VISTA - An Image Processing Architecture

by

Terrance R. Ingoldsby

A THESIS
SUBMITTED TO THE FACULTY OF GRADUATE STUDIES
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF
MASTER OF SCIENCE IN COMPUTER SCIENCE

DEPARTMENT OF COMPUTER SCIENCE

CALGARY, ALBERTA
MAY, 1987

ISBN 0-315-38007-1

THE UNIVERSITY OF CALGARY

FACULTY OF GRADUATE STUDIES

The undersigned certify that they have read, and recommend to the Faculty of Graduate Studies for acceptance, a thesis entitled "Vista - An Image Processing Architecture" submitted by Terrance R. Ingoldsby in partial fulfillment of the requirements for the degree of Master of Science.

Supervisor
Dr. Anton Colijn
Department of Computer Science

Professor J.R. Parker
Department of Computer Science

Dr. Graham D. Lodwick
Department of Surveying Engineering

May 15, 1987

ABSTRACT

This thesis examines methods of applying multiprocessing to the field of image processing. Image processing is identified as an excellent task to perform via multiprocessing since images can be subdivided into small regions that can often be treated as separate images. By exploiting this property, it is possible to distribute the processing of the sub-regions amongst a number of separate processors.

A variety of multiprocessor computer architectures are surveyed and their ability to execute common image processing algorithms efficiently is evaluated. The factors that make particular architectures efficient at certain kinds of image processing problems are identified and a powerful multiprocessor architecture, *Vista* is synthesized. An implementation of *Vista* is described and its performance evaluated.

It is found that the *Vista* architecture can be constructed using inexpensive, readily available components. A wide variety of image processing algorithms can be executed in near constant time for a fixed image size.

ACKNOWLEDGEMENTS

I'd like to thank the many people who in any way contributed to the *Vista* project. Professors, students and technical support people have truly given me their advice, and when I was discouraged, friendship.

In particular, this project could never have succeeded without the assistance of Jim Parker. He did much work with no recompense except my heartfelt thanks.

The construction of *Vista* would have been an overwhelming challenge without the technical advice of Patrick J. Irwin, *Supertech* and Bart Hicks (who seems quite comfortable living in frequency space).

I would be remiss if I did not thank my parents, who sacrificed much to help me along the way.

My most special thanks to my wife, Mónica for the years of help, advice and love, and for keeping the home fires burning when I was not there to tend them. Thanks to my daughter Christina for sharing her daddy with a computer, and to Ashley who arrived just in time for the grand finale.

May 15, 1987

Terry

Table of Contents

List of Figures

CHAPTER 1

Improved Image Processing via Advanced Computer Architectures

Most research projects are born out of the need to overcome a problem which has arisen in the course of other work. This project is no exception. The field of *image processing* is an area of computer science whose application finds use in a wide variety of disciplines. These include, but are by no means limited to, medical technology, remote sensing by aircraft and space probes, transmission of images via noisy communication links, and many other diverse applications. Additionally, current research is focussed on the machine interpretation of images; ie. the machine analysis of image data to the extent that automated identification of image content can be performed. This automated interpretation of image content, often called *machine vision*, has almost unlimited potential for use.

1.1. Architectural Considerations in Image Processing

Images, by their nature, contain great amounts of data and information. The processing of this information by conventional computers is frequently a time consuming task. This severely limits the use of *image processing* and *machine vision* in areas where large numbers of images must be processed in a short period of time. Since certain physical constraints limit the ultimate speed of conventional, single processor computers, it is widely accepted that markedly improved performance will only be possible by the use of parallel (multiprocessor) machines. Numerous attempts have been made to create multiple processor machines for use in *image processing* [1 2] with varying degrees of success. These efforts have used widely varying approaches. Some use common, easily available components while others use highly customized systems which are frequently very expensive. As might be expected, the specialized, expensive systems generally give better performance than the systems using 'garden variety' parts. This expense limits the use of *image processing*.

It was with the hope of extending the tool of *image processing* to users who required very high performance, but without great expense, that the *Vista*[†] project was born. The aim of the *Vista* project was to

[†]Vista - Spanish noun - vision, view or scene

examine the endeavors of other researchers, learn from their experiences and thus design a powerful but inexpensive *image processing* architecture. To prove the feasibility of the proposed architecture, a working implementation was built. This thesis seeks to analyze the problems encountered in creating a high performance *image processing* machine and to describe the solutions that can be used to overcome them. In particular, a rationale will be provided for the design decisions made in the *Vista* architecture.

1.2. Image Processing with Multiprocessors

Two great problems stand in the way of effective multiprocessor systems. The first is the question of how to efficiently divide a given task amongst the various processors, and the second is to devise a means of communicating information between processors and processes. Clearly, both problems are highly interdependent. The architecture of the machine may limit the ways in which tasks may be divided or interprocessor communication may occur. The distribution strategy must take advantage of both machine structure and the characteristics of the algorithms to be executed. Thus it would seem that a parallel processing machine should be designed with some consideration of the algorithms it is most likely to execute. The identification of these algorithms is, in itself, a non-trivial task since the most popular algorithms might not be used if other more powerful algorithms could be executed in a reasonable period of time.

1.3. Interprocessor Communication

Multiprocessor computer systems frequently need to pass information between the various processors in the system. A variety of interprocessor communication methods have been devised. The communication strategies commonly proposed involve the passing of information in one of three ways. The first, and simplest method, is to connect a number of processors on a common bus. When one processor has a message for another, it simply sends the message across the bus. Unfortunately, *bus saturation*, a condition where the bus is transferring information at 100 per cent capacity, can occur with relatively low numbers of processors. The exact number of processors accommodated before this occurs depends on the actual design of the bus and the algorithm being executed. One virtue of this method is that the component systems required are easily (and commercially) available.

The second technique, which utilizes a communication scheme that provides a separate channel between each and every processor, is in some sense the best, at least from a point of view of

communication. Unfortunately, this connection scheme, known as the *crossbar switch*, is impractical since it requires approximately N^2 switch units to service N processors.[3] Thus for even small numbers of processors the number of switching elements quickly becomes unwieldy. It should be noted that some such machines have been built, the most well known of which is probably the C.mmp machine.[4] Several switching strategies have been proposed which provide nearly the same connection facilities as the *crossbar switch*, but with reduced component count.[5] Although these schemes are an improvement over the standard *crossbar switch*, they are still difficult to implement for large numbers of processors. Even with practical numbers of processors, it is apparent that such a strategy would likely be somewhat expensive.

A third method involves the passing of information in packets. A given processor is connected to a particular subset of the other processors in the system. If it needs to communicate with one of these processors, it sends the message directly to it. If communication with other processors, not directly accessible, is required, the message must be relayed along a path of processors. Clearly it is desirable to connect the processors and arrange the tasks such that a minimum number of steps is involved in transmitting the information. Much effort has gone into devising a variety of connection schemes[6] [7] including meshes, pyramids, trees, etc. For large numbers of processors these limited connection strategies are feasible to construct. Much research is needed to devise good connection topologies, as well as software to place processes in appropriate places within the topology and to permit efficient routing of messages between processors. Until large numbers of processors and switching elements can be implemented on single integrated circuits, constructing such architectures from discrete components will remain costly and difficult.

1.4. Design Criteria

It was desirable to devise an architecture that could be easily constructed using readily available components. This would serve to advance the state of the art by permitting a prototype machine to be built that could be used for experimenting with different algorithms and techniques. It was tolerable that the *Vista* machine be general purpose only within a particular field or narrow range of fields. As was mentioned previously, in this case it is for the processing of images. A justification for the choice of *image processing* will be offered shortly.

To aid in the design of the *Vista* architecture, a set of criteria was established. The criteria were that the system:

1) Should be implementable using existing, available technology; preferably reasonably priced components (due to budget constraints).

2) Should not be tied to any particular device or component (eg. processor).

3) Must provide demonstrably better performance than single CPU machines.

4) Must be flexible and easily programmed within the chosen field(s) of interest.

These four, simple criteria, lead to early rejection of both the *crossbar switch* and packet transmission methods of communication. The former because of the expense, difficulty of construction (really best suited to VLSI) and the fact that parts for this scheme are not easily available. Those *crossbar switches* that are available were usually developed for the telecommunication industry, and are not particularly well suited to digital computers. The latter was rejected as having too many problems to resolve. It is not likely that the difficulties associated with packet systems will be solved for many years to come, and the desire was to build a working machine. Additionally, no consensus has emerged on what connection strategy is best, and once a particular design was constructed, revising the connection system would be nearly impossible. It would seem that, for the time being, packet architecture research would best be accomplished via simulations. This leaves the lowly bus communication method as the choice for the *Vista* system. A method was devised that reduced the amount of information transferred across the bus, thus minimizing bus contention. The techniques used, described in chapter 3, form the crux of the *Vista* system.

1.5. Suitability of Multiprocessors to Image Processing Problems

Some computer algorithms are computation bound. In these cases, performance will be dictated by how fast the necessary computations can be performed. In other cases, particularly where large data sets are involved, the speed of the operation may be dictated to some extent by the speed at which data can be transferred to the computer.[8] For example, a geophysical data processing company recently developed a custom specialized bit slice processor that executed certain programs at 80 Million Instructions per Second (Mips), only to discover that it was very difficult to move the data quickly enough to fully exploit the power of its high performance CPU.[9] Although no proof is offered, it seems that a particular process is

more likely to be Input/Output (I/O) bound if large data sets are involved.

At the very least, it is clear that the processing of data cannot occur more quickly than the acquisition of the data. Computer images typically involve many hundreds of thousands, or even millions of picture elements (pixels). These images are typically acquired by digitizing a real image and then transmitting it from the digitizing device to the computer via a serial or parallel data channel (bus). Because of certain engineering constraints, parallel channels are rarely more than 32 bits wide. Even if the channel is of high bandwidth, transmission of the data can take a sizable length of time (~hundredths of a second). These time lags are imperceptible in most cases to humans, but are long in terms of computer time. If data are being processed for display to humans in a time so short as to appear instantaneous (say 1/30 second), then any further reduction in processing time will not be noticed. One readily apparent way to minimize the data transmission time is to connect each pixel in the data sensor directly to a processor. The advantage of this technique is that all pixels can be transferred to the processors simultaneously. It is of interest that research is being carried out on machines which involve one processor/pixel.[10] Combining the data sensor and the processor unit may be practical some day, but not in the near future. Thus, for the time being, image data will likely continue to be transmitted by buses. Having hinted that the processing of images is a reasonable activity to pursue with a multiprocessing machine, a brief examination of the operations commonly performed on images is in order.

1.6. Image Processing Properties and Objectives

Image processing usually attempts to accomplish one of two things. Either certain features of an image are caused to stand out (enhancement) or statistical properties are employed to remove 'noise' that may have somehow corrupted the image. *Image processing* by machine has existed since the early 1900's when methods were developed for removing noise from photos transmitted across trans-oceanic cable.[11] The advent of the space age and its attendant satellite probes, coupled with the development of the modern computer, inspired more sophisticated methods for restoring and enhancing the images received from remote sensors. A wide variety of standard, cookbook techniques for achieving certain effects were developed. Most of these algorithms share the common characteristic of operating on a large number of pixels (or groups of pixels) in much the same way. Because of this property, some highly specialized

machines have been created which take advantage of the repetition inherent in some algorithms. Unfortunately, these machines are often so specialized in architecture as to be difficult to reprogram for other algorithms.

Perhaps the most attractive reason to choose *image processing* as an application for multiprocessing is because of a property which will be referred to as *locality*. Briefly stated, in many cases it is a good assumption that pixel values in one part of an image have little dependence on values in another, distant region of the image. While this assumption is by no means universally applicable, many *image processing* algorithms exploit this tendency. This in turn allows one to assign different regions of an image to different processors with relatively little modification of existing algorithms. These regions may be rows, columns, or 'patches', but if a processor does not need data from outside its assigned region, little interprocessor communication is required. *Vista* is designed to exploit this factor to the utmost. Aside from this feature of *locality*, many *image processing* algorithms operate on all pixels (or groups of pixels) in much the same way. This means that processes could easily be distributed so as to provide near equal load to the various processors in the system.

1.7. Other Potential Applications of the Vista Architecture

A challenging new field in Computer Science is that of *machine vision*. Whereas the function of *image processing* is usually to bring out or enhance existing information in an image in order to make it more intelligible to humans, *machine vision* seeks to cause the machine to interpret the information in the image directly. Since to date limited success has been achieved in creating algorithms, parallel or otherwise, to accomplish machine vision, it is difficult to anticipate whether or not the *Vista* architecture will be useful in this field. Since *Vista* is flexible, it will be more useful than many of the special purpose image processing machines which currently exist. These special purpose units are frequently unable to cope with algorithms that involve large numbers of branches (decisions) based on image content.

One concept that may be used in machine vision is that of *adaptive memory*. This technique involves using smart memory cells that could be told to perform operations on themselves. *Vista* can appear to be an *adaptive memory* system, if not at the memory cell level, then at least with small blocks of cells. This idea also fits in well with some of the new so called *object oriented* languages. Even if *Vista* is not directly

applicable to *machine vision*, it would seem likely that *machine vision* would require pre-processing of an image by standard *image processing* techniques. In this case, *Vista* will at worst be useful as a 'front end' to a second vision machine.

1.8. The Vista Project

As is the case with all computer architectures, *Vista* represents a compromise of many conflicting requirements. Most notably, the decision was made to sacrifice the performance that, given technological advances, will be possible in the future, for the privilege of constructing and experimenting with a multiprocessor using the technology that is currently available. To provide an understanding of the *Vista* project, the remaining chapters correspond to the various phases of *Vista* research. Chapter 2 examines the characteristics of common *image processing* algorithms. This study is made in order to identify traits that are common to the various *image processing* algorithms, and exploit these characteristics in the design of an *image processing* machine. Chapter 3 provides a brief survey of some of the architectures that have been proposed as solutions to *image processing* problems. In Chapter 4 the strengths and weaknesses of these proposals are analyzed and, coupled with the knowledge of image operations garnered in Chapter 2, a new architecture, *Vista*, is proposed. Chapter 5 describes a prototype implementation of *Vista* while Chapter 6 discusses the software needed to program the prototype. The final chapter summarizes what was learned from the *Vista* project, what improvements might be made, and proposes new directions that future *Vista* research might take.

CHAPTER 2

Image Processing Algorithms

In order to apply multiprocessing techniques to image processing, it is necessary to determine which characteristics are common to most image processing algorithms. This in turn forces the identification of the algorithms most likely to be used in an image processing system. Unfortunately this is not as simple as it might seem. One way to do this, and the method which will be pursued herein, is to consult a reasonable sample of the literature that discusses image processing and from this identify which algorithms are universally discussed. This is not a totally adequate method since some algorithms, currently seldom used because of computational expense, might well prove to be popular if more powerful (eg. parallel) computing machines were available. No better selection scheme presents itself, so notwithstanding the limitations noted, discussion will proceed as indicated.

It is convenient to divide the field of image processing into classes of operations. While no universal consensus exists on how to partition the operations, most authorities follow approximately the divisions suggested by Castleman.[12] It should be emphasized that some overlap exists between the operation classes and that they will be used chiefly to lend structure to the study of image processing algorithms.

Two general goals are usually kept in mind when an image is processed. The first is to enhance the image. Enhancing an image is generally construed to mean emphasizing certain characteristics of the data being processed so as to make them 'stand out'. In other words, the image is modified so as to be more easily interpreted by humans or machines. The enhanced image may or may not look like a normal image. For example, images obtained via infrared sensing devices are often displayed by assigning various gray levels or colours to particular temperature ranges. If colour is used, it is not uncommon to assign so called 'cool' colours (such as blue) to regions of lower temperatures and 'hot' colours (such as red) to hot spots. An image so enhanced and displayed might appear very different than would a normal photograph.

It should not be assumed that more conventional images cannot be enhanced. Consider an image which consists of a number of objects on top of a similarly shaded and coloured background. This would

normally make the identification of the foreground objects difficult. By enhancing the image so as to increase the contrast difference between the two shades, the task is made much easier.

The second major category of image processing concerns itself with the removal of noise and distortion from data. The noise may be random, such as might be expected in the case of data transmitted from distant space probes, or it may be systematic, such as occurs when a sensing device distorts or changes data in a predictable, consistent fashion. In either case, the goal is to restore the image, as far as possible, to an uncorrupted state. This can be done by using statistical properties of the data or knowledge about the deficiencies inherent in the sensor.

A variety of techniques exist for enhancing and restoring images. Most of these techniques are more useful in performing operations in one category than the other. Still, some overlap exists which may allow use in either category. For example, frequency domain filtering techniques (discussed in Section 2.4.1) are useful for both edge enhancement and noise removal. For this reason the assorted techniques are considered without distinction to category of application.

2.1. Point Operations

The simplest operation that can be performed on an image involves mapping an input pixel value to an output value. The mapping of a given pixel requires no information from any other pixels. Rather, given the input value, the new output is either calculated as a function, or looked up in a table. Furthermore, the function (or table) used is the same for all pixels in the image, regardless of spatial position. The operation requires data from a single pixel point only, hence the term *point operation.*

While very simple, *point operations* are nonetheless useful. In fact, they are probably the most common of all image operations. When performing image enhancement, they may be used to accentuate particular data values or gray levels in the image. This technique is known as *thresholding*. Another popular application is to 'stretch' the contrast of a poorly exposed photograph. Image restoration often involves compensating for non-linearities which exist in sensing equipment. When the non-linearity is spatially invariant, *point operations* are effective.

For obvious reasons, *point operations* are computationally very cheap. Since the number of possible values a pixel may contain is limited by the resolution of the sensing device, look-up tables are commonly

used to perform the mapping. The analog to digital (A/D) converters found in sensors are commonly 8 bit devices, and rarely exceed 12 bits; table sizes therefore range from 256 to 4096 elements. The mapping is easily achieved by modern microprocessors with very few instructions. For example, using only a small 8 bit processor, such as the MC6809, the operation could be accomplished as follows:

```
        LDX #IMAGE_ADDRESS ;Get pointer to image
        LDY #MAP_TABLE ;Pointer to table
        LDU #NUM_PIXELS ;Number of pixels
LUP:    LDA ,X ;Get the pixel value
        LDA A,Y ;Translate value
        STA ,X+ ;Store new value
        LEAU -1,U ;Decrement counter
        BNZ LUP
```

One iteration of the loop requires only 23 machine cycles or about 11.5 μsec/pixel. A 256 x 256 pixel image could be mapped in about 0.75 seconds. More advanced microprocessors could easily accomplish the same operation in 1/10 to 1/100 the time.

Although the algorithm used for point mapping is readily amenable to parallelism, it is not at all clear if it is worth the effort. Conceivably each pixel of the image could be assigned to a separate processor, each with its own conversion table. In this manner the entire image could be processed in a few microseconds. If there are less processors than pixels in the image, a likely situation, then each processor could be made to process several pixels, allocated by any reasonable scheme. Unfortunately, if the output of the computations is displayed for human analysis, very little performance improvement will be noted, since even a single processor machine operates near the limits of human perception. Nonetheless, in cases where the output is to be analyzed by machine (eg. robot vision), the speed-up may be useful.

2.2. Algebraic Operations

Another important category of operations manipulates images by performing algebraic operations on the (corresponding) pixels in two or more images. One or more of the images used in the computations may be artificially (computer) generated, or all may be images obtained from sensing equipment. The

operations used are usually a linear combination of the following:

$$C(x,y) = A(x,y) + B(x,y)$$
$$C(x,y) = A(x,y) - B(x,y)$$
$$C(x,y) = A(x,y) \times B(x,y)$$
$$C(x,y) = A(x,y) \div B(x,y) \tag{2.1}$$

There is no reason why other, non-algebraic operators might not be used instead of +, −, ×, ÷ if they are useful in a given application.

Although the possibilities are limitless, there are several common ways in which the operators are applied. Sometimes, several nearly identical images are operated on. A good example is a technique frequently used for removing noise from images received from remote sensors, such as spacecraft. Over the span of time required to digitize an image, a space probe's 'view' often changes very little. By averaging several nearly identical images the random noise becomes less noticeable, thus improving image quality. If the images are not spatially aligned, due to spacecraft movement, techniques exist to align the two; these will be discussed in Section 2.3. Another possibility involves the combining of two (apparently) unrelated images. A trivial example of this might be the superposition of longitude/latitude lines onto an aerial photograph.

The final common approach is to artificially create an image which is then used to modify another. For example, a mask might be created with the value 1 at all pixels within a region of interest, and 0 everywhere else. When the mask is multiplied by another image, only the region of interest remains. The utility of this will become apparent in Section 2.4 where *transformation operations* are discussed. A variation of this idea is used when it is necessary to compensate for a sensing device which has varying sensitivity or non-linearity at different coordinate points. An algebraic filter could be prepared to correct for these shortcomings. This filter might multiply the input image by values ranging from 0 to 1.0, depending on the spatial location.

As in the case of *point operations*, large amounts of parallelism exist in *algebraic operations*. Here, however, the operations being performed are often far less trivial than the simple mappings employed in *point operations*. In fact, since floating point arithmetic may be involved, the computations would require

a sizable length of time for a single processor machine. Because of the lack of interaction between pixels in a given image, a simple scheme to divide the task between various processors is easily created. This strategy involves practically no interprocessor communication.

To divide the task requires only to divide the input images into spatially identical regions. Corresponding regions are passed to the various processors along with a description of the operations to be performed. All processors may thus process their regions concurrently. Since the regions can be made equal in area, each processor will perform nearly identical amounts of computation. This equal load 'balancing' is a desirable characteristic of distributed algorithms.

2.3. Spatially Dependent Operations

All of the operations considered thus far have not depended in any way on the spatial coordinates of an image. While useful, they do not take advantage of the fact that adjacent pixels are somewhat dependent on one another. Almost any image (except perhaps random noise) does exhibit regional dependency. Additionally, to this point the algorithms discussed have always mapped input pixels to their original spatial positions in the output. This is quite limiting in many situations. Consider the following possible applications of spatially dependent operations.

Suppose an image, contaminated by noise, has been acquired. Further suppose that this noise is random in nature, and (as is common) tends to be of high frequency (ie. affects only one or two pixels in a region). The noise might be made less noticeable by interpolating the values of noisy pixels from their neighbours. Of course there is no way of knowing which pixels have been affected by noise, so all pixels must be interpolated. Since pixels not contaminated by noise will usually have values somewhat similar to uncontaminated neighbours, interpolation will cause relatively little change in value. Even if one of the neighbours is noisy, the majority of adjacent pixels have correct values; the detrimental effect will be reduced. This sort of strategy produces a smoothing effect on the image. A wide variety of interpolation methods can be employed; each having particular characteristics (computational cost, number of adjacent pixels used, weighting factors, etc.) recommending them for use in the appropriate circumstances.

Although not as obvious as in the previous categories of operations, concurrency can also be found in the above example. One approach would be to first set up two frame buffers; one containing the input

image and the other storage for the smoothed image. The images could be divided into a number of overlapping regions. Each region would be centred at the pixel to be processed and would include all pixels needed for the interpolation. The regions would usually be small, since most interpolations require only the central pixel and those pixels immediately adjacent to it. Regions would be assigned to any free processor which would compute the interpolated value and store it in the appropriate location in the output frame buffer. As before, interprocessor communication is not required.

In the previous example, the 'spatial' aspect of the operation was due to the differing values of pixels at different spatial locations. Other kinds of spatial dependency exist. For example, it is sometimes necessary to spatially translate an image. This translation process is known as *geometric transformation*. When the translation is an integral number of pixels in the x and y directions, and it is the same throughout the image, the process is relatively simple. Some mapping strategy can be invented which, given the original x, y coordinates, determines the new x, y values. The situation is rarely this simple; frequently the amount of translation varies at different positions in the image, and often a pixel will be translated to new locations 'between' legitimate pixel points. Thus the mapping is dependent on spatial position, and should be capable of interpolating pixel values from neighbouring pixels when translation occurs to points 'between' pixels.

If translations were random or discontinuous (in the sense that initially adjacent pixels translated to widely separated locations) then it is doubtful that meaningful concurrency could be found. In fact, even a single process would be messy because of the discontinuous functions involved in the mapping. As an aside, let it be noted that this very strange, non-continuous mapping is proposed for use by the commercial satellites to scramble 'Pay-TV'.[13] Fortunately, it is rarely necessary to consider such exceptional situations. The more usual case occurs when an image is figuratively printed on a rubber sheet, and then stretched in the desired directions. In any particular translation scheme, no pixel will be translated more than Δx, Δy pixels from it's original location. To allow concurrency in processing the algorithm, as before, input and output buffers are provided. Each processor is given a copy of the data from a region centered around the pixel being translated and $2\Delta x \times 2\Delta y$ in size. The output pixel location is mapped (or more specifically, reverse mapped) to find the location of the input pixel. Its value is either read or interpolated as necessary. Once again no interprocessor communication is necessary.

2.4. Mathematical Transformations

Sometimes a completely different strategy is required from those examined thus far. It can be beneficial to exploit certain mathematical properties of the image by changing its coordinate frame. This is not an uncommon approach in many disciplines. For example, it is not unusual in physics to change from a Cartesian coordinate system to polar or spherical coordinates. This is done because many problems which are difficult in one coordinate frame are straightforward in another. Prior to the age of pocket calculators, it was not uncommon to calculate the product of large numbers by adding the logarithms of the numbers (found in a table) and then taking the anti-log of the sum. Some calculators still use this technique for calculating x^y. Analogous procedures exist for image processing.

The most popular transform used in image processing is without question the Fourier Transform. Some others are the Walsh, Hadamard and Hotelling Transforms. Because of its popularity, and the fact that most other transforms are used in analogous ways, attention shall be focussed on the Fourier transform and the algorithm used to calculate it.

2.4.1. The Fourier Transform

It is interesting to note that image processing by computer is a relatively recent phenomenon. Before the use of digital computers, images were processed optically; for some purposes (especially where speed is essential) optical image processing is still used.[14] Many of the techniques used in optical image processing relied on the fact that a Fourier transform is easily computed using a converging lens. When an object is placed in the front focal plane of a converging lens, its Fourier transform is present at the back focal plane. By manipulating the transformed image appropriately, operations that are very difficult to perform on an untransformed image are easily carried out (this will be demonstrated shortly). Synthesizing this transform operation in a digital computer is entirely possible, and offers some advantages (and admittedly disadvantages) over the optical case. To more fully appreciate what can be accomplished via the Fourier transform, attention is turned to what it is, why it is useful, and how it may be computed in a digital computer. Although images are two dimensional in nature, and therefore require two dimensional transforms to operate on them, first consider the one dimensional (1D) Fourier transform; the concepts discussed are easily extended to two dimensions (2D).

Consider the function F: f(x) = A sin(x), shown in Figure 2.1a. This is certainly one way of describing the function; another might be to simply say that it is a sine wave of amplitude A, frequency 2π, and phase shift 0 (w.r.t. the origin). In other words, it may be assumed that the properties of a sinusoidal wave are well known and all that is necessary is to specify the parameters that may differ from sine wave to sine wave. Of course it is not obvious how this method might be used for other functions, since an infinite number of different functions exist, many of which require many parameters to specify their behaviour. It is offered without proof that any function that is continuous over the region of interest may be expressed as a sum of sinusoidal waves of different amplitudes, frequencies and relative phases. This includes practically all the functions (or data sets representing unknown functions) that might be measured by instruments (such as digitizing cameras) in the physical world.[15] In Figures 2.1b, 2.1c, and 2.1d an example is given illustrating how a square wave may be synthesized by summing sinusoidal waves. It can now be seen that, similar to the method proposed for Figure 2.1a, the square wave approximation in Figure 2.1d might be described by specifying the parameters of the sine waves which comprise it. The parameters may themselves be expressed as a function. This is necessary since a large, or even infinite number of sinusoidal waves may be needed to compose a given function. Notice that since three parameters (amplitude, frequency and phase) are necessary to describe a sinusoidal wave, a single 1-D 'real' function is not capable of a complete description. For example, if a real, 1-D function were used then the abscissa might indicate frequency and the ordinate axis amplitude. A function would then describe a continuum of sinusoidal

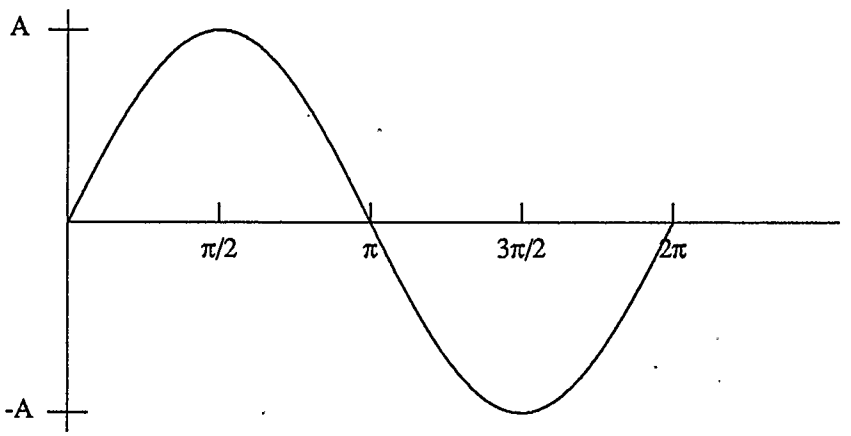

Figure 2.1a - F: f(x) = A sin(x)

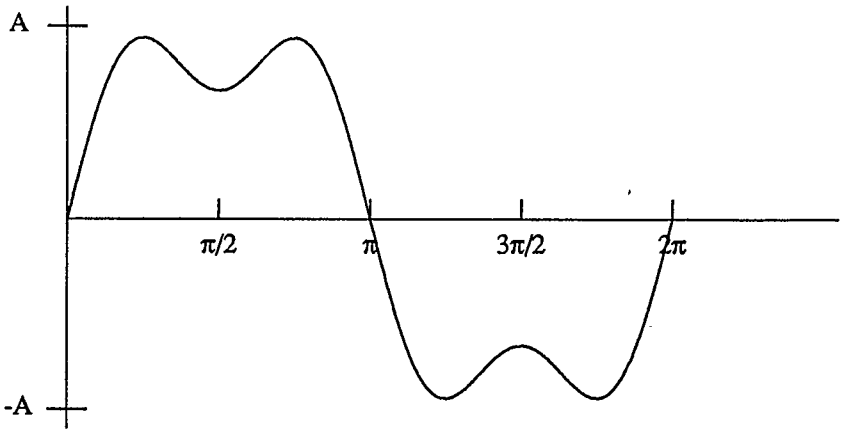

Figure 2.1b - F: f(x) = A (sin(x) + sin(3x)/3)

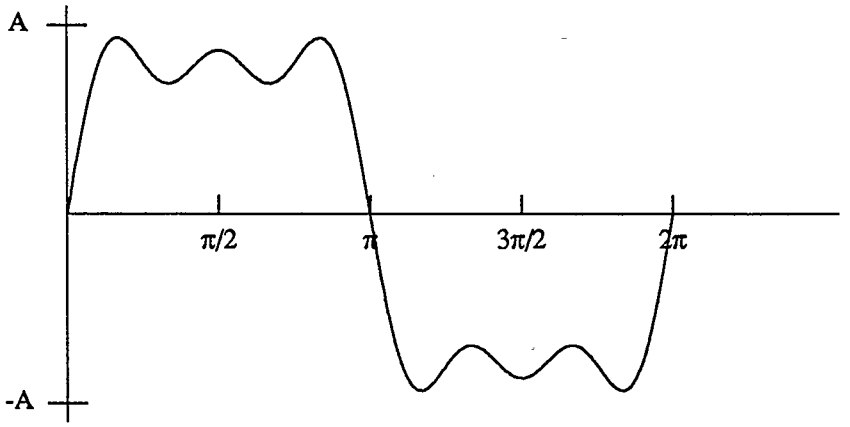

Figure 2.1c - F: f(x) = A (sin(x) + sin(3x)/3 + sin(5x)/5)

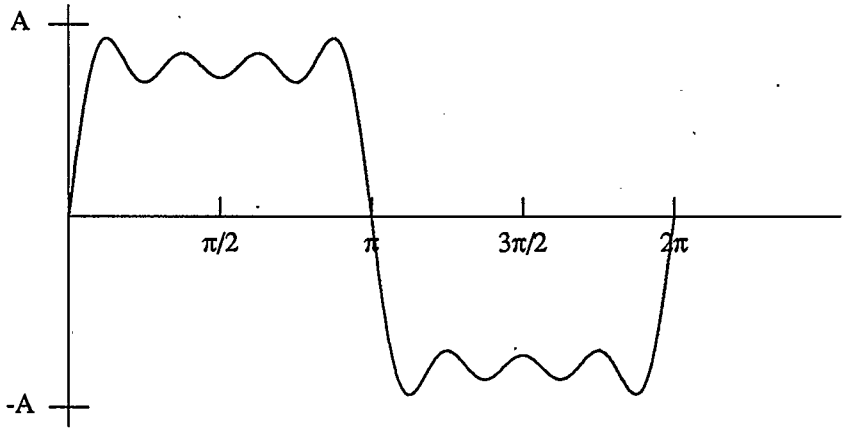

Figure 2.1d - F: f(x) = A (sin(x) + sin(3x)/3 + sin(5x)/5 + sin(7x)/7)

waves of various frequencies and amplitudes, but the relative phase of these waves would remain ambiguous. A second function describing phase as a function of frequency would be necessary to completely

describe the sinusoidal waves.

Alternately, a complex function may be used to completely specify the characteristics of the sinusoidal waves needed to synthesize a particular function. This is possible because a sinusoidal wave of any phase value may be specified by a sum of sine and cosine waves. If the real part of a complex function is made to describe the amplitude of cosine waves required and the imaginary part the amplitude of the sine waves then together they specify the amplitude and phase of the sinusoidal waves. Complex functions are thus commonly used to specify the amplitude, frequency and phase of a continuum of sinusoidal waves. The mechanism used to move from the conventional representation of a function to this rather strange sum of waves is known as the Fourier transform.

Exactly why this transformation of coordinates is desirable is not immediately obvious. The common representation of a function, often known as the time domain (since it specifies a function's value w.r.t. time) is close to how most humans visualize the world around them. It is largely due to this familiarity that so much can be accomplished in the time domain; manipulations of functions are analogous to the world as perceived by humans. It should not be too surprising that if some familiarity is developed with the alternate coordinate system, known as the frequency domain, many interesting operations can likewise be carried out. While not presuming any degree of mathematical rigor (those interested may consult the cited references) an attempt will be made to illustrate how thinking in frequency space can facilitate many operations.

An understanding of the basic concepts of frequency domain filtering can be gained by studying the field of audio reproduction. Consider again the square wave approximation shown in Figure 2.1d. Notice how the 'flat' part of the square wave is largely defined by the low frequency components comprising it. The rising and falling edges depend mostly on the high frequency components to give them shape. It turns out that to perfectly synthesize a square wave by summing sinusoidal waves requires an infinite number of sinusoidal components (some involving 'infinite' frequency).

When high fidelity audio reproduction is desired, one important factor is the frequency response of the amplifiers used in the reproduction process. Square waves (or more precisely waves whose vertical edges are sufficiently steep so as to appear square to the testing equipment used) are applied to the input of an audio amplifier. The output of the amplifier is then examined with an oscilloscope. Any deficiency that

affects the high frequency components of the square wave (such as high frequency roll-off) causes easily visible distortion in the transition edges. Low frequency shortcomings manifest themselves in distorted 'flat' edges. Thus it may be inferred that, if it is desirable to 'smooth' a function, it is only necessary to remove the high frequency components. Conversely if only the edges are of interest then the low frequency components of the signal may be attenuated. These operations are similar to passing the signal through an inductor or capacitor, respectively. Of course, these operations may have some side effects on other parts of the function being processed, but largely accomplish the objectives.

Until recently it has been more convenient to perform audio signal processing in the time domain. With the increasing use of computer technology in the audio field, this may eventually change to the frequency domain. In these cases it will be necessary to convert the audio signal from time to frequency domain by way of a Fourier transform. After conversion to frequency space the various frequency components of the signal can be amplified or attenuated as is required. After the desired operations have been performed in the frequency domain, it is necessary to somehow get the function back into a time domain representation if it is to be meaningful to most humans. Fortunately it can be shown that an inverse Fourier transform exists that will perform the desired coordinate change. In fact, it is one of the remarkable properties of the Fourier transform that the inverse transform is computed in almost exactly the same fashion as the original forward transform. For example, except for the sign of the exponential, the two equations are identical.

$$f(t)=\frac{1}{\sqrt{2\pi}}\int_{-\infty}^{\infty}F(x)e^{-i2\pi tx}dx \qquad F(x)=\frac{1}{\sqrt{2\pi}}\int_{-\infty}^{\infty}f(x)e^{i2\pi tx}dt \tag{2.2a,b}$$

Analogously to the 1-D Fourier transform there exists a 2-D Fourier transform. This transform is the same as that alluded to earlier in the reference to optical image processing. It is now evident how optical image processing might be performed. By placing an image at the front focal plane of a converging lens, the Fourier transform of the image appears at the back focal plane. A specially prepared filter, opaque at the undesired frequency component coordinates and transparent elsewhere, can be used to remove certain frequencies. If a second lens is placed such that the Fourier transform generated by the first lens is located at the second lens's front focal plane, then the second lens will calculate the inverse transform at its back focal plane. (See Figure 2.2). If no masking or filtering had been performed on the Fourier transform of

the image, then the final output would be identical to the input. Because of the filtering, the output image will be modified consistent with having had certain frequencies removed. To smooth an image, for example, a mask could be prepared which blocks the high frequency components. This technique is known as spatial filtering.

Anyone with experience in the use of optical equipment will at once appreciate both the advantages and disadvantages of spatial filtering. The transforms are computed in a few nanoseconds! On the other hand, optical equipment is notoriously fussy. Of more fundamental concern is the difficulty in preparing the filter masks needed for each operation. Simple amplitude filters (affecting all phases equally) are quite possible, albeit with the nuisance of, in most cases, processing photographic emulsions. More complicated masks, affecting both amplitude and phase, are very difficult or impossible to make. It is for these reasons that spatial filtering is often synthesized in a computer; almost any mask can be generated easily and quickly.

The main difference between the real, analog world and the digital world is the requirement that functions be discrete rather than continuous. For use by computers, any device that senses an event occurring in the real world must necessarily convert continuous function values to discrete numeric values sampled at (usually small) finite intervals in time. If the sampling frequency is too coarse, or the range of

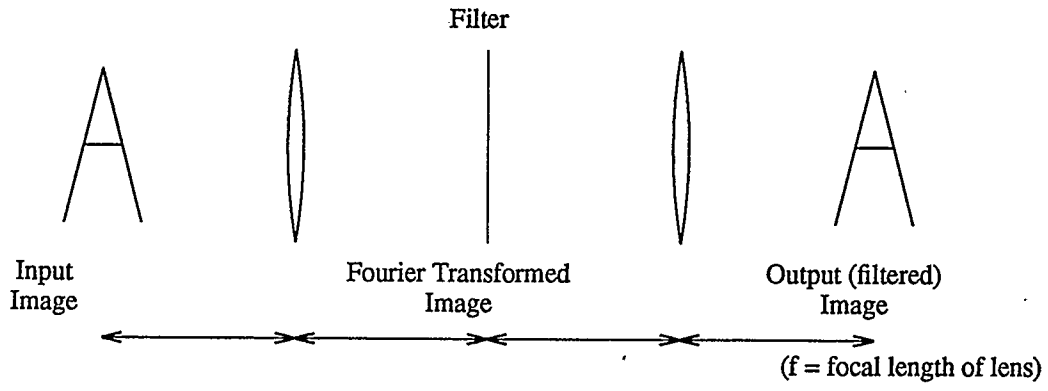

Figure 2.2 - Optical Image Processing

numeric values used does not have sufficient dynamic range to adequately represent the function, many problems can occur. The purpose of the present discussion is to demonstrate how Fourier transforms may effectively be computed using multiprocessor computers, but much work has been done on the ramifications of undersampling and restricted dynamic range[16].

The first attempts to simulate the Fourier transform in a computer attempted to replace the integral with a summation.

$$f(t) = \frac{1}{\sqrt{N}} \sum_{x=0}^{N-1} F(x) exp[-i 2\pi t x / N] \quad where\ t=0,1,...,N-1 \tag{2.3}$$

This Discrete Fourier Transform (DFT) computed the desired values, but was very expensive computationally, requiring $O(n^2)$ calculations for a single dimension DFT, 'n' being the number of data samples analyzed. This severely limited its applicability, since even large computers were incapable of computing transforms of large data sets in any reasonable amount of time. In 1964 J.W. Cooley and J.W. Tukey invented an algorithm for computing the 1-D Discrete Fourier Transform in O(n log(n)) time. This made it feasible to compute even large data sets using computers. It is interesting that Cooley and Tukey were not the original inventors of the algorithm that now bears their name; it was originally invented by G.C. Danielson and C. Lanczos in 1942 for use with hand calculators but was subsequently forgotten[17]. For obvious reasons, the new algorithm, as well as all similar ones, are known as Fast Fourier Transform (FFT) algorithms. FFT algorithms compute exactly the same results as an ordinary DFT. Aside from the speed of computation only two significant differences exist between the two; a DFT can be computed for any number of data points whereas an FFT can only be used on certain size data sets. The 'magic' sizes vary depending on the exact FFT algorithm used, but many (including Cooley-Tukey) insist that the data set size be a power of 2. Of less importance to the field of image processing is the requirement that data samples be equally spaced when the FFT is used. The DFT has no such restriction.

Although the FFT algorithm proposed by Cooley and Tukey was only for single dimensional data, the extension to two dimensional data is simple. A 2-D DFT can be calculated by taking 1-D DFTs of all the rows in an image, followed by 1-D DFTs of the columns. This is due to the fact that the formula for the 2-D DFT

$$F(u,v) = \frac{1}{N}\sum_{x=0}^{N-1}\sum_{y=0}^{N-1} f(x,y)exp[-i2\pi(ux+vy)/N] \tag{2.4}$$

can be separated and expressed in the form

$$F(u,v) = \frac{1}{N}\sum_{x=0}^{N-1} exp[-i2\pi ux/N]\sum_{y=0}^{N-1} f(x,y)exp[-i2\pi vy/N] \tag{2.5}$$

allowing the 2-D DFT to be calculated in two stages. Since the FFT is interchangeable with the DFT, it is of course used instead.

Although the reasoning used to derive the FFT algorithm from the DFT algorithm is interesting, it will not be considered here. Rather, an attempt will be made to outline what the algorithm does and to demonstrate that the potential exists for computation using parallel processors.

The FFT algorithm is best described diagrammatically (see Figure 2.3). If the columns in the signal flow graph are numbered from 0 to 4, then the data at column 0 are the original input data and the data at column 4 are in a scrambled form. This is

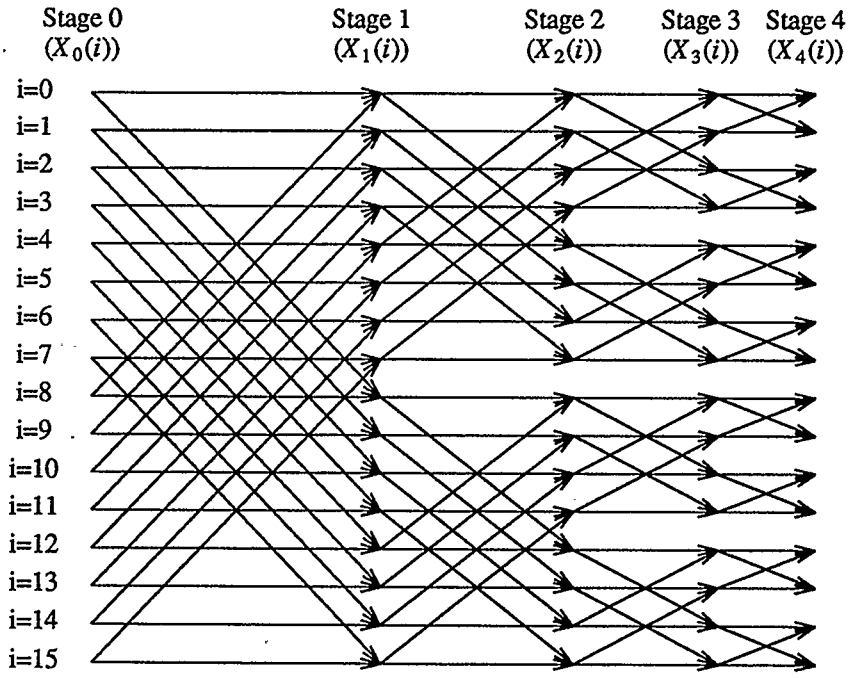

Figure 2.3 - Signal Flowchart for Cooley-Tukey FFT Algorithm

Horizontal arrows 'bring' the value at the tail of the arrow multiplied by a twiddle factor. The twiddle factor is a function of data set size, 'stage' and 'i', but does not depend on data values. Diagonal arrows 'bring' the value found at their tails. Where horizontal and diagonal arrows meet, the sum is calculated.

easily corrected using an O(n) computation. This complication will be ignored for the time being. Columns 1 through 3 represent the data at intermediate stages of computation. Consider how a particular data point is computed. Each node in the graph (except of course the input node) is determined by adding the value 'brought' to it by a diagonal arrow to a second value which is obtained by multiplying the value 'brought' by a horizontal arrow with a value known as the *twiddle factor*. For example at level 1, data point 4 is calculated by:

$$X_1(4) = X_0(12) + wX_0(4) \tag{2.6}$$

where w is a *twiddle factor*. The *twiddle factors* do not depend on the data; rather, they are characteristic of a particular FFT size and can be precalculated and stored in a table for repeated use. It is thus seen that each data point requires one multiplication and one addition. It can also be seen that each data point is operated on $\log_2(n)$ times. This means that nlog(n) multiplications and additions are required. (In practice, a periodicity in the twiddle factors allows half the multiplications to be omitted; to simplify the discussion this is ignored).

The conventional method for calculating the FFT is to calculate each column in sequential order. This is not the only possible sequence.[18] [19] From the signal graph it can be seen that it might be possible for each data point to have its own processor. Some sort of communication scheme could be devised such that as each processor computed a value it would transmit the value to the appropriate processors for the next column; one being itself and the other dependent on the column and row in the signal graph. This would reduce the calculation to O(log n) time for the 1-D FFT! Unfortunately, the communication scheme would have to be quite sophisticated, possibly some variation of the crossbar switch. All processors would need to be capable of communicating simultaneously with another processor. This switching system would be very expensive.

If the absolute maximum in parallel processing is not demanded, more economical methods are available. Notice that at each column, the diagonal paths sub-divide into regions. For example, notice that in column 2, data points 0 through 7 depend only on data points 0 through 7 in column 1. No values from other data points in column 1 are needed. This means that after the first column is calculated, the data could be divided amongst two processors which could thereafter work independently. After the next column was complete each of these processors could similarly divide their load between two more proces-

sors. This could continue until the entire FFT was complete. This would require at maximum 'n' processors; however, if this many were not available the sub-dividing of tasks could stop at any column. In fact this may be desirable, since each additional group of processors has only half the workload of the previous group. In any case, this requires no processor to compute more than 2n data points, ie. O(n) (see Figure 2.4). Although not as efficient as the previous example, the performance gain is substantial without the communication overhead of the first method.

The methods described above are worthwhile only when it is necessary to compute large 1-D FFTs. Most image sensing devices currently available provide less than 1024 x 1024 size images. Since each 2-D FFT is computed as a sequence of 1-D FFTs, it is not clear that the benefits would outweigh the additional overhead, particularly with such small data sizes. For the 2-D FFT, much more can be gained by having a separate processor calculate each row (and column) in the image. If insufficient processors are available, each processor could calculate a number of rows (columns). Only when a one processor/row ratio is achieved will it become worthwhile to pursue the 1-D parallelism techniques. In any case, the current expense of the requisite number of processors would prohibit such a system.

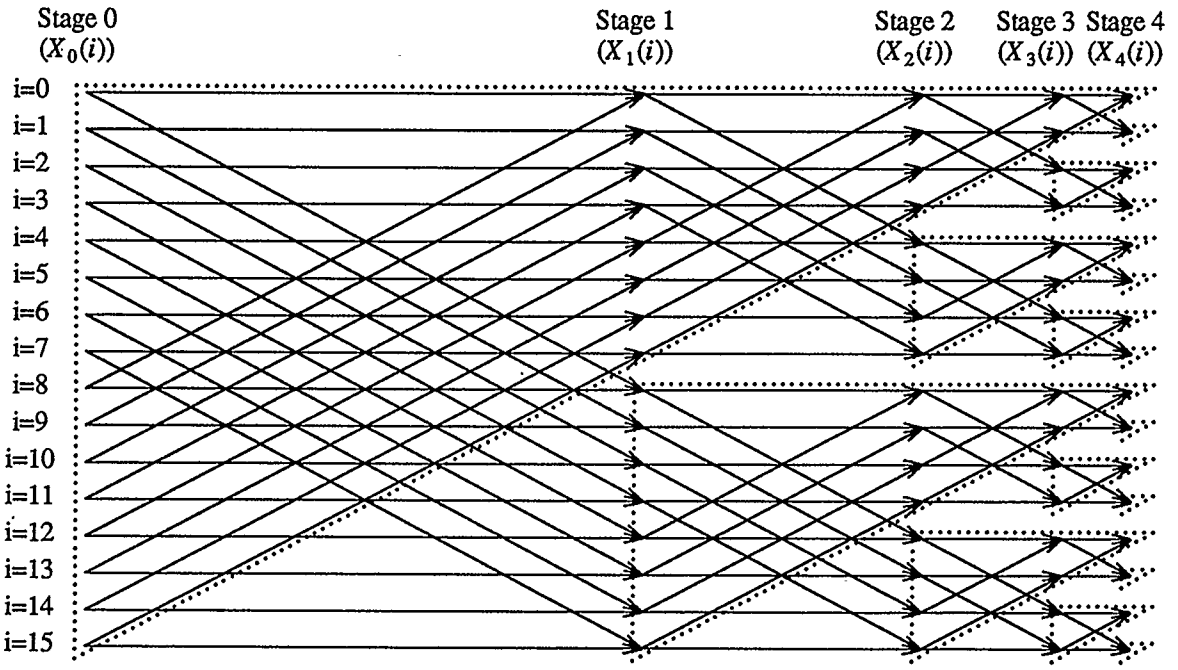

Figure 2.4 - FFT Signal Flowchart divided into sub-processes

2.4.2. Other Transforms

Although the most popular transformation used in image processing, the Fourier transform (and its digital counterpart, the FFT) is by no means the only transform used. Others include the Walsh, Hadamard and Hotelling transforms. Some of these (eg. Walsh and Hadamard) possess properties similar to the Fourier transform. This allows fast O(nlog(n)) algorithms to be devised for them. For others (eg. Hotelling), no fast algorithm exists. Most, especially those similar in character to the Fourier transform, can be computed by a number of independent single dimension transforms. This makes them readily amenable to parallelism using the same approach shown for the Fourier transform.

Although the techniques examined are by no means the only ones applied to image processing, they do represent a good sample of methods in current use. Chapter 3 examines how some designers have approached the problem of parallel image processing, while Chapter 4 uses the characteristics of image processing algorithms to propose a new image processing architecture.

CHAPTER 3

Image Processing Architectures

Having examined the salient characteristics of popular image processing algorithms, the question must now be asked, "What machine architecture would be most suitable to implement these algorithms?". In an attempt to answer this question, a survey will be made of some of the image processing architectures proposed by other researchers (Sections 3.2.1 - 3.2.4). The strengths and weaknesses of each architecture will be analyzed and the best ideas synthesized into a new and powerful image processing architecture (Chapter 4).

3.1. Architectural Classification of Computer Architectures

Before considering specific architectures, it is necessary to establish the terminology which will be used to describe the wide variety of architectures to be considered. For the most part, Flynn's[20] classification scheme will be used. Flynn classified architectures based on the manner in which instructions and data flow through the machine. The four basic classes of machines described by Flynn are: Single Instruction Single Data (SISD), Single Instruction Multiple Data (SIMD), Multiple Instruction Single Data (MISD), and Multiple Instruction Multiple Data (MIMD). A 'normal', Von Neumann computer is one example of an SISD machine since single streams of instructions and data flow through the machine. An instance of an SIMD machine is the so called *array* processor, where one instruction affects a number of data elements simultaneously. MISD machines are frequently pipeline architectures, where a single stream of data flows through a sequence of processing units, all operating concurrently. The most esoteric of the classes, MIMD, describes those machines which can simultaneously execute several or many instructions on more than one data element. MIMD machines are at once the most difficult machines to design and use, both from hardware and software perspectives, and those which offer the greatest potential for performance.

Flynn's classification method, while useful, is often too general to allow meaningful comparisons between different architectures. Other more specific taxonomies have been proposed to describe the con-

nection structures used in multiple processor machines.[21] While these classification methods are useful, the nearly infinite number of interconnection strategies taxes the ability of any classification strategy to make useful comparisons between specific architectures.

Other important factors besides performance should be considered when evaluating a computer architecture. In the case of multiprocessor machines, traits such as fault tolerance, software transparency, and modularity are all important. For example, if a machine incorporates a large number of processing units, the probability is significant that one or more of them may not always be operational. How well the machine copes with this problem may determine its feasibility. Related to this is the question of whether the software can automatically cope with varying numbers of processors. This in turn relates to the 'modularity' of a design. A design is said to be modular if few changes are needed to other components when an additional processor is added. Another aspect of modularity is whether or not an additional processor significantly enhances system performance.

3.2. Effective Image Processing Architectures

Recalling Chapter 2, several different types of image processing operations are identified. One class of algorithms permits pixels within small regions to be processed independently. A region process may use the values of certain pixels within the region to process other pixels in the region, but no information is required from other regions, thus allowing concurrent processing of regions. Another class of algorithms, those which perform mathematical transforms on the data, often requires communication within rows or columns, but outside of the row or column under consideration, no intercommunication is required. This allows many transforms to proceed concurrently. A successful image processing architecture should take advantage of these traits, as well as being easy to program, reliable (fault tolerant) and cost effective.

Before designing a new architecture, it is wise to consider what others have done before, thus benefitting from their experience. It would be difficult to survey all of the different architectures that have been proposed; therefore a representative sample will be chosen for discussion.

3.2.1. Bus Based Multiprocessors

Most contemporary computers are implemented using a 'bus' structure. A bus is a collection of (conceptually) parallel wires. Each wire provides a communication path for a particular computer signal,

such as data, address lines and control lines. To these wires are connected (in parallel) all components of the computer system that require those particular signals. The greatest problem with bus oriented multiprocessor architectures is their tendency to bottleneck due to bus saturation. Any bus has a limit to the amount of information that can be transferred across it in a given time. If several processors use the bus to fetch both instructions and data from a shared memory, then they may be forced to wait until other traffic on the bus clears. This clearly limits the number of processors that can be beneficially interconnected via a bus. The bus architecture is not without its compensating features. Up to the point of bus saturation it is very modular. Additional processors can be added with little or no modification to other existing equipment. Many attempts have been made to sidestep the problems inherent in the bus architecture.

3.2.1.1. FLIP - A Flexible Bus Oriented Image Processing Machine

One effort to overcome the limitations of bus based systems is found in the FLIP system.[22] The FLIP system consists of three parts. (See Figure 3.1). A large frame buffer for image storage, a Peripheral data Exchange Processor (PEP), and 16 Flexible Individual Processors (FIP). All three units are attached to the bus of a host computer. In addition to the main host bus, several other buses exist between system

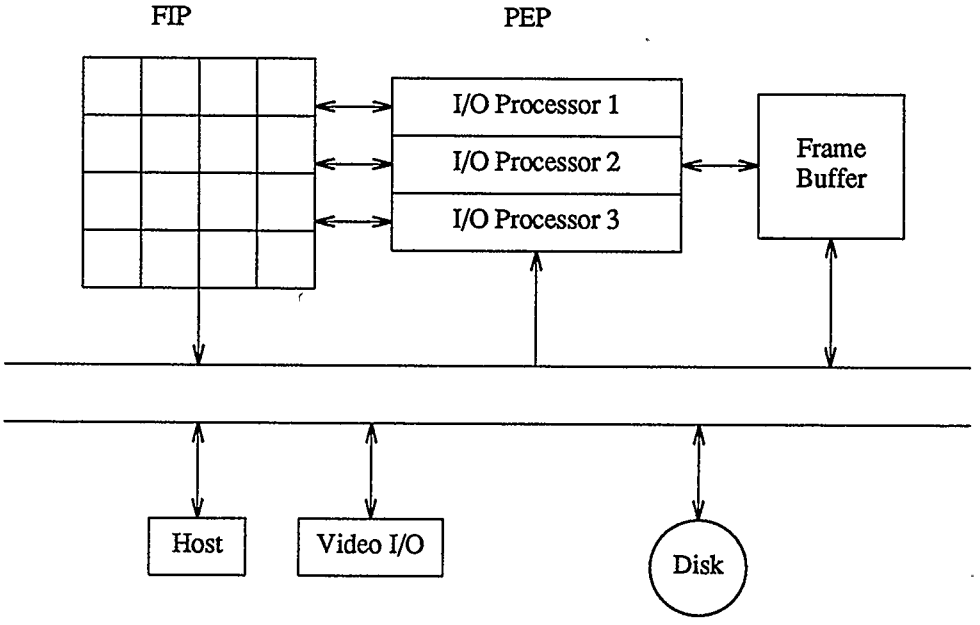

Figure 3.1 - The FLIP Architecture

components. A separate, internal bus exists between the FIP units, allowing them to communicate with each other directly. The FIPs receive data and information from the image memory and the host computer via the PEP. The PEP acts as a high speed cache, loading information from the host computer bus, and transmitting it on to the FIPs. To achieve sufficient bandwidth between the PEP and FIPs, the PEP outputs data via three high speed buses which in turn service 16 medium speed buses (one per FIP). The intuitive notion that using several buses should reduce bus contention is supported by Lang[23] who claims that N processors using more than N/2 buses is nearly as effective as a crossbar switching scheme.

FLIP is usually programmed using the *data flow* paradigm. That is, the 16 FIP units pass data packets to each other over the internal bus. These packets contain not only data, but information about the task which should be performed on the data. Due to the communication scheme, FLIP can be used to implement a wide variety of other paradigms. Flexibility is one of the strongest features of the FLIP system. FLIP seeks to overcome the bus saturation problem by using multiple buses, and employing a special data transfer unit. It largely succeeds at this, though it should be noted that only a modest number of processors can be so connected (the actual implementation used 16). System performance is approximately 10 Million Instructions per Second (MIPs).

3.2.1.2. PICAP - A Bus Oriented MIMD Machine

Researchers at the Linkoping University in Sweden use another approach to overcome bus overload.[24] Their PICAP II machine employs a very high speed bus (40 Mbyte/second) to which is connected an equally fast image memory system and a number of special purpose processors. Each processor may operate independently from the others, but is capable of performing only specialized tasks. For example, a *filter processor* is used to perform the necessary filtering operations.

Although PICAP II does in many ways achieve its objectives, it nonetheless has serious deficiencies. To achieve the needed memory speed, only two possibilities exist. The frame buffer may be made of very high speed RAM, an intolerable expense for the memory size required to store images. This is particularly relevant to the PICAP design, since PICAP allows its processors to simultaneously work on several images, or intermediate images, at a given time. The alternative to using fast RAM is to construct the memory in an interleaved fashion. This technique involves arranging memory into distinct banks, such that adjacent

memory addresses lie in different banks. If, as is common, memory locations are addressed sequentially, the effective speed of the memory can be made to appear N times faster that it really is, where N is the number of memory banks. This was the alternative chosen by PICAP's designers. Unfortunately, it is not always convenient to use algorithms that access memory in such a rigid format. If a program should make successive accesses to memory locations that lie within a single bank, memory speed deteriorates to the rated speed of the memory circuitry.

Another problem with PICAP is its use of highly specialized processors. This makes the design highly fault intolerant, since if even one processor fails, it is impossible for the others to take up the slack.

There are many redeeming features in PICAP, however. Most lie in the design of the specialized processing units. Consider the aforementioned *filter processor*. It consists of a SIMD (array) processor which accesses values from a local memory buffer. The local memory buffer contains a strip of the image which has been placed there prior to performing the filter operation. This allows the filtering operation to proceed without use of the high speed system bus. Since a particular datum may be used several times, many system bus accesses may be eliminated.

3.2.2. Data Flow Architectures - Flexible Pipelines

Recently, a growing number of researchers have been advocating the use of the *data flow* paradigm. The *data flow* paradigm in many ways represents a flexible pipeline processing mechanism. To understand the differences between an ordinary pipeline and a *data flow* pipeline, consider the time-worn analogy of an assembly line.

In an assembly line the product being constructed is passed from assembly station to assembly station. At each station, a particular, rigid task is performed. For example, in an automobile assembly line one station may build the chassis, the next may install the drive train, another the body, the interior, and so on. In this type of assembly line, all cars come out of the factory identical. Each station can perform only one particular, never changing task. This is totally inflexible, especially if the workers at the different stations are capable and equipped to do a variety of tasks.

A few, minor modifications to the assembly line discussed above can result in a much more productive factory. As before, the assembly line is comprised of a number of work stations. These stations are

now manned by workers who possess the skills and tools needed to perform any automobile assembly operation. This time the flow of automobiles through the plant is somewhat different: the unit under assembly is forwarded to the next assembly station with a list of needed operations attached to it. Upon arrival at a station, the workers read the list and perform a necessary task, then pass the unit on to the next station with a new list of operations to be performed. In this way many different automobiles may be produced, some with different colours, motors, etc.. This pipeline is different than the first because each station is flexible in the tasks it can perform, and the task it does perform is based on information sent with the unit under construction.

The same characteristics that make the flexible assembly line desirable are also useful in a computer architecture. Instead of the conventional pipeline, where each processor performs a fixed task, it is possible to transmit command instructions along with the data as it flows from processor to processor. In this way the processors perform different tasks depending on the data received. This is known as the *data flow* paradigm.

3.2.2.1. The ImPP Data Flow Architecture

Recently, a *data flow* pipelined processor has been implemented using Very Large Scale Integration (VLSI) technology by the NEC Corporation.[25] This processor chip, known as ImPP (Image Pipelined Processor) consists of 10 pipeline modules interconnected by a one directional bus which is capable of transmitting both data and their associated instruction tokens. An upgraded, commercial version is reportedly now available for use.

Several characteristics stand-out in the ImPP design. Firstly, the so called *Von Neumann bottleneck* is eliminated. The *Von Neumann bottleneck* occurs when a processor is forced to wait while instructions are fetched. This normally is not a major concern when a single processor is fetching instructions from a memory bank, but can become a serious problem if several processors are attempting to get instructions from a single memory unit. The *data flow* architecture avoids this by transmitting commands along with the data. These commands are then executed using instructions contained in each individual processing unit. Another potential advantage, yet to be realized, is the possibility of creating software that would automatically be partitioned for concurrent execution. Many compilers already make use of a *data flow*

graph structure to aid in generating executable code. *Data flow* graphs are close in structure to *data flow* architectures, thus providing the hope that programs can be written in a non-concurrent fashion, allowing the compiler to determine the concurrency.

Despite their growing popularity, *data flow* pipelines are not without drawbacks. Some algorithms, particularly mathematical transforms, are not yet easily coded on *data flow* architectures. Additionally, just as in ordinary pipelines, it is necessary to balance the load equally among all processors in the pipeline, or performance may suffer greatly due to a single overloaded processor.

3.2.3. Mesh Architectures

There are a number of variations of the mesh design, but all versions interconnect a large number of processors in a symmetrical fashion. The most common topology is a grid, but hexagons, trees and other schemes have been proposed.

3.2.3.1. MPP - A Cellular Logic Array

All designs dealt with thus far have consisted of relatively few processors. Mesh architectures frequently consist of three orders of magnitude more processors. The Massively Parallel Processor (MPP)[26] consists of over 16,384 processors arranged in a 128×128 grid. Each processor is a very simple single bit unit capable of only a limited number of operations, a 1024×1 bit RAM, six registers and an adder. Each processor can communicate with the four units above, below and on either side of it. The MPP is an SIMD architecture, so all 16,384 processors execute the same instruction on the data contained in their registers and memory. Most instructions are executed in 100 nsec.

Since most images have several bits of data associated with each pixel, it is necessary in most cases to perform each calculation as a series of smaller, serial calculations. For example, to add two 8 bit images requires a sequence of eight single bit additions for each pixel in the image. Some researchers feel that this is an advantage; only the number of bits in use are calculated. Since most machines use a word size that is an integral power of 2, they claim that operations on pixels represented by a non-integral power of 2 bits is wasteful. Others feel that the need to perform operations in a serial fashion is non-intuitive. This of course could be solved by making each processor have a larger word size, such as 8 or 16 bits. Unfortunately processors of this complexity would be nearly impossible to interconnect using discrete components and are

currently beyond the capabilities of VLSI engineering for chip level implementation. With the component density of integrated circuits increasing yearly, this limitation may not always be insurmountable.

The most attractive feature of MPP and other similar architectures is the great similarity between the form of image data and the architecture of the machine. In many cases, this allows algorithms to be devised which closely fit the architecture, and are easily and efficiently implemented. Unfortunately, many other algorithms, such as the Fourier transform, are not so transparently applied. It must be conceded that in most benchmarks MPP outperforms other image processing architectures. In cases where the algorithm does not fit the architecture, MPP often still wins because of the massive number of processors used. Performance in these cases comes from throwing a large amount of computing power at an inefficient solution. Whether this is cost effective can only be decided by the end user.

Due to the relationship between the processor array and the structure of the data (bit map) MPP and other similar architectures are highly fault intolerant. The large number of processors also makes it likely that a processor may fail. MPP provides for this eventuality by having spare columns of processors which may be switched in should a processor fail. This further adds to the expense and complexity of the system.

3.2.4. Systolic Arrays - Making the Most of a Memory Fetch

A generalization of the pipeline architecture is the systolic array. Like an ordinary pipeline, values are received from source cells, computations performed and results passed on to the next cell in the pipeline. Unlike conventional pipelines, a *systolic array* can take other forms besides a one dimensional array. Two, three or even higher dimension arrays are at least theoretically possible. Additionally, a conventional pipeline only passes results to the next processor in the pipe, whereas in a *systolic array* both input and results may be passed on. One reason that *systolic arrays* are efficient is due to the fact that as much 'mileage' as is possible is derived from each memory access. Since both input and results are propagated through the pipe, one memory fetch can be used by many processors at different stages throughout the pipe. This reduces the effect of memory speed on the time required to process data.

While *systolic arrays* are quite good at a variety of image processing tasks, they are not without drawbacks. As is pointed out by Hillis,[27] arrays of higher than two dimensions may be impractical to construct. Due to the fixed connection strategy used in *systolic arrays*, they are sometimes handicapped by

their inability to send results directly where they are most needed. Of course given a particular task it would probably be possible to design a *systolic array* that would transmit data to the desired nodes, but that arrangement might not prove suitable for other algorithms. An example of this is seen in the manner in which Fourier transforms are commonly computed on *systolic arrays* [28] [29]. Instead of using the Fast Fourier Transform algorithm, the less efficient Discrete Fourier Transform algorithm is generally used. Performance is gained by the large number of processors available. A linear *systolic array* of N processors can perform an N point DFT in O(N) time. A two dimensional *systolic array* can compute a one dimensional transform in $O(N^{1/2})$ time! Of course achieving this performance requires the creation of a systolic array with the necessary (large) number of processors. This, in turn, requires VLSI technology (particularly in the case of the 2D array).

CHAPTER 4

Synthesis of an Efficient Image Processing Architecture

The architectures examined in Chapter 3 are representative of techniques which have been used in order to achieve high performance image processing. Each architecture previously described has certain characteristics. By understanding the characteristics that make them desirable, as well as their shortcomings, it is hoped that an improved architecture, *Vista*, may be created.

Basically three classes of machines have been examined. MIMD machines such as PICAP, are extremely powerful in concept, but often the particular implementation does not allow the full theoretical power to shine through. Many bus oriented MIMD processors are severely limited by the *Von Neumann* bottleneck. This occurs because every data item typically requires many instructions to manipulate it. If many processors are attempting to process a large data set, and instructions must be fetched over a common bus, a *Von Neumann* bottleneck is sure to occur. Also, a given algorithm may require each element of the data set to be fetched several times. Where large data sets are involved, as in image processing, a very large number of bus transactions may be necessary.

SIMD machines, such as MPP, have a conceptual advantage in certain cases since humans often think in an SIMD way. For example, if there exists a 2D array of data, it may be desirable to add '1' to all elements of the array. The typical, non-computer programmer would probably describe such an operation in exactly that way, ie. add '1' to all array elements. This would, of course, be exactly how such a problem would be specified to an SIMD machine. At the same time, the SIMD structure limits the ways in which the parallelism inherent in a problem may be exploited. A particular solution might be inherently parallel and still not be expressible in an SIMD environment. Since a many data are manipulated with few instructions, SIMD architectures do provide relief from the *Von Neumann* bottleneck mentioned earlier.

MISD machines in all their incarnations tend to reduce *Von Neumann* bottleneck, but they, like SIMD units, cannot efficiently implement some algorithms. In many of the more sophisticated designs, they do make full use of each data fetch performed. This eliminates unnecessary or redundant data fetches.

Some architectures are heavily algorithm dependent. Upon reading the literature, it quickly becomes noticeable that papers discussing particular styles of architectures often discuss certain kinds of algorithms to the exclusion of others. The underlying cause of this seems to be whether or not the communication paths within the architecture in question are suitable for a particular algorithm and whether or not the concurrency of the algorithm 'fits' the architecture well. What is really needed is an architecture that can behave like any of MIMD, SIMD or MISD as required!

4.1. Solutions to Common Pitfalls in Bus Oriented Multiprocessors

In considering how such a versatile architecture could be designed different options were considered. All but one required great expense, esoteric communication schemes, and usually application of VLSI technology. Thus attention was focussed on whether or not a bus oriented machine could be made to fulfill the goals established in Section 1.4. Since the main stumbling block to a multiprocessor bus architecture is bus contention, consideration was given as to how this might be reduced.

Most engineers tend to counter the bus saturation problem by increasing the amount of information the bus can carry in a given time. Unfortunately, this approach meets with quite limited success. The difference in performance between a somewhat mediocre bus and a high performance bus is only about one order of magnitude. The cost and design challenges faced when dealing with a high speed bus are often several orders of magnitude greater than the low performance bus. Instead, the approach was taken that, rather than increasing bus capacity, the need for bus traffic would be reduced. If this could be accomplished the same net effect would be garnered. Indeed, if additional performance were still desired, then a high speed bus could be used as well.

Kung[30] notes, "The ultimate performance goal of a special-purpose system is - and should be no more than - a computation rate that balances the available I/O bandwidth with the host". He continues to explain that a system can operate no faster than it can receive data *unless* "multiple computations are performed per I/O access . . . the repetitive use of a data item requires it to be stored inside the system for a sufficient length of time". Thus, the I/O problem is related not only to the available I/O bandwidth, but also to the available memory internal to the system. Herein lay the key as to how a bus system might be made to render high performance image processing.

The first step is to increase the multiprocessors' I/O channel capacity. This scheme was partially used by FLIP which employed a special processor unit (PEP) that had the dedicated task of getting data to the flexible processors. Nonetheless the FLIP flexible processing units could not process while data were being received or transmitted. An architecture would be more efficient if the processing elements could perform computation with I/O carried on transparently in the background. This seemed to require that all data needed for a particular calculation be loaded in a single I/O operation. These data would correspond to a region, row or column in an image, depending on the algorithm involved. To eliminate the repeated loading of any given data item, and to provide storage for the area of the image under consideration, sufficient memory would have to be available on board each multiprocessor unit. Additional performance could be gained by making the onboard memory very high speed. Since the memory could be small compared to the frame buffer, cost would not be prohibitive. Each processor would be able to make data fetches very quickly, thus performing computations in a minimum of time. The final decision was to make the architecture MIMD. This was done to satisfy the criterion of flexibility since an MIMD machine can often appear to function as MIMD, SIMD and MISD.

Having chosen to use a bus to implement the new Vista architecture, it is useful to consider the problems that develop when a conventional bus oriented multiprocessor is used to execute image processing algorithms.

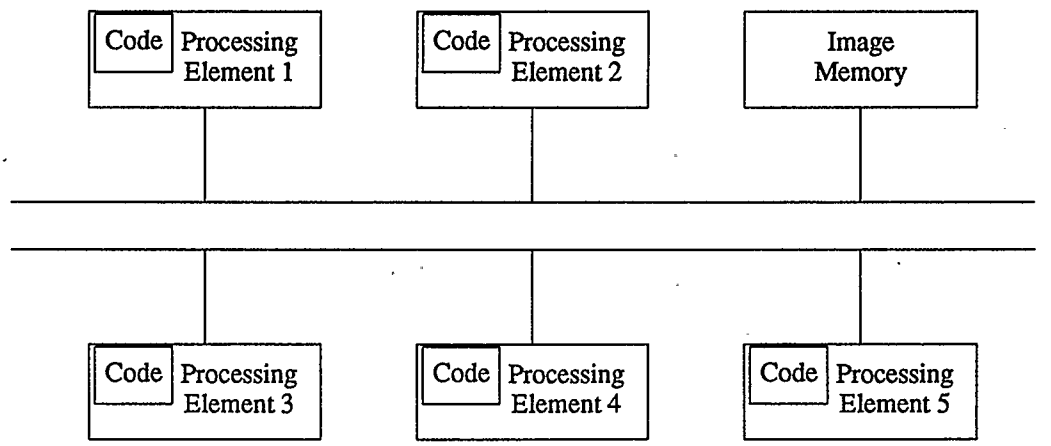

Figure 4.1 - Conventional Bus Oriented Multiprocessor

Suppose a multiprocessor machine, such as that shown in Figure 4.1 is used to perform a neighbourhood smoothing operation on an image. The simple architecture used consists of a number of processors, each with private code memory, connected to a bus. Also connected to the bus is a frame buffer containing the image data to be processed and storage for the output image. To process a particular pixel the weighted sum of the surrounding pixels is averaged together to give a smoothed value for the pixel in question. For example (see Figure 4.2), if pixel 'F' were being processed, its value would be given by

$$F=c_1A+c_2B+c_3C+c_4E+c_5F+c_7G+c_8I+c_8J+c_9K \quad \textit{where } c_1,\ldots,c_9 \textit{ are constants} \qquad (4.1)$$

Performing this operation requires ten bus transactions, nine fetches from an input frame buffer and one store to an output frame buffer. Suppose now that pixel 'G' is to be processed in the same way. It is now necessary to fetch pixels B, C, D, F, G, H, J, K and L across the bus, compute the new value for G and store the result. Notice, however, that six of the nine fetches performed were somewhat redundant, since pixels B, C, F, G, J and K were fetched during the smoothing of pixel F. If the two smoothing operations were both performed by the same processor, and the processor had some onboard memory, the redundant fetches could be eliminated. In this simple case the communications required over the bus would be reduced by 60 per cent. Other image processing algorithms achieve even greater reductions in bus activity. The degree to which communication is reduced depends on how many times the algorithm in question uses an item of data.

Two situations arise in image processing in which onboard memory may substantially reduce bus activity. The first was illustrated by the neighborhood operation mentioned above. In this case it is seen that the minimum regions used by the smoothing operation overlap; ie. some of the pixels used to calculate

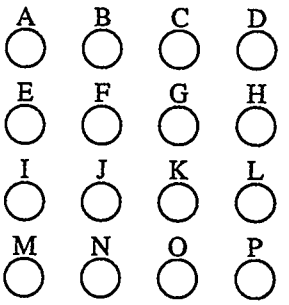

Figure 4.2 - A Small Region of an Image

F are also used to calculate G. If onboard memory is sufficient to store a region of the image containing several adjacent minimum regions, bus activity will be reduced since the values fetched across the bus will be used a number of times per bus transaction. Thus it is seen that bus activity will be reduced by the presence of onboard memory. Some algorithms have no overlap in their minimum regions. Onboard memory can still provide a performance improvement in these cases.

Consider the case of the 2D FFT algorithm. It is convenient to think of the 2D FFT as being composed of two separate sub-operations; the computation of the 1D FFT's of the rows in the image, and the subsequent calculation of the 1D column FFT's. The rows (and columns) may be thought of as the minimum regions of the algorithm, since a row (column) contains all the information needed to compute a 1D FFT. In this case the minimum regions are completely disjoint; rows (columns) do not overlap one another.

The Cooley-Tukey 1D FFT algorithm operates on an array of N data items, where N is a power of 2. During the execution of the algorithm, each location in the array is accessed O(log(N)) times. If sufficient onboard memory exists on a processing element, then the entire array of data to be processed can be loaded into the local memory at the start of computations and the FFT can be performed without further accesses via the bus until the operation is complete and the data are sent back to global memory. Thus there are only O(N) bus transactions instead of O(Nlog(N)) as would be the case with no local memory. If N=256, a typical value, savings of about an order of magnitude result.

The two cases examined are both similar in that bus transactions are reduced because a region of data is loaded into local memory at the start of an algorithm, and the data elements are accessed several times during the execution of the algorithm. In some sense, the burden may be thought of as having been transferred from the global system bus to the bus connecting local memory to the processor element. Since there is only one processor connected to local memory, while there are many connected to the global system bus, this is an acceptable tradeoff. It also allows the processors to use very high speed RAM for the small, local memory, while retaining inexpensive, slow RAM for the global frame buffer.

Supposing that the architecture of Figure 4.1 has been improved by the addition of local memory, another problem is uncovered. While the data is being loaded into local memory from the frame buffer, the processor involved is fully occupied with the data loading operation. No computation is being performed

during this stage. If the algorithm desired is computationally expensive, the local processors may not require access to the bus for a period of time. It is even possible that the bus will lie idle while computations are being carried out. When it is necessary to send the processed data back to the frame buffer, a burst of bus activity occurs during which the processing element responsible performs no useful computations. It would be desirable to smooth out the bus activity and in some way relieve the processing elements from the burden of transferring data from the global store to their local memory.

4.2. The Vista Image Processing Architecture

Armed with an understanding of the importance of using each bus transaction to the fullest, and a knowledge of the characteristics of popular image processing algorithms, it is possible to propose a new architecture that is capable of surpassing previous bus oriented multiprocessors in image processing applications. In fact, the performance of the proposed architecture, *Vista*, equals that of many multiprocessors which use sophisticated (and expensive) communication topologies. To fully understand how this architecture functions, and what makes it so efficient, its component parts will be described (Section 4.2.1), followed by an example (Section 4.2.2) showing the details of executing an image processing algorithm. Finally, Section 4.2.3 analyzes the performance that can be expected from Vista, and the factors that affect it.

4.2.1. Vista Components

As can be seen in Figure 4.3, Vista is composed of five distinct types of modules: a bus, master control unit (MCU), region transfer unit (RTU), image memory, and Tachyon† slave processors. Each type of component performs a certain function in the image processing system.

Vista uses an ordinary parallel bus for communication. Many such buses are commercially available (eg. Multibus, VMEBus, UNIBus, etc.), and Vista can be implemented using any one of them. As in all bus architectures, bus speed is ultimately the limiting factor in system performance; Vista's characteristics allow even a modest bus architecture to deliver outstanding performance.

The Master Control Unit (MCU) has two responsibilities. It provides the interface with the Vista user, and includes an operating system which allows the user to control the hardware of the entire system.

†Tachyon - a particle postulated to move with a velocity > light speed

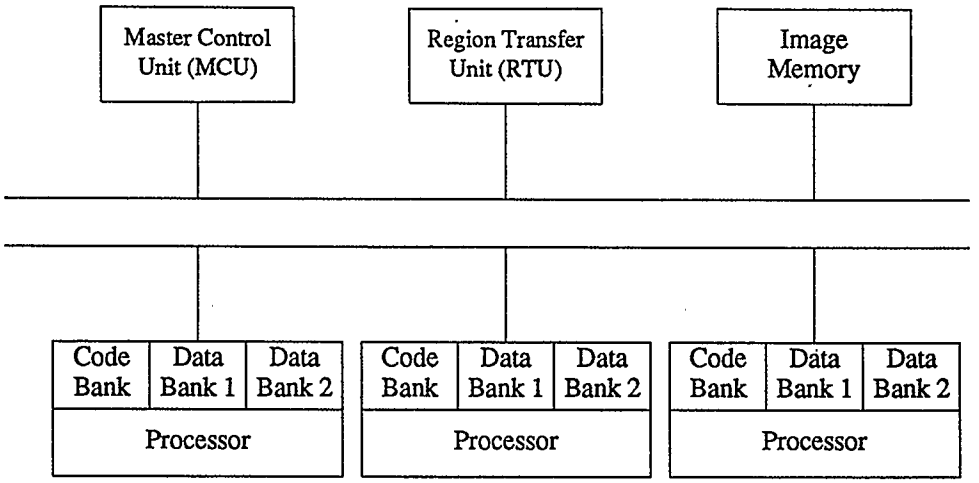

Figure 4.3 - VISTA Block Diagram

The MCU controls and directs the other components in the Vista system with the exception of the common bus; the common bus has its own protocol which arbitrates simultaneous requests for bus use. As will be shown, these never occur in the Vista architecture.

The Region Transfer Unit (RTU) is a special purpose I/O processor module which is capable of transferring regions of information, principally between the image memory bank and the Tachyon processors. It operates under the control of the MCU and is capable of transferring arbitrarily shaped image regions, such as rows, columns, rectangles or hexagons. To achieve maximum effectiveness it operates at or near the capabilities of the bus used to implement Vista.

Image memory is, as the name suggests, a large memory bank used for storing raster images. The speed of the memory should match the performance of the bus. That is, memory should be fast enough to service the bus at bus transfer speed. On most common buses, this does not require expensive static RAM; inexpensive dynamic RAM is usually adequate.

The key to Vista's performance lies in the design of the Tachyon slave processors. A Tachyon processor consists of five sub-components. At its heart lies a high speed microprocessor suited for arithmetic computations. Recently a variety of such units have emerged which fill these requirements (eg. Texas Instruments' TMS32010 and TMS32020, Motorola's DSP 56000, National Semiconductor's LM32900). Three banks of memory reside on each Tachyon, two data banks and one program bank. The memory

should be matched to the performance of the onboard microprocessor. In many cases, this will imply the use of high speed static RAM. Fortunately, due to the limited size of the memory required, no great expense is incurred.

The memory is arranged in a unique fashion. The program bank may be connected either to the common bus or to the Tachyon microprocessor, but not both at the same time. The data banks are designed such that when one is connected to the common bus, the other is connected to the Tachyon microprocessor, and vice versa. The size of these memory banks is fixed in a given implementation. The program bank should be large enough to implement all anticipated image processing algorithms. Experimentation has shown 4 K words of memory to be quite adequate but there may exist some algorithms which require slightly more. The data banks should be identical in size, and large enough to contain a *minimum region* of the image, a *minimum region* being defined as all data points (pixels) needed by any image processing algorithm whose use is anticipated. For example, the *minimum region* associated with a 2D FFT is a row or column (ie. a row or column is used in computing the 1D FFT's that make up a 2D FFT). A simple neighbourhood operation would require the pixel being smoothed as well as the surrounding pixels used in computing the smoothed value. As will be demonstrated shortly, considerable performance improvements can result if the data bank is large enough to store several *minimum regions.*

Experience has shown that memory requirements are easily satisfied using a few Kilowords of high speed static RAM. Since the amount of high speed RAM required is relatively small, the expense is quite tolerable. An incidental benefit of having separate program and data areas is to allow Harvard style architecture microprocessors to be used in the Tachyon. These processors achieve very high speed by overlapping program and data fetches.

The final component of the Tachyon is the control unit. This unit receives commands from the MCU which determine how the Tachyon memory is configured, whether the Tachyon processor is active or stopped, allow synchronization and provide a means for the Tachyon unit to inform the MCU when an assigned task has been completed.

It is not immediately obvious that the Vista components work in a complementary fashion that yields performance superior to the bus based multiprocessors examined thus far.

4.2.2. A Sample Execution of an Image Processing Algorithm with Vista

To understand the interaction between the various Vista components, consider an example of how the previously discussed region smoothing operation might be executed on Vista. Assuming the image to be processed has been previously loaded into the Vista frame buffer, processing could proceed in the following manner. First, the MCU sends out a command to all Tachyon units to halt their processors. This accomplished, commands are sent from the MCU to the Tachyons causing the program memory banks to be connected to the common bus. The MCU then proceeds to load Tachyon programs into the respective Tachyon processors. These programs might well be different for each Tachyon; in this particular example all Tachyons will likely use the same program. The Tachyon code might be loaded directly by the MCU from its internal memory, or indirectly by the RTU under command of the MCU. Since the program load operation will only need to be repeated when a different image operation is required, the program may be invoked many times without the need of reloading.

It is now necessary to divide the image into regions to be processed. In a typical neighborhood operation the number of pixels needed to compute a given pixel's value will be quite small. The data memory banks in the Tachyon are large enough to contain a sizeable number of neighbourhood regions, so to make optimal use of I/O operations the screen might be divided into horizontal strips containing a number of regions. For example, if the neighbourhood operation in question considered only the pixels immediately adjacent to the pixel being processed, a strip might be defined as being three rows high and the width of the screen wide. Suppose that each data memory bank in the Tachyon is large enough to contain at least 3K pixels. If the image in question is 512×512 pixels, then one strip will require only 1.5K of memory for storage. The MCU program would divide the image into 512 regions, each to undergo processing by a Tachyon.

To simplify the explanation, initially suppose that only one Tachyon unit exists in the system. To process the input image the MCU would command the RTU to transfer the first strip from the source image (in image memory) to the Tachyon data bank which is connected to the common bus. If the data bank is twice the size of the image regions, it can be conceptually partitioned into two sections, one for input data, the other for processed data. After a short interval the RTU confirms that the transfer is complete. The MCU then sends a signal to the Tachyon causing the data bank connected to the bus to 'swap' with the data

bank connected to the Tachyon microprocessor. This allows the microprocessor access to the data. The MCU transmits a message to the Tachyon telling it to begin execution of the program in its program bank (which contains the neighbourhood algorithm). While the MCU is waiting for the Tachyon to complete its processing of the region, it commands the RTU to load the next input region into the Tachyon data bank that is currently connected to the common bus. After the RTU confirms this operation the MCU waits until the Tachyon signals that its program has finished execution. The MCU again sends control signals to the Tachyon causing the data banks to be exchanged. The Tachyon is then instructed to begin execution again on the data that was preloaded into the data bank. As before, while waiting, the MCU commands the RTU to transfer the processed data out of the Tachyon and to the destination area of image memory, and new data from the source image to the Tachyon memory bank. Since the actual control signals to the Tachyon take relatively little time, as long as the RTU can move data quickly enough, the Tachyon will be processing data nearly 100 per cent of the time, never needing to wait for I/O. Even in the single Tachyon case this is quite efficient since it causes the large, comparatively slow image memory to appear as fast static RAM to the Tachyon microprocessor.

When several Tachyon units are available, all may process concurrently provided that the RTU can move data to and from the frame buffer quickly enough to supply the needs of the Tachyons. Of course, at some point any additional Tachyons added to the system will spend time idling since the RTU won't be able to keep up with the Tachyons. The number of Tachyons that can be efficiently serviced is determined by three factors.

1) Bus speed clearly limits the RTU in its ability to transfer information.

2) The speed of the Tachyon units. Clearly if they are made faster the RTU and bus will not be able to keep up with as many of them.

3) The quantity of information needed to be transferred across the bus for a given algorithm. As was pointed out earlier, it is essential that each item of data be used fully to avoid the need for repeating I/O on data which were previously transmitted.

In the example discussed, the image was divided into horizontal strips which overlapped one another. While quite efficient in the horizontal direction (ie. moving horizontally across the strip required no additional transmission of data) the division scheme proved inefficient in the vertical direction (each row was

transmitted three times across the bus). This division scheme could easily be improved on by transmitting rectangular regions instead of horizontal strips. This would necessitate retransmission of only those pixels on the perimeter of the rectangular region, resulting in a great saving of I/O capacity. These savings would allow the saved I/O capabilities to be used to service additional Tachyons. This emphasizes the need to divide the image into subregions in an intelligent fashion.

A surprising characteristic of the Vista system is that it performs best on computationally intensive tasks. To understand this, consider first the extreme case of performing a NULL operation on an image (ie. input = output). In this case, the operation of the Tachyons will be (neglecting control time) instantaneous. In essence, the RTU will be unable to keep up with even a single Tachyon unit. In this case, the time required to process the image will depend entirely on how fast the image regions can be transmitted from the frame buffer to the Tachyons, and back to the frame buffer. Additional Tachyon processor units will not change the processing time. It is noted that the parallel processing power of the system has been lost in this trivial case, but that since the operation was trivial, it is still performed very quickly. For example, a bus capable of transmitting 5MBytes/second could transmit a 512 x 512 image back and forth in 0.1 seconds. Of course if the regions were designed inefficiently such that retransmission of data was performed, this time would increase somewhat. Still, the performance would be very near 'real-time' for a trivial (or nearly trivial) operation.

Something quite different occurs when a computationally expensive algorithm is used. When the Tachyon units are occupied a large portion of the time, the RTU is able to service a larger number of Tachyons while still ensuring that none ever have to wait for data to be loaded or unloaded from the data banks. This brings to bear the power of parallel processing on the problem and high performance is achieved. If many Tachyons are available, their combined processing power will eventually be sufficient to keep up with the RTU. Any additional Tachyons added to the system will not increase performance, since the RTU will be unable to service them without forcing other Tachyons to wait for servicing. Thus it can be seen that, just as with the NULL operation, expensive operations are also carried out in the time it takes to transmit the image to the Tachyons and back to the frame buffer (presuming that sufficient Tachyons are available).

4.2.3. Theoretical Performance of Vista

It is desirable to quantify the performance obtainable via the Vista architecture. As a thorough examination of this aspect of Vista is found elsewhere,[31] a less rigourous approach is taken herein.

Suppose on a particular Vista implementation it is necessary to process a certain algorithm. It is found that the RTU is capable of servicing exactly four Tachyon units adequately. Execution of the algorithm can be broken into three phases:

1) Startup
2) Main
3) Shutdown

Given an image, subdivided into N identically shaped regions, it is clear that using the Vista architecture to process the image requires transmitting the N regions back and forth across the bus. Since bus transfer speed is fixed, this transmission time, t_{tran}, is easily calculated:

$$t_{tran} = \frac{2NS_R}{B} \tag{4.2}$$

where S_R = size of each region and B = min(bus speed, RTU speed).

If throughout the image processing operation the bus were busy continuously, then the total time, t_{total}, would be exactly t_{tran}. Unfortunately, the bus sometimes lies idle. Thus the total time is given by:

$$t_{total} = t_{tran} + t_{bus\ idle} \tag{4.3}$$

where $t_{bus\ idle}$ is the time the bus spends idle during the image processing operation. In general it is quite difficult to compute $t_{bus\ idle}$. For certain, specific cases the computation is quite straightforward.

Defining t_T as the time required for a Tachyon to process a region, and t_{rtu} as the time required by the RTU to fill and empty a Tachyon buffer, ie.

$$t_{rtu} = 2S_R/B \tag{4.4}$$

then it can be seen that to solve a given problem n Tachyons will be required to 'keep up' with the RTU, where

$$n = \frac{t_T}{t_{rtu}} \tag{4.5}$$

To simplify the situation assume that N >> n (ie. the number of regions is large compared to the number of Tachyons employed) and that n is an integer. With such restrictions, execution proceeds as illustrated in

Figure 4.4. In the case where n = 4, it can be seen that during the startup phase the bus is idle ½t_{rtu}. During the main phase the bus is never idle. In the final, shutdown phase, the bus lies idle ½(2n - 1)t_{rtu}. That is

$$t_{bus\ idle_{startup}} = \tfrac{1}{2}t_{rtu} \tag{4.6a}$$

$$t_{bus\ idle_{main}} = 0 \tag{4.6b}$$

$$t_{bus\ idle_{shutdown}} = \frac{2n-1}{2}t_{rtu} \tag{4.6c}$$

thus

$$t_{bus\ idle} = nt_{rtu} = 2nS_R/B \tag{4.7}$$

Therefore

$$t_{total} = t_{tran} + t_{bus\ idle} = 2(N+n)S_R/B \tag{4.8}$$

Since N >> n it can be seen that (in this special case)

$$t_{total} \approx t_{tran} \tag{4.9}$$

The assumption that N >> n is a perfectly reasonable assumption since the number of regions in an image usually ranges from about 100 to 1000 while the number of Tachyons that can be applied to a problem will typically be no more than 1/10 the number of regions. The restriction that *n* be an integer

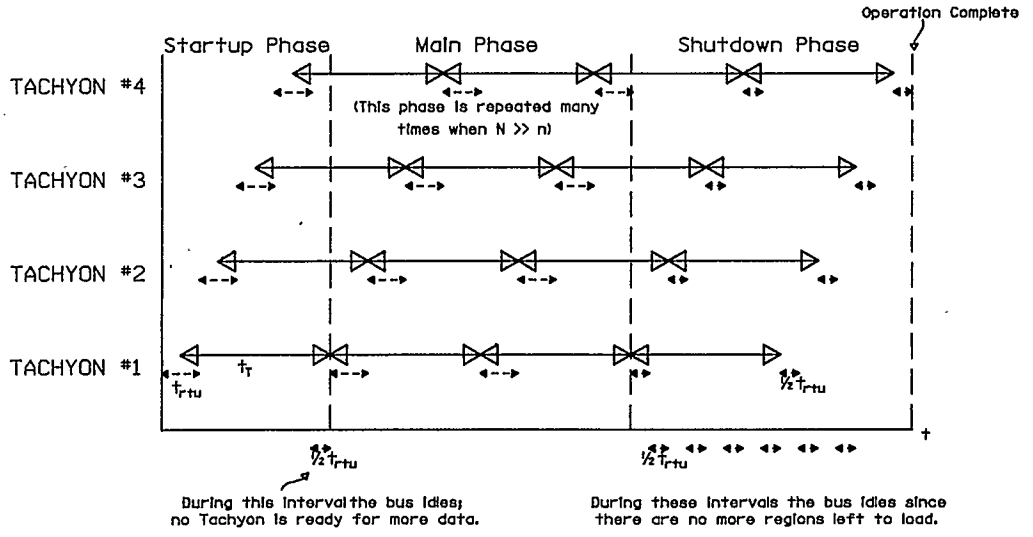

Figure 4.4 - VISTA Timing Diagram Case n=4, N=16

simplifies the calculations but does not change the basic outcome that the image processing time is approximately image transmission time (where N >> n). Thus it is seen that Vista offers nearly O(1) performance for algorithms with similar numbers of regions (roughly true for fixed image size) and N >> n.

The ability of the Vista machine to process algorithms of widely varying complexity in constant time is quite unusual. Of course it is possible to degrade Vista's performance if certain degenerate cases occur; these cases seldom occur in image processing situations.

The most obvious way in which Vista may suffer degraded performance occurs when the regions processed by the various Tachyons have large areas of overlap. In this case, many pixels must be retransmitted across the system bus, severely loading the bus and reducing the number of Tachyons the RTU can successfully service. Since most images exhibit the property of locality, this interdependency of regions does not normally occur.

Another cause of performance loss occurs when the time required to process a region is larger than the time needed to move the entire image back and forth between the frame buffer and Tachyons. If this occurs, the image processing time will be approximately that required to process a region. This is so because if sufficient Tachyons are available, one may be assigned to each region of the image and the regions processed concurrently. In practice, this situation does not seem to arise with common image processing algorithms. Even expensive tasks, such as the FFT (of a single region), can be performed in approximately 1/5 the time required to transmit the image across the bus. This does, however, provide another reason why the Tachyon processors should be high performance units; slow Tachyons might allow performance to suffer on computationally intensive tasks.

In a nutshell, the strength of Vista lies in its ability to reduce the amount of I/O necessary by ensuring that retransmission of data is cut to the bare minimum. Processors are relieved of the burden of fetching data across the system bus and are able to carry out processing almost continuously. Vista achieves high performance by being well adapted to the characteristics of images and image processing algorithms.

Vista also is noteworthy for its versatility. In the case described above Vista functioned in a manner analogous to an SIMD machine. While it is true that the same instructions were not executed simultaneously at all Tachyon processors, conceptually it might be imagined that they were, since all regions were subjected to the identical instruction sequence. Vista can, of course, be programmed to act more as an

MIMD machine, with each Tachyon executing a different program. This might be useful for machine vision where some Tachyons might preprocess sub-images looking for certain characteristics. Those sub-images that met the qualifications might be processed further by other Tachyons executing different programs. SIMD pipeline simulation is easily possible. The RTU could move the output of one Tachyon to the next in the pipeline sequence. Thus it is seen that Vista can be used in many different ways. It is even possible to change Tachyon programs 'on the fly'. More discussion of these concepts will take place in Chapter 6.

Vista can be viewed in a number of different ways. From the point of view of the MCU, the Tachyons appear as double buffered I/O units. From the Tachyon's perspective, the data banks appear to be paged, with the MCU and RTU performing the paging function. Vista's modularity makes it easily programmable and fault tolerant. If a Tachyon unit fails, others can 'take up the slack'. Of course the other components of the system might well cause system failure should they cease to function.

CHAPTER 5

A Vista Implementation

To test the feasibility of the Vista architecture introduced in Chapter 4, it was necessary to create a prototype implementation. This chapter discusses the advantages and disadvantages of using particular components to build the prototype system. The alternatives available at the time of design are described and the reasons behind the selections made are detailed. The resulting implementation incorporates the important concepts outlined in Chapter 4, but does not pretend to be a polished, production unit. As a prototype, its purpose was chiefly to demonstrate the practicality of the Vista architecture and to act as a learning tool for a future production version of Vista.

5.1. Component Selection

During the design of the prototype, several restrictions were imposed which had an effect on system performance. In the Vista architectural description found in Chapter 4, the Master Control Unit (MCU) and Region Transfer Unit (RTU) were separate devices which operated independently. The MCU commanded the RTU to transfer regions of data between the image memory and the Tachyon processors. The RTU then advised the MCU when the requested operation had been completed. This scheme allowed the MCU to perform other tasks while the RTU was executing a command. In most cases, however, the MCU would remain idle, since few operations not related to Tachyon control can be performed while the image is being processed. The only task considered essential during image processing is that of user interface; the user should be capable of controlling the function of the computer at all times. During the actual processing of an image, it is not unreasonable to restrict the user interface requirements to 'typeahead' and interrupt (abort) capabilities.

The RTU should be capable of transferring arbitrarily shaped regions of an image between image memory and the Tachyon units. This can be accomplished by a custom *direct memory access* (DMA) style device, but is more easily implemented using a programmable control unit; a microprocessor. Upon observing that the MCU and RTU perform few tasks concurrently, it was decided to combine the two into a

single unit. The combined MCU/RTU would contain a microprocessor capable of making the decisions needed for control of the Tachyons and transferring data between memory banks. Almost any microprocessor currently in use is capable of meeting the control requirements, but some models are better than others at transferring regions of data quickly.

5.1.1. Selection of the MCU/RTU Microprocessor

Because of the large size of images, most older, 8 bit microprocessors would be inappropriate; they are capable of addressing only 64K regions of memory efficiently. More modern 16 bit processors have far larger address spaces which can be used to access all of the data in an image efficiently. Since a 16 bit processor transfers twice as many bits of information at a time as an 8 bit processor, it follows that, all else equal, 16 bit processors should also transfer data twice as quickly as their 8 bit counterparts. In practice, due to its more recent design, a 16 bit unit is usually more than two times faster than an 8 bit unit.

Not all 16 bit processors are created equal. In general, processor performance is inversely related to the length of time the CPU has been available. The more modern units feature advanced addressing modes, prefetching of instructions, and many other features. A side issue in choosing an appropriate processor for a task is the consideration of the software tools at the designer's disposal. For the reasons cited, initially it seemed desirable to use a Motorola MC68000, or MC68010 microprocessor. The MC68000 and MC68010 have address spaces sufficiently large to accommodate the anticipated image sizes. Cross assemblers and cross compilers were readily available on the mainframe computer system at hand (a VAX 11/780 UNIX system). Since the MCU/RTU requires no unusual interface, it was anticipated that a commercially available MC68000 based board could be obtained for use with the bus chosen.

While clearly capable of performing MCU/RTU functions, the MC68000 was not the only eligible candidate. When a prototype is being constructed the objective is to verify the feasibility of a design at minimum expense, while conserving resources for the production units. It is also important that prototype software and hardware will be transferable to the design of production units with minimal changes. It was for these reasons that the Intel family of processors was considered.

At the time when components were being selected, the possible Intel 16 bit processors available were the iAPX86, iAPX186, and iAPX286. The special purpose iAPX89 I/O processor was also briefly con-

sidered because of its ability to transfer data quickly, but was discounted due to difficulties in transferring arbitrarily shaped regions and lack of suitability for general purpose and control functions.

The iAPX86 is the entry level 16 bit processor available from Intel. As one of the first commercially available 16 bit microprocessors, it is not surprising that some of the amenities found in units released subsequently are lacking. The major shortcoming in the processor's design, at least in this application, is the size and nature of the addressing system. The physical address space is limited to 1 Megabyte of memory, which is marginally adequate for medium resolution image processing. Unfortunately, only 64K segments of data can be accessed at a given time. Accessing other regions of memory involves changing certain base registers called 'segment registers'. This complicates code, since tests must be performed frequently to ensure that the boundary of a segment has not been reached, necessitating reloading of an appropriate segment register. Nonetheless, if only a small number of Tachyons were available (which would almost certainly be the case in a prototype), the iAPX86 would be quite adequate.

When the Vista design was proven, it would be possible to upgrade the MCU/RTU by using one of the more powerful members of the Intel family, such as the iAPX286 or iAPX386. Both the iAPX286 and iAPX386 are capable of executing iAPX86 code without changes, but at substantially improved cycle times (the iAPX286 is as much as six times faster than the iAPX86[32]. This would allow already developed software to be used, while permitting even higher performance in the future as new code employing the augmented capabilities of the new processors is developed. The iAPX386 is a particularly exciting possibility for future Vista machines. Introduced in 1986, the iAPX386 is a sophisticated, state of the art processor capable of very high performance. Segment size has been extended to 4 Gigabytes, making it an ideal choice for image oriented applications.

Software tools for use with the iAPX family were not available on the local mainframe, but IBM PC's (which use the software compatible iAPX88) can be used to generate code for the iAPX86. Since IBM PC's are very popular, this would make it possible for a large number of users to write software for production Vista machines. The final factor which caused the iAPX86 to be chosen was the grant program Intel offers to researchers using Intel equipment. Although not the optimum choice, the iAPX86 exhibits sufficient performance to demonstrate the feasibility of the Vista system.

The specific iAPX86 processor board chosen was the Intel iSBC86/05. This board incorporates an iAPX86 (8 Mhz), space for EPROM, 8K of static RAM, a programmable parallel port and a serial port. An additional serial port was added to increase communication capacity.

5.1.2. Bus Selection

Having chosen the iAPX86 it was necessary to choose a bus architecture to provide the communication path and arbitration between system components. Since the iAPX86 (8 Mhz) is capable of transferring linear regions of 16 bit data at 4.7 x 10^5 transfers/sec (ie. ~1 MBytes/sec), it is clear that bus bandwidth need be no greater this. For data transfers of non-linear regions, the iAPX has far lower performance. Various buses were considered, including Multibus, VME bus, Q bus and others. All would have been suitable, but Multibus was chosen since it is the bus supported by Intel (and thus would interface to Intel processor boards) and because it was relatively straightforward to implement the Multibus interface on the Tachyon board. The Multibus standard is capable of transferring 10 MBytes/sec, so the limiting factor in data transmission would be the iAPX86. Additionally, the Multibus has proven very popular in industry, and an almost unlimited selection of boards are available from various manufacturers. The only drawback of the Multibus is that it is not suitable for supporting the iAPX386, which is expected to run on Multibus II. Multibus II had only just been announced at the time of the Vista prototype design, and was not chosen because no hardware was available for it at that time.

5.1.3. Construction Technology

A more serious restriction to Vista's performance arose because of the limited time and facilities available to construct the Tachyon processor units. It was evident from the outset that the number of components needed to implement a Tachyon would require a physically 'tight' design to fit on the Multibus circuit card. If the board were created using printed circuit technology, a multi-layer board would be required. Designing a multi-layer printed circuit board is best done with the use of Computer Aided Design (CAD) tools. This is especially important if high speed logic is used in the hardware design since capacitance and inductance effects are far more noticeable under these conditions. Due to the difficulty in changing a printed circuit board once it is created, it is essential that a logic simulator be available to debug the hardware design before implementation. Since none of these software tools were available, 'wire

wrap' technology was used to implement the Tachyon board. Wire wrap is relatively easy to modify when design errors are found. Unfortunately, wire wrapping a board of the complexity of a Tachyon is a very time consuming and error prone task. This precluded the construction of more than one Tachyon board using wire wrap. The use of wire wrap served to bring the Tachyon design to a state where printed circuit boards could be designed for production units.

5.1.4. Tachyon Design

Using a single Tachyon board would mean that many of the multi-processing concepts could not be explored. Other problems, such as region transfer, bus synchronization and bus bandwidth could be investigated, and the hardware design debugged. Furthermore, useful software could be developed and a system capable of medium performance image processing established. The software would be easily transferable to production units with only minor modifications required.

One further minor restriction was placed on the Tachyon design. The Vista standard suggests that the MCU should communicate control information to the Tachyons via the bus. This requires that all the control circuitry be present on the Tachyon boards. The advantage of this scheme is that it allows almost any commercial microprocessor board to be used as the MCU (or MCU/RTU). As improved microprocessor boards become available, the new boards may be substituted with no hardware modifications required.

The particular iAPX86 board used in the Vista prototype, the iSBC86/05, incorporated a parallel control port. It was decided that the parallel port would be used to communicate the control information with the Tachyon processor unit. This would relieve some of the complexity of the control circuitry needed on the Tachyon board. Although making substitution of the MCU/RTU board more complicated, this was felt to be an acceptable sacrifice for a prototype design.

While a multi-Tachyon implementation of Vista does not require that the Tachyons be exceptionally fast (see Section 4.2.2), there are certain advantages in using fast Tachyons. In the case of the Vista prototype, since only one Tachyon board was built, the faster it was, the more quickly images could be processed. In the more general case it decreases the number of Tachyons required to keep the RTU busy, thus minimizing *shutdown* time (Section 4.2.3) Using large numbers of Tachyons also increases the capacitive load on the bus, requiring more powerful (expensive) bus drive circuitry. Additionally, each incarnation of

the Tachyon requires bus interface circuitry. The cost of the bus interface circuitry is largely independent of Tachyon speed and represents a near constant cost per Tachyon board. Thus it is more cost effective to build a few fast Tachyon processors than many slow Tachyons. Also, it may not always be possible to divide a particular task into subtasks, thus preventing the use of multiprocessing. In this case, the greater the performance available from a Tachyon, the faster the processing will take place. For these reasons it was decided to implement the Tachyon using a high performance processor unit.

Several processors were considered for use in the Tachyon board. The Tachyon processors do not require large amounts of memory (as compared to total image size). They do need to be able to perform arithmetic computations very quickly. At the time of the Tachyon design, a new processor had recently been released by Texas Instruments, the TMS32010. The TMS32010 is a special purpose processor oriented to digital signal processing applications. Most instructions, including multiplication, are executed in a single, 200 nsec clock cycle. Performance can approach 5 Million Instructions per Second (MIPs). The instructions are tailored to the needs of computation intensive problems. For example, instructions exist to perform multiplication and accumulation operations in a single cycle. Values may be loaded into registers or stored in memory via a shift register, thus combining two operations into a single cycle. These, and other special instructions make the TMS32010 highly suited to image processing applications.

The TMS32010 is not without its eccentricities. To characterize it as unusual would be somewhat of an understatement. To make its high speed possible, the TMS32010 is a Harvard architecture processor, having separate program and data memory areas. The data memory is only 144 16 bit words in size and is located 'on chip' (ie. within the TMS32010). The program memory area is only 4K words in size. This is usually adequate for ordinary digital signal processing applications, but for the Tachyon units, some adaptations had to be made.

Unlike a true Harvard architecture, the TMS32010 does allow values to be transferred between program and data memory, albeit with a significant speed penalty. This feature was used to implement the Tachyon processor unit. The program memory space is divided into two regions. A 3K word region is dedicated to the storage of code (programs). This corresponds to the program bank of the Tachyon architecture. The remaining 1K words of the address map are assigned to hold the data to be processed. Two 1K word banks of memory exist on the Tachyon board. When one of these banks is connected to the

TMS32010, the other is connected to the Multibus, and vice versa. These 1K banks correspond to the data banks in the Tachyon design. (See Figure 5.1).

This arrangement required very careful design. The first difficulty involved the nature of the Multibus. The Multibus standard (IEEE 796) states that both 8 and 16 bit transfers must be possible. Furthermore, following Intel convention, 16 bit values are stored in memory with the significant byte stored first (eg. the value 1234H would be stored as 3412H). This is contrary to the philosophy of the TMS32010. These difficulties were overcome easily, largely due to the excellent application notes provided by Intel[33] A greater challenge involved the splitting of the address space into the Tachyon program and data areas. The TMS32010 requires memory with access times of about 70 nsec. Splitting the address space into two

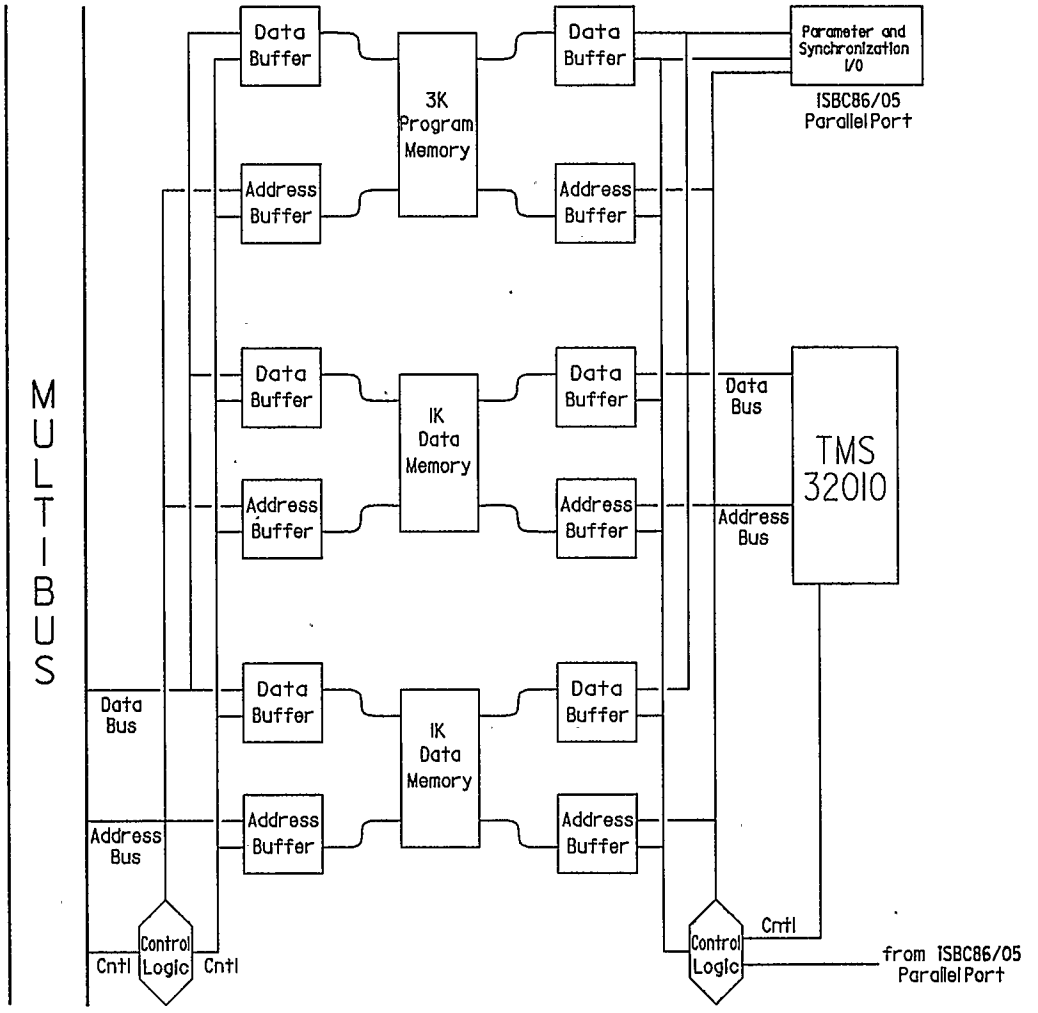

Figure 5.1 - Tachyon Block Diagram

regions requires that address decoders be incorporated into the circuitry. Since the memory is multiplexed between two sources (the Multibus and the TMS32010) address and data buffers are also required between the TMS32010 and memory. These decoders and buffers rob precious time from the 70 nsec available. If these difficulties are not sufficient, Texas Instruments multiplexed the control pin which indicates whether an output or memory write instruction is in progress. The output instructions send information to one of eight output ports. These ports are not built into the TMS32010 chip, but rather must be added on using additional hardware. When an output instruction occurs, the address of the output port is placed on the TMS32010 bus and the data strobed into the port using the WE*[†] line. Unfortunately this is the same method used to write values to program memory. The only difference between the two operations from a bus perspective is that output operations occur to bus addresses 0 through 7, and memory writes to addresses 7 through 4095. This means that still more circuitry is required to differentiate the two kinds of operations. The net effect of all the additional circuitry is to to leave very little time available for the memory to respond to TMS32010 bus requests. This necessitates using 55 nsec RAM and 74AS/ALS series logic to gain the needed speed. As much as is possible, the various decoding and buffering operations occur in parallel. This sometimes requires more gates than would otherwise be needed, but is essential if the timing specifications of the TMS32010 are to be met.

One further, annoying consequence of the multiplexed memory read/output 'WE*' line occurs during output port operations. During memory write and output operations, it is within the limits of the TMS32010 timing parameters that the 'WE*' line may become active before the data being written are valid. This is not a problem during memory writes, since the correct value is ultimately stored, and the intermediate value of the particular memory location during the store operation is inconsequential. Output to a port is another matter. If a simple data latch is used for an output port, glitches may occur during output port operations. This is a consequence of the fact that data bus values are not guaranteed valid at the beginning of output instructions. This can cause port values to be unstable during output operations. This necessitates double buffering the output port such that output values cannot change until the end of the output instruction.

[†]WE* is a TMS32010 control signal. High=memory READ, low=memory WRITE.

The interest shown in the output port is more than academic. The output port forms part of the synchronization mechanism employed on the prototype Tachyon board. It is used to signal the MCU whether the Tachyon is busy, or waiting to be assigned a new task. Status codes can also be passed back to the MCU/RTU indicating errors, or other important values that need to be known by the controller. An input port also exists by which the Tachyon can receive parameters from the MCU which direct exactly how an algorithm should be executed. For instance if a Fast Fourier Transform is being performed, the input port allows the Tachyon to determine whether a forward or reverse transform should be performed.

The MCU commands the Tachyon to begin processing an image via a special input pin on the TMS32010, known as the BIO* pin. The instruction set of the TMS32010 includes an instruction known as BIOZ. BIOZ causes a branch to occur to a specified address if the BIO* pin is at state '0' when the instruction is executed. The processor can thus be put in a waiting loop which exits when the MCU drives the BIO pin low. This, as is all MCU communication, is directed through the I/O port of the iSBC86/05 board.

Many minor problems occurred during the construction and testing of the Tachyon prototype board. Most were of an absent minded nature; connecting logic TRUE lines to inputs requiring logic FALSE input (and vice versa), poor connections, etc. The few problems of a more serious nature were usually the result of the small amounts of time allowed by the TMS32010 timing specifications to accomplish certain operations. A logic analyser was employed several times to clarify what was happening during bus operations. Without such a tool, it is difficult to see how debugging could proceed on such a device as the TMS32010. It must be emphasized that when constructing a device of this nature, the highest standards of construction must be used. The prototype board was wired using low capacitance/low resistance silver plated Tefzel wire. Care was given to IC placement and wire routing.

To provide for the storage of images being processed, a 256K byte memory card, the Intel iSBC056A was added to the system. This is the final component that, strictly speaking, forms part of the Vista system, but several other components are needed to make Vista useful.

5.2. Mass Storage and User I/O

Due to lack of time, it was not possible to add mass storage (a disk drive) to the Vista system, or an interface to a digitizing device. To circumvent these shortcomings, one of the serial ports on the iSBC86/05 board was connected to a mainframe computer system. Software was written which allowed the mainframe to act as a fileserver for Vista, uploading and downloading code and data on demand. Since the mainframe computer was equipped with a digitizing camera, it was possible to digitize an image into the mainframe from where it could be sent to Vista. Since images are large and the bandwidth of serial ports small, many minutes are required to move images between the two computers. Nonetheless the communication scheme was sufficient to establish the feasibility of Vista.

Future Vista machines could easily be supplied with mass storage by simply adding an appropriate interface card, available from diverse manufacturers, and writing the appropriate driver software. A similar approach could be taken to incorporate an image digitizer unit.

To allow users to communicate with Vista, the other serial port on the iSBC86/05 is configured to communicate with a terminal. Through this port, commands could be issued to the Vista system and the results of the computations examined.

5.3. Prototype Performance

A 2D FFT program was written to provide a benchmark of the Vista prototype's performance level. A picture was digitized into a 256 x 256 pixel 8 bit gray level image. The image was divided into 256 columns and rows and a 2D FFT was computed as described in Section 2.4.1. Since the number of Tachyons (1) was insufficient to keep the RTU constantly transferring data, the approximation given in Section 4.2.3 could not be used.

It was determined (by simulation) that the TMS32010 FFT program computes a 1D 256 point FFT in 23 msec. Since 256 column FFT's and 256 row FFT's were required, it was anticipated that the 2D FFT of the entire image would be computed in

$$t = 512\,x\,23\,msec = 11.8\,sec \tag{5.1}$$

which was very close to the 12.5 sec measured by a hand-held stopwatch during an actual execution. The reason for the slight discrepancy is explained in Section 6.3.1.

It is interesting to speculate as to the performance of the Vista implementation if more Tachyons had been available. To compute this it is convenient to break the computation into two parts, the row and column sub-FFT's. This is necessary since the iAPX86 based RTU is more efficient at transferring rows (adjacent words) than columns.

By examining the cycle times of the RTU region transfer program it was possible to determine that the RTU is capable of transferring one word every 2.125 μsec or 470588 words/sec during row transfers. During column transfers it is possible to transfer 1 word every 5.25 μsec or 190476 words/sec. For purposes of computing the FFT each pixel in the 256 x 256 image was stored as a 'real' word and an 'imaginary' word, resulting in a total image size of 128 Kwords. The entire image must be transferred across the bus twice so a total of 256 Kwords are transferred in each phase of the FFT. The region size (for both rows and columns) was 512 words, which could be processed by a Tachyon in 23 msec. This yields a rate of 43478 words/sec processed by the Tachyon necessitating

$$n_{row} = \frac{470588}{43478} = 11\ Tachyons \tag{5.2}$$

and

$$n_{column} = \frac{190476}{43478} = 4.4\ Tachyons \tag{5.3}$$

The number of Tachyons required in both the row and column portions of the transform are substantially less than the number of regions (256), so the approximation found in Section 3.2.1.4 can be safely used yielding

$$t_{row} = \frac{(131072 \times 2)}{470588} = 0.56\ seconds \tag{5.4}$$

and

$$t_{column} = \frac{(131072 \times 2)}{190476} = 1.4\ seconds \tag{5.5}$$

Therefore the total time to compute a 256 x 256 point 2D FFT would be

$$t_{FFT} = 0.56 + 1.4 = 1.96\ seconds \tag{5.6}$$

This is an impressive figure when compared to contemporary single processor machines.

While having serious deficiencies for practical use, notably the primitive method of receiving data, and the restriction to a single Tachyon processor, the prototype machine has fulfilled its purpose in refining

and proving the Vista design. Vista software and development tools have been written which will be easily transferred to future Vista machines. These tools are the subject of discussion in Chapter 6.

CHAPTER 6

Vista Software

Having discussed in some detail the hardware used to implement the Vista prototype, consider now the software required to make Vista function. To facilitate discussion, the software will be divided into the following classes: development tools, Vista system software, and application programs.

Before considering the various classes of software, it is necessary to establish the philosophy with which the Vista prototype has been programmed. Many of the machines discussed in Chapter 2 were connected to and operated under the control of a conventional mainframe computer system. This scheme has both advantages and drawbacks. The greatest advantage of using specialized processors, such as Vista, under control of a large well developed system, is the immediate availability of advanced software tools and utilities. The file system, user interface and other support software that accompany almost any major computer system can be used for both developing and executing programs on a Vista machine. If Vista were a polished, production system, this would be the implementation of choice. In the case of a prototype system, several problems emerge with this strategy.

First, and certainly not least, is the understandable reluctance of systems personnel to connect unproven hardware to the highly vulnerable bus of the mainframe computer. In the case of the Vista prototype, an RS232 interface was provided precisely to avoid this problem. Although much slower than a direct connection to the mainframe system bus, it prevented the Vista prototype from having an adverse affect on the mainframe system's hardware. In any case, if Vista were to operate under the control of the mainframe system, some sort of software interface would have to be provided.

The local mainframe available for use with Vista, a VAX 11/780 UNIX system, allows new devices to be attached to the operating system by means of specialized *device driver* routines. Unfortunately, the UNIX operating system only allows device drivers to be loaded at 'boot' time. As well, since these drivers operate in a privileged system state, undetected bugs can cause system crashes. The difficulty in debugging experimental drivers (requiring a re-boot for each new version) coupled with the hazards to other mainframe users made this option unacceptable.

Another option is to write pseudo-device drivers to control Vista. These pseudo-device drivers might be developed as ordinary UNIX programs that would send and receive messages and data via the RS232 communication connection. This is a perfectly acceptable method of controlling the Vista prototype in the sense that it poses no danger to the mainframe system, and simultaneously allows the developer access to the sophisticated tools available on the UNIX system.

One notable drawback exists in the scheme proposed above. By sending commands and receiving replies via the UNIX system, only indirect communication exists between the developer and the Vista prototype. This lack of direct contact with the prototype complicates debugging since the chain of communication makes it more difficult to ascertain where problems lie; in the Vista machine or the communication protocol.

It should be noted that the Intel iSBC86/05 computer card used for the MCU/RTU arrived with absolutely no software; the Electrically Programmable Read Only Memory (EPROM) sockets were empty. Whether Vista was configured to operate under direct operator control or indirectly through a mainframe computer system, very low level bootstrap software had to be written. This very low level software is especially difficult to debug since it interfaces directly with computer hardware. Also, it must be in place before any higher level routines can be written. For these reasons, it was decided that the Vista system should operate under direct user control (via a terminal).

The decision to employ direct control had fortuitous consequences; during the development of the Vista software major renovations were made to the UNIX system's communication lines, disrupting communications to the Vista research lab for several months. Because Vista was configured to function autonomously, it was possible to continue development without a communication connection to a mainframe system.

6.1. Development Tools

Having established that Vista was to function autonomously, under direct developer control, consider the development software needed to construct such a system. Since two different microprocessors are used in the Vista prototype, the Intel iAPX86 and the Texas Instruments' TMS32010, it is clear that two classes of development tools are also needed. Obtaining the appropriate tools was a major challenge to Vista

development.

The iAPX86 (often known as the 8086) is a very popular microprocessor. For some reason, however, no cross assembler or compiler could be found that would run on the available VAX 11/780 UNIX system. The iAPX86 is in many respects an unusual processor to write code for. It features a wide variety of addressing modes and the instructions vary in length. The processor divides the physical address space into regions where data, code and stack requirements are stored. Depending on the instruction executed, different regions (called segments) are accessed. The default segment accessed by a particular instruction may, in many cases, be overridden by special segment override prefixes. Fortunately, all of these intricacies are largely invisible to programmers due to the sophisticated assembler language defined by Intel. Although it would have been possible to write an iAPX86 cross assembler for the UNIX system, because of the unusual features of the processor it was felt that a complete language implementation would require substantial time to develop. To avoid this loss of time, another alternative was sought.

A very popular series of microcomputers, the IBM PC family, use the iAPX88 processor as their CPU. The iAPX88 is in every way software compatible with the iAPX86 used in Vista. iAPX88 assemblers are standard tools found on many IBM PC systems, and it was initially hoped that one of these, the popular *MASM* (by Microsoft), could be used for Vista development. Unfortunately this was not as straightforward as expected. *MASM* was designed to generate code for use with the IBM PC operating system. Due to the nature of the operating system, no fixed addresses are allowed in programs. Rather, all addresses and segment values are specified by the operating system at load time. Oddly enough, specifying absolute addresses in source code does not cause error messages to be generated by the assembler. Only the output object code is in error, having hyphens, '-' substituted where the addresses ought to be. To overcome this difficulty, a filter program was written that parses the assembler output listing, building a symbol table of address values (from the source portion of the listing). The filter program creates a revised listing with the absolute addresses substituted for the hyphens. Thus a method was found to generate iAPX86 code for the Vista system.

To enable EPROM's to be programmed with the assembler object code, a second filter program was written. This filter program parses the revised assembler listing and creates two files containing the object code in a special ASCII format (Intel hex format). The two files contain the even and odd address

instructions and data generated by the assembler. It is necessary to separate the instructions into even and odd address values because of a peculiarity of the iAPX86; the iAPX86 requires two EPROM's containing the even and odd code and data values to be used. The ASCII format files are then sent to an EPROM programmer which stores the appropriate values into the integrated circuit. The EPROM's are then ready to be plugged into the iSBC86/05 for testing of the code.

The development tools for the TMS32010 processor were equally difficult to acquire. No TMS32010 assembler was available that would run on the Vax 11/780 UNIX system. Although a cross assembler was available for use with IBM PC's, obtaining it would have been both time consuming and expensive. Fortunately the assembler language of the TMS32010 is quite straightforward and an interested associate, Professor James Parker, wrote a TMS32010 cross assembler suitable for use on the UNIX system.

Other peculiarities of the TMS32010 make it somewhat difficult to develop code for this processor. In particular the debugging process is complicated by the fact that all code must be executed from RAM; EPROM is not fast enough for use with the TMS32010. Thus if a simple monitor program were written that allowed debugging of TMS32010 programs, the problem remained of debugging the monitor. This situation is not unusual in itself, but is complicated by the fact that the user has no direct contact with the Tachyon (TMS32010) computer; the Tachyons are controlled by the MCU/RTU. Debugging would be exceedingly tedious since no direct feedback would exist to indicate the intermediate results of program execution. The solution to this problem was found in the form of a TMS32010 simulator, also written by James Parker. The simulator, written for the UNIX system, allowed code to be single stepped, examining registers and memory locations during execution. The simulator was invaluable for debugging TMS32010 code.

6.2. System Software

Since the Vista prototype was intended to operate independently of other computer systems, it was necessary that a simple operating system, or at least a monitor, be written. The system software would be executed on the iAPX86 processor, since it acted as the system master. Due to the fact that the purpose of the prototype was only to provide a development environment for Vista, it was not important that the

operating system be compatible with other existing systems. Neither was great sophistication or complexity required. The requirements of the operating system were quite basic. The operating system should allow data and code to be loaded into the Vista machine and stored to some external memory unit. Common MCU/RTU functions, such as moving regions of data, would also be appropriate system services. Finally, a simple user interface would have to be provided for use by developers. This interface would be most beneficial if it included tools to provide debugging of iAPX86 (MCU/RTU) code, since no simulator existed for this processor.

The system software, known as BMON86 (Bootstrap MONitor for iAPX86) fulfilled these requirements. The BMON86 monitor was written (by the author) in iAPX86 assembler language and 'blasted' into EPROM using the process described above. Serial I/O was buffered and interrupt driven allowing processing and I/O to proceed concurrently. To permit application code and data (both iAPX86 and TMS32010) to be loaded into Vista a fileserver protocol was developed that treated the UNIX system as a large, but slow, disk drive. Vista could request files from the UNIX system and store results on it via an RS232 serial port. This was not an entirely satisfactory solution, since the large data sets associated with image processing tend to require long periods of time to be transmitted over a standard serial port. Still, since the Vista machine did not have a disk drive of its own, it sufficed for a prototype. Additionally it proved convenient since the UNIX system was attached to a digitizing camera and this could then be used with Vista. Finally, the TMS32010 code was generated (as described previously) on UNIX and was conveniently available for transmission to Vista. The iAPX86 application code generated on the IBM PC was first transmitted to UNIX via modem and then to Vista using the BMON86 fileserver.

The BMON86 monitor was written in such a fashion to be easily extendible. A standard parameter protocol was established that defined how subroutines should call system services. Additional user interface commands are added by simply adding entries to a system table. New user interfaces can be implemented dynamically (by application programs), without any changes to EPROM code. It is similarly quite simple to add new interrupt driven device drivers to the system should additional hardware become available. In short, BMON86 is a simple yet effective tool for development of prototype Vista applications.

6.3. Application Programs

Vista application programs necessarily involve two sets of software; MCU/RTU control code and Tachyon region processing code. The former directs the operation of the Tachyon(s) and the latter performs the computations which actually process a region of an image. The exact nature of the Tachyon code varies greatly depending on the kind of image processing operation required. It is therefore difficult to show a generalized example of Tachyon code. A specific example will be given shortly in the hope that it suffices to demonstrate how a Tachyon might be programmed.

There is one aspect of Tachyon code that can be generalized; ie. the portion of the Tachyon program that interfaces with the MCU/RTU. MCU/RTU programs are quite general in nature since their chief function is to provide data and synchronization commands to the Tachyons.

Consider the following detailed description of a method that was used to interface the MCU/RTU and Tachyon units in the Vista prototype. Where relevant, shortcomings with the method are noted and suggestions for improvements are made.

First, a standard Tachyon region processing program is loaded into the Tachyon program memory bank by the MCU/RTU software. The Tachyon processor is stopped while the program is being loaded. When the processor is started, the Tachyon program performs whatever initialization is needed, then outputs a non-zero value via its output port to signal the MCU/RTU that it is not busy, and is ready to process data. The Tachyon process then enters a synchronization loop, awaiting a signal from the MCU/RTU to indicate that it should proceed to process data. The MCU/RTU, whenever possible, loads a data bank with data to be processed and 'gives' the bank to the Tachyon processor. If a parameter needs to be passed to the Tachyon process, the MCU/RTU transmits the value to the Tachyon input port. The MCU/RTU then pulls the 'BIO' pin on the TMS32010 processor low, causing the Tachyon process to fall out of its synchronization loop and begin processing. Immediately after falling from the synchronization loop, the Tachyon process outputs a zero value to the MCU/RTU via the Tachyon output port, indicating that the Tachyon processor is busy. The MCU/RTU then releases the BIO pin and proceeds to check other Tachyon processors (if they exist) to see if they need servicing. The MCU/RTU may also unload any data that was processed previously from the (currently) idle Tachyon data bank, and preload it with more information to be processed. When the Tachyon process has terminated, a non-zero value is output to the

MCU/RTU indicating that the data have been processed, and possibly indicating a status value. The Tachyon software then returns to the synchronization loop and awaits the assignment of a new task. The MCU/RTU eventually polls the Tachyon output port and determines that the Tachyon has completed its operation. The MCU/RTU gains control of the data bank containing the processed information and instructs the Tachyon to begin processing the new data that has been preloaded into its current data bank. While it is being processed by the Tachyon, the MCU/RTU stores the processed data in the frame buffer, and loads the idle bank with new information to be processed.

This synchronization technique, while quite adequate for a single Tachyon, should be improved on when more Tachyons are added. The first problem with the afore-described technique occurs when either the Tachyon process terminates very quickly, or the MCU/RTU is very slow in response (as might be the case if it were trying to service a number of Tachyons). The MCU/RTU is supposed to release the BIO line as soon as the Tachyon signals that it is busy. If the Tachyon process were trivial, it might terminate, branch to the synchronization loop and, if the MCU had not yet released the BIO line, restart the task and re-process the data a second time. Since the MCU/RTU can monitor the Tachyon busy status and respond within a few microseconds, it is doubtful that meaningful Tachyon programs that short would ever be used. If, however, the MCU/RTU attempted to perform other operations while still polling the Tachyon busy line, the MCU/RTU response time might be inadequate. This problem could be solved easily in one of two ways.

An obvious software solution would involve an additional handshake between the MCU/RTU and the Tachyon. After the MCU/RTU forces the BIO pin low, signalling the Tachyon to begin execution, the Tachyon would reply with a busy signal, just as before. Before proceeding with processing of the data, however, the Tachyon would wait for the MCU/RTU to reset the 'proceed' signal it had just sent. This would prevent the problem of the Tachyon processing the data twice, but would slow down the Tachyon process slightly. A better solution involves a small hardware modification.

The MCU/RTU control line driving the TMS32010 BIO line could be connected to the BIO pin via a non-retriggerable monostable multivibrator, or a non-retriggerable down counter (clocked by the TMS32010 processor clock). When the MCU/RTU would attempt to drive the TMS32010 BIO line, it would create a short pulse at the TMS32010 BIO pin. This pulse would be long enough for the Tachyon to

catch the signal from within the synchronization loop but short enough that no plausible Tachyon program could ever terminate before the pulse was reset (say five clock cycles). Since the hardware actually producing the pulse would be non-retriggerable, the pulse could not be repeated until the MCU/RTU reset its BIO driver control line. This would seem to be the preferable method, since the Tachyon unit could proceed immediately without worrying about the MCU/RTU response time.

Another likely improvement in MCU/RTU to Tachyon communication strategy involves more hardware modifications. To eliminate the MCU/RTU polling (needed to determine when the Tachyon process is finished) it would seem reasonable to add an additional output port to the Tachyon processor board. This port could be connected in such a fashion so as to cause an interrupt on the MCU/RTU when the Tachyon process terminated. The MCU/RTU interrupt service routine could queue the Tachyon for servicing when the MCU/RTU was free to do so. This would allow the MCU/RTU to service those processes found in the queue without concerning itself with determining exactly when they had finished.

Due to the nature of the Vista architecture all Tachyon programs are executed under control of the MCU/RTU, and are not under direct user control. In fact, since the TMS32010 employs a Harvard architecture which requires separate program and data spaces, the data being stored 'on-chip', it is impossible to ever directly examine some data variables used in programs. It is also impossible to 'single step' Tachyon programs, examining status after each step. These difficulties make advanced software debugging tools, such as a simulator, absolutely essential, if Tachyon programs are to be created in any reasonable length of time.

6.3.1. A Sample Application Program

Perhaps the best way to illustrate how Vista software might be written is by way of example. A very common (and expensive) image processing operation is the 2-D Fast Fourier Transform (FFT). As was explained in Chapter 3, an N x N point FFT may be computed by calculating 2N 1-D FFT's. In most higher level languages, the implementation of an FFT is a straightforward task involving the use of floating point arithmetic. The TMS32010 (Tachyon) processor is not capable of performing floating point arithmetic in hardware, and floating point operations implemented in software tend to be very slow. It is not clear that the great precision typically offered by floating point operations is necessary when the input

values are not precisely known, as is the case when data has been obtained via limited precision analog to digital (A/D) convertors. In many cases A/D convertors have only 8 to 10 bits of resolution. For this reason an integer FFT was coded for the Tachyon processor(s).

The concepts used in the assembler language integer FFT implementation are easily extended for use in high level languages. This is sometimes necessary when the ultimate in speed is required and the machine executing the high level code does not have floating point hardware. For this reason a brief examination of the characteristics of an efficient integer FFT are appropriate.

The main difference between a floating point FFT implementation and an integer FFT implementation is the lack of dynamic range available for calculations. It is therefore crucial that the integer FFT algorithm be implemented in such a way that overflow never occurs if accuracy is to be preserved. At the same time it is possible to be too conservative in the use of dynamic range causing a loss of precision and cumulative round off errors. To understand how these problems affect the FFT algorithm, and how they may be overcome, consider first the floating point FFT program found in Appendix I. Appendix I contains a documented listing of an FFT written in a little known language (BASIC09) that may be read as pseudo-code.

Note that the program found in Appendix I computes the FFT as shown in the signal flow diagram found in Chapter 2 (Figure 2.3). The program consists of three nested loops. The outermost loop calculates the different columns of the signal flow diagram. The middle loop computes the regions of data within each column that have no relationship to other regions (refer to Figure 2.4). The innermost loop performs the actual computation of intermediate values. It consists of the following instructions:

a.reel = datval(datapt).reel

a.imag = datval(datapt).imag

b.reel = wp.reel*datval(datapt+fftsize).reel - wp.imag*datval(datapt+fftsize).imag

b.imag = wp.imag*datval(datapt+fftsize).reel + wp.reel*datval(datapt+fftsize).imag

datval(datapt).reel = a.reel + b.reel

datval(datapt).imag = a.imag + b.imag

datval(datapt+fftsize).reel = a.reel - b.reel

datval(datapt+fftsize).imag = a.imag - b.imag

where wp.reel and wp.imag are values between +/- 1.0.

If the data values (datval()) take arbitrary values, it is difficult to predict the range of the resulting computations. If the data are limited to values between +/- 1.0 it can been seen that datval(datapt).reel, datval(datapt).imag, datval(datapt+fftsize).reel, and datval(datapt+fftsize).imag will range between +/- 2.0. If a factor of 0.5 were then applied to the datval() values it would guarantee that intermediate datval() would always remain within the range of +/- 1.0. Since each data item would be scaled in this manner lg(N)† times throughout the entire FFT algorithm, each data item would be divided by N over the course of the algorithm.

An interesting feature of the FFT algorithm is the convention of dividing the data by N in the forward (reverse) transform but not in the reverse (forward) transform. This is done so that if data are forward and then reverse transformed, the original values are recovered. (Alternately the data may be divided by sqrt(N) in both directions). This division by N is automatically accomplished if the method described above (division by 2 at each stage) is utilized. Note that if input data fall within the range +/- 1.0 and are divided by N over the course of the FFT the output data will also fall within the +/- 1.0 range. Fortunately, it is possible to make a more restrictive estimate of output data values because of certain properties of the Fourier transform.

Parseval's theorem states that the energy (or power) before transforming is the same after transforming. In the case of the DFT it is necessary to take into account the scale factor involved so in this case the energy before transforming is N times the energy after transforming. This reduction in power has a useful side effect. Data (initially ranging from +/- 1.0) can be forward transformed (dividing by N) yielding values still in the +/- 1.0 range. The transformed values can then be reverse transformed (without dividing by N) without intermediate or final values ever falling outside the +/- 1.0 range.

These properties of the FFT algorithm make it very easy to implement using integer arithmetic. The integer bit patterns available are used to represent the real values from -1.0 to 1.0. Unfortunately, the way in which this is usually done is somewhat wasteful. If an integer on a particular machine consists of 'n' bits, then 1.0 can be represented by the integer value $2^{(n-2)}$ and -1.0 by the two's complement of $2^{(n-2)}$.

$$01000000_2 = 1.0_{10} \quad n=8 \tag{6.1a}$$

†lg() is used to denote $\log_2()$.

$11000000_2 = -1.0_{10}$ **(6.1b)**

Unfortunately this representation never utilizes close to 50 per cent of the possible bit patterns. This is equivalent to saying that a loss of precision occurs (since the scheme described is capable of representing values from -2.0 to (almost) 2.0). A better arrangement represents -1.0 as $2^{(n-1)}$ and approximates 1.0 by $2^{(n-1)} - 1$. This causes a loss of precision for the value 1.0 but improves precision on all other values.

$$01111111_2 \approx 1.0 \quad (0.9922) \quad n=8 \tag{6.2a}$$

$$10000000_2 = -1.0 \tag{6.2b}$$

Division by 2 (required at each intermediate stage of the forward FFT) is very easily accomplished in integer arithmetic by an arithmetic right shift of 1 bit. In the specific case of the TMS32010 processor it is even easier since the TMS32010 architecture allows shifts to be performed as part of load/store operations with no extra CPU cycles required.

Use of the above concepts allowed an efficient FFT to be coded for the TMS32010 which used the highest possible precision while preventing overflow at any stage. To achieve the highest performance the TMS32010 FFT routine relied heavily on tables to eliminate the necessity of calculating trigonometric values and bit reversal values (used in descrambling). As an example of performance, the TMS32010 FFT implementation is capable of performing a 256 point 1-D FFT in approximately 23 msec.

As was pointed out earlier, an N x N 2-D FFT can be computed by performing 2N 1-D FFT's. For the TMS32010 FFT implementation described above this means a 256 x 256 point FFT would take 512 * 0.023 sec = 11.8 seconds (see Section 5.3). This value was very close to that actually measured in test runs on the Vista prototype (12.0 sec to 12.5 sec). The fact that the time was not consistent is to be expected since several of the TMS32010 instructions (involving transfers from program to data space in the Harvard architecture) take indeterminate amounts of time depending on machine state at time of execution.

While shortcomings were demonstrated by the current Vista implementation (indeed, that is the purpose of a prototype), it was possible to show that the Vista system is at once feasible and capable of high performance at a modest cost. If both the Vista hardware and the software development tools were improved, it is clear that a high performance image processing machine would result.

CHAPTER 7

Conclusions, Hindsight, and Future Plans

As a particular phase of a research project draws to a close, it is important to take stock of what has been learned, and to plan the direction that further research will take. Although based on a standard bus technology, Vista demonstrates that powerful multiprocessor architectures can be implemented at a reasonable cost.

Vista has shown the advantages of tailoring an architecture to meet the needs of the family of problems to be solved. This results in savings in the amount of interprocessor communication needed, a particularly important concern for bus structured machines. The benefits of relieving the processors of the burden of sending and receiving data and messages are also evident.

The fact that Vista is capable of performing most image processing operations in O(1) time (for a fixed image size) could have an interesting effect on software development. Currently programmers often are forced to choose algorithms whose performance is inferior to more expensive algorithms simply because the time constraints imposed by the problem at hand preclude the use of the superior algorithm. The Vista architecture will allow software engineers to choose the best algorithm, rather than accepting inferior performance as a tradeoff for speed.

Vista is not without defects. Although fault tolerant of Tachyon failures, failure of the MCU, RTU or bus would likely cause system failure. Memory could be made fault tolerant if it were partitioned into a number of (user transparent) blocks. If a block failed it could be disconnected from the system automatically. Modern memory components have reached a stage of high reliability, however, so it is doubtful these measures would be worth the effort.

Although the Vista project has demonstrated the basic feasibility of the Vista architecture, certain facets require further experimentation. In particular, methods need to be found to ensure that adjacent regions are processed consistently near their boundaries. For example, an edge enhancement algorithm would have to ensure that edges extending over several regions joined smoothly.

7.1. Development Tools

One thing that has come to be appreciated greatly in the development of Vista, is the importance of development tools. In particular, software simulation of the machine under development is essential. Modern high performance microprocessors and their associated logic execute instructions so rapidly as to make debugging via hardware tools nigh impossible. Even highly sophisticated logic analyzers provide only very limited glimpses into the execution of software due to the limited duration of time they are capable of recording. Although logic analyzers are highly useful in finding hardware bugs, software simulation allows step by step execution of new code under control of the developer. It seems that the first step in future development of Vista will be to enhance the existing simulation tools, and create new tools which model the entire Vista machine, instead of small portions of it (as is the case currently).

7.2. Hardware Improvements

Concerning improvements to the Vista hardware, several difficulties have emerged in the current Vista implementation that should be taken into account in the next design iteration. The current size of the Tachyon memory banks is adequate for most image processing operations encountered; however, certain situations do crop up in which larger, or at least re-balanced (ie. a re-proportioning of total TMS32010 address space assigned to data/code) memory banks would be beneficial. Currently the program memory bank is 3K words in size. Since even a fairly complicated algorithm, such as the FFT can be implemented in less than half that space, some of this could be allocated to the data memory bank. The data memory bank, although sufficiently large to accommodate most minimum regions needed by the majority of image processing algorithms, is not large enough to perform FFT's larger than 256 points easily. Since many images are larger than 256 x 256 pixels, it would be beneficial to double the data bank size to 2K words. This would allow FFT's of 512 x 512 images to be performed. An added benefit is that the augmented data banks would be capable of containing more minimum regions in other image processing algorithms. This in turn, would decrease the areas of overlap that would need to be transmitted more than once across the bus, thus reducing traffic on the bus.

The synchronization scheme is also a candidate for improvement. Although polling is sufficient when experimenting with a single Tachyon board, it would be more convenient if this burden could be

accomplished automatically when several Tachyons are used. One of the ideas discussed in Section 5.3 might be employed to tell the Tachyons to begin processing. Upon completion, the Tachyons could signal the MCU via an interrupt, thus relieving the need for the MCU to poll the Tachyon status flags constantly. The Tachyons themselves might well be upgraded. Several new digital signal microprocessors have recently become available which would decrease the number of Tachyons needed for a given level of performance. The TMS32020 or TMS32025 would be particularly good replacements, since they are capable of executing the already developed TMS32010 code. From a hardware perspective they also incorporate additional control signals that makes multiprocessing easier.

7.3. Transparent Multiprocessing

An area that would almost certainly become the focus of future Vista development is the development of multiprocessing software. Currently, the burden is on the programmer to specify exactly how the processing and synchronization should occur. This attention to tedious details is acceptable when demonstrating that an architecture is feasible, but would require excessive development time if Vista were to be used in a practical environment. Some consideration has been given as to how the burden of these details might be shifted from the programmer to the system software.

The division of images into sub-regions for the purpose of multi-processing is determined by the nature of the image processing algorithm employed. Nonetheless, most algorithms divide their image in a limited number of ways. For example, images are often subdivided into rows, columns, rectangles and perhaps hexagons. Although other shapes are certainly possible, the aforementioned strategies would suffice for the majority of image processing needs. Software tools could thus be provided that, given certain parameters (such as x, y size of a rectangle) would automatically divide the image into subregions, and store descriptors to these regions in a queue. This queue, containing descriptors of the regions still requiring processing, would allow a methodical way of assigning regions to idle Tachyons. The Tachyons, as they complete a processing task, interrupt the MCU and are placed on a queue of Tachyons who require unloading of processed data. As the processed data is moved back to the frame buffer, items are deleted from the 'Processed Regions' queue. The RTU determines the shape and size of the region to be transferred from information in the descriptor. (See Figure 7.1). As long as the regions can be processed in any order, all the programmer would be required to do would be to write the actual code that processed a

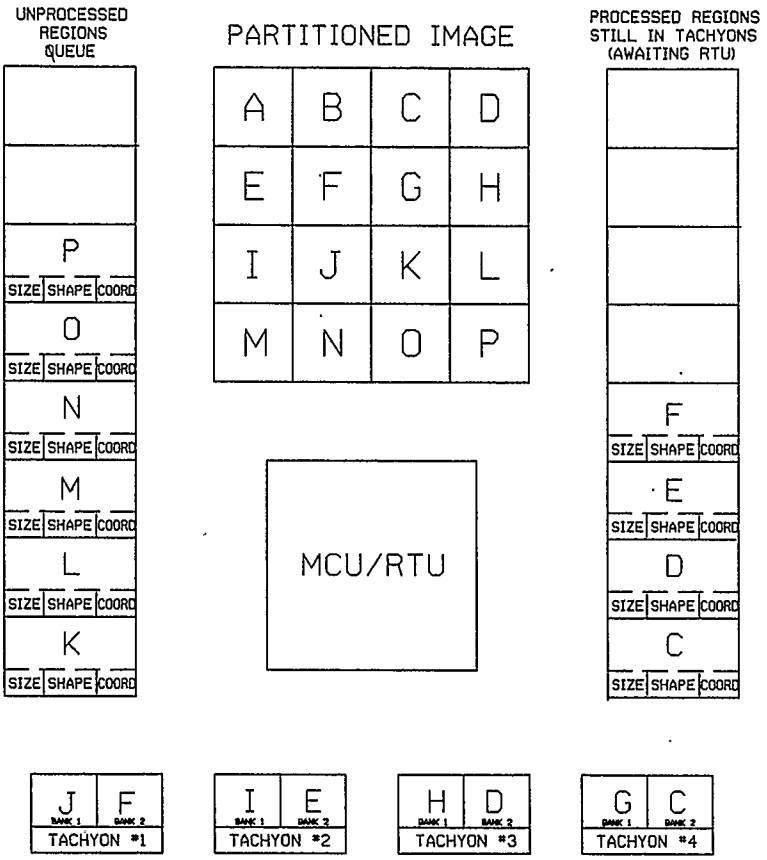

Figure 7.1 - Vista Scheduling System

region, and describe how the image is to be divided. Thus, most of the effort of writing multi-processing code would be shifted to the system software.

Most image processing algorithms do not specify any particular order for the processing of the image sub-regions. For those that do, it would be possible to add a field to the region descriptors that would contain a list of the regions that had to be processed first. Another queue of completed jobs consisting of the regions already processed would also be added. Thus, when a descriptor was to be removed from the queue of regions awaiting processing, the queue of completed jobs would be examined to determine if the jobs specified in the unprocessed region's descriptor were present. If they were not, then the unprocessed region would be moved to the end of the queue to be checked again later. Of course this scheme allows the possibility of deadlock, unless the order of processing is specified carefully. Since most image processing algorithms that might require specific order of processing of regions would probably do so in a straightfor-

ward fashion, it is not unreasonable to insist that the programmer specify the processing order in such a fashion that deadlock would not occur. Although automatic prevention of deadlock might be difficult, detection would be quite simple. If all processors were idle (meaning no region was being processed) and no item in the queue could proceed, then deadlock has occurred.

Since maintenance of the above mentioned queues would require significant processor attention, it might be necessary to insist that the MCU and RTU be separate, independent units. If the queues were maintained in memory local to the MCU, no increase in bus traffic would result, and performance would not be compromised by the automatic scheduling of regions.

The Vista architecture has shown great promise in the field of multiprocessing of images at reasonable hardware and software costs. It is hoped that this architecture will find practical application in the near future.

References

[1] Mannos, James L. (Editor) Design of Digital Image Processing Systems, *Proceedings of SPIE*, Vol 301, Aug 27-28, 1981, San Diego, California

[2] Preston, Kendall Jr., Uhr, Leonard (Editors) *Multicomputers and Image Processing - Algorithms and Programs*, Academic Press, New York 1982

[3] Hillis, W. Daniel *The Connection Machine*, The MIT Press, 1985 Cambridge Massachusetts Third Printing, 1986

[4] Oleinick, Peter N. *Parallel Algorithms on a Multiprocessor* UMI Research Press, Ann Arbor Michigan, 1982.

[5] Lang, T. Bandwidth of Crossbar and Multiple Bus Connections for Multiprocessors, *IEEE Transactions on Computers* C31 Dec 1982 pp. 1227-1234

[6] Anderson, George A., Jensen, E. Douglas Computer Interconnection Structures: Taxonomy, Characteristics; and Examples, *Computing Surveys* Vol 7 No 4 Dec 1975 (ACM) pp. 197-213

[7] Ahuja, Narendra, Swamy, Sowmitri Multiprocessor Pyramid Architectures for Bottom-Up Image Analysis, *IEEE Transactions on Pattern Analysis and Machine Intelligence*, Vol PAMI-6 No 4, July 1984 pp 463-475

[8] Kung, H.T. Why Systolic Architectures, *Computer*, Jan 1982, pp. 37-46

[9] Bancroft, John A personal conversation with T. Ingoldsby in the spring of 1986

[10] Reeves, Anthony P. Parallel Algorithms for Real-Time Image Processing, *Multicomputers and Image Processing* Edited by Preston, K. and Uhr, L. pp. 7-18

[11] R.C. Gonzalez, Wintz, Paul, *Digital Image Processing*, Addison-Wesley Publ. London 1977

[12] Castleman, Kenneth R. *Digital Image Processing*, Prentice-Hall 1979 N.J.

[13] Sheets, William Graf, Rudolf Pay-TV Descrambling, *Radio-Electronics*, July 1986 Vol 57 No 7, pp 44-46

[14] Lee, S.H. (Editor) Topics in Applied Physics *Optical Information Processing Fundamentals*, Vol. 48, Springer-Verlag Berlin 1981

[15] Brigham, E. Oran *The Fast Fourier Transform*, Prentice-Hall, N.J.

[16] Gonzalez, R.C. Wintz, Paul *Digital Image Processing*, Addison-Wesley Publ. London 1977

[17] Higgins, R.J. Fast Fourier transform: An introduction with some minicomputer experiments, *American*

Journal of Physics, Vol 44, No. 8, August 1976 pg 766

[18] Pease, Marshall C. An Adaptation of the Fast Fourier Transform for Parallel Processing, *Journal of the ACM*, Vol 15 No 2, April 1968 pp 252-264

[19] Swarztrauber, Paul N. FFT algorithms for vector computers, *Parallel Computing* (1984) pp 45-63, Elsevier Science Publishers B.V. (North Holland)

[20] Bowen, B.A. Buhr, R.J.A. *The Logical Design of Multiple Microprocessor Systems*, Prentice-Hall 1980 N.J.

[21] Anderson, George A. Jensen, E. Douglas ' Computer Interconnection Structures:Taxonomy, Characteristics and Examples, *ACM Computing Surveys*, Dec 1975, Vol 7, No 4, pp 197-213

[22] Gemmar, Peter Image Correlation: Processing Requirements and Implementation Structures on a Flexible Image Processing System (FLIP), *Multicomputers and Image Processing - Algorithms and Programs*, Edited by Kendall Preston Jr. and Leonard Uhr, Academic Press 1982 New York,

[23] Lang, Tomas Valero, Mateo Alegre, Ignacio Bandwidth of Crossbar and Multiple Bus Connections for Multiprocessors, *IEEE Transactions on Computers*, C31 Dec 1982 pp 1227-1234

[24] Kruse, Bjorn Gudmundsson, Bjorn Antonsson, Dan PICAP and Relational Neighborhood Processing in FIP, *Multicomputers and Image Processing - Algorithms and Programs*, Edited by Preston Jr., Kendall Uhr, Leonard Academic Press, N.Y. 1982 pp 31-46

[25] Temma, Tsutom et al Data Flow Processor Chip for Image Processing, *IEEE Transactions on Electron Devices*, Vol ED-32 No 9 September 1985

[26] Batcher, Kenneth E. Design of a Massively Parallel Processor, *IEEE Transactions on Computers*, Vol C-29 No 9, Sept 1980

[27] Hillis, W. Daniel New Computer Architectures and Their Relationship to Physics (or Why Computer Science is No Good), *International Journal of Physics*, Vol 21, Nos. 3/4 1982 pp 255-263

[28] Zhang, Chang nian et al Multi-Dimensional Systolic Networks for Discrete Fourier Transform SIGARCH ewsletter, Volume 12, Issue 3, June 1984, *The 11th Annual International Symposium on Computer Architecture Conference Proceedings*, pp 215 - 222

[29] Kung, H.T. Special-purpose devices for signal and image processing: an opportunity in very large scale integration (VLSI) *Proceedings of SPIE*, Vol 241 Real-Time Signal Processing III (1980)

[30] Kung, H.T. Why Systolic Architectures, *IEEE Computer*, January 1982, pp 37-46

[31] Parker, J. "Analysis and Simulation of a Common Bus Multiprocessor" An unpublished paper by Prof J. Parker, U of Calgary

[32] Intel iAPX286/10 data booklet #210253-008, May 1985

[33] Intel Corporation OEM Systems Handbook, Santa Clara, California, 1985

APPENDIX A

A BASIC09 FFT Program

```
PROCEDURE fft
(* Performs an 'in-place' FFT as described in Brigham's *)
(* book, 'The Fast Fourier Transform'. *)
(* The FFT algorithm is basically a divide and conquer *)
(* algorithm. This can be seen in the signal flow graphs *)
(* used to represent the FFT. In each column of the signal *)
(* graph are a number of sub-FFTs. The elements in each *)
(* sub-FFT can be thought of as an entirely new data set *)
(* which needs to transformed, independent of other sub-FFTs *)
(* in that column. Each successive column causes the number *)
(* of sub-FFTs to double, but the size of these sub-FFTs to *)
(* halve. This continues until the sub-FFTs are of size 1 *)
(* when no further processing is required. At this point *)
(* the FFT is complete. *)
(* Clearly, if no processing of the data occurs, subdividing *)
(* the original data array will have had no effect. It is *)
(* necessary to process each element in a given column by *)
(* multiplying it by a factor, W^p, and adding it to a *)
(* corresponding element in an adjacent (soon to be) *)
(* sub-FFT. These elements are the so-called 'Dual Node *)
(* Pairs'. *)
(* For each sub-FFT is is necessary to calculate the factor *)
(* Owing to the symmetry inherent in W^p, it is possible to *)
(* compute W^p for two sub-FFTs using only a single (complex) *)
(* multiplication. This is because W^p = -W^(p+N/2). *)
(* This effectively cuts the number of multiplications in *)
(* half. *)
(* To accomplish the above, 3 loops are necessary. The *)
(* outermost loop corresponds to the columns in the signal *)
(* flow graph. Clearly it must have lg(numpts) iterations *)
(* if the original is to be sub-divided until the sub-FFT *)
(* size reaches 1. The middle loop causes the appropriate *)
(* number of groups of data to be processed and sub-divided. *)
(* The inner loop actually processes the data s.t. it is *)
(* ready for the next subdivision. *)
(* *)
(* Since there are lg(numpts) columns, and in each column *)
(* there occur ~numpts/2 multiplications and ~numpts *)
(* additions, it can be seen that the order of the FFT is *)
(* nlog(n). *)
(* *)
(* Unfortunately after the FFT has been computed, the *)
(* data lies in scrambled form. It is then unscrambled *)
(* using 'scrambler' *)

TYPE COMPLEX = reel, image:REAL  (* real is a reserved word *)

(* datval contains the points to be transformed. *)
```

```
(* numpts is the number of points (2^int).  If *)
(* forward = TRUE then a forward FFT is performed.  If *)
(* forward = FALSE, a reverse FFT occurs and all data *)
(* values are divided by numpts. *)

PARAM datval(32):COMPLEX
PARAM numpts:INTEGER
PARAM forward:BOOLEAN

DIM column, subfft, datapt:INTEGER
DIM numcols, numffts, fftsize:INTEGER
DIM expi:REAL
DIM startsub, endsub:INTEGER
DIM scrampos:INTEGER
DIM npi_2:REAL
DIM a,b,wp:COMPLEX
(* a, b are temporary variables.  wp is the twiddle factor *)
DIM i,j:INTEGER

BASE 0

numcols = FIX(LOG(numpts)/LOG(2))

IF forward THEN
  expi = 1.0
ELSE
  expi = -1.0
  FOR i=0 TO numpts-1
    datval(i).reel=datval(i).reel/numpts
    datval(i).imag=datval(i).imag/numpts
  NEXT i
ENDIF

(* Begin In-Place Cooley-Tukey Algorithm *)
fftsize=numpts/2
npi_2=2.0*PI/numpts
numffts=1

FOR column=1 TO numcols
  startsub=0
  endsub=fftsize

  FOR subfft=1 TO numffts
    (* Compute the factor (W^p) for this sub-FFT *)
    (* Notice that using:  scrampos=startsub/fftsize *)
    (* is equivalent but cheaper than: *)
    (*   scrampos=startsub/2^(numcols-column)  *)
    (* W^p is sometimes called the 'twiddle' factor. *)

    (* First shift the index: *)
    scrampos=startsub/fftsize
    (* and now bit reverse it: *)
    RUN scrambler( numpts+0, numcols+0, scrampos )
    wp.reel=COS(npi_2*scrampos)
    wp.imag=expi*SIN(npi_2*scrampos)

    FOR datapt=startsub to endsub-1
```

```
      (* Now process both current element *)
      a.reel=datval(datapt).reel
      a.imag=datval(datapt).imag
      b.reel=wp.reel*datval(datapt+fftsize).reel -
          wp.imag*datval(datapt+fftsize).imag
      b.imag=wp.imag*datval(datapt+fftsize)reel +
          wp.reel*datval(datapt+fftsize).imag

      datval(datapt).reel=a.reel+b.reel
      datval(datapt).imag=a.imag+b.imag

      (* and the dual-node pair *)
      datval(datapt+fftsize).reel=a.reel-b.reel
      datval(datapt+fftsize).imag=a.imag-b.imag

    NEXT datapt
    startsub=startsub+2*fftsize
    endsub=endsub+2*fftsize
  NEXT subfft
  fftsize=fftsize/2
  numffts=numffts*2
NEXT column

(* FFT is now complete, but scrambled.  Fix it. *)
FOR i=0 TO numpts-1
  j=i
  RUN scrambler(numpts+0,numcols+0,j)
  IF j >= i THEN
    a.reel=datval(i).reel
    a.imag=datval(i).imag
    datval(i).reel=datval(j).reel
    datval(i).imag=datval(j).imag
    datval(j).reel=a.reel
    datval(j).imag=a.imag
  ENDIF
NEXT i
END

PROCEDURE scrambler
(* This routine scrambles and unscrambles the locations of *)
(* data in the FFT array (bit reverse) *)
PARAM numpts, orderfft, position:INTEGER
(* 'numpts' and 'orderfft' MUST be passed by value. *)
(* 'position' MUST be passed by reference. *)

DIM i,tempos:INTEGER

tempos=0

orderfft=orderfft-1

FOR i=0 TO orderfft
  numpts=numpts/2
  IF position>=numpts THEN
    tempos=tempos+2^i
    position=position-numpts
  ENDIF
```

```
NEXT i

position=tempos

END

PROCEDURE dummy
(* A sample driver program *)
TYPE COMPLEX=reel, imag:REAL

DIM values(32):COMPLEX
DIM i,j:INTEGER
BASE 0

FOR i=0 TO 31
  values(i).reel=SIN((4.0*i*PI)/32)
  values(i).imag=0
NEXT i

RUN fft(values,32,TRUE)

FOR i=0 TO 31
  PRINT "i=";i;"  ";values(i).reel, values(i).imag
NEXT i

END
```