 209–230, Academic Press, 1986.

[Kun84] H.T. Kung. Systolic algorithms for the CMU Warp processor. In *Proc. 7th Int. Conf. on Pattern Recognition*, pages 570–577, 1984.

[Lam74] L. Lamport. A new solution of Dijkstra's concurrent programming problem. *Comm. ACM*, 17:453–455, 1974.

[Lee83] Chuang Chang Lee. Elimination of redundant operations for a fast sobel operator. *IEEE trans. Sys. Man Cyb.*, SMC-13(3), 1983.

[Lev80] Martin D. Levine. Region analysis using a pyramid data structure. In S. Tanimoto and A. Klinger, editors, *Structured Computer Vision Machine Perception through Hierarchical Computation Structures*, pages 57–100, Academic Press, 1980.

[Mar82] David Marr. *Vision.* W.H. Freeman Co., 1982.

[Mar84] John Markoff. Risc chips. *Byte*, 191–206, November 1984.

[McC63] B.H. McCormick. The Illinois pattern recognition computer — ILLIAC III. *IEEE Trans.*, EC-12:791–813, 1963.

[McC85] A.J. McCollum. *Parallel computer structures for high speed image processing.* Technical Report, UWIST, February 1985.

[MH80] D. Marr and E. Hildreth. Theory of edge detection. *Proc. Royal Soc. London*, B207:187–217, 1980.

[MM82] J.V. McCanny and J.G. McWhirter. On the implementation of signal processing functions using one-bit systolic arrays. *Electron. Lett.*, 18:241–243, 1982.

[MMS79] J.G. Mitchell, W. Maybury, and R. Sweet. *Mesa Language Manual (Version 5.0).* April 1979.

[Mot81] Motorola. *MC6845B Cathode Ray Tube Controller.* 1981.

[Mye77] G.J. Myers. The case against stack oriented instruction sets. *ACM Sigarch News*, 7–10, August 1977.

[NB80] R. Nevatia and K.R. Babu. Linear feature extraction and description. *Comput. Graph. Image Proc.*, 13:257–269, 1980.

[NCR84] NCR Corporation. *Geometric Arithmetic Parallel processor, NCR45CG72.* Drayton, Ohio, 1984.

[NF81] Alexandru Nicolau and Joseph A. Fisher. Using an oracle to measure parallelism in single instruction stream programs. *The 14th Annual Microprogramming Workshop*, October 1981.

[Per79] R.H. Perrott. A language for array and vector processors. *ACM Trans. on Prog. Lang. Syst.*, 2:177–195, 1979.

[Per87] R.H. Perrott. *Parallel Programming.* Addison-Wesley, 1987.

[PFP85] J. W. Poulton, H. F. Fuchs, and A Paeth. Pixel planes graphic engine. In Neil Weste and Kamran Eshraghian, editors, *Principles of CMOS VLSI design*, chapter 9, pages 448–480, Addison Wesley, 1985.

[Pic84] Philip D Picton. Comment on 'elimination of redundant operations for a fast sobel operator'. *IEEE trans. Sys. Man Cyb.*, SMC-14(3), 1984.

[Pot82] J.L. Potter. Pattern processing on staran. In K.S. Fu and T. Ickikaw, editors, *Special Computer Architectures for Pattern Processing*, pages 87–101, CRC Press Inc, Baton Rouge, Florida, 1982.

[Pre70] J.M.S. Prewitt. Object enhancement and extraction. In B.S. Lipkin and A. Rosenfeld, editors, *Picture Processing and Psychopictorics*, pages 75–149, Academic Press, 1970.

[PW87] Franklin P. Prosser and David E. Winkel. *The Art of Digital Design: An Introduction to Top-Down Design.* Prentice-Hall International, 2 edition, 1987.

[PZ86] R.H. Perrott and A. Zarea-Aliabadi. Supercomputer languages. *Computing Surveys*, 18(1):5–22, March 1986.

[Rac87] Racal-Redac. *Visula ISIS Primary User's Guide.* 1987.

[Ran81] S. Ranade. Use of quadtrees for edge enhancement. *IEEE Trans. Syst. Man Cybern.*, 11(5), May 1981.

[Red73] S.F. Reddaway. DAP - a distributed array processor. In *1st Annual Symp. on Computer Architecture*, pages 61–65, Florida, 1973.

[Res82] Cray Research. *Fortran (CFT) Reference Manual.* 1982.

[RM82] I. N. Robinson and W. R. Moore. A parallel array processor architecture and its implementation in silicon. In *Proc IEEE Custom Integrated Circuits Conference*, pages 41–55, 1982.

[Rob65] L.G. Roberts. Machine perception of three-dimensional solids. In Tippett J. et al, editor, *Optical and Electro-optical Information Processing*, chapter 9, pages 159–197, MIT Press, 1965.

[Rob77] G.S. Robinson. Edge detection by compass gradient masks. *Comput. Graph. Image Proc.*, 6:492–501, 1977.

[Rob87] Phillip Robinson. How much of a risc? *Byte*, 12(4):143–150, April 1987.

[Ros70] Azriel Rosenfeld. Connectivity in digital pictures. *J. ACM*, 17(1), 1970.

[RS81] S. Ranade and M. Shneier. Using quadtrees to smooth images. *IEEE Trans. Syst. Man Cybern.*, 11(5), May 1981.

[Sam69] J. Sammet. *Programming languages: history and fundamentals.* Prentice-Hall, 1969.

[Sam79] H. Samet. *A distance transform for images represented by quadtrees.* Technical Report TR-780, University of Maryland, College Park, Maryland, USA, 1979.

[Sam81] H. Samet. Connected component labelling using quadtrees. *J. ACM*, 28(3), July 1981.

[Sam82] H. Samet. Neighbor finding techniques for images represented by quadtrees. *Comput Gr. Image Process.*, 18(1), January 1982.

[Sam83] H. Samet. A quadtree medial axis transform. *Comm. ACM*, 26(9), September 1983.

[Sam84] Hanan Samet. The quadtree and related hierarchical data structures. *ACM Computing Surveys*, 16(2), June 1984.

[SBM62] D.L. Slotnick, W.C. Borck, and R.C. McReynolds. The solomon computer. In *AFIPS Conf. Proc.*, pages 97–107, 1962.

[Sem86] Plessey Semiconductors. *PDSP16401 2-dimensional edge detector.* May 1986.

[Sho73] J.E. Shore. Second thoughts on parallel processing. *' Comput. Elect. Eng.*, 1:95–109, 1973.

[SOA73] Y. Suzuki, K. Odagawa, and T. Abe. Clocked cmos calculator circuitry. *IEEE Journal of Solid-State Circuits*, SC-8(6):336–340, December 1973.

[Ste75] K.G. Stevens. CFD - a fortran-like language for the Illiac IV. *SIGPLAN Not. (ACM)*, 10(3):72–80, 1975.

[Ste87] Jeffrey Steinberg. A virtual VAX does it in parallel. *Digital Review*, 2–5, March 23 1987.

[Str78] W.D. Strecker. VAX-11/780: a virtual address extension to the DEC PDP-11 family. In *Proc. Computer Conference*, pages 967–980, June 1978.

[Tan80] Steven L. Tanimoto. Image data structures. In *Structured Computer Vision*, chapter two, pages 31–56, Academic Press, 1980.

[Tan83] S.L. Tanimoto. A pyramidal approach to parallel processing. *Proc. 10th Ann. Int. Symp. on Computer Architecture, Stockholm, June*, 372–378, 1983.

[Tex84] Texas Instruments. *TMS4464 65,536-word by 4-bit dynamic RAM.* 1984.

[TK80] Steven L. Tanimoto and Allen Klinger, editors. *Structured Computer Vision.* Academic Press, 1980.

[Tur36] A. M. Turing. On computable numbers, with an application to the *Entscheidungsproblem. Proc. London Mathematical Soc. Ser. 2*, 42, 1936.

[Uhr78] L. Uhr. Recognition cones and some test results; the imminent arrival of well-structured parallel-serial computers; positions, and positions on positions. In A. R. Hanson and E. M. Riseman, editors, *Computer vision systems*, pages 363–377, Academic Press, New York, 1978.

[Ung58] S.H. Unger. A computer oriented towards spatial problems. In *Proc. Inst. Radio Eng. (USA)*, pages 1744–50, 1958.

[Uni81] Unit-C. *Parallel-Pascal product description.* 1981.

[VIT85] VITA. *The VMEbus specification REV C.1.* 1985.

[Wat74] D.M. Watson. *The application of cellular logic to image processing.* PhD thesis, University of London, 1974.

[WC72] Wulf W.A. and Bell C.G. C.mmp - a multi-mini-processor. *AFIPS Conf. Proc. FJCC pt. II*, 41:765–777, 1972.

[WE85] Neil Weste and Kamran Eshraghian. *Principles of CMOS design — a Systems Perspective.* Addison Wesley, 1985.

[Wei83a] Weitek. *Designing with the WTL 1032/1033.* 1983.

[Wei83b] Weitek. *Performing Floating Point Division with the WTL 1032/1033.* 1983.

[Wei84a] Rheinhold P. Weicker. Drystone: a synthetic systems programming benchmark. *Comm. ACM*, 27(10):1013–1030, October 1984.

[Wei84b] Weitek. *WTL1032 High-Speed 32-Bit IEEE Floating Point Multiplier, WTL1033 High-Speed 32-Bit IEEE Floating Point ALU.* 1984.

[Whi11] A.N. Whitehead. *An introduction to mathematics.* Oxford University Press, 1911.

[Whi85] Colin Whitby-Strevens. The transputer. In *Proceedings of the 12th International Symposium on Computer Architecture*, pages 292–300, 1985.

[WHR82] A. Y. Wu, T. H. Hong, and A. Rosenfeld. Threshold selection using quadtrees. *IEEE Trans. Pattern Anal. Mach. Intell.*, 4(1), January 1982.

[Wir77] N. Wirth. Modula: a language for modular multiprogramming. *Software P and E*, 7, 1977.

[Wir81] N. Wirth. Pascal-s: a subset and its implementation. In D.W. Barron, editor, *Pascal - The Languate and its Implementation*, chapter 12, pages 199–358, John Wiley and Sons, 1981.

[Wir83] Niklaus Wirth. *Programming in Modula-2.* Springer-Verlag, second edition, 1983.

[Bal81] D. H. Ballard. Generalizing the Hough Transform to detect Arbitrary Shapes. *Pattern Recognition*, 13:111–122, 1981.

[LT82] D. Lavie and W. K. Taylor. Effects of border variations due to spatial quantisation on binary-image template matching. *Electronics letters*, 18(10):418–420, May 1982.

C362/86

Engineering trade-offs in the design of a real-time system for the visual inspection of small products

E R DAVIES, MA, DPhil, CPhys, MInstP
Department of Physics, Royal Holloway and Bedford New College, Egham, Surrey
A I C JOHNSTONE, BSc
Department of Computer Science, Royal Holloway and Bedford New College, Egham, Surrey

SYNOPSIS This paper presents an algorithm for the rapid inspection of small products, and considers its optimal implementation. Speed and cost limits constrain optimisation relatively simply, but concurrency makes the situation considerably more complex. However, an inspection system that makes particularly efficient use of a set of hardware processors has been designed. The host CPU is not merely used for control and data logging, but takes an integral role in the main image analysis task. The emphasis of the paper is on removing arbitrariness in the design of hardware for industrial inspection systems.

1 INTRODUCTION

The past decade has seen enormous growth in the applications of computers to manufacturing engineering. Automated assembly and automated inspection are becoming mature technologies: both customarily use vision, since sensors such as TV and linescan cameras are capable of providing prodigious quantities of relevant information at high data rates, and algorithms are known which will analyse much of this data sufficiently reliably for the purpose of control. In automated assembly, vision provides data on the positions and orientations of products, and can at the same time check them for defects. In automated inspection, the immediate purpose is that of dimensional checking and quality control. More specifically, it aims (a) to identify and reject products containing defects, (b) to provide feedback on product characteristics (such as size or surface texture) so as to keep manufacture within specified tolerances, and (c) to provide statistics on manufacturing parameters (1).

There are two important trends in manufacturing: one is that products are becoming increasingly complex and sophisticated; the other is that the consumer is becoming significantly more demanding with regard to quality. As a result there is a move towards 100 per cent inspection of products.

If there is to be 100 per cent control of quality at a time when products are becoming increasingly complex, this necessarily throws a heavy load on computing machinery. Indeed, the enormous amounts of data in typical images, coupled with the high throughput rates, mean that special electronic hardware is needed for visual inspection. Such hardware is costly, and there are two basic strategies for providing it: the first is to maintain generality by building multi-processor systems containing (for example) large numbers of microprocessors; the other is to employ special dedicated hardware systems which are less adaptable but may be considerably cheaper in individual applications. This paper studies these basic strategies and analyses the speed/cost tradeoffs in hardware for automated visual inspection. In addition, it describes the real-time system we have designed to inspect products such as biscuits at costs compatible with the low profit margins of foodproduct manufacture.

2 ALGORITHMS FOR PRODUCT INSPECTION

Many inspection problems involve three main tasks: (1) image acquisition, (2) product location, and (3) product scrutiny and measurement. In this section we bypass the problem of acquisition and concentrate on product location and scrutiny.

To locate products in a grey-scale image it is very common to threshold the intensity and thus obtain a binary image from which objects can be located with relatively little further processing. This scheme is only suitable if the lighting system is carefully configured and products appear silhouetted e.g. as dark objects against a light background, so that the intensity histogram is bimodal. Many inspection systems have been designed on this basis, but attention is now shifting towards more complex products for which this approach is unlikely to be successful. For this reason we concentrate here on systems based on edge detection. This approach is generally much more robust, being able to negotiate problems due to shadows, overlapping products, etc.

One common approach to object location is to use a simple edge detector to locate the boundaries of objects, and then to link broken edges in order to create complete (connected) outlines of objects. Having obtained complete object outlines, these can be thinned down to single pixel width, and a tracking algorithm can be used to follow the outlines of individual objects, hence generating a set of

polar (r,θ) plots (1,2). Efficient one dimensional pattern matching algorithms can then be used to identify objects, at the same time determining their orientation by finding how much the observed profile needs to be shifted before it matches a standard template. It is worth noting that object scrutiny can then proceed, at least in part, by checking how close the match with the standard template is.

This approach, which we shall call Algorithm A, has the disadvantage that image noise, and the nature of the objects and the lighting, together with possibilities of overlapping objects and other artefacts, can make it difficult to link broken edges together. Indeed, there are arguments that indicate that it is not in general possible to perform this function successfully: in any case we have found many practical instances of failure. However, in the restricted set of cases in which the algorithm can be used, it operates extremely rapidly, since the tracking algorithm need not visit every pixel but can by-pass much of the image. An unfortunate consequence of this is that the algorithm does not take in enough data to be particularly robust. In our work, we have aimed at high accuracy and extremely high levels of robustness, since these are the qualities we have found particularly in demand in industrial applications of vision: we have therefore avoided the tracking schemes used by Algorithm A.

An alternative approach which we shall call Algorithm B also involves edge detection, but instead of linking broken edges together directly it proceeds with the Hough transform strategy of locating objects by accumulating (at a reference point within each object) the evidence that is available for their existence (3,4). This gives a particularly robust strategy for locating objects, though it is not so fast as Algorithm A. However, Algorithm B is still highly efficient, since full use is made of locally available edge orientation information to compute candidate positions for object reference points. Here we illustrate this technique by reference to circular object location. For a circular object, each edge pixel that is found permits computation of a candidate centre point a distance equal to R (the radius) along the edge normal in 'parameter space' (4). (In this application parameter space is isomorphic to image space.) When all edge pixels in an image have been processed, it is only necessary to search parameter space for clusters of candidate centre locations and to average them to find accurate positions for the centres of circular objects.

Once products have been found using Algorithm B, they may conveniently be scrutinised using the radial histogram method, which involves computing an average radial intensity profile of the product, and matching this against a suitable template (5,6). This is again a one-dimensional approach, and is analogous to that of matching one-dimensional shape profiles in Algorithm A, though it aims to provide information on product reflectivity and size rather than shape (5-7). Naturally, Algorithm A could also be augmented by the use of radial intensity histograms, if this were appropriate.

These arguments show that Algorithm B should be considerably more robust and accurate for object location than Algorithm A. We have made extensive practical tests of the situation, particularly for the location of round foodproducts such as biscuits, and have verified that this is so. Specifically, Algorithm B has been found to be exceptionally tolerant of broken and overlapping products, and those having other shape defects such as protuberances around their edges (5). In what follows we assume that Algorithm B is to be used because of its superior robustness and accuracy.

3 THE SPEED PROBLEM

For our tests on biscuit inspection, Algorithm B was augmented to include assessment under four main headings - roundness, radius, amount of chocolate cover, and general acceptability according to a radial intensity correlation coefficient. The initial version of this algorithm took about one minute to run on a PDP-11/34A. Subsequent optimisation of the algorithm strategy brought the execution time down to ~5 seconds, without resorting to hand massage of machine code. Although the original algorithm was written in Pascal, it became clear that attempts to optimise the machine code would (for this algorithm) result in a speedup factor of less than two: a 3 second overall execution time appeared to be a limiting case. This is to be expected for the following reasons. The edge detection part of Algorithm B requires some 16 image accesses for each pixel in the 128x128 image. Although the access time of the framestore is around 1 microsecond, instruction fetches and overheads within the program loops reduce the average throughput to around 100000 pixels/second. As a result the minimum execution time of the edge detector is of the order of 2.6 seconds. Product scrutiny then requires some 4-5 accesses over the relevant area (about 3000 pixels in our application), which will take at least another 0.1 seconds. The Hough transform calculations are only applied to some 200 points, but their high computational cost will add another 0.1 seconds to the total execution time. Clearly even with ideal code generation there is a lower bound on the overall processing time of about 2.8 seconds. Changing to a 68000 or other commonly available microprocessor would not affect this substantially.

At best, software optimisation is subject to severely diminishing returns, and further speedup must rely on enhancement of the hardware implementation. As stated in section 1, this has to be obtained either by use of several central processor units (CPUs) or by specially designed dedicated electronic hardware. To inspect biscuits at typical rates of 10-20 per second, a speedup factor ~100 must be attained.

For industrial applications, cost has to be kept low, and it is useful to see how generality can be maintained subject to this

constraint. With this in mind we examine a number of alternative processing architectures.

4 MULTIPROCESSOR SYSTEM DESIGN

4.1 SIMD architecture

When considering fast hardware for image analysis applications, it is natural to start with the SIMD machine, since this architecture would appear to match the hardware to the algorithm most accurately (8). The SIMD (or 'Single Instruction Multiple Data') architecture when applied to image analysis ideally involves use of one processing element (PE) per pixel, the PEs being arranged in an array isomorphic with the image being processed. Such a machine is able (for example) to invert or threshold an image in one instruction cycle, since all processors operate simultaneously on their respective pixels. This type of machine is also able to operate rapidly on images to remove noise by local averaging, or to find edge pixels rapidly by operations within 3x3 neighourhoods. In addition, it can efficiently build up distance functions or find object skeletons by sequences of 3x3 neighbourhood operations. A typical SIMD machine (9) is able to perform these 3x3 operations efficiently since each processing cell has direct links with its 8 neighbours (only 4 in the case of some machines such as the ICL DAP (10)), so the required data is immediately available.

Most SIMD arrays include an activity bit for each PE which allows selective application of processing steps to different areas of the image by disabling individual PEs. However, this clearly wastes the power of the SIMD machine. Unfortunately, all but the lowest level image processing operations require selective processing of the image. The Hough transform calculations required for Algorithm B form an interesting extreme case in which around 200 special image points (edge pixels) trigger a floating point calculation, at the end of which a single point in parameter space must be accessed. In principle the floating point calculation could be performed at every point by the SIMD array, but the only available way of performing the subsequent random access of the parameter plane is by propagation techniques (9). Typically 40 propagation cycles per point would be required in this application for each of the 16384 pixels in the image. (Note that an attempt to re-organise Algorithm B so that propagation routines are used to locate circle centres leads to significant loss of generality, since Algorithm B itself is immediately generalisable to detect any object shape (4).) A conventional sequential processor would be slow at calculating the edge image, but could then efficiently execute the 200 floating point calculations and directly access the parameter space. In addition, technology constraints dictate the use of simple bit-serial processors in current SIMD machines, and these would require many cycles to execute the required floating point calculation. Clearly, the pure SIMD solution would be much less efficient.

An alternative hybrid strategy would be to perform the edge calculation in the SIMD array, and then to read the results out sequentially into a conventional processor which would perform the floating point calculation and update the parameter space. The economics of this approach would be dictated by the relative costs of the SIMD and sequential machines and the bandwidth of the communication channel. Current SIMD arrays are still rather expensive devices, which discounted their use in our application.

This analysis shows that SIMD architectures are of limited use for processing tasks that cannot efficiently exploit their regular topology. The simplicity of the individual PEs, and the absence of long distance communication links within the image make them particularly unsuitable for geometrical calculations on object features. Thus the SIMD architecture is currently inappropriate for many tasks of image analysis that might be needed in industrial inspection, even though it might be well adapted to various image processing tasks in a general imaging environment.

4.2 Multi-processor systems

General multi-processor structures provide resources that may be used concurrently in an unrestricted fashion, unlike the SIMD machine where all resources operate in lockstep. As with all forms of parallel implementation, the efficiency of a multi-processor system will be dictated by the effectiveness of the functional partitions. Interactions between functions will require either transmission of data between processes or access to shared memory spaces. In the one case there is a potential data bottleneck due to lack of bandwidth in the communications channel, and in the other, processes may stall during contention for shared memory. Therefore the speed of a multi-processor system containing N processors is never increased by the ideal factor N unless there is no process interaction, which is unlikely to be the case in a system doing useful work. High efficiency will be obtained by minimising process interaction. Naturally, there is the risk that a system containing N processors and capable of increasing speed by the factor ~100 noted in section 3 will be rather an expensive solution.

4.3 Pipelined processing systems

Pipelined processing systems form an interesting sub-class of multi-processor systems which can be useful for the repetitive execution of a given set of operations. This is typically the case for industrial inspection systems, where the same algorithm is applied to each frame of data as it comes off the camera. In a pipeline, individual frames of data are passed along a chain of processors so that in an N-processor system, N different data sets are being processed at any one time.

Since all processors pass their completed data set on up the chain at the end of a fixed time slot, pipelines are only as fast as the slowest processor in the chain. To be optimal, all procesors should complete in the same amount of time. For a video-based system, an obvious approach would be to execute in

integral numbers of TV frames. For high level parts of the algorithm, such as the Hough transform calculations, subdivision into equal execution time processes would be virtually impossible to achieve. Finally, the approach requires significant bus switching logic and local memory, as well as the hardware processors, which are themselves liable to be costly. Thus pipelined systems pose a serious partitioning problem, and in addition to lacking generality are likely to constitute a rather expensive solution to the speed problem.

4.4 General processing capability

We have concluded in our work that, contrary to many of the suppositions about image analysis (based on what is frequently valid in image processing per se), the ideal type of processing system is a highly general multicomputer system, which is abstract in the sense of not being tied to any specific imaging representation. Again this is not achievable within the budget of most industrial inspection systems. For algorithms such as Algorithm B, the best compromise seemed to be to make optimum use of a single CPU by linking it with a set of hardware accelerators selected for maximal generality coupled with applicability to the problem in hand. In this context, Algorithm B was seen as constituting a useful case study in algorithm analysis and multiprocessor system design: this will be discussed in more detail below.

5 FURTHER ANALYSIS OF ALGORITHM B AND ITS IMPLEMENTATION

Table 1 gives a breakdown of the functions in Algorithm B. The 'description' indicates the size of the neighbourhood employed in imaging operations. It also indicates those processes that are one-dimensional: these are marked since they involve loops containing a significant number of operations, but not as many as for two-dimensional image processing in 1x1 or 3x3 neighbourhoods.

The two other headings in the table, time for execution in software on an LSI-11/23 and cost of hardware implementation, are somewhat notional since it is difficult to divide the algorithm rigorously into completely segregated sections. For example, it has been assumed that various overhead costs such as that of a backplane, rack and power supply have already been covered: we shall largely ignore such complications in what follows. Overall, the figures presented here should be sufficiently accurate to form the basis for useful decisions on cost effectiveness of hardware. Finally, costs are based on chip and other component prices, and do not include logic design or p.c.b. layout. However, on the whole the cost of design and layout work is proportional to the number of connections, which is itself roughly proportional to component cost. This means that our results will be substantially correct, since the analysis below is independent of scaling.

As a simple starting approximation, any function that is implemented in fast hardware will be assumed to run in zero time. To find the most cost-effective means of speeding up the system, we should therefore consider a sequence of options in each of which one additional function is implemented in hardware, successively reducing the load on the host CPU. To achieve this systematically, we should examine the speed-cost product (or cost/time ratio) of every function, and in successive options implement in hardware the function currently having the lowest value of this parameter: the rationale for this is to preferentially replace in hardware those functions that are slow and whose cost is relatively small, by applying a criterion function with suitable weighting values.

This simple procedure is made somewhat more complex by the significant economies that are possible when implementing functions 6-10, e.g. by using common pixel scanning circuitry. Specifically, any subset of the functions 6-10 can operate with a single interface, scanning circuit and radial position lookup table (which gives a value for radial position once x and y displacements relative to the circle centre are known). On the other hand, any subset of these functions that is not implemented in hardware engenders a time overhead in software. A full analysis of the problem would require a large number of functional partitions to be examined in order to find the optimum system configuration. However, this exhaustive search procedure need not be performed in this instance since the time overhead is much greater than the sum of the software times for functions 6-10. This means that once the initial cost overhead has been paid it will clearly be optimal to implement all of these functions in hardware. For this reason we group functions 6-10 together in the remainder of this paper. Table 2 summarises the position.

Table 2 shows that the cost/time ratios divide themselves into four main categories: (1) those of the order of 1 £/ms which are clearly worth implementing in hardware; (2) those between ~5 £/ms and 25 £/ms which will also have to be implemented in hardware to get a reasonable speed system; (3) those around 100 £/ms which it would be worth implementing if a very much faster system were needed; and (4) those above 1000 £/ms which it would probably never be economical to implement in hardware. If option 1 were chosen, the total cost of the system would be £9000 and the algorithm would run in 0.7 seconds; if option 2 were used, the system would cost £13700 and would run in 0.1 seconds; if option 3 were chosen, the system would cost £23700 and would run in 0.002 seconds, whereas with option 4 the system would cost £27700 and would run in zero time (in the current approximation). Here we have assumed that the base cost of computer plus camera, frame store, backplane, power supply, etc is some £6000 and that this will permit the algorithm to run in ~5.0 seconds as indicated in Table 2.

In the above analysis we assumed that those functions implemented in electronic hardware run in zero time. This will not be entirely valid in practice, and the most serious errors will be for image neighbourhood operations - particularly those for neigh-

bourhoods of size 3x3. Taking 150 nsec as the fastest time for pixel access (as with our implementation using the VME bus), we see that a 3x3 neighbourhood operation in a 128x128 image takes some 25 msec. With suitable local storage this could be reduced to ~8 msec or even to ~3 msec. For a 1x1 neighbourhood, pixel access times would be ~3 msec. Next, let us assume that the actual processing is carried out by TTL circuitry in some tens of nanoseconds per pixel location; then the processing time will be less than 1 msec. Thus quite straightforward circuitry could be used to implement each function in times as short as 3-4 msec: this goes some way to justifying, and extending, the approximation we made earlier.

We now interpret our finding that the cost/time ratios fall into four main categories. Broadly, the first category (cost/time ~1 £/ms) arises for imaging operations in 3x3 neighbourhoods, which are well worth implementing in hardware. The second category (cost/time in the range 5 to 25 £/ms) arises for faster imaging operations in 1x1 neighbourhoods. The third category arises for one-dimensional operations which involve less processing, and certain rather time-consuming floating point operations. And the fourth category is a general data processing category with non-repetitive operations that run so fast they are unlikely to be worth implementing in dedicated hardware. Specifically, functions 5,12,13,14 require tedious logic and/or floating point arithmetic, which means that one is competing with the cost-effectiveness of mass-produced CPUs if one implements them in hardware: in general it is not worth doing this.

Function 4 is at the high end of category 2 since it involves relatively few pixels and is essentially a one-dimensional rather than an imaging operation: in addition, its cost is rather high because it performs quite complex arithmetic.

5.1 More rigorous investigation of hardware-software tradeoffs

We now attempt a more rigorous analysis of the effectiveness of implementing the various functions in hardware. A complete breakdown of the overall cost/time ratio sequence is given in Table 3. t and c are the times and costs of the functions. Assuming an overhead cost of £6000 (see above), T and C are the overall times and costs resulting from implementing in hardware all functions down to the one indicated: the minimum value of T is taken as 0.030 seconds and is based on realistic values for the imaging and 1-D operations, as discussed earlier. Looking at the C*T product should now give an indication of the optimal tradeoff between hardware and software: this occurs for 13 functions implemented in hardware.

It is important to realise that minimising the C*T product only gives a general indication of the required hardware-software tradeoff. A lot depends on the original specification for the inspection system: it might be that the main aim is to meet a certain cost or speed rather than to produce a 'bargain package' that might do well in the market place. In our work, we have aimed particularly at foodproduct inspection, where it seemed to be vital to minimise costs while keeping speeds moderately high (5). For this reason, we aimed at an overall cost of less than £10000. By implementing functions 1,3,6-11 in hardware, we found we could get within a factor 3.6 of the optimal tradeoff (C*T product). However, another important factor arose in this analysis: that was the declining cost of faster CPUs. Table 4 shows the same C*T calculation for an LSI-11/73 host processor replacing an LSI-11/23. In this case the optimum tradeoff again occurs for 13 functions implemented in hardware. However, our compromise of implementing only functions 1,3,6-11 in hardware is now within a factor 1.8 of the optimal tradeoff. It seems fair to assume that these factors will become even more attractive with future CPUs.

5.2 Further factors in hardware design

Some further improvement in performance was obtained by making use of the fact that the host processor and the dedicated hardware can operate concurrently. (Ideally we would gain a factor two in speed by using two processors, but it is clear that our design criteria involve mandatory partitions in the algorithm which are inimical to such a large gain in speed.) In particular, we found that function 2 can run in the host CPU while function 3 runs in hardware, and function 13 can run on the CPU while functions 6-11 run concurrently in hardware. Figure 1 gives an execution map of our implementation, showing that our final allocation of functionality to hardware and software is able to make significant gains in efficiency and speed. This further justifies implementing relatively few functions in hardware.

In our implementation of Algorithm B, we have achieved 25 msec for function 3 (edge detection), and 10 msec for functions 6-11: we are currently upgrading these to roughly double the speeds. At that stage the timings will be as indicated in Figure 1, and at a total cost of £12500 (using an LSI-11/73 with functions 1,3,6-11 in hardware) we will have a system capable of inspecting 11-12 products/second using Algorithm B.

5.3 Generality of the functions implemented in hardware

Algorithm B was partitioned into sections that correspond to a significant degree of generality. First, edge detection itself is a highly general image analysis function (11); second, the Hough transform procedure used for object location is generalisable to a variety of shapes (4); third, the radial histogram approach has the potential for being used even in cases where cylindrical symmetry does not exist, since it can be used to provide a rotationally invariant 'signature' characteristic of one or other part of an object in the region of an easily locatable feature. Finally, certain thresholding operations (e.g. counting the number of pixels darker or lighter than certain threshold values) are exceptionally easy to implement yet generally

useful for object scrutiny.

Clearly, function generality is a crucial factor which will frequently override the C*T criterion in deciding on the priorities for building dedicated hardware. We have kept this in mind while deciding which functions to implement in hardware in our visual inspection work.

6 CONCLUSION

This paper has presented an algorithm for the rapid inspection of small products such as biscuits. It has analysed how this algorithm may optimally be partitioned between dedicated hardware and software. Detailed specifications such as strict speed or cost limits have been seen to constrain the basic optimisation procedure, and function generality is also a critical factor. In addition, it has been found difficult to decide systematically the best ways of incorporating concurrency into the design when processors take radically different forms: however, we have been able to design an inspection system that makes efficient use both of the host CPU and of a limited number of hardware processors. The approach we have adopted seems somewhat unusual in that we have proved it best to retain use of the host CPU for a proportion of the processing rather than to set about building everything in dedicated hardware: specifically, the host CPU is not merely used for control and general data logging, but is used to take an integral role in the main image analysis task. Ultimately, the aim of our work is to develop the methodology of digital hardware design for industrial inspection applications, and at the same time to arrive at optimal designs rather than ones that contain arbitrary sets of ad hoc processors.

Acknowledgements

The authors are grateful to the SERC and to United Biscuits and Unilever for financial support during the course of this work.

REFERENCES

(1) DAVIES, E. R. A glance at image analysis - how the robot sees. Chartered Mechanical Engineer, Dec 1984, 32-35

(2) PARKS, J. R. Industrial sensory devices. Ch. 10 in BATCHELOR, B. G. (editor) Pattern Recognition - Ideas in Practice. Plenum: New York, 1978, 253-286

(3) HOUGH, P. V. C. Method and means for recognising complex patterns. US Patent 3069654, 1962

(4) BALLARD, D. H. Generalising the Hough transform to detect arbitrary shapes. Pattern Recognition, 1981, 13, no.2, 111-122

(5) DAVIES, E. R. Design of cost-effective systems for the inspection of certain foodproducts during manufacture. Proc 4th Conference on Robot Vision and Sensory Controls, London, 9-11 Oct 1984, 437-446

(6) DAVIES, E. R. Radial histograms as an aid in the inspection of circular objects. IEE Proceedings D, 1985, 132, no. 4, Special Issue on Robotics, 158-163

(7) DAVIES, E. R. Precise measurement of radial dimensions in automatic visual inspection and quality control - a new approach. in BILLINGSLEY, J. (editor) Robots and Automated Manufacture, IEE Control Engineering Series 28. Peter Peregrinus Ltd: London, 1985, 157-171

(8) DAVIES, E. R. Image processing - its milieu, its nature and constraints on the design of special architectures for its implementation. in DUFF, M. J. B. (editor) Computing Structures for Image Processing. Academic Press: London, 1983, 57-76

(9) FOUNTAIN, T. J. CLIP4: a progress report. in DUFF, M. J. B. and LEVIALDI, S. (editors) Languages and Architectures for Image Processing. Academic Press, London, 1981, 283-291

(10) HUNT, D. J. The ICL DAP and its application to image processing. in DUFF, M. J. B. and LEVIALDI, S. (ibid) 1981, 275-282

(11) DAVIES, E. R. Circularity - a new principle underlying the design of accurate edge orientation operators. Image and Vision Computing, 1984, 2, no. 3, 134-142

Table 1 Breakdown of algorithm B

	function	description	time (sec)	cost (£)	c/t ratio (£/ms)
1.	acquire image	1x1	-	1000	-
2.	clear parameter space	1x1	0.017	200	11.8
3.	find edge points	3x3	4.265	3000	0.7
4.	accumulate points in parameter space	1x1	0.086	2000	23.3
5.	find averaged centre	-	0.020	2000	100.0
6.	find area of product	1x1	0.011	100	9.1
7.	find light area (no chocolate cover)	1x1	0.019	200	10.5
8.	find dark area (slant on product)	1x1	0.021	200	9.5
9.	compute radial intensity histogram	1x1	0.007	400	57.1
10.	compute radial histogram correlation	1-D	0.013	400	30.8
11.	overheads for functions 6-10	-	0.415	1200	2.9
12.	calculate product radius	1-D	0.047	4000	85.1
13.	track parameters and log	-	0.037	4000	108.1
14.	decide if rejection is warranted	-	0.002	4000	2000.0
	time for whole algorithm		4.960		

Table 2 Revised breakdown of algorithm B

	function	description	time (sec)	cost (£)	c/t ratio (£/ms)
1.	acquire image	1x1	-	1000	-
2.	clear parameter space	1x1	0.017	200	11.8
3.	find edge points	3x3	4.265	3000	0.7
4.	accumulate points in parameter space	1x1	0.086	2000	23.3
5.	find averaged centre	-	0.020	2000	100.0
6-11.	set of functions with same overhead	1x1	0.486	2500	5.1
12.	calculate product radius	1-D	0.047	4000	85.1
13.	track parameters and log	-	0.037	4000	108.1
14.	decide if rejection is warranted	-	0.002	4000	2000.0
	time for whole algorithm		4.960		

Table 3 Speed–cost trade-off figures for LSI-11/23 based system

order	function	t (sec)	c (£)	T (sec)	C (£)	C*T (£-sec)
0	-	-	6000	4.990	6000	29940
1	3	4.265	3000	0.725	9000	6530
2	6-11	0.486	2500	0.239	11500	2750
3	2	0.017	200	0.222	11700	2600
4	4	0.086	2000	0.136	13700	1860
5	12	0.047	4000	0.089	17700	1580
6	5	0.020	2000	0.069	19700	1360
7	13	0.037	4000	0.032	23700	760
8	14	0.002	4000	0.030	27700	830

Table 4 Speed–cost trade-off figures for LSI-11/73 based system

order	function	t	c	T	C	C*T	gain
		(sec)	(£)	(sec)	(£)	(£-sec)	
0	-	-	7000	2.154	7000	15080	1.99
1	3	1.835	3000	0.319	10000	3190	2.05
2	6-11	0.207	2500	0.112	12500	1400	1.96
3	2	0.006	200	0.106	12700	1350	1.93
4	4	0.035	2000	0.071	14700	1040	1.79
5	12	0.017	4000	0.054	18700	1010	1.56
6	13	0.016	4000	0.038	22700	860	1.58
7	5	0.007	2000	0.031	24700	770	0.98
8	14	0.001	4000	0.030	28700	860	0.97

The last column in this table shows the overall gain in speed relative to the corresponding LSI-11/23 option in Table 3.

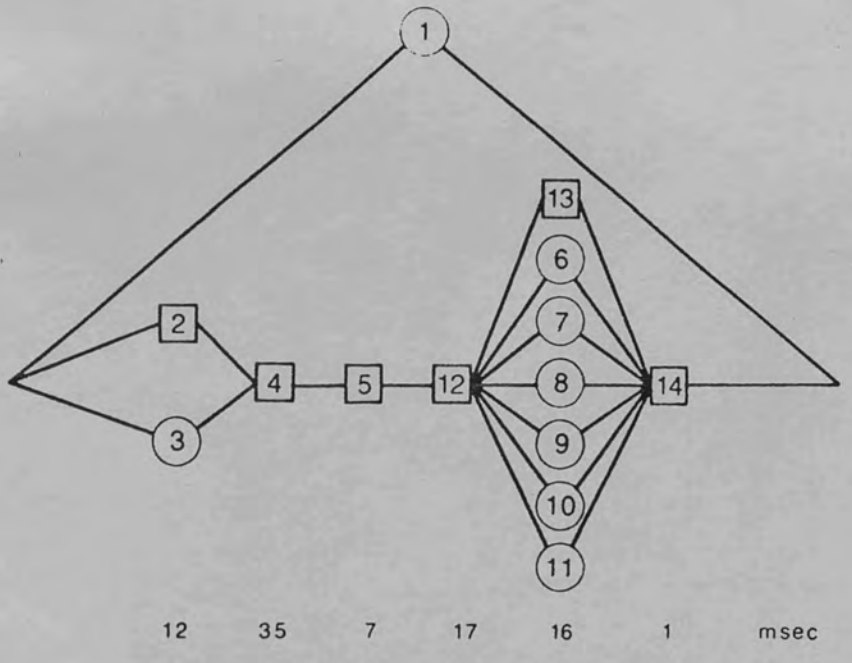


Fig 1 Execution map of algorithm B showing its implementation, making use of:

(a) pipelining of image acquisition and algorithm execution;
(b) simultaneous execution in hardware and software;
(c) sharing of scanning overhead and data I/O for functions six to ten.
Also indicated are the execution times of individual processes totalling 88 ms

□ operations involving host CPU
○ operations executed in dedicated hardware