(12)

AD-A135 732

Semiannual Technical Summary

Multi-Dimensional Signal Processing Research Program

31 March 1983

Prepared for the Department of the Air Force
under Electronic Systems Division Contract F19628-80-C-0002 by

Lincoln Laboratory

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

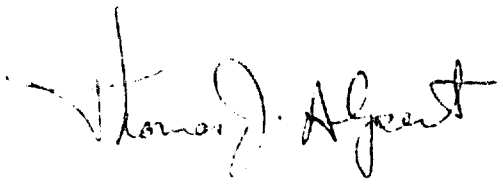
LEXINGTON, MASSACHUSETTS

83 12 13 045

The work reported in this document was performed at Lincoln Laboratory, a center for research operated by Massachusetts Institute of Technology, with the support of the Department of the Air Force under Contract F19628-80-C-0002. A part of this support was provided by the Rome Air Development Center.

This technical report has been reviewed and is approved for publication.

FOR THE COMMANDER



Thomas J. Alpert, Major, USAF
Chief, ESD Lincoln Laboratory Project Office

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

LINCOLN LABORATORY

MULTI-DIMENSIONAL SIGNAL PROCESSING RESEARCH PROGRAM

SEMIANNUAL TECHNICAL SUMMARY REPORT TO THE ROME AIR DEVELOPMENT CENTER

1 OCTOBER 1982 — 31 MARCH 1983

ISSUED 11 OCTOBER 1983

LEXINGTON MASSACHUSETTS

ABSTRACT

This Semiannual Technical Summary covers the period 1 October 1982 through 31 March 1983. It describes the significant results of the Lincoln Laboratory Multi-Dimensional Signal Processing Research Program, sponsored by the Rome Air Development Center, in the areas of multiprocessor architectures for image processing and algorithms for object detection and region classification in aerial reconnaissance imagery.

TABLE OF CONTENTS

LIST OF ILLUSTRATIONS

1. INTRODUCTION AND SUMMARY

The Lincoln Laboratory Multi-Dimensional Signal Processing Research Program was initiated in FY 80 as a research effort directed toward the development and understanding of the theory of digital processing of multi-dimensional signals and its application to real-time image processing and analysis. A specific long-range application is the automated processing of aerial reconnaissance imagery. Current research projects that support this long-range goal are image modeling for target detection and multiprocessor architectures to implement image processing algorithms.

This Semiannual Technical Summary discusses our work in several areas. In Section 2 we present a discussion of our recent efforts in improving the target detection algorithm developed in FY 82. The improvements lie in modifying the algorithm to detect targets of size comparable to that of the estimation window.

In Section 3 we present some preliminary theoretical results in the area of vector-valued image processing. This approach assumes that sensor measurements give rise to multispectral or vector-valued image picture elements (pixels). Techniques based on signal processing and detection theory are then developed which differ fundamentally from classical feature extraction and pattern recognition techniques.

Sections 4 and 5 contain discussions of multiprocessor architectures that are useful for processing image data. In Section 4 we present an overview of a multiprocessor architecture consisting of 16 nodal processors that communicate through a butterfly interconnection network, and we report on our efforts to develop an integrated circuit to implement the basic switching element. Section 5 focuses on the emerging nodal processor architecture itself and discusses how certain architectural principles could be used to obtain a high-performance nodal processor.

Finally, in Section 6 we elaborate on some recent improvements to our image processing facility that we have found useful in the development of algorithms for the processing of aerial reconnaissance photographs.

2. TARGET DETECTION

During the current reporting period, we have continued our work in target detection along two lines. First, we have coded the algorithm for detection of anomalous areas in the C language and incorporated it as a module in our image processing facility on the VAX computer (see Section 6). Secondly, we have begun to explore methods for using linear prediction to detect larger targets. Each of these activities is discussed separately below.

2.1 ANOMALY DETECTION

The algorithm for detection of anomalous areas in aerial photographs (see References 1 and 2) was originally developed in the APL language on the Lincoln Laboratory Amdahl 470 central computer. The APL language provided a suitable environment for development of the algorithm since it is interactive and permits algorithm variations to be coded and tested rapidly. We have now incorporated the algorithm as a module in our image processing laboratory facility. The algorithm was completely rewritten in the C language and compiled for more efficient execution. The new version of the algorithm is flexible with respect to parameter selection and is convenient to use for someone not totally familiar with its details or with the APL language.

The algorithm as coded still requires relatively large amounts of computer time due to the lengthy matrix calculations. We are currently seeking ways to reduce the computation by using algorithms that are recursive and exploit symmetry.

2.2 LARGE TARGET DETECTION

The algorithm referred to in the previous section was developed for the purpose of detecting point targets or anomalies. Such targets are either a single pixel or at most a few pixels in diameter. The philosophical point of view taken for this problem was that statistical parameters for the background would either be known *a priori* or could be estimated from the data while target parameters (since targets may be a variety of different types) were basically unknown. If one then examines a small area of the image, one could ask the question "Is this area pure background, or is it not?" If the area is not pure background, then it must contain a target. The branch of statistics that deals with such questions is that of significance testing. When significance testing is applied to this problem and combined with certain other assumptions, a 2-D adaptive linear prediction algorithm results. This algorithm compares the sum of normalized linear prediction errors surrounding each pixel to a threshold to detect the "targets."

This approach can be extended to the detection of larger objects (say tens of pixels wide). Philosophically, one can still take the position that background parameters can be estimated and that target parameters are unknown and apply a significance test to small areas of the image. However, since in this case target parameters significantly affect the estimation of background parameters, a slightly different tack has to be taken.

Consider first a situation where a target has a portion of its boundary roughly parallel to the left edge of the image as shown in Figure 2-1. The approach considered here is to detect points along the left edge of the object boundary by an algorithm that processes data horizontally from the left side of the image. The remaining boundary points are found in a similar manner by processing in three other directions.

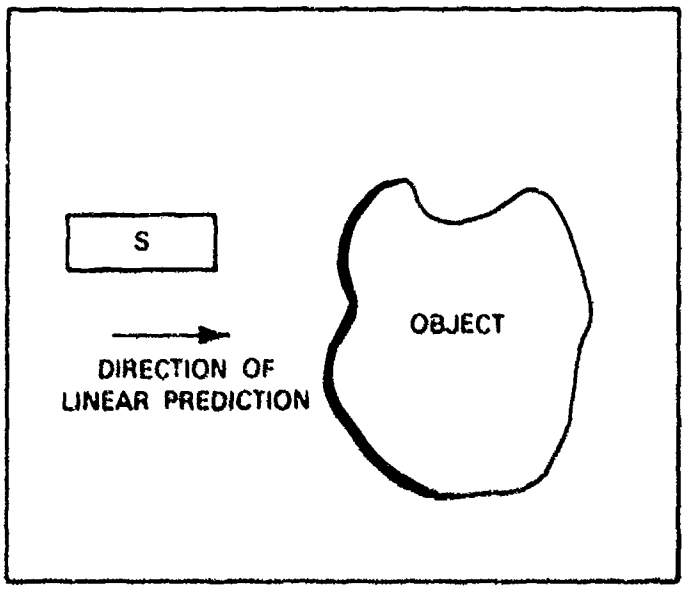


Figure 2-1. Boundary detection by significance testing.

We will attempt to detect points along the left edge of the object boundary (heavy line) by considering small rectangular regions one row wide and performing a significance test. If the region S corresponding to a given point fails the test, then the point is considered to be a boundary point or an interior point of the object. If we were to scan the image from top to bottom, left to right, performing this test at each pixel, we would arrive at some number of points which are candidates for boundary or object points. Because the test does not give perfect results, we expect that this procedure would produce a number of points in the image that are not truly object or boundary points (false alarms) and miss a number of the true boundary or object points. One way to improve the procedure therefore is to couple the decisions for nearby points by some additional structure imposed on the problem. We can do this by modifying the significance test to incorporate a "prior" probability that depends on the assignment of neighboring points.

Recall that our implementation of the significance test for images involves comparing the joint probability density function for points in the region S to a threshold. That is, we look at

$$p(\underline{x}) > \lambda$$

where p represents the background probability density function. If the inequality is true, we conclude that only background is present. Otherwise, we decide that there are some target points in the region S. In the modified significance test, we make the comparison

$$P_0 \cdot p(\underline{x}) > \lambda$$

where P_0 is the prior probability of background. Clearly when P_0 is large, we are more likely to decide on background and when P_0 is small we are more likely to decide on a target. If the density function $p(\underline{x})$ can be approximated by a multivariate Gaussian form, then some straightforward procedures (see Reference 2) permit reformulation of the test in terms of the error residuals of linear prediction. In particular, the modified significance test for constant false alarm takes the form

$$\sum_S \left[\frac{\epsilon^2(n,m)}{\sigma^2_{n,m}} \right] - 2\ell n\ P_0 < \lambda'$$

where λ' is a threshold derived from λ. The error residuals ϵ and variances σ^2 can be computed adaptively from the data as described in Reference 2.

The decisions made by applying the significance test to neighboring points can now be linked through the prior probabilities P_0. In particular, we assume that the prior probability evolves in a Markov-like manner, i.e., its value depends on the decisions made at previously considered points. If we allow P_0 to be dependent on points in a region which is a non-symmetric half plane located above and to the left of the given point (see Figure 2-2), then in scanning the image from the upper left corner successive decisions can be made without iteration.

We have implemented a rudimentary version of this algorithm on the VAX facility and are beginning to test its performance on data with known statistical characteristics. We are currently studying how differences in parameters that specify the statistical character of the background and target data affect our ability to detect the boundary.

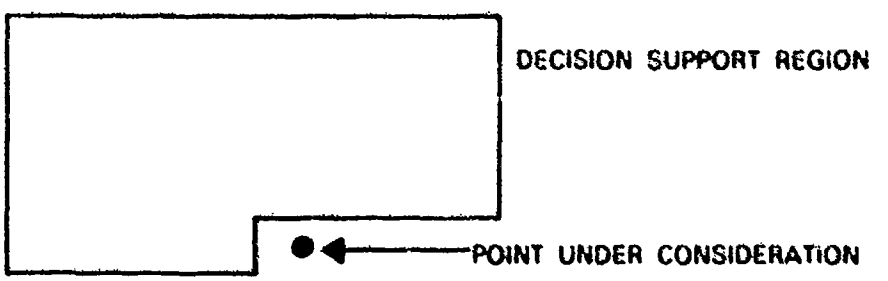


Figure 2-2. Support region for evolution of prior probability.

3. VECTOR-VALUED IMAGE PROCESSING

During the current reporting period, we have been considering the extension of the existing algorithms for segmentation and target detection to the case of vector-valued images. By a vector-valued image we mean a set of related images in pixel alignment which are treated as a single entity for purposes of image processing (see Figure 3-1). Examples would be the set of three color components representing a color image, the multiple channel images taken at differing wavelengths by certain types of scanning sensors, and a set of images taken from different sensors (e.g., radar, IR, video) that have been registered for purposes of joint processing.

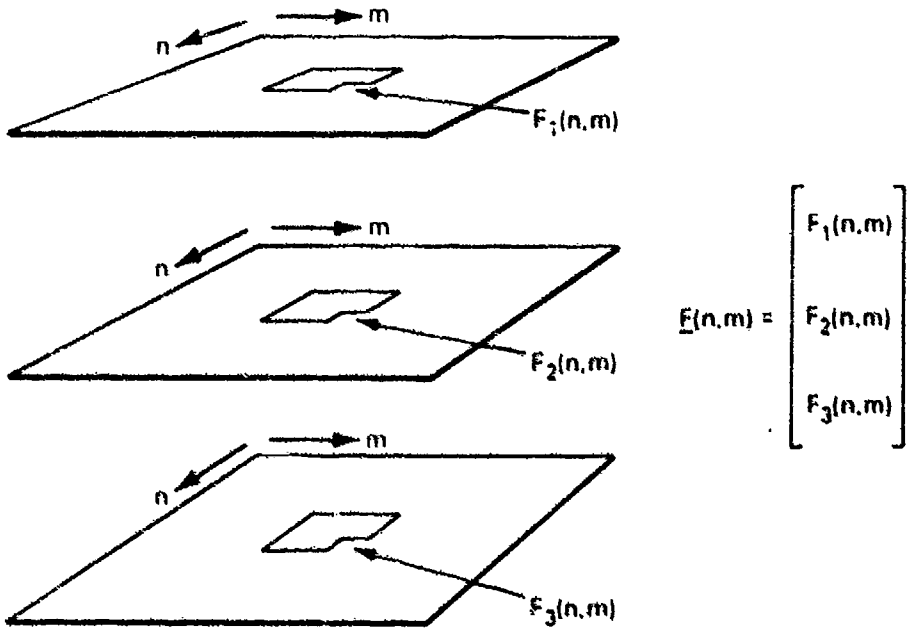


Figure 3-1. Vector autoregression model for images.

Although a considerable amount of work has been done on vector-valued images using pattern recognition ideas (extracting features and performing classification and clustering on the features), very little work has been done in modeling images as vector-valued spatial random processes and examining the related problems of filtering, prediction, estimation and so on for this class of images. Our attempts to extend the existing algorithms for segmentation and object detection to vector-valued images are a first step in this direction.

3.1 EXTENSIONS OF SEGMENTATION AND DETECTION ALGORITHMS

Both the segmentation algorithm and the object detection algorithm are based on an autoregressive model for images. The vector form of this model is

$$\underline{F}(n,m) = - \sum_{\substack{i=0 \\ j=0 \\ (i,j)\neq(0,0)}}^{\substack{M-1 \\ N-1}} A_{ij}\, \underline{F}(n-i,\, m-j) + \underline{W}(n,m) \qquad (1)$$

where $\underline{F}$ is the modeled image and $\underline{W}$ is a white noise driving term. Both $\underline{F}$ and $\underline{W}$ represent vector quantities with K components and the A_{ij} are $K \times K$ matrices. The model is illustrated in Figure 3-1 for K = 3. A typical point with coordinates (n,m) is shown in each of the three image planes and the region of summation in Equation (1), also known as the region of support, is shown as a rectangular area that excludes the given point. The particular support illustrated and used in Equation (1) is referred to as quarter plane support and will be the specific form considered in the rest of this discussion. Other forms of support are also possible.

The driving term $\underline{W}$ is not illustrated in Figure 3-1. However, one can picture this as another set of planes consisting of white noise. For each point (n,m) in the image a set of white noise terms is selected from the corresponding point in the noise planes and used in the model. It is important to note that although the white noise terms are uncorrelated spatially, these terms are generally correlated between image planes. Thus, the white noise vector $\underline{W}$ at any point (n,m) is described by a $K \times K$ covariance matrix S_w which is not, in general, diagonal.

Once a model of the form (1) is postulated, then it is possible to generalize the segmentation and object detection algorithms to the vector image case. We will not go through derivations here but merely state the results.

3.2 VECTOR M.A.P. SEGMENTATION

Recall that the maximum *a posteriori* (MAP) segmentation algorithm attempts to find regions in the image with homogeneous texture (see Reference 3). The algorithm does this by assigning "states" to the pixels; when a pixel has state s_i it is interpreted as belonging to a region with texture class i. Let $\underline{F}^{(i)}(n,m)$ represent the vector image $\underline{F}$ after removal of the mean of class i.

Define the vector prediction error $\underline{E}^{(i)}(n,m)$ as

$$\underline{E}^{(i)}(n,m) = \underline{F}^{(i)}(n,m) + \sum_{\substack{\ell=0 \\ j=0 \\ (\ell,j)\neq(0,0)}}^{\substack{M-1 \\ N-1}} A_{\ell j}\, \underline{F}^{(i)}(n-\ell,\, m-j) \quad . \qquad (2)$$

Then the MAP segmentation algorithm assigns states s(n,m) to pixels by iteratively evaluating the expressions

$$[\underline{E}^{(i)}(n,m)]^T\ [\Sigma^{(i)}]^{-1}\ \underline{E}^{(i)}(n,m) + \ell n\ |\Sigma^{(i)}| - 2\ell n\ \text{Prob}\ [s(n,m) = i|S_{n,m}] \qquad (3)$$

where $\Sigma^{(i)}$ is the prediction error covariance matrix and $S_{n,m}$ is a set of states for pixels surrounding point (n,m). At each stage of the iteration, s(n,m) is assigned the label i_0 where i_0 is the value of i that minimizes expression (3). For the two-class segmentation, this procedure results in a comparison of (3) for i = 0 to a similar expression for i = 1 and an assignment of states accordingly. The final states arrived at by iteratively applying (3) correspond to a segmentation of the image into regions.

3.3 VECTOR TARGET DETECTION

The target detection algorithm (see References 1 and 2) uses statistical significance testing to decide if a small area of an image S centered at a point (n,m) is pure background or if it contains something else (a "target"). The algorithm performs this test using the error residuals of linear prediction. Let $\underline{E}^{(0)}(n,m)$ be defined by (3) where in this case the mean of the image is estimated locally and subtracted from $\underline{F}$ to obtain the zero mean image $\underline{F}^{(0)}$. Then the vector image form of the object detection algorithm makes the comparison

$$\sum_{(n,m)\epsilon S} [\underline{E}^{(0)}(n,m)]^T\ [\Sigma^{(0)}]^{-1}\ \underline{E}^{(0)}(n,m) \quad > \quad \lambda\ . \qquad (4)$$

If the quantity on the left is above threshold, an object is announced to exist at point (n,m). Otherwise, no object is assumed to have been present. Observe that in both this algorithm and the segmentation algorithm (above), the coupling between image planes appears in two ways. It appears first in the linear prediction coefficients A_{ij}, since any component of the error residual $\underline{E}(n,m)$ in a given plane generally depends on data in all planes. Secondly, it appears in the Equations (3) and (4) where the error terms are coupled through the prediction error covariance matrix Σ. It will be seen later that even when simplifying assumptions are made in the autoregressive model that lead to a decoupling of the linear prediction coefficients, the coupling appearing in Equations (3) and (4) is still present. Thus, when dealing with vector images it is generally suboptimal to perform a scalar version of the algorithm on separate image planes and attempt to combine results.

3.4 ALGORITHM PARAMETER ESTIMATION

The vector image processing algorithms described above require the computation of matrix linear prediction parameters. Computation of these parameters involves solution of so-called Normal equations which, although linear, can be of high dimensionality in the vector case. In this section, we examine the form of these equations, some of their properties, and discuss possible procedures for simplifying their solution.

IMAGE VECTORS

$$\underline{f} = \begin{bmatrix} \underline{f}_1 \\ \underline{f}_2 \\ \vdots \\ \underline{f}_N \end{bmatrix} \quad \text{WHERE } \underline{f}_n = \begin{bmatrix} \underline{F}(n,1) \\ \underline{F}(n,2) \\ \vdots \\ \underline{F}(n,M) \end{bmatrix} \quad \text{WHERE } \underline{F}(n,m) = \begin{bmatrix} F_1(n,m) \\ F_2(n,m) \\ \vdots \\ F_K(n,m) \end{bmatrix}$$

CORRELATION MATRIX

$$\mathbf{R} = E[\underline{f}\,\underline{f}^T] = \begin{bmatrix} R(0) & R(1) & \cdots & R(N-1) \\ R(-1) & R(0) & \cdots & R(N-2) \\ \vdots & & & \\ R(-N+1) & & \cdots & R(0) \end{bmatrix}$$

BLOCK TOEPLITZ
KM × KM BLOCKS

WHERE

$$R(k) = E[\underline{f}_n \underline{f}_{n-k}^T] = \begin{bmatrix} R(k,0) & R(k,1) & \cdots & R(k,M-1) \\ R(k,-1) & R(k,0) & \cdots & R(k,M-2) \\ \vdots & & & \\ R(k,-M+1) & & \cdots & R(k,0) \end{bmatrix}$$

BLOCK TOEPLITZ
K × K BLOCKS

WHERE

$R(k,\ell) = E[\underline{F}(n,m)\,\underline{F}(n-k,m-\ell)]$ IS NOT TOEPLITZ

12957-04

Figure 3-2. Row ordering.

The statistical characterization of vector images begins with grouping the various intensity values $F_k(n,m)$ that make up the vector image into one larger vector. This vector will have NMK components. Although many orderings for these components are possible, two such orderings appear to be most useful. The first ordering, which is shown in Figure 3-2, will be called row ordering. In this case, the N vector rows of the image are stacked into one large vector. Each row vector consists of M column elements, each of which is a K-dimensional vector quantity. Thus, this ordering is similar to that which would be used for scalar images except the image elements forming the components of the vector are not scalars but K-dimensional vectors.

The second ordering that we shall consider is shown in Figure 3-3 and will be called component ordering. In this case, each image plane is scanned by rows and represented by a vector. These vectors are then stacked in component order to arrive at the vector representation for the set of images. Both orderings appear to be useful and are related by a permutation transformation. We shall discriminate the two orderings notationally by using unprimed variables for row ordering and primed variables for component ordering.

Figures 3-2 and 3-3 also show the form of the correlation matrices (or covariance matrices) for the two orderings. Since the correlation matrix is the expectation of the outer product of the vectors, a particular ordering of the vectors induces a corresponding partitioning of the matrix. If it is assumed that the images are stationary in the spatial dimensions, then their correlation matrices have certain symmetry properties. In particular, for row ordering, the matrix is block Toeplitz with block Toeplitz blocks. The innermost blocks are not Toeplitz in general. For component ordering, the overall matrix is neither block Toeplitz nor block symmetric. The blocks however *are* block Toeplitz and are comprised of Toeplitz blocks.

The Normal equations of linear prediction are most conveniently written with row ordering. In this case, the equations assume the form

$$\mathbf{RA} = \mathbf{S} \tag{5}$$

where $\mathbf{R}$ is the row-ordered correlation or covariance matrix and the matrices $\mathbf{A}$ and $\mathbf{S}$ are given by

$$\mathbf{A} = \begin{bmatrix} A_0 \\ A_1 \\ \cdot \\ \cdot \\ \cdot \\ A_{N-1} \end{bmatrix} \quad ; \quad \mathbf{S} = \begin{bmatrix} S_0 \\ 0 \\ \cdot \\ \cdot \\ \cdot \\ 0 \end{bmatrix} \tag{6}$$

IMAGE VECTORS

$$\underline{f}' = \begin{bmatrix} \underline{f}^1 \\ \underline{f}^2 \\ \bullet \\ \bullet \\ \bullet \\ \underline{f}^K \end{bmatrix} \quad \text{WHERE } \underline{f}^k = \begin{bmatrix} \underline{f}^k_1 \\ \underline{f}^k_2 \\ \bullet \\ \bullet \\ \bullet \\ \underline{f}^k_N \end{bmatrix} \quad \text{WHERE } \underline{f}^k_n = \begin{bmatrix} F_k(n,1) \\ F_k(n,2) \\ \bullet \\ \bullet \\ \bullet \\ F_k(n,M) \end{bmatrix}$$

CORRELATION MATRIX

$$R' = E[\underline{f}' \, \underline{f}'^T] = \begin{bmatrix} R_{11} & R_{12} \bullet\bullet\bullet R_{1K} \\ R_{21} & R_{22} \bullet\bullet\bullet R_{2K} \\ \bullet \\ \bullet \\ \bullet \\ R_{K1} & R_{K2} \bullet\bullet\bullet R_{KK} \end{bmatrix} \quad \text{NM} \times \text{NM BLOCKS}$$

WHERE

$$R_{\ell k} = E[\underline{f}^\ell \underline{f}^{kT}] = \begin{bmatrix} R^{\ell k}_0 & R^{\ell k}_1 \bullet\bullet\bullet R^{\ell k}_{N-1} \\ R^{\ell k}_{-1} & R^{\ell k}_0 \bullet\bullet\bullet R^{\ell k}_{N-2} \\ \bullet \\ \bullet \\ \bullet \\ R^{\ell k}_{-N+1} & R^{\ell k}_{-N+2} \bullet\bullet\bullet R^{\ell k}_0 \end{bmatrix} \quad \begin{array}{l} \text{BLOCK TOEPLITZ} \\ \text{WITH TOEPLITZ BLOCKS} \\ \text{M} \times \text{M BLOCKS} \end{array}$$

WHERE

$R^{\ell k}_j = E[\underline{f}^\ell_i \, \underline{f}^k_{i-j}]$ 'S TOEPLITZ

1296271-4

Figure 3-3. Component ordering.

where

$$A_n = \begin{bmatrix} A_{n0} \\ A_{n1} \\ \cdot \\ \cdot \\ \cdot \\ A_{nM-1} \end{bmatrix} \quad ; \quad S_0 = \begin{bmatrix} \Sigma \\ 0 \\ \cdot \\ \cdot \\ \cdot \\ 0 \end{bmatrix} \tag{7}$$

and where $A_{00} = I$ and Σ is the prediction error covariance matrix. Note that the quantities **A** and **S** are MNK-by-K matrices.

The structure of these equations seems to have many possibilities for obtaining a solution to (5) by methods other than brute force. The development of recursive solutions to the Normal equations has led to interesting results in both the vector time series cases and in the (scalar) two-dimensional image cases (References 4, 5, and 6). These kinds of solutions are often ideally suited for recursive estimation of parameters as is required for the target detection algorithm. We are currently seeking to extend some of these results to the vector image case.

Some special results for solution of the Normal equations can be obtained when the covariance matrix **R** has some further structure. This is the case of when correlation between image planes is separate from correlations within each plane. Hunt and Kubler[7] have indicated that in many cases such a separation of correlations is well justified empirically. This separability of correlation is best represented in terms of component ordering where the matrix **R'** has the direct product form

$$\mathbf{R}' = R_s \times R_c = \begin{bmatrix} r_{11} R_s & r_{12} R_s \cdots r_{1K} R_s \\ r_{21} R_s & r_{22} R_s \cdots r_{2K} R_s \\ \cdot & \\ \cdot & \\ \cdot & \\ r_{K1} R_s & r_{K2} R_s \cdots r_{KK} R_s \end{bmatrix} \tag{8}$$

where R_s is a matrix of spatial correlation parameters within each image and the r_{ij} are the components of R_c representing correlations between image planes. In this special case, one can show that the prediction coefficient matrices A_{ij} in (2) are diagonal with equal diagonal elements. Thus, the prediction problems for the various planes are decoupled and within

each plane the optimal linear predictive filters are identical. However, the prediction error covariance matrix Σ that appears in Equations (3) and (4) is *not* diagonal and is in fact equal to R_c. Thus, the prediction errors are correlated from plane to plane and this fact must be acknowledged in a vector-oriented image processing algorithm.

The foregoing analysis indicates that there are a number of potentially interesting topics associated with vector image analysis and that there is potential for improving results for single images by treating the analysis of a set of images as a vector image processing problem. This area is one that seems to have been overlooked in most of the classical work in image analysis.

4. MULTIPROCESSOR ARCHITECTURES FOR IMAGE PROCESSING

As part of the Multi-Dimensional Signal Processing Research Program sponsored by RADC, we have been investigating multiprocessor architectures for image processing and other multi-dimensional signal processing applications. Typical image processing problems involve data sets comprised of several hundred thousands (or millions) of pixels (picture elements) and large amounts of computation (several hundreds of processor instructions per pixel). To satisfy the real-time requirements of some applications and the general requirement for rapid turnaround, the multiprocessor architecture must be capable of supporting rapid computation on large data sets.

To utilize fully the processing power of the multiprocessor system, it is imperative that the system be straightforward to program. This will allow new ideas to be implemented, tested, evaluated, and refined in a timely fashion. Thus, the two primary requirements of the multiprocessor architecture are high speed (in an operational sense, not necessarily in an instructions/second sense) and ease of programming.

We have been considering a multiprocessor architecture consisting of sixteen nodal processors connected by an interprocessor communications network (see Figure 4-1). The number of processors is somewhat arbitrary from an architectural point-of-view; sixteen was chosen because it is a power of two of nontrivial yet manageable size. This architecture implies that there is no shared memory; each nodal processor has its own, private data memory and data are passed between processor memories using the communication network. The architecture of the nodal processors is discussed in Section 5.

The preliminary choice for the interprocessor communication network is a butterfly network, which necessitates a power-of-two number of processors (see Section 4.1). In the future other communication networks, such as a fibre-optic bus, may prove useful for interprocessor communication. Thus, the multiprocessor architecture must be modular so that communication networks can be interchanged without requiring significant hardware or software changes.

A preliminary guess indicates that the 16-processor system could execute roughly 200M instructions/second. This rate is somewhat less than that required for operational real-time image analysis. (At this time our best guess is that an operational system will be required to execute 1G-10G instructions/second.) If the individual instructions are powerful enough, it is conceived that a second-generation 64-processor system could meet the operational throughput requirements.

(The range of 1G-10G instructions/second was arrived at very crudely. A photointerpreter wishing to process a high-resolution image consisting of 4K-by-4K pixels in 1 s with 100-1000 instructions/pixel requires a multiprocessor bandwidth of 1.6G-16G instructions/second. The MIES channel capacity of 144M bps results in a pixel rate of 18M pixels/second, assuming 8 bits/pixel. Using 100-1000 instructions/pixel for processing results in a processor bandwidth requirement of 1.8G-18G instructions/second. The range of 100-1000 instructions/pixel is probably reasonable for many current image processing algorithms. If

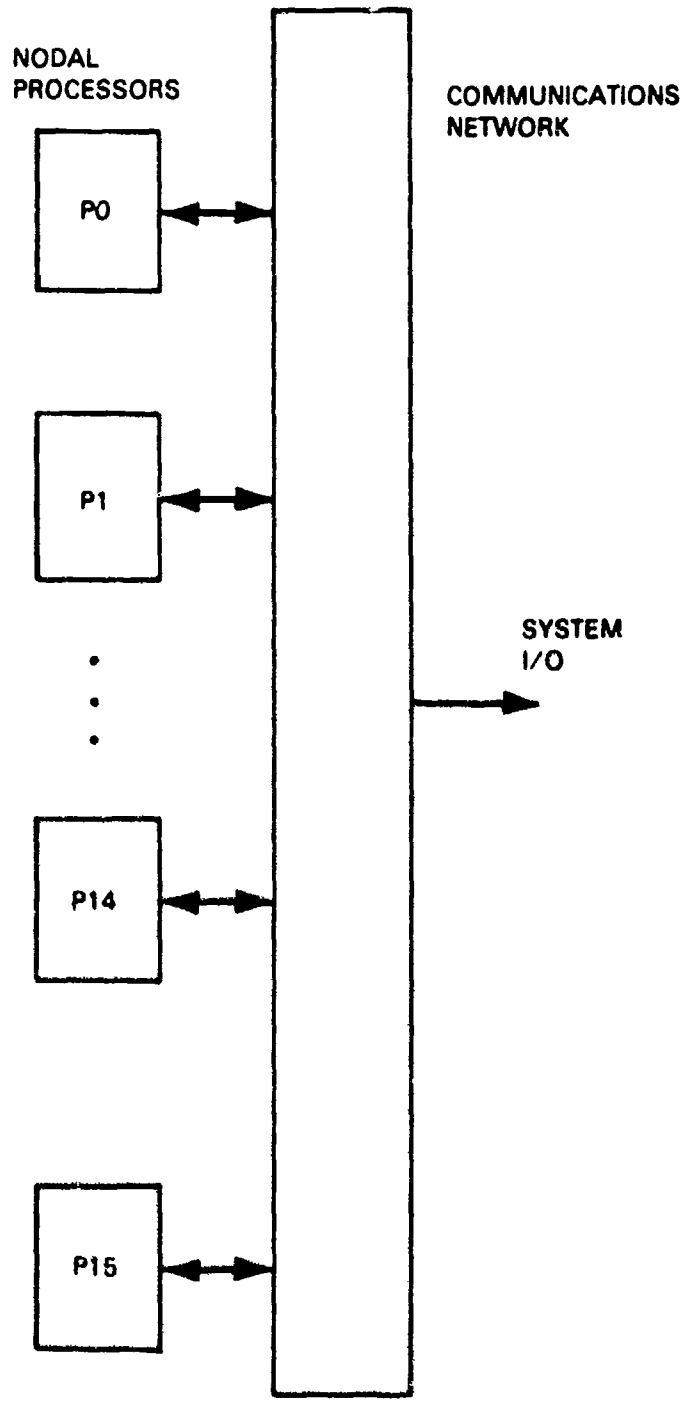


Figure 4-1. A high-level block diagram for a multiprocessor system.

the instruction set is a particularly powerful one tuned to the image processing application, the number of instructions/pixel required to implement any particular algorithm will be relatively small. On the other hand a reduced-instruction-set architecture will require a relatively large number of instructions/pixel to do the same computation.)

4.1 BUTTERFLY INTERCONNECTION NETWORK

A butterfly interconnection network is useful for providing communication between N processors, where N is equal to two raised to some integer power. In contrast to a crossbar switching system whose complexity grows as N^2, the butterfly interconnection network has a complexity that grows only as $(N/2)\log_2 N$. An example of a butterfly interconnection network is shown in Figure 4-2 for N = 16. The nodal processors are labeled P0 through P15, but for convenience of illustration, the outputs of the nodal processors are found on the left side of the figure while the inputs to the processors are found on the right side of the figure.

Although the butterfly network does not possess the complete connectivity of the crossbar switch, it nevertheless supports several important classes of simultaneous communications between pairs of nodal processors. Our experience has shown that this type of network can support the communication necessary for the efficient multiprocessor implementation of important signal processing operations such as fast Fourier transforms, convolutions, and processing pipelines. For image processing applications, efficient use of a multiprocessor system can be made in many cases simply by partitioning the image data so that each nodal processor can work independently on its own sub-images with the butterfly network providing any communications required by the nodal processors working on neighboring sub-images.

The butterfly interconnection network, shown in Figure 4-2, can be used to support up to sixteen parallel communications between sixteen pairs of nodal processor outputs and inputs. (In general, a butterfly network can support up to N parallel communications, where N is the number of processors.) In addition, it will support a broadcast mode where any single nodal processor output can transmit messages to the inputs of all the nodal processors.

Each circle in Figure 4-2 represents a single butterfly switch and its associated control logic. Each switch can be set to one of four external states (straight, crossed, upper-broadcast, and lower-broadcast), as shown in Figure 4-3. We are currently designing a custom LSI integrated circuit that will implement one two-input, two-output butterfly switch. As currently planned, each chip will be capable of switching two 8-bit-wide data paths as well as the necessary control signals. This will permit the chips to implement butterfly networks of any size up to and including N = 256. Each chip will also have the capability of acting in "slave" mode, using the control signals of another chip (its "master") to determine its external state. This permits two chips to implement a 16-bit-wide switch, three chips to implement a 24-bit-wide switch, and so on.

Figure 4-4 shows the high-level block diagram for the two-input, two-output butterfly switch. The control plane will accept request signals (CreqA and CreqB) from either of the two input ports (labeled PORT A CONTROL and PORT B CONTROL in Figure 4-4) and,

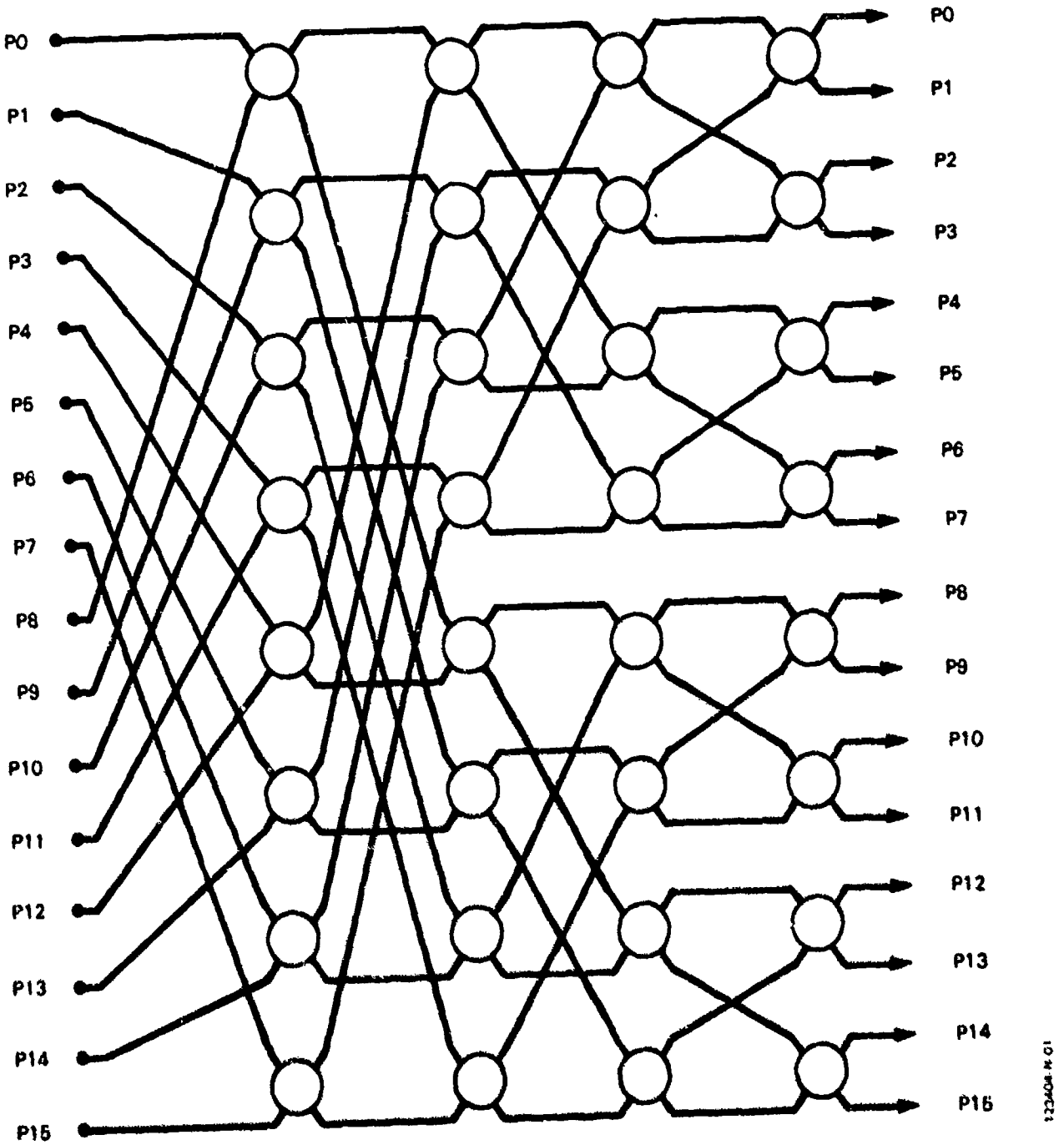


Figure 4-2. A butterfly interconnection network.

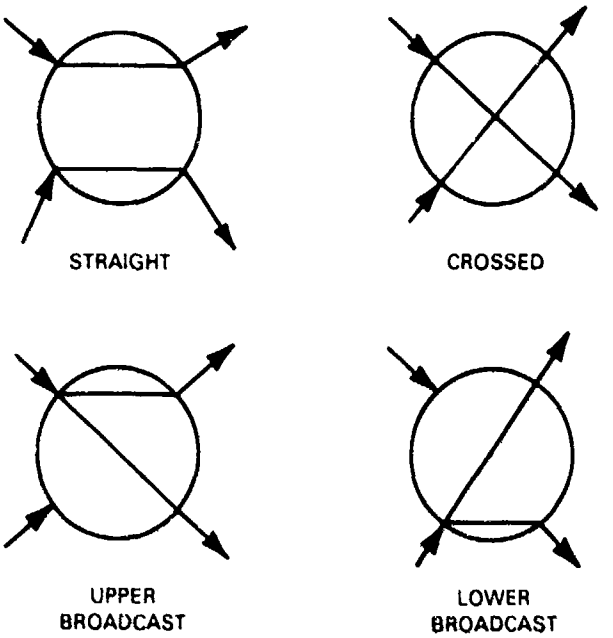


Figure 4-3. External states of a two-input, two-output switching element.

after setting itself to the appropriate external state, will route the request to the next switch in the network. When the requested connection has been established, an acknowledgment signal is propagated back through the network using the CackA, CackB, CackC, and CackD lines. The data plane is used to switch 8-bit-wide data (DAA, DAB, DAC, and DAD lines) as well as data strobe signals (DstbA, DstbB, DstbC, and DstbD) and data acknowledgment signals (DackA, DackB, DackC, and DackD).

The tagA and tagB lines are wired to one of the bits of the DAA and DAB signals, respectively. The particular bit selected depends on the rank of the butterfly network in which the switch lies. During the initialization of a connection, the DAA or DAB signal is used to carry destination addressing information. During this period, the corresponding tag signal indicates which switching mode is being implicitly requested by the sender (straight or crossed). The reset signal can be used to reset the switch to a cleared initial state and the priority signal (pr) determines which request is to be serviced first in cases of contention. Not shown in Figure 4-4 are signals used to request a broadcast mode configuration.

The control plane generates two bits of information (labeled x and y in Figure 4-4) that are used to set the switch into one of its four external states. Figure 4-5 shows how each bit of the DAA and DAB inputs is switched to the DAC and DAD outputs. The control plane is designed as an asynchronous finite-state machine that uses an NMOS programmable logic array (PLA) and a static NMOS register. A second PLA is used to select between the internal (on-chip) x and y control signals or externally generated x and y control signals which enable the switch's data plane to be used in "slave" mode.

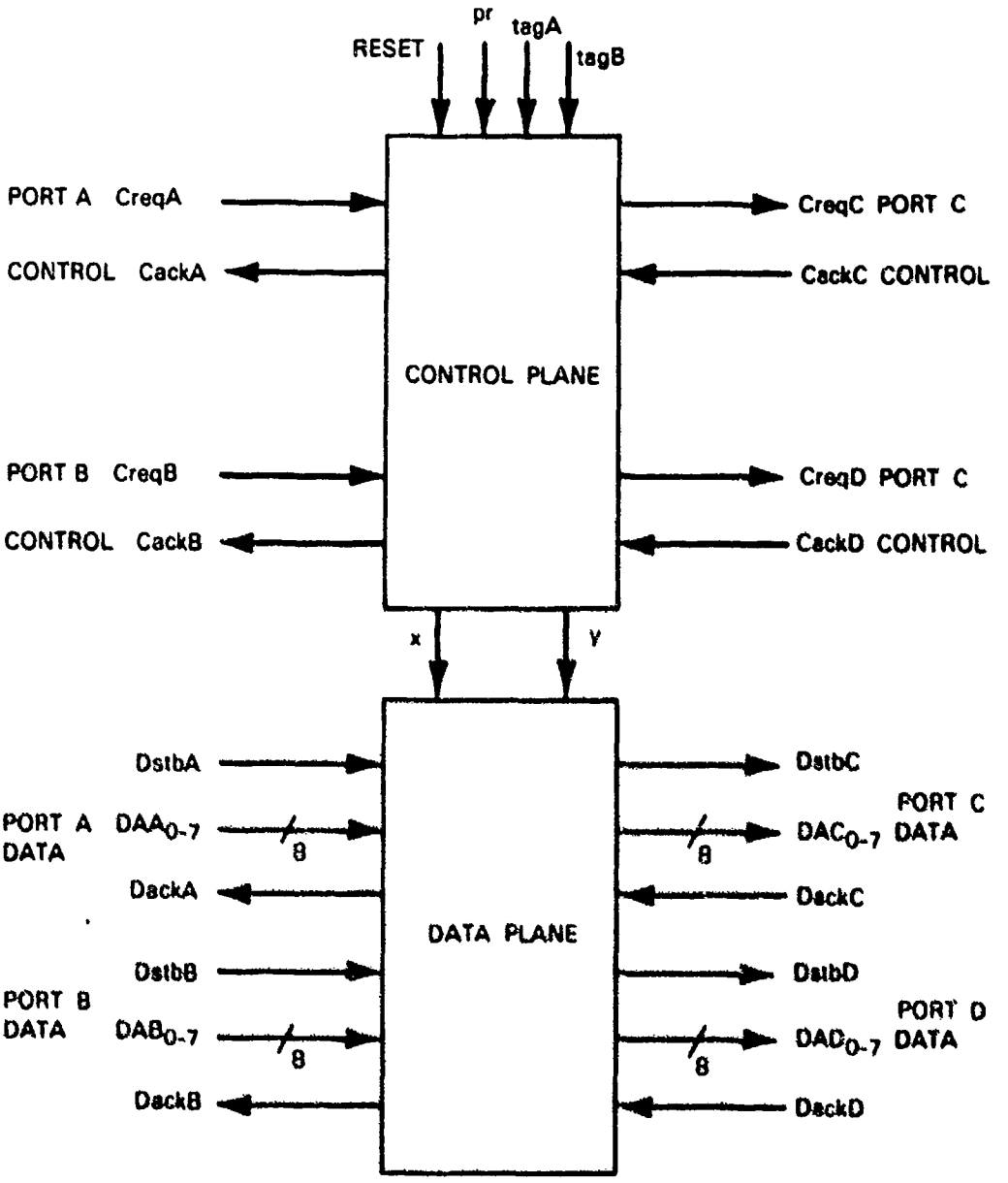
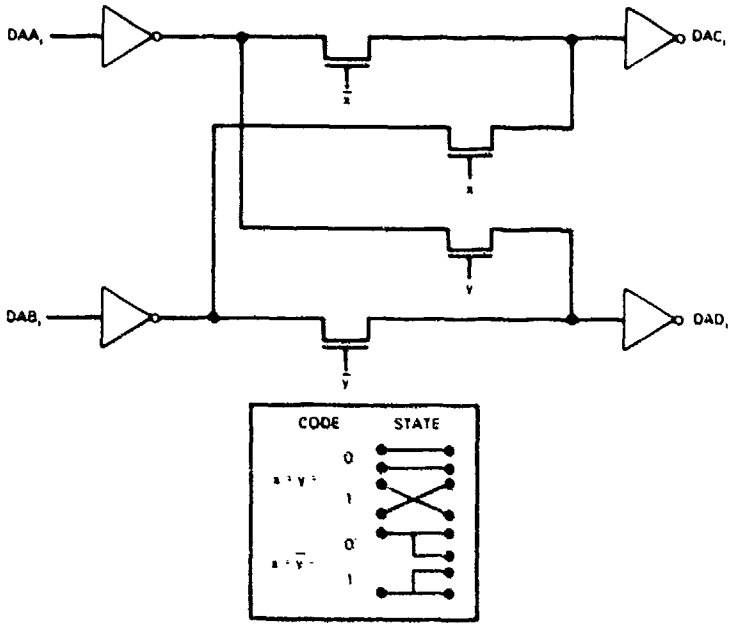


Figure 4-4. Block diagram of the switching element.



129674-N

Figure 4-5. K-chip butterfly switching element (shown for the i^{th} data bit).

When a processor wishes to send a block of data to another processor, it first puts the appropriate addressing information on its data output lines (these lines lead to a DAA or DAB input of a switch in the first rank of the butterfly network) and then raises its request line. The request propagates through the network as each switch uses the addressing information to set itself to the appropriate state. When the receiving processor sees the request line go high, it responds by raising its acknowledgment line. The acknowledgment signal is routed back through the network to the sending processor. When the sending processor sees the acknowledgment line go high, it knows that the connection has been established and that it can begin transmitting data.

To actually transmit the data, the sending processor places the data value on its data output lines and then raises its data strobe signal. The data strobe signal tells the receiving processor that the data lines may now be read. The sending processor later lowers the data strobe line, places the next piece of data on its data output lines, and then raises the data strobe line again, thus repeating the process. The data acknowledgment line may be used after every piece of data to acknowledge receipt by the receiving processor, or it may be used after the entire block of data has been received. The processor-level protocol will determine which procedure is to be used; the switch design will support both protocols.

5. NODAL PROCESSOR ARCHITECTURE

In this section, we shall discuss the architecture of the nodal processor, sixteen of which will comprise the multiprocessor system. The objective is to advocate a number of architectural features that we feel will help achieve the dual goals of high performance and ease of programming. The perspective is one of a user who is interested in developing and refining image processing algorithms with a minimum of effort and having them execute very rapidly.

As a basic ground rule, we have tried to separate architectural issues from implementational issues. Some architectural principles may ultimately have to be comprised in the interest of efficient and timely hardware implementation. However, with the advent of VLSI and VHSIC technology, implementation techniques will doubtless change once again. The development of an image processing architecture may influence future chip sets, thus permitting the implementation of architectural features that may have to be comprised with today's technology. Therefore, our objective thus far has been to develop the best architecture possible without many preconceptions about implementation details.

One approach to building a fast processor is to use the fastest available logic family for the implementation. For a fixed architecture the basic machine cycle time will determine the speed of execution. There are mitigating concerns, of course. Fast logic in general implies more power consumption, more demanding board layout, lower-level integration, and less reliability. Thus, it will probably be necessary to compromise between raw logic speed and other measures such as integration level or power consumption.

5.1 ARCHITECTURAL REQUIREMENTS FOR A NODAL PROCESSOR

Based on the premise that the nodal processors in a multiprocessor image processing system will have to be fast computers in their own right which are capable of handling large arrays, we can outline some of the architectural requirements for the nodal processors. Traditional high-speed array processor architectures have been very "horizontal," using techniques such as separate program and data memories, separate hardware for address computation and index register manipulation, and pipelined fetch, decode, and execution of instructions. Typically, however, commercially available array processors have proven difficult (and thus expensive) for users to program, and they have been primarily used as pre-programmed function boxes for host computers. Undoubtedly many of these architectural features will be used in the nodal processor architecture, provided that they do not impair the ease with which the machine can be programmed.

To permit simple, straightforward programming, the nodal processor architecture should efficiently support a high-level language. In addition, the machine language itself should be relatively high-level. It is important to isolate the programmer as much as possible from the particular implementation of the nodal processor. The programmer should be concerned with specifying operations and data objects to be used as operands and not with the details of address computation or index register manipulation.

Since the nodal processors will be handling large arrays of image data, the accessing of array elements must be very efficient. Similarly, it is important that the processor hardware provide a mechanism for controlling program loops that accrues very little overhead. To

encourage structured, top-down programming discipline the processor hardware must support very efficient subroutine call and return linkages. An assembly language programmer should be able to transfer control to a subroutine and provide a list of arguments in a single instruction. Local and temporary variables used inside a subroutine should be allocated dynamically to permit re-entrant (recursive) subroutines and efficient interrupt handling.

The machine-level instruction set should be flexible so that it can be tuned for different applications. In particular, it should provide some simple array handling instructions such as clearing an array or adding two arrays. A set of several array instructions could relieve the programmer of a significant amount of detailed code generation in an image processing application and allow him/her to concentrate on the important parts of the program.

It would be desirable if machine-level instructions and subroutine calls had a common format from the programmer's viewpoint. This would permit the implementation of new machine-level instructions first as subroutines for testing and evaluation of their utility.

The nodal processor architecture should make some provision for including special-purpose computational devices tailored for specific applications. For example, a high-speed multiplier/accumulator could be considered a "special-purpose" device, though for image processing and multi-dimensional signal processing applications it is a requirement. Other special-purpose devices might include an FFT butterfly box, a 16-point FFT, or a CORDIC rotation element. A special-purpose processor may also be necessary to facilitate I/O with the outside world as well as the interconnection network. The important point is to allow sufficient flexibility to permit the nodal processors to be retrofitted with special-purpose components to increase computational throughput for a particular application.

5.2 MACHINE-LEVEL INSTRUCTIONS

In this section we shall outline most of the machine-level instructions which are envisioned as being necessary for the nodal processor to implement. There are two major subsections, one dealing with arithmetic and logic operations to be performed on data objects and the other dealing with instructions used to control the program flow. A detailed discussion of what is a data object and how it is accessed is deferred until later.

5.2.1 Arithmetic and Logic Operations

In many cases execution of a machine instruction will take two source operands, perform an operation on them, and store the result in a designated destination (or equivalently set the destination operand equal to the result). Thus, three data objects need to be specified in a typical arithmetic/logic instruction. Other instructions have a single source operand and a destination, requiring the specification of only two data objects. On some occasions the destination will be one of the source operands, such as in the case of incrementing a variable. On other occasions the result of the operation need only be stored temporarily because it will be an operand for the next instruction or an instruction to be executed shortly.

Some computer architectures use one of the source addresses as the destination address at all times, while more primitive architectures use an accumulator register as one source as well as the destination. However, programs written for these architectures usually require a

significant number of LOAD, STORE, or MOVE instructions which simply transfer data without doing any useful computation. A three-address architecture should alleviate much of this overhead, resulting in fewer instructions needed to accomplish a particular computation. The use of three-address instructions is consistent with the philosophy of a horizontal architecture, permitting the source and destination addresses to be computed in parallel.

There is a line of reasoning which indicates that instructions should be permitted to have an arbitrary (within reason) number of source operands and destinations. This would permit the user to define super-instructions (or macro-instructions) tailored to particular applications. It would also permit consistency of form for instructions and subroutine calls, allowing one to emulate super-instructions easily. The array instructions discussed below could be considered as a set of super-instructions.

Below is a list of more or less traditional arithmetic/logic operations implemented in some fashion on most conventional computers. Given their widespread utility it makes sense for the nodal processor to provide them as well. At this point the list should probably not be regarded as being complete.

ARITHMETIC OPERATIONS

Instructions with two source operands and one destination.

ADD - Add the two source operands and set the destination equal to the sum.

SUB - Subtract the second source operand from the first source operand and set the destination equal to the difference.

MUL - Multiply the two source operands and set the destination equal to the product.

DIV - Divide the first source operand by the second source operand and set the destination equal to the quotient.

Instructions with two source operands and no destination.

COMPAR - Compare. Subtract the second source operand from the first source operand and use the difference to set the condition flags. Do not store the difference anywhere.

Instructions with one source operand and one destination.

NEG - Negate the source operand and set the destination equal to the result.

INC - Increment the source operand by one and set the destination equal to the result.

DEC - Decrement the source operand by one and set the destination equal to the result.

COPY - Set the destination equal to the source operand.

ABS - Set the destination equal to the absolute value of the source operand.

Instructions with no source operands and one destination.

ZERO - Set the destination equal to the value zero.

ONE - Set the destination equal to the value one.

LOGICAL INSTRUCTIONS

Instructions with two source operands and one destination.

AND - Take the bit-by-bit logical "and" of the two source operands and set the destination equal to the result.

OR - Take the bit-by-bit logical "or" of the two source operands and set the destination equal to the result.

XOR - Take the bit-by-bit "exclusive-or" of the two source operands and set the destination equal to the result.

NAND - Take the logical complement of the logical "and" of the two source operands and set the destination equal to the result.

NOR - Take the logical complement of the logical "or" of the two source operands and set the destination equal to the result.

NXOR - Take the logical complement of the "exclusive-or" of the two source operands and set the destination equal to the result.

BIS - Bit Set. Functionally identical to *OR*.

BIC - Bit Clear. Take the logical complement of the first source operand and "and" it with the second source operand. Set the destination equal to the result. This will clear any bits in the second operand corresponding to logical "ones" in the first operand, leaving the remaining bits unchanged.

LSHL - Long Shift Left. Shift the second source operand to the left by the number of bits specified by the first source operand and set the destination equal to the result.

LSHR - Long Shift Right. Shift the second source operand to the right by the number of bits specified by the first source operand and set the destination equal to the result.

LROTL - Long Rotate Left. Rotate the second source operand to the left by the number of bits specified by the first source operand and set the destination equal to the result.

LROTR - Long Rotate Right. Rotate the second source operand to the right by the number of bits specified by the first source operand and set the destination equal to the result.

Instructions with one source operand and one destination.

SHL - Shift Left. Shift the source operand one bit to the left and set the destination equal to the result.

SHR - Shift Right. Shift the source operand one bit to the right and set the destination equal to the result.

ROTL - Rotate Left. Rotate the source operand one bit to the left and set the destination equal to the result.

ROTR - Rotate Right. Rotate the source operand one bit to the right and set the destination equal to the result.

COMPL - Complement. Take the logical complement of the source operand and set the destination equal to the result.

Instructions with no source operands and one destination.

CLEAR - Clear all the bits of the destination.

SET - Set all the bits of the destination to logical "one."

It may be useful to consider adding other instructions such as incrementing or decrementing a source operand by a small number (say 15 or less), doubling the value of a source operand, or halving the value of a source operand.

In many instances the destination will be identical to one of the source operands. The programmer can specify this simply by using the same data object for both the source and destination.

5.2.2 Array Instructions

For image processing as well as other array processing applications, it seems prudent to provide the nodal processor architecture with a small but powerful set of instructions for performing array operations. These operations would include element-by-element addition, subtraction, multiplication, and division of two arrays. As with the scalar arithmetic operations discussed in the previous subsection, the array instructions would have two source operands and a destination, with provisions for the destination being equivalent to one of the source operands. Other operations, such as the absolute value array operation listed below, take a single source array as well as a destination that may or may not be the same as the source array. If they seem useful, it should be possible to incorporate array logical operations similar to the scalar logical operations listed earlier.

BASIC ARRAY ARITHMETIC INSTRUCTIONS

AADD - Add the corresponding elements of the two source arrays and store the sum in the corresponding elements of the destination array.

ASUB - Subtract the elements of the second source array from the corresponding elements of the first source array and store the difference in the corresponding elements of the destination.

AMUL - Multiply the corresponding elements of the two source arrays and store the product in the corresponding elements of the destination array.

ADIV - Divide an element of the first source array by the corresponding element of the second array and store the quotient in the corresponding element of the destination array.

AABS - Store the absolute value of the elements of the source array in the corresponding elements of the destination array.

The array instructions listed above should work on arrays of any dimensionality and size as long as they are consistent with those of the other arrays used in the same instruction. Other array instructions such as those listed below involve the computation of a scalar result from one or more source arrays.

OTHER ARRAY PROCESSING INSTRUCTIONS

INNER - Inner product. Multiply the corresponding elements of the two source arrays and sum the resulting products. Store the sum in the destination.

ASUM - Add all the elements of the source array and place the resulting sum in the destination.

ASUM2 - Add the squares of all the elements of the source array and place the resulting sum in the destination.

AMAX - Store the value of the largest element in the source array in the destination.

AMIN - Store the value of the smallest element in the source array in the destination.

In the *AMAX* and *AMIN* instructions, it may be worth considering storing the indices of the maximum and minimum array values somewhere as well as the values themselves. As with the basic array instructions, these instructions should be able to handle arrays of arbitrary dimensionality and size.

For image processing applications, it will be very useful to be able to select a subarray out of an array to use as the source operand in an array instruction.

Other array operations, such as 2-D convolution, would be useful as well, but because they involve source and destination arrays of varying sizes, they may be more difficult to implement as machine-level instructions. Such operations are good candidates for becoming "instructions" that are initially emulated by subroutines.

It would also be very useful to provide a mechanism, either in hardware, software, or some combination of the two, that would allow a programmer to specify a complicated operation, or set of operations, to be performed on individual array elements and then invoke the complicated operation as an array "instruction" to be performed over all the array elements. This would relieve the programmer of the necessity of writing code that explicitly loops over all elements in the source arrays. The interpretive array processing language APL provides this facility and it has proven very useful for the efficient development of image processing programs.

5.2.3 User-Defined Instructions

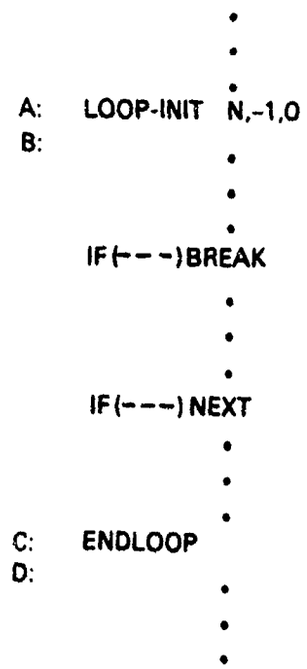
Earlier we alluded to providing users with the option of defining new instructions tailored to specific applications to be added to the instruction set of the nodal processor. Presumably this could be accomplished by providing enough unused op-codes in the instruction set and a mechanism for allowing users to load their own micro-code. Before going to the labor-intensive step of writing microcode, users will no doubt want to emulate their new instructions in some fashion so they can test their usefulness in the context of some applications software. This argues for providing a subroutine calling mechanism which has the same format as an ordinary instruction.

Users may want to micro-code additional instructions not only to provide complicated arithmetic operations but also to control special-purpose processors integrated into the nodal processor for their particular application. For example, if a CORDIC rotation element is added to the nodal processor, it will be necessary to provide machine-level instructions to send data to it, control it, and get answers from it. This may imply a need for control paths and data paths in the nodal processor design in anticipation of various special-purpose processors.

5.2.4 Controlling Program Flow

In this section, we shall discuss instructions that are used to control the flow of programs written for a nodal processor. There are essentially three classes of these instructions: those used for simply transferring control, those used for controlling loops, and those used in conjunction with subroutine calls.

JUMP and *BRANCH* are instructions used to transfer control from one part of a program to another. A *JUMP* instruction sets the program counter equal to the instruction address contained in the *JUMP* instruction. (Usually the program counter is simply



LOOP CONTROL STACKS

LOOP COUNT	LOOP INCREMENT	LOOP LIMIT	BREAK ADDR	NEXT ITERATION ADDR	TOP OF LOOP
N	-1	0	D	C	B
•	•	•	•	•	•
•	•	•	•	•	•
•	•	•	•	•	•

129675-N

Figure 5-1. Organization of loop control information.

incremented to fetch the next instruction in sequence.) A *BRANCH* instruction transfers control to an instruction located at some address relative to the *BRANCH* instruction by adding the contents of the program counter to the offset contained in the *BRANCH* instruction. Thus, the *BRANCH* instruction acts like a relative jump.

If instruction words vary in length, then separate *BRANCH* and *JUMP* instructions are useful because most transfers of control are local, and the *BRANCH* instruction will require fewer bits to represent the offset than the equivalent *JUMP* instruction will require to represent an absolute instruction address. On the other hand, if all instructions are the same length, that length must be long enough to incorporate *JUMPs* to any absolute instruction address, and the necessity of having a separate *BRANCH* instruction is eliminated.

In practice the programmer would not use the *JUMP* or *BRANCH* instruction directly, but would rely on an assembler or compiler to translate *IF-THEN-ELSE* constructs into the appropriate tests and conditional jumps. A jump, of course, is directly analogous to a *GOTO* statement in high-level language, the use of which is frowned upon by the advocates of structured, top-down programming. In addition, the programmer must be prevented from *JUMP*ing out of a loop directly for reasons that we shall discuss later. This is easily done if an assembler or compiler stands between the programmer and the machine-language.

Loops are important constructs in many types of programming, but especially in signal processing applications programming. It is important to develop an architecture that efficiently supports the execution of instruction loops. From the programmer's point of view, most loops can be written using the following loop control instructions: *LOOP-INIT*, *LOOP-TEST*, *ENDLOOP*, *REPEAT-LOOP*, *BREAK*, and *NEXT*. Since loops are always perfectly nested, auxiliary variables used to control the execution of the loop, such as the loop counter, loop increment, loop limit, break address, top-of-loop address, and next-iteration address, can be stored on a set of parallel stacks (see Figure 5-1). New values of the variables are pushed onto these loop control stacks when a loop is entered, and they are popped off when the loop has been satisfied.

The *LOOP-INIT* instruction indicates the beginning of a loop and the addresses of the data objects whose values are to be used as the initial value of the loop counter, the loop increment, and the loop limit. (These values are inherently integers.) Note that infinite loops, to be exited by a conditional *BREAK* instruction, can be formed by specifying a loop increment of value zero. The *LOOP-INIT* instruction would include a test to see if the loop counter already exceeded the loop limit. (Here "exceeded" must be used in a sense consistent with the sign of the loop increment.) If it does, control would pass immediately to the first instruction after the loop (the break address). This defensive programming tactic may save effort during debugging and execution.

The *ENDLOOP* instruction indicates the physical end of a loop. When an *ENDLOOP* instruction is executed, the loop counter is incremented by the loop increment, then it is tested against the loop limit. If more iterations of the loop are to be executed, control is transferred to the top-of-loop address. If not, control is transferred to the first instruction after the *ENDLOOP* instruction and the loop control stacks are popped. (This is equivalent to executing a *BREAK* instruction.)

For infinite loops to be exited by a conditional *BREAK* instruction, the *REPEAT-LOOP* instruction can be used instead of the *ENDLOOP* instruction. Execution of the *REPEAT-LOOP* instruction simply causes a *BRANCH* or *JUMP* to the top-of-loop address.

Because high-speed processors tend to be pipelined, there is a disruption of the instruction fetch-decode-execute pipeline when control is transferred to other than the next instruction in sequence. It may be possible to circumvent this problem at the bottom of loops by realizing that the instruction to be fetched will generally be the one whose address is stored in the top-of-loop register rather than the next address in sequence.

The *LOOP-INIT* and *ENDLOOP* instructions will be adequate for most of the loops in image processing applications programs. However, in some cases the *LOOP-TEST* instruction, which may occur anywhere in the loop, may be useful. The *LOOP-TEST* instruction is basically a conditional *BREAK* instruction. The loop counter is tested against the loop limit (taking the sign of the loop increment into account) and, if the loop is to be exited, control is transferred to the first instruction after the *ENDLOOP* instruction and the loop control stacks are popped.

The *BREAK* instruction alluded to earlier can be used to bail out of a loop at any time. It can be executed conditionally based on satisfaction of the loop variables (which is what essentially happens in the *LOOP-INIT, LOOP-TEST,* and *ENDLOOP* instructions) or based on other conditions such as the results of arithmetic/logic operations. When *BREAK* is executed, control is transferred to the break address, usually the first instruction after the *ENDLOOP* or *REPEAT-LOOP* instruction, and the loop control stacks are popped.

The *NEXT* instruction is used to stop the current iteration of the loop and immediately begin the next iteration. Control is transferred to the instruction at the next-iteration address, which is typically the *ENDLOOP* address. For loops terminated by the *REPEAT-LOOP* instruction, the next-iteration address is the same as the top-of-loop address.

As discussed above, the *LOOP-INIT* instruction must initialize several parameters used in controlling the loop. Instruction addresses such as the break address, next-iteration, and the top-of-loop address could be encoded in the *LOOP-INIT* instruction as offsets from the address of the *LOOP-INIT* instruction. It should be straightforward for an assembler to deduce and compute these offsets from assembly language code so that the programmer need not be bothered with this task. Other parameters such as the initial value of the loop counter, the loop limit, and the loop increment can be accessed as if they were ordinary data objects.

To prevent the loop control stacks from getting out of synch, programs are not permitted to get out of loops except by using some variant of the *BREAK* instruction. In particular, a program should not *JUMP* or *BRANCH* out of the scope of a loop.

Structured software tends to have many subroutines and consequently many subroutine calls and returns. Thus, it is important to have an efficient subroutine linkage mechanism so that the overhead for short subroutines is not too large.

One approach that has appeared in the computer literature (and has been alluded to earlier in this section) involves making a subroutine call look like an ordinary instruction

invocation. This approach allows the application programmer to develop new "instructions," which are implemented as subroutines, that are tailored to a specific application. The new "instruction" could eventually be implemented in micro-code if warranted by frequency of usage and performance requirements. This approach is similar in many respects to the subroutine linkage mechanisms used in threaded-code architectures. It has the potential advantage of permitting the called routine to be one of a variety of subroutine types that can handle the bookkeeping of the subroutine linkage in different ways. The calling routine simply supplies the "name" of the called routine and a list of data objects to be used as arguments, in the same way that an ordinary instruction supplies the "name" of the operation to be executed (as embodied in the op-code) and a list of data objects to be used as arguments.

A subroutine call can be implemented by having an op-code with a leading 1 to distinguish it from ordinary instructions (whose op-codes are assumed to have a leading 0) followed by an 8- or 10-bit subroutine "name." The subroutine "name" is used to look up the subroutine starting address in a table, and control is transferred there. The subroutine call will contain pointers to the "names" of the data objects to be used as arguments and it may also be desirable for it to contain the number of arguments as well.

The first instruction in a subroutine must be an *ENTER-SUBR* instruction. The *ENTER-SUBR* instruction is responsible for setting up the environment of the called subroutine and stacking the return address back to the calling routine. Naturally, control is returned to the routine by executing a *RETURN* instruction. It must clean up the subroutine's environment, restore the environment of the calling program, and pop the return address off the return stack to the program counter thus transferring control back to the calling program. In some sense it is the opposite of the *ENTER-SUBR* instruction, and consequently different types of *RETURN* instructions can implemented each corresponding to a different type of *ENTER-SUBR* instruction. The calling routine has no knowledge of what kind of *ENTER-SUBR* and *RETURN* instructions lie ahead in the called subroutine.

Bear in mind that the *RETURN* instruction must also clean up the loop control stacks if it is executed within the scope of a loop. This problem could be avoided if it were illegal to issue a *RETURN* instruction within the scope of a loop just as it should be illegal to *JUMP* or *BRANCH* out of a loop. This would force the programmer to get out of loops by using the *BREAK* instruction or its relatives, which would keep the loop control stacks from being incorrectly manipulated.

Providing access to data objects in the calling routine is an important aspect of subroutine linkage. We shall assume that the *CALL* instruction will include the number of arguments to be passed as well as the locations of the "names" of the data objects to be used as the arguments. The *ENTER-SUBR* instruction must therefore access the "names" and store them in locations used inside the subroutine.

Since the calling program may be a subroutine as well, the details of subroutine linkage may be viewed as establishing the environment for the called subroutine within the environment for the calling subroutine and dismantling it when control is returned to the calling subroutine. One mechanism for doing this involves the use of an argument stack. Typically

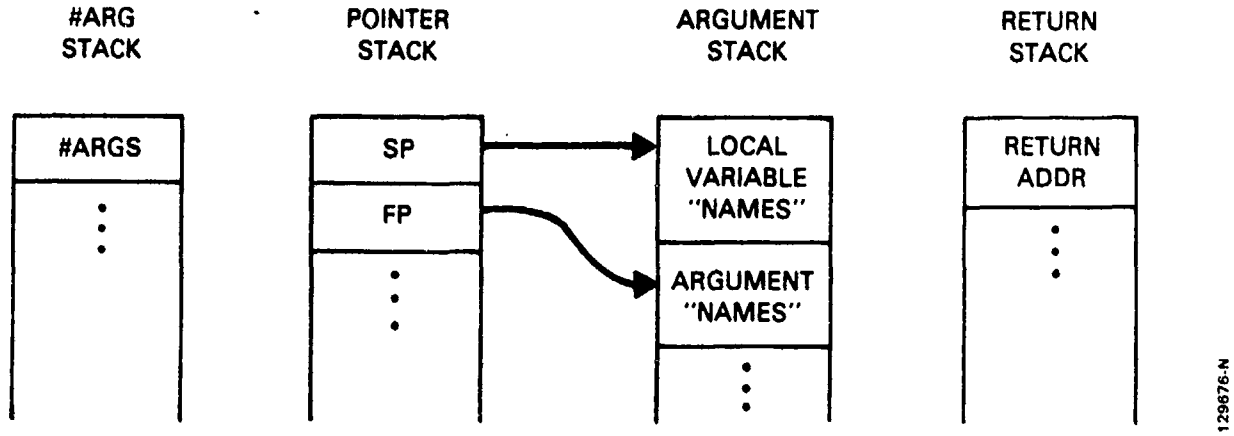


Figure 5-2(a). Four stacks used for subroutine linkage.

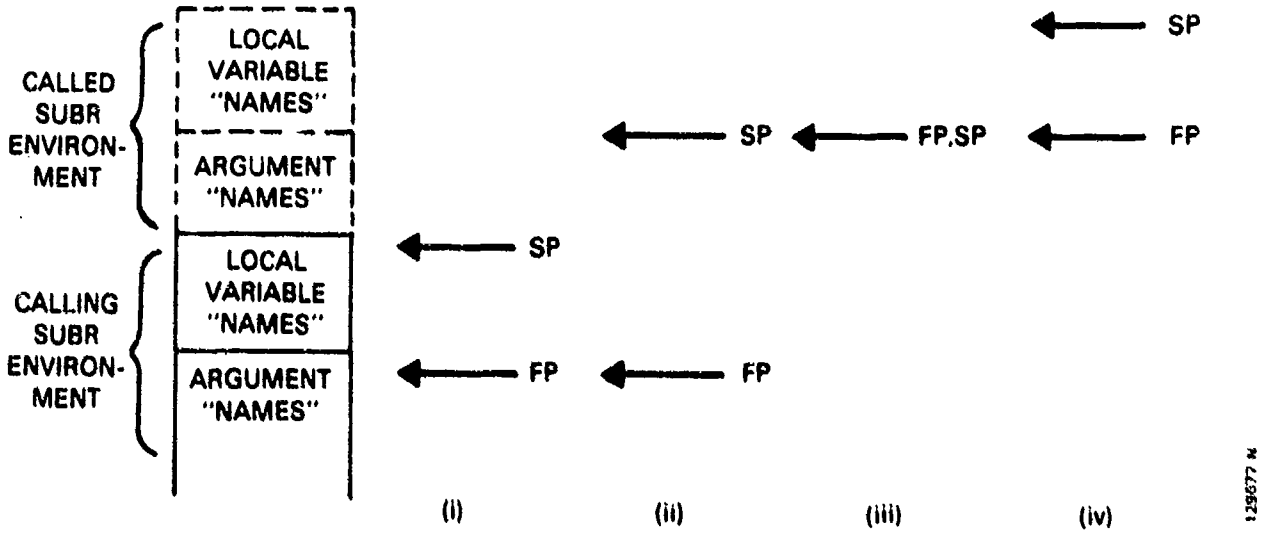


Figure 5-2(b). Evolution of FP and SP (i-iv) during a subroutine call.

three pointers are associated with an argument stack: the top-of-stack pointer (SP), the frame pointer (FP), and the global pointer (GP). The "names" of data objects used in the calling routine are accessed by references to the registers which comprise the argument stack. (The registers of the argument stack contain the data object "names.") For example, (SP) would refer to the register at the top of the stack, 4(FP) would refer to the fourth register above the register pointed to by the frame pointer, -1(FP) would refer to the register below the register pointed to by the frame pointer, and 7(GP) would refer to the seventh register above the register pointed to by the global pointer. (Typically the global pointer points to the register at the bottom of the argument stack.) Another type of "name" is a literal. In this case the data object is simply the scalar value supplied by the appropriate bits of the literal "name."

When the *CALL* instruction is encountered, it contains the "name" of the subroutine to which control will be passed (which gets turned into an instruction address by table-lookup), the number of arguments, and the registers that contain the "names" of the data objects to be used as arguments. Control is passed to the first instruction in the called subroutine, which must be an *ENTER-SUBR* instruction of some type. The *ENTER-SUBR* instruction is responsible for setting up the environment of the called subroutine. For example, it must allocate storage on the argument stack for the "names" of all the local data objects used in the called subroutine. In addition, it must allocate storage in the data memory for scalar values, arrays, records, and other data structures, and set up the appropriate links between the "names" of the local data objects and the places where their values are stored.

One simple method to effect the linkage of argument names is as follows: When control is passed to the *ENTER-SUBR* instruction, the program counter still points to the *CALL* instruction. Using the program counter as a pointer, the *ENTER-SUBR* instruction learns the number of arguments to be passed and the registers containing the "names" of the data objects to be used as arguments. The *ENTER-SUBR* instruction pushes the "names" of the data objects onto the argument stack and pushes the number of arguments onto another stack (the #arg stack). At this point it is convenient to think of the SP and the FP as being the top two registers on yet another stack (the pointer stack). The current value of the SP is now pushed onto the pointer stack, effectively setting the FP equal to the SP. Then the SP is incremented to allocate space for the "names" of the local data objects to be used in the subroutine. The *ENTER-SUBR* instruction then allocates storage in the data memory for the local data objects and makes the link between the "names" of the local data objects and the places where their values are stored. The return address is computed by incrementing the value of the old program counter (so that it points to the instruction after the *CALL* instruction) and pushed onto the return stack. Then the program counter is set to point at the first instruction after the *ENTER-SUBR* instruction and execution continues in the subroutine [see Figures 5-2(a) and (b)].

The *RETURN* instruction essentially reverses the effects of the *ENTER-SUBR* instruction. First, the storage used for the local data objects is deallocated. The pointer stack is popped, effectively restoring the FP of the calling routine and resetting the SP to deallocate the registers used for local data object "names" by the called routine. Then the number of arguments is popped off the #arg stack and subtracted from the SP to restore it to the

value that it had at the time the *CALL* instruction was executed. Finally, the return address is popped off the return address stack into the program counter so that execution resumes with the first instruction after the *CALL* instruction.

5.2.5 Input-Output Instructions

There are two aspects of nodal processor I/O: communication with the outside world and communication with other nodal processors in the multiprocessor system. Here we shall comment only on the latter. In terms of architectural requirements, we want the nodal processor to handle the protocols for I/O as little as possible for two primary reasons. First, we do not want to slow the nodal processor down with bookkeeping tasks related to I/O; and second, we want the flexibility to interface improved interprocessor connection networks as they are developed with a minimum of hardware and software changes to the nodal processors themselves.

These requirements argue for a direct memory access (DMA) capability with a separate I/O processor to handle the necessary communications protocols. The nodal processor should simply be able to request that data be sent to another processor or be received from another processor. In addition, it should be possible to interrupt the nodal processor to inform it that data have been received from another processor.

The detailed requirements for the I/O instructions have not been worked out. However, it is probably important to have them operate in at least two modes. In the first mode an I/O instruction is used to initiate some I/O operation. Then, after perhaps doing some useful computation, the processor executes a *WAIT* instruction which suspends its execution until a condition flag is set by the I/O processor indicating that the I/O operation has been completed. Alternatively, the I/O instruction could enable an interrupt which becomes active when the I/O operation has been completed.

5.3 DATA OBJECTS

An underlying assumption of the nodal processor architecture is that it must support a fairly large data memory (256K to 1M bytes) for image processing applications. For direct addressing, this implies addresses as least 20 bits in length, and perhaps 24 or even 32 bits to provide for future expansion of the memory size or to incorporate any address tagging required by the architecture. With three address instructions, the number of bits needed for specifying source and destination addresses thus ranges from 60 to 96 bits, resulting in very wide instruction words. In order to keep the instructions width down, some architectures make use of address registers. These registers, which contain the addresses of the desired operands, are few enough in number so that they can be specified with a small number of bits in an instruction word. Operands are thus accessed indirectly. However, address registers must be saved and restored during context switches like subroutine calls and returns and interrupt servicing, and this may imply a high level of overhead.

In this section we shall outline the simple concept of using data objects as arguments for the arithmetic/logic instruction. By using data objects we are trying to prevent the applications programmer from getting involved with data addressing. In any one program the number of distinct objects (such as scalar variables and arrays) is relatively small (say

the order of a thousand or so, sometimes much less) compared with the number of possible data addresses. Since programmers are much more productive when thinking and working at a high level, it may make sense for them to refer to and manipulate data objects by "name." The "name" of a data object could be used to look up any pertinent information about the data object (such as its type and current value) in a table.

The address of a location in the data memory could be interpreted as a "name" which permits the value stored in that location to be accessed. As mentioned above, such a "name" would be long, perhaps as long as 32 bits, and would only provide access to the values of scalar data objects. The length of a name could be shortened by assuming that a maximum of 4096 (for example) data objects would occur in any one program. Thus, only 12 bits need to be used to identify any one data object. The 12-bit "name" could be used to access a table containing the address in data memory where information about the data object in question is stored. This would permit data objects to consist of more than one value (such as records and arrays) as well as other ancillary information (such as data type, array length, and array dimensionality). In the paragraphs below we shall discuss one possible mechanism that tries to embody the notion of having programmers regard the number that the processor manipulates as data objects.

As mentioned earlier the "names" of data objects would be contained in an argument stack that would be accessed by a stack pointer (SP), a frame pointer (FP), and a global pointer (GP). For data objects which possess only a single value, the data object "name" can be used to look up the appropriate address which is then used to access the desired value.

In order to insulate the programmer from the nuisance of computing the addresses of array elements, multi-valued data objects such as arrays and records will be accessed in an unconventional manner. The address referenced by the data object "name" will point to the "header" that contains all the information necessary to access data from the "named" array. The "header" will include things like the base address in the data memory where the values are stored, the number of dimensions of the array, the number of storage cells along each dimension, the number of words (or bytes) of data memory used for a single storage cell, the type of values stored in the array, etc. In addition, it can contain index registers as well as the effective address corresponding to the current values of the index registers. This permits a separate, noninterfering set of index registers for accessing each distinct array (see Figure 5-3).

When an array data object is referred to by "name," the value returned by the data memory is the value indicated by the current values of the index register in the "header" of the array. The values of the index registers may be altered by special instructions (e.g., *MODIFY-INDEX*) that will store the values of other data objects into the index registers. This can get a little cumbersome if one is bouncing all over an array, accessing values at random. On the other hand, if one is accessing values from the array in a structured way, it would be useful to incorporate auto-incrementing, auto-decrementing, and zeroing of the index registers when an array "name" is used to access the value of an array element. The data memory must be signaled to indicate the desired alteration to the index registers of

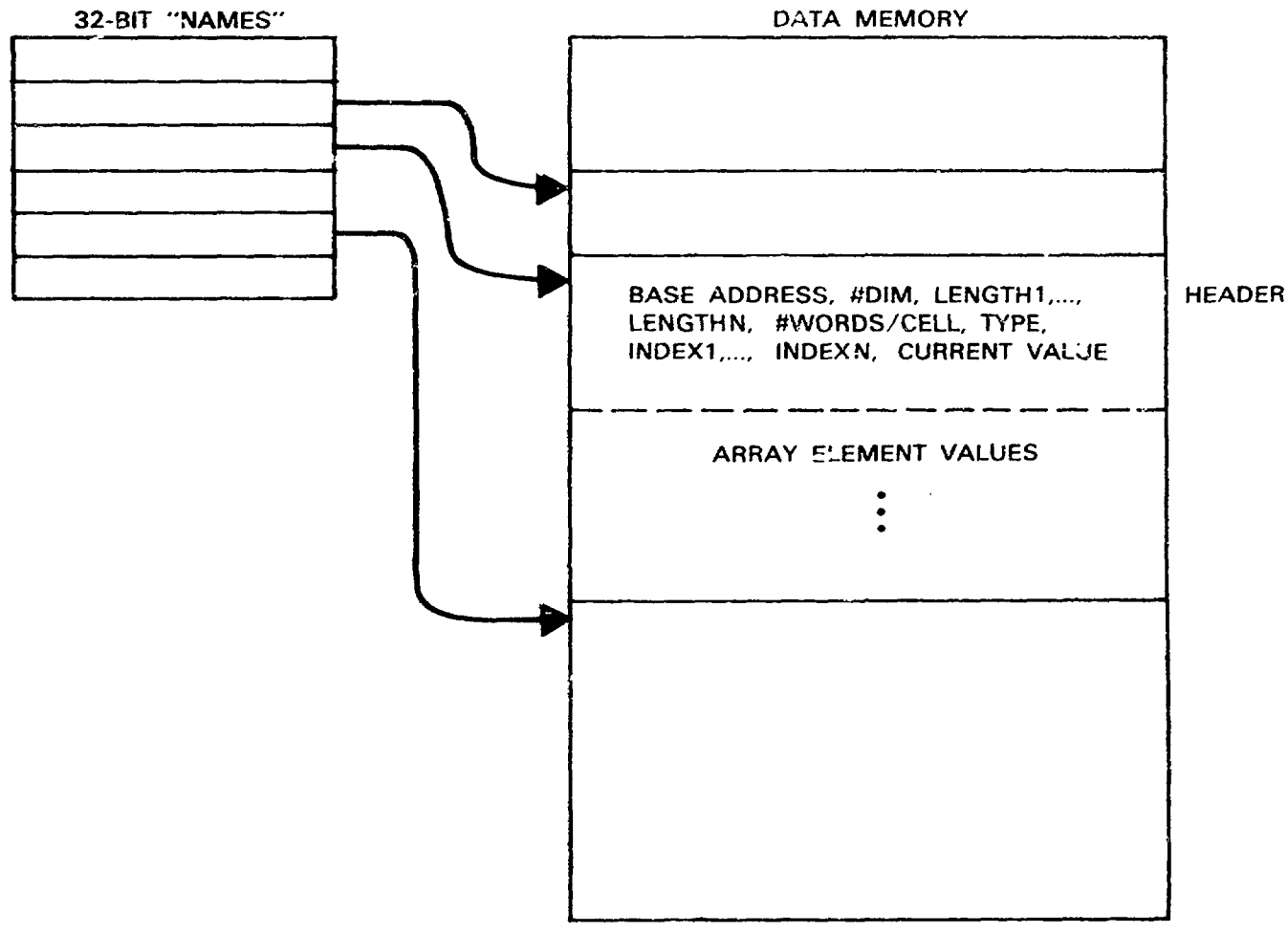


Figure 5-3. A simple scheme for accessing values from array data objects.

the "named" array, the index registers must be changed, and the new effective address must be computed before the "named" array is accessed again.

The simple scheme discussed above illustrates some of the ideas behind treating arrays as data objects rather than a collection of values. However, with the simple scheme, all references to data objects are doubly indirect. An instruction refers to some register in the argument stack such as 4(FP) which contains a data object "name" whose decoded address leads to the place where the value is stored. (For arrays the access ends up being triply indirect because the effective address in the array "header" must be used to retrieve the appropriate value.) We are currently exploring various schemes for reducing the number of levels of indirection required for accessing the values associated with both scalar and array data objects while retaining the essence of object-based addressing.

For many image processing operations it is desirable to access pixels within a small, usually rectangular window centered at some point in the image (see Figure 5-4). To facilitate this, the nodal processor should be capable of specifying the location of elements within an array in terms of a window location and an offset within the window as well as the usual method of specifying an offset from the array origin. The location and size of the window could be stored in the array header in the data memory along with a separate set of index registers to be used to contain the offset with respect to the window location. Any additional information to facilitate the address computation could also be stored in the header.

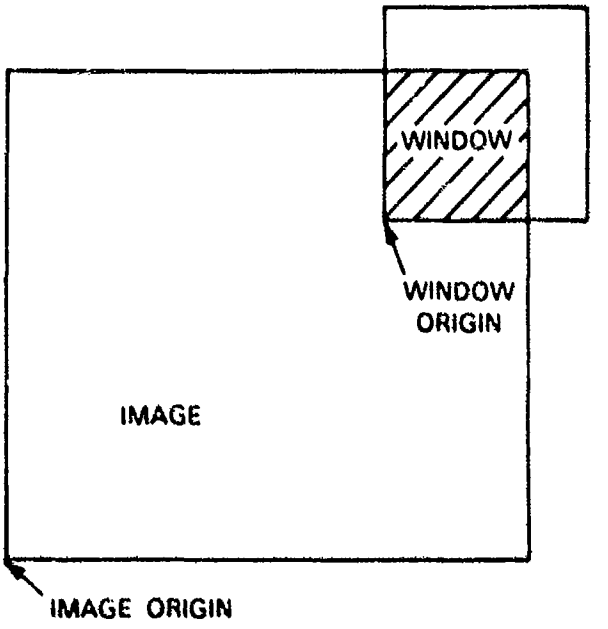


Figure 5-4. Accessing windowed image data.

Special instructions will be needed to set up windows within arrays and manipulate their location. For example, a *DEFINE-WINDOW* instruction would be used to specify the size and initial location of a window. Note that parts of the window may not overlap with the array. This can occur when the center pixel of the window is positioned at a corner of the array. If the data memory is asked to return a value that is within the window but not within the array, it must return an error value. A *MOVE-WINDOW* instruction is needed to alter the location of the window within the array. The arguments of the *MOVE-WINDOW* instruction are used to specify the window location within the array. Some consideration should be given to the possibility of moving the window by one pixel along any dimension without having to resort to a full-blown *MOVE-WINDOW* instruction. This mechanism could be extremely useful in image processing operations that compute the result of a function of the pixel values within a sliding window.

It might also be useful to provide a mechanism to allow multiple windows into the same array. There are some image processing operations that make use of concentric windows (a 3-by-3 inside a 5-by-5, for example) and the availability of multiple windows might make it very easy for a programmer to implement such an operation. In order to implement multiple windows into the same array, it might be easy to create a dummy array with a header whose pointers point to the values of the desired array. The second window would make use of the header of the dummy array. Alternatively, the array header could be generalized to provide for multiple windows, but some mechanism must be provided that indicates which window to use when the array "name" is used as an argument in an instruction.

5.4 DATA MEMORY

It should be somewhat evident from many of the foregoing discussions that the data memory must do much more than store bits. Basically it must be a big memory with some amount of computational capability. It must accept a "name" and return the appropriate value. It must keep track of the parameters necessary to access values in arrays and other data structures. It must respond appropriately to instructions such as *MODIFY-INDEX*, *DEFINE-WINDOW*, and *MOVE-WINDOW*. It must be capable of rapidly doing computations of the form ADDR = BASE + INDEX1 + INDEX2 * OFFSET 2... It must be able to compute three addresses in parallel to support the three-operand instructions efficiently.

There is no reason to presuppose that the data memory must consist of sequential storage locations like the memory of a conventional computer. The rapid retrieval of multi-dimensional array elements may imply a more sophisticated memory organization. For example, a memory could be designed that would fetch an entire image neighborhood rather than a single pixel. Alternatively, a memory system may contain a linked list of array elements. Each element could then provide access to neighboring elements, giving the data memory a local connectivity that could be useful for image processing operations. The total number of bits needed to store a given quantity of data would be larger than that used in a conventional sequential memory, but that is the price for faster potential access to multi-dimensional array elements.

6. IMAGE PROCESSING LABORATORY

During the current reporting period, we have begun to establish a comprehensive laboratory facility for supporting image processing research. Until now our computational support for the research was split between Lincoln Laboratory's Amdahl 470 central computer and a VAX 11/780 within our Division. The COMTAL Vision One/20 display was typically used in a stand-alone mode with tape input. The laboratory facility will now be integrated around the VAX and the COMTAL and provide a flexible testbed for development and testing of new algorithms.

A block diagram of the laboratory facility is shown in Figure 6-1. A number of general and special-purpose image processing modules interact individually or in tandem with a data base of images. These modules read images from the data base and may generate new images that are added to the data base. Typical functions performed are target detection, segmentation, enhancement, edge detection, filtering, and so on. The resulting images can be displayed and manipulated interactively on the COMTAL and transferred via tape to the Laboratory's central computer or other facilities as necessary. A hard-copy camera (see discussion below) that produces high-quality photographs is attached to the COMTAL and can be shared with other display devices in the area.

Special APL and LISP environments will also be present. The APL environment, which itself provides a powerful set of tools for image processing, has access to the data base and provides important additional capability. (Both the target detection and the segmentation algorithms were originally developed in APL.) LISP seems to be useful for both image processing and architecture simulation studies that we expect to carry out in the future.

Our specific activities during this reporting period have been:

(a) *Development of a workable APL facility on the VAX.* This includes development of some basic low-level APL utilities (such as I/O functions), transferring existing APL programs and libraries from the 470 to the VAX, and conversion of these programs as necessary. Most of this work has now been completed.

(b) *Recoding the target detection algorithm.* The original APL research version of the target detection algorithm was brought over to the VAX as part of step (a) above. However, it was desirable to have a compiled version of this algorithm for more production-oriented applications. We have, therefore, rewritten the algorithm in the C language and made it available as an image processing module in the system. The compiled version has more flexibility with respect to parameters, and interfaces well with the UNIX operating system.

(c) *Coding several generic image processing modules.* A number of image processing modules for performing such functions as linear prediction, Karhunen-Loeve transformation and general matrix transformations have been coded and placed in service. These and future modules are designed to take advantage of the unique command-oriented and stream-directed I/O features of the UNIX

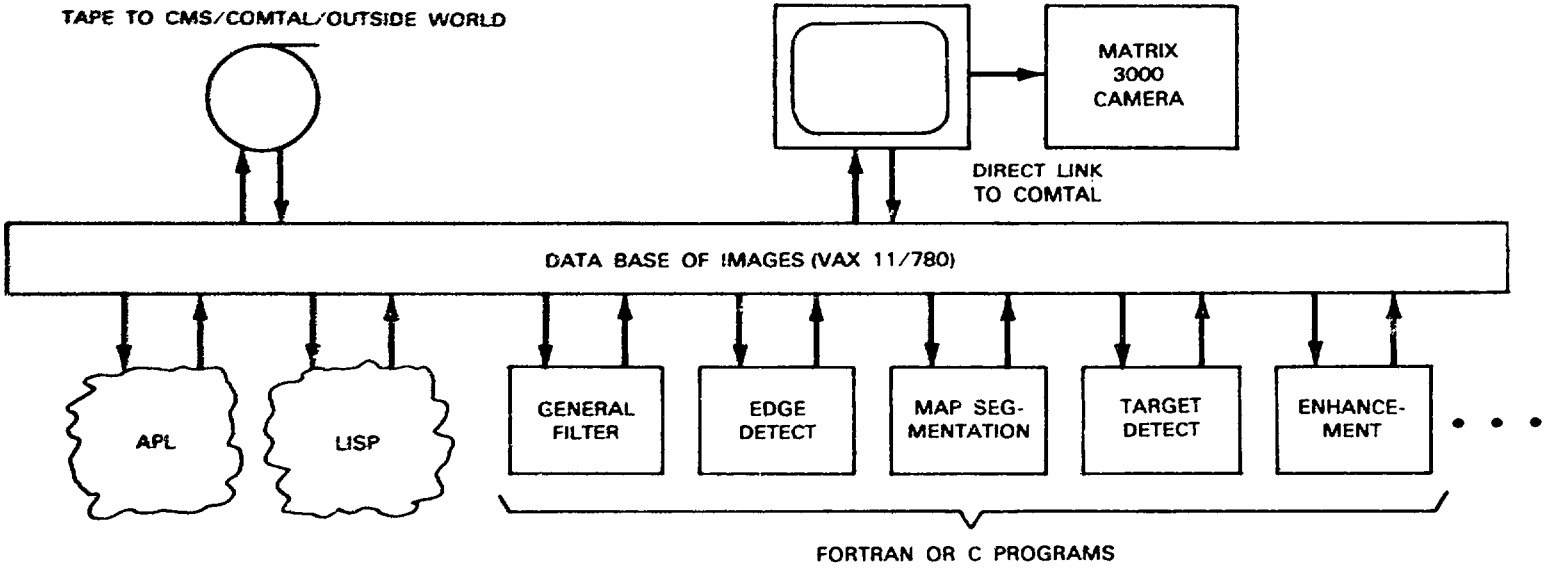


Figure 6-1. Image processing facility.

environment. Through use of these UNIX features, it is possible to interactively apply several operations, represented by distinct program modules, in pipelined fashion without link editing these modules into a larger program or writing intermediate files. We have also coded several more basic programs in C. These perform such functions as input/output of images in standard form, matrix inversion, etc., and will support future software development.

In line with the above activities, we have added to the hardware devices available in the image processing laboratory through the purchase of a MATRIX model 3000 hard-copy recording device and some additional terminals. Figure 6-2 is a picture of the facility showing an APL terminal, COMTAL keyboard and display, and the MATRIX hard-copy device. The hard-copy unit provides high-quality color or black and white photographs (prints or transparencies) in formats ranging from 35 mm to 8 by 10. Either conventional film or Polaroid instant film may be used.

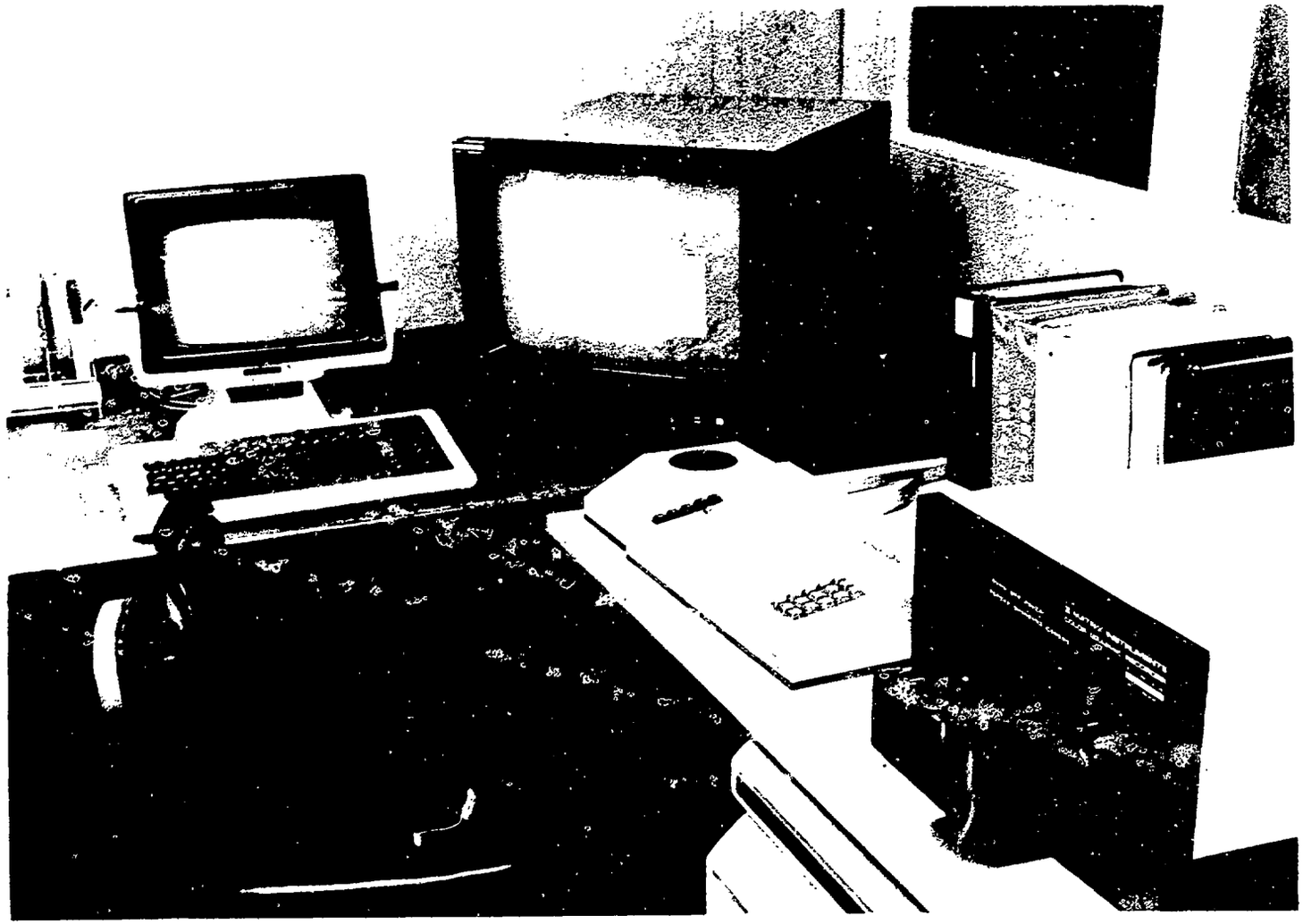


Figure 6-2. Image processing facility user area.

REFERENCES

1. Semiannual Technical Summary, Multi-Dimensional Signal Processing Research Program, Lincoln Laboratory, M.I.T. (30 September 1982), DTIC AD-A126333/4.

2. T.F. Quatieri, "Object Detection by Two-Dimensional Linear Prediction," Technical Report 632, Lincoln Laboratory, M.I.T. (28 January 1983), DTIC AD-A126340/9.

3. C.W. Therrien, Computer Vision, Graphics, and Image Processing **22**, 313 (1983).

4. M. Wax and T. Kailath, "Efficient Inversion of Doubly Block Toeplitz Matrices," Proc. IEEE Intl. Conf. on Acoustics, Speech, and Signal Processing, Boston, 14-16 April 1983.

5. H. Akaike, SIAM J. Appl. Math. **24**, 234 (1973).

6. C.W. Therrien, IEEE Trans. Acoust., Speech, and Signal Processing **ASSP-29**, 454 (1981), DTIC AD-A105406/3.

7. B. Hunt and O. Kubler, "The Theory of Optimal Multi-Spectral Image Restoration" (to be published).

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER ESD-TR-83-035	2. GOVT ACCESSION NO. AD-A135732	3. RECIPIENT'S CATALOG NUMBER
4. TITLE *(and Subtitle)* Multi-Dimensional Signal Processing Research Program		5. TYPE OF REPORT & PERIOD COVERED Semiannual Technical Summary 1 October 1982 — 31 March 1983 6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) Dan E. Dudgeon		8. CONTRACT OR GRANT NUMBER(s) F19628-80-C-0002
9. PERFORMING ORGANIZATION NAME AND ADDRESS Lincoln Laboratory, M.I.T. P.O. Box 73 Lexington, MA 02173-0073		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS Program Element No. 62702F Project No. 4594
11. CONTROLLING OFFICE NAME AND ADDRESS Rome Air Development Center Griffiss AFB, NY 13440		12. REPORT DATE 31 March 1983 13. NUMBER OF PAGES 56
14. MONITORING AGENCY NAME & ADDRESS *(if different from Controlling Office)* Electronic Systems Division Hanscom AFB, MA 01731		15. SECURITY CLASS. *(of this report)* Unclassified 15a. DECLASSIFICATION DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT *(of this Report)*

Approved for public release; distribution unlimited.

17. DISTRIBUTION STATEMENT *(of the abstract entered in Block 20, if different from Report)*

18. SUPPLEMENTARY NOTES

None

19. KEY WORDS *(Continue on reverse side if necessary and identify by block number)*

target detection
adaptive contrast enhancement
image processing architectures

20. ABSTRACT *(Continue on reverse side if necessary and identify by block number)*

This Semiannual Technical Summary covers the period 1 October 1982 through 31 March 1983. It describes the significant results of the Lincoln Laboratory Multi-Dimensional Signal Processing Research Program, sponsored by the Rome Air Development Center, in the areas of multiprocessor architectures for image processing and algorithms for object detection and region classification in aerial reconnaissance imagery.