Design Study of RIP 1: An Image Processor for Robotics

Rafael Bracho
Arthur C. Sanderson

CMU-RI-TR-82-3

The Robotics Institute
Carnegie-Mellon University
Pittsburgh, Pennsylvania 15213

May 1982

This research was partially sponsored by the National Science Foundation under research grant number ECS-7923893.

Table of Contents

List of Figures

List of Tables

Abstract

A multiprocessor system, specially designed for image processing in a Robotics environment, is proposed. Preliminary evaluation of this system suggests it is possible to perform image segmentation and to extract the necessary information to control a robot at speeds approaching real-time, using 8-bits of gray-level information in the raw signal. The proposed *Robotics Image Processor 1 (RIP 1)* can deliver a command to a robot approximately every 100 ms.

The system consists of six processors sharing a high-speed time-multiplexed bus (*RAPID Bus*). Four of them are 16-bit microprocessors (MC68000-based) while the other two are high-performance bit-slice signal processors. The architecture of these signal processors (*RISPs*) is presented in enough detail to be compared against four other architectures. Results show that RISP performs well for typical image processing algorithms and that, due to its power and flexibility, it may be used for other digital signal processing applications such as speech and EEG modeling.

It is shown that RIP 1, due to its key features of speed and programmability, could be used in vision-based robot control or other situations where high speed processing of large amounts of data is needed. Results suggest that it is especially well suited for industrial inspection problems where rapid decisions based on visual information must be made.

1. Introduction

1.1 Image processing in robotics

Traditionally speaking, a robot has often been thought of as a machine with a very strong similarity to human beings. In science fiction, robots attain humanoid characteristics: they walk erect, talk, see, act, sometimes even *feel* the same way humans do. Evaluation of industrial applications now suggests that sensor-based robots offer major advantages in flexibility and precision of robotic systems.

A vision system consists of several parts. In its most simple form, it must have a receptor and an analyzer. In the case of human vision, the receptor would be the eye while the analyzer is the brain (it should be pointed out that some analysis is performed in the retina.) In a robot, the receptor is usually a TV camera while the analyzer is a computer. The field of discrete mathematics that deals with the analysis of the pictures "seen" by the camera, is called *Digital Image Processing*.

In a computer, an image is often represented by an array of numbers. Logically speaking, this two-dimensional array represents the quantized value of the brightness function sent by the camera as it sweeps across the screen. It is obvious that there is a finite number of such values or *pixels* (picture elements). In a typical arrangement, an image is represented by 65536 pixels, organized as a 256 x 256 square array, while each pixel takes one out of 256 possible values (8 bits of information).

Digital image processing deals with these arrays by performing various mathematical manipulations on them, extracting features and describing the image. The results of such manipulations are further analyzed to extract the meaning of what is being seen by the TV camera. This next stage in the analysis is performed by a set of *pattern recognition* algorithms. There is a final stage of computation in robotics: *control*. By now the image has been analyzed in such a way that the computer has extracted the necessary information from it, and it is time to execute an action depending on what the robot "sees". Figure 1-1 shows the algorithmic pipeline of the robot-control problem.

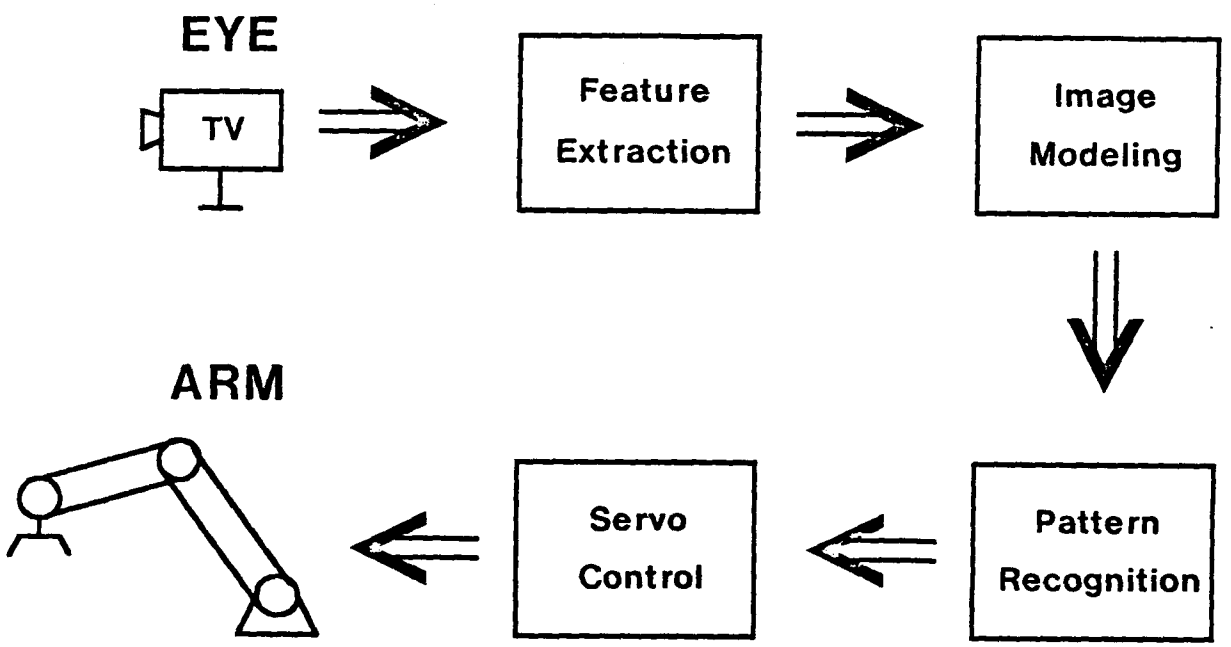

Figure 1-1: Computational tasks in the robot-control problem

1.2 Needs and goals

Now consider the following problem: We want to control a robot so it makes decisions in *real-time*, i.e., so it can manipulate objects, avoid collisions, etc. The problem of doing it in real-time forces us to give commands to the robot approximately every tenth of a second (~ 100 ms), due to typical mechanical time constants of a robot. It is reasonable to assume that we have one-third of those 100 ms for the image preprocessing† while the other two-thirds are consumed by the pattern recognition, interpretation and control tasks.

The problem of analyzing a 256 x 256 image in 33.3 ms is not a trivial one. It means analyzing a pixel every 500 ns. That, in turn, means that the computer has two to eight machine cycles, depending on its technology, to work on a pixel. Of course, there are a number of 'tricks' that can be used to effectively speed up the computation. A number of them are discussed in chapter three.

---

† It is customary to apply the term *image preprocessing* to all the processing done to the 'raw' image; that is, to the array of pixels that come directly from the digitizer

A final requirement is that, due to the unpredictable nature of the robot's environment, we should retain as much useful information as we can. This means, in terms of image preprocessing, that we want to do the analysis on all eight bits per pixel. In the past, there have been systems developed that first reduce the number of bits per pixel and then work on several pixels at a time (see section 2.2).

Our goal is to develop a system with the following characteristics:

1. Capable of performing image preprocessing algorithms in real-time, i.e., able to analyze a complete TV frame every 33.3 ms.

2. Capable of utilizing the relevant information contained in the 8 bits of quantized brightness per pixel.

3. Capable of doing pattern recognition and control fast enough so the robot is given a command every tenth of a second.

2. Image preprocessing algorithms

In this chapter we will outline a few algorithms commonly applied to raw images. It should be noted that this is neither an exhaustive survey, nor is it mathematically rigorous.

A few words about notation are in order:

- We will denote matrices by uppercase boldface letters, such as **U**, **V**, etc. The (i,j)th. element of the **U** matrix is denoted by $u_{i,j}$. The jth. column is denoted by $\mathbf{u}_j$.
- If **U** is a matrix, $\mathbf{U}^T$ is its transpose, $\mathbf{U}^*$ is its complex conjugate and $\mathbf{U}^{-1}$ is its inverse.
- Let $\{u_{i,j}\}$ be a N x N sequence, we define the N^2 x 1 ordered vector $\boldsymbol{u}_r$ as:

$$\boldsymbol{u}_r = [u_{1,1}\ u_{1,2} \cdots u_{1,N}\ u_{2,1} \cdots u_{N,N}]$$

- where the subscript r denotes *row ordering*. We could similarly define the *column ordered* vector $\boldsymbol{u}_c$.

2.1 General model

Many image processing algorithms are simply linear operations performed on the square array of pixels. The output image field is formed from linear combinations of pixels of the input image field. Such operations include filtering and unitary transforms. Other transformations, like thresholding and edge detection, are distinctly nonlinear.

The linear transformations can be represented as matrix operations on an N x N array of pixels. Thus if the image is the (square) array **U**, we have:

$$\mathbf{V} = \mathbf{T}\,\mathbf{U}$$

which may be a filtering, transforming, or some other operation on the original image **U**.

We should note that, in the most general case, the number of operations needed is proportional to N^4, for the case where the output matrix **V** is of the same dimension as **U**. A number of shortcuts can be taken depending on the characteristics of the linear operator **T**. If **T** is separable such that it can be represented in the direct product form

$$\mathbf{T} = \mathbf{T}_r \otimes \mathbf{T}_c$$

then the number of operations is proportional to N^3. It is important to note that, even in this case, for a typical 256 x 256 image, we are talking about 2^{24}, or ~16×10^6 operations.

In the case of unitary transforms, a further reduction might be possible, with the aid of FFT-like algorithms. In this case, the number of operations is proportional to $N^2\log(N)$. For a 256 x 256 image, this represents around 2^{19}, or ~500×10^3 operations.

For the non-linear case, some of the algorithms can also be modeled as an operation on the N x N image, the superposition of an L x L matrix. This (typically smaller) matrix is "rastered" through the image, producing an output array that is a not necessary linear function of the input array. An example of this is thresholding (where L is taken to be 1,) or certain types of edge detection (with typical values of 3 or 5 for L). Some cases of segmentation also lie within this general model.

A characteristic of this type of algorithm is that it usually leaves the dimension unchanged. In other words, the output array of pixels is also N x N. Therefore the number of operations needed is N^2 (from the "raster" effect) times the number of computations needed within the L x L matrix. In the case of $L = 3$, we are talking about 9 multiplications and 8 additions per pixel, in other words, $9N^2$ multiplications and $8N^2$ additions. It is important to note that the next stage of processing would also require a number of operations proportional to N^2; the robot-control problem, as it was previously mentioned, has several stages of computation.

In the next few sections, we will briefly describe some of the algorithms mentioned.

2.2 Brightness Transformations and Thresholding

There are a number of algorithms that deal with transforming the brightness function. In terms of our general model, most of them can be thought of as linear operations on the image matrix. If we represent the picture as a square array **U** of dimension N x N, this means performing linear combinations on the $u_{i,j}$ elements.

Probably the simplest transformation we can perform on the brightness function is to count the number of pixels that have a particular value. If we plot the number of pixels against the brightness, we obtain a histogram which gives us an idea of the distribution of shades within the gray spectrum. Sometimes this histogram is bimodal with one peak close to pure black, probably representing a dark background, and another peak near white, perhaps representing a light object. When the histogram doesn't have this bimodal shape, the image's objects are very similar in color, either to the background or among themselves. A number of algorithms have been developed to make the bimodal shape "appear" in the histogram. Examples include scaling and the use of the Laplacian operator. Since there are N^2 pixels, we need to increment up to 256 counters (for 8 bit pixels) N^2 times. A

problem is that the addressing takes some computation since we are using the value of the pixel as address of the counter to be incremented.

Sometimes the histogram is too "compact", in the sense that the range of values found in the pixels is not 256. This happens when there is a uniform shadow over the image or the contrast is too low (perhaps the iris of a camera was too closed.) In this case, a simple transformation that gices good results, is to scale everything so the full dynamic range is used. This involves searching for the maximum and minimum values. Unless a trivial 0 or 255 is found, we have to test *all* the N^2 pixels in order to find:

$$\alpha = \text{MIN}(u_{i,j}) \quad \forall i,j = 0,1,...,255$$

and

$$\beta = \text{MAX}(u_{i,j}) \quad \forall i,j = 0,1,...,255$$

Then the transformation, which again has to be performed in all N^2 pixels, is:

$$u^*_{i,j} = (u_{i,j} - \alpha)*(M / (\beta - \alpha))$$

where

$$M = 2^n - 1 \quad \text{(n bits per pixel)}$$

A more sophisticated approach is that of histogram *equalization*. In this case it has been shown [1] that if we perform the transformation:

$$s^* = \int_0^s p_u(\xi)\, d\xi$$

where $p_u()$ is the prob. density fcn. of $\{u_{i,j}\}$ and s (s^*) are the brightness values of the pixel before (after) the transformation. The resulting probability density function for $\{v_{i,j}\}$, $p_v()$, is **uniform** in the range [0,1] (it has been assumed that all pixels are normalized to lie in the range [0,1]). For the discrete case, which is the one we are interested in, we have the probability mass function:

$$p_u(\zeta_k) = \frac{n_k}{N^2} \quad 0 \leq \zeta_k \leq 1; \quad k = 0,1,...,M$$

where (M + 1) is the number of gray levels (256), ζ_k is the kth level and n_k is the number of pixels with this gray level in the image; note that the histogram is a plot of $p_u(\zeta_k)$ *vs.* ζ_k. The discrete form of the histogram equalization transformation is, therefore:

$$s_k^* = \frac{1}{N^2}\left[\sum_{j=0}^{k} n_j\right]$$

It is important to note that the previous transformation requires almost $2N^2$ additions and N^2 n-bit shifts (assuming that *n* is a power of two.) This is in addition to the N^2 increments needed to obtain the histogram. Also, the histogram is never completely uniform, although a good approximation is usually obtained.

Weszka, et al. [2] have suggested the use of a Laplacian operator to aid in the threshold detection. In a continuous image field,the Laplacian operator:

$$\nabla^2 F(x,y) = \partial^2 F(x,y) / \partial x \partial y$$

forms the second partial derivative of the image field along the image coordinate directions. As an example, consider an image region in the vicinity of an object in which the luminance increases from a low plateau level in a smooth ramp-like fashion. In the flat regions, the Laplacian is zero and along the ramp it is nearly zero. However a large positive value will occur in the transition region from the low plateau to the ramp and large negative values in the transition from the ramp to the high plateau. A gray level histogram formed of only those points of the original image that lie at coordinates corresponding to very high or low values of the Laplacian tends to be bimodal with a distinctive valley between the peaks. This is due to the fact that the histogram only contains gray levels occurring at the beginning and end of gray level slopes. This approach has been tried, with some success, in cases where the shadows produced by the objects are the source of different gray shades.

Once the histogram has been obtained, we can use it to change our image into a binary image. Remember that, hopefully, we obtained a bimodal histogram. The idea is to select a threshold θ somewhere between the two peaks found in the histogram. Any pixels whose brightness is less than the threshold, i.e. $u_{i,j} \leq \theta$, are assigned the binary value of zero. Conversely, a value of one is given to all pixels brighter than the threshold. The main advantage to this method is the data compression since no matter how many bits were required to describe each pixel, the new image uses only one bit per pixel. This is why this type of image is called a *binary* image. It should be obvious that at the same time we are doing data compression, we are also suffering a loss of information. There could be objects in the image that have very similar shades of gray. This method of thresholding could possibly assign the same binary value to them. The same is true with black or white objects when the lighting is far from ideal.

Computationally speaking, we have to test every pixel in the image which means we have N^2 comparisons to make. As far as setting the threshold, there is some simple hardware that can be added to the digitizer that would automatically send a binary image to the computer. The main problem is that a human operator typically decides where the threshold should be, therefore adding a subjectivity factor.

The idea of thresholding could be extended to several levels. In a multi-threshold situation each pixel is assigned one out of $N+1$ values, where N is the number of thresholds. The number of bits required is $INT(\log_2(N+1))$ where INT() is the function that gives the smallest integer greater than its argument.

Some attempts have been made to provide for automatic thresholding. One idea [3] is to "fit" a curve to the section of the histogram between the two peaks. For example, one could fit a quadratic equation $y = ax^2 + bx + c$, where a,b and c are constants. This could give a good approximation of the histogram's valley. The minimum of the valley would be found at the point $\theta = -b/2a$. Other approaches have been tried, such as trying to fit gaussian distribution curves to the histogram, etc.; the goal being to find the best point to place the threshold.

In summary, if we are willing to sacrifice the loss of information inherent in the procedure, thresholding offers the advantage of being able to process several pixels at a time. In addition, if we stay with the binary image, all the possible operations on pixels are boolean in nature. Special purpose hardware can easily be built to perform them in real time. From the robotics point of view however, the binary image is not a sufficient representation for many problems. As described in section 1.2, we would like to retain the gray level information that comes from the camera.

2.3 Filtering

When a signal is passed through a linear system, a convolution between itself and the system's impulse response occurs. This linear operation is called *Filtering*. In digital signal processing, there are two main types of filters: those with *infinite* impulse response, or **IIR**, and those with *finite* impulse response, or **FIR**.†

There are many ways to realize a one-dimensional digital filter. When a two-dimensional filter is

---

†There is a third type of filters, the so called *Lattice*, which are recursive but have a different realization from the traditional IIR. These, however, are not used much in image processing.

desired however, the problem becomes more complex. For example, in the one-dimensional case, when we want to implement a particular filter, we first obtain the Z-transform of the desired transfer function, and then perform a **polynomial factorization** which gives us a series of first- and second-order pole/zero factors to be cascaded. The problem in the two-dimensional case is that this factorization doesn't necessarily exist, since the filter's transfer function depends on two variables and, in the general case, there are cross-products of them.

In the case where there is such a factorization, the problem is reduced to two one-dimensional filters. In an image field this means that we may filter in the row and column directions *independently*. In the more general case however, we have to perform a two-dimensional convolution or transform the image and do the filtering in the frequency domain. We will talk about transforms in section 2.4.

2.3.1 IIR filters

The IIR filters offer several advantages in the one-dimensional case. They may be obtained from existing analog filters via simple transformations and they usually require a much lower order to achieve a particular frequency response; *they can be unstable*, however. The stability issues have been discussed for the one-dimensional case but when the signal is two-dimensional, stability cannot be guaranteed except for very few special cases.

Computationally, an IIR filter can be expressed by a recursion formula. The number of operations per pixel depends on how many terms this formula has. Since each pole introduces a term, at least in the one-dimensional case, then we can say that the number of operations per pixel depends on the order of the filter. IIR filters are seldomly used in image processing. An exception is the Kalman filter which leads to recursive implementations that have been used widely in one-dimensional digital signal processing.

A problem in applying Kalman filters to two-dimensional signals is that computational loads grow linearly with the number of pixels. Woods et al. [4] introduced two new approximations to a Kalman filter. One, called the strip processor, updates a line segment at a time; the other, the reduced update Kalman filter (RUKF,) is a scalar processor. In the latter case, the equations were obtained for the undistorted signal-in-noise observation model. The signal model was of the AR type with a general or nonsymmetric half-plane (NSHP) coefficient support. The resulting filter equations were seen to constitute an NSHP recursive filter. The filter was shown to be weakly optimal [5] and also approximately strongly optimal in several examples. Recent work [6] extended the RUKF to deconvolution-type problems where we observe a noisy and distorted version of the signal and wish

to estimate the noise-free signal. They accomplished this through modeling the signal distortion as an FIR filtering with NSHP support.

2.3.2 FIR filters

The FIR filters do not have poles and therefore are not susceptible to stability problems. The one-dimensional techniques of windowing, etc. have been applied to images with some success. It is nonetheless important to consider the case where the filtering is performed by convolving the image field with a mask.

A typical low-pass mask, used for image smoothing, would be:

$$H = \begin{matrix} +1 & +2 & +1 \\ +2 & +4 & +2 \\ +1 & +2 & +1 \end{matrix}$$

This mask is centered around the pixel we are analyzing. The convolution of the image with the mask performs the low pass filtering.

$$v_{i,j} = \frac{1}{16}\left[\sum_{k=0}^{N}\sum_{i=0}^{N} u_{k,l}\, h_{i-k+1,j-l+1}\right]$$

Note that since the mask is biasing the brightness of the pixel, we include the 1/16 factor in front of the summations.

High-pass masks have been used for edge crispening. The idea is to enhance the brightness of the pixel we are interested in when compared to its neighbors. A typical high-pass mask that doesn't bias the image is:

$$H = \begin{matrix} +1 & -2 & +1 \\ -2 & +5 & -2 \\ +1 & -2 & +1 \end{matrix}$$

All these cases require $8N^2$ summations and $9N^2$ multiplications and leave the image's dimension unchanged so the next stage of computation will also take a number of operations proportional to N^2.

2.4 Transforms

In this section we will discuss a number of *unitary* transformations. A unitary transformation is a specific type of linear transformation in which the basic linear operation is exactly invertible and the operator kernel satisfies certain orthogonality conditions. The forward unitary transform of the N x N image field U results in another N x N field V as defined by:

$$v_{i,j} = \sum_{k=0}^{N} \sum_{l=0}^{N} u_{k,l}\, \alpha_{k,l;i,j}$$

where $\alpha_{k,l;i,j}$ represents the forward transformation kernel. The reverse or inverse transformation provides a mapping from the transformed domain to the image space as given by:

$$u_{k,l} = \sum_{i=0}^{N} \sum_{j=0}^{N} v_{i,j} \beta_{i,j;k,l}$$

where $\beta_{i,j;k,l}$ denotes the inverse transformation kernel. If the kernels are orthonormal to all other forward and inverse kernels, the transform is said to be unitary. If the kernels can be written in the form

$$\alpha_{k,l;i,j} = \alpha^{C}_{k;i} \alpha^{R}_{l;j} \text{ and } \beta_{i,j;k,l} = \beta^{C}_{i;k} \beta^{R}_{j;l}$$

then the transformation is said to be separable. A separable two-dimensional unitary transform can be computed in two steps. First a one-dimensional unitary transform is taken along each row and then a second one-dimensional unitary transform is taken on the partial result. Unitary transforms may be represented as matrix operations on the image array. Then the forward transform is:

$$V = UA$$

while the inverse transform is:

$$U = VB$$

Obviously, $B = A^{-1}$. For a unitary transform, $A^{-1} = A^{*T}$.

Examples of transforms include Fourier, Cosine, Sine and Hadamard transforms. They all have

been used for image representation purposes. Since all these are unitary transforms, it is possible to compute them using FFT-like algorithms. As discussed in section 2.1, by using these algorithms it is possible to change the number of operations from $\sim N^4$ (for general transforms) and $\sim N^3$ (for separable unitary transforms) to $\sim N^2 \log_2(N)$.

2.5 Edge detection

In this section we will explore a series of algorithms that deal with the problem of detecting the edges of an object that appears in the image. An edge is defined as a change or discontinuity in the brightness function. We will only consider *local* discontinuities. The *global* ones will be treated in section 2.7.

A common approach for edge detection has been to first *enhance* the edges and then to threshold. If we choose the threshold to be a brightness value less than the edge brightness, only the edges will have a binary "one". Selecting the threshold is a problem since too high a threshold will miss important edges while a threshold set too low will misinterpret noise as edges in the image.

A variety of edge enhancement techniques can be utilized to accentuate the edges before the thresholding operation. One of the simplest techniques is discrete differencing. Horizontal edge sharpening can be obtained by the running difference operation, which produces an output image **V** from the relation:

$$v_{i,j} = u_{i,j} - u_{i,j+1}$$

Similarly, vertical sharpening results from:

$$v_{i,j} = u_{i,j} - u_{i+1,j}$$

Diagonal sharpening can be obtained by subtraction of diagonal pairs of pixels.

Horizontal edge accentuation can also be accomplished by forming the differences between the slopes of the image amplitude along a line, according to the relation:

$$v_{i,j} = \left[u_{i,j} - u_{i,j-1}\right] - \left[u_{i,j+1} - u_{i,j}\right]$$

or

$$v_{i,j} = 2u_{i,j} - u_{i,j-1} - u_{i,j+1}$$

Similar equations exist for vertical and diagonal slope differences.

Two-dimensional discrete differentiation can be performed by convolving the original image array with a 3 x 3 *compass mask*. There are eight such masks, each tailored to enhance edges in a particular direction. For example, the **North** mask is:

$$H = \begin{matrix} +1 & +1 & +1 \\ +1 & -2 & +1 \\ -1 & -1 & -1 \end{matrix}$$

while the **Southwest** mask is:

$$H = \begin{matrix} +1 & -1 & -1 \\ +1 & -2 & -1 \\ +1 & +1 & +1 \end{matrix}$$

This is the L x L submatrix case we mentioned in section 2.1. Computationally speaking, we have to perform L^2 multiplications and $L^2 - 1$ additions ***per pixel***, with N^2 pixels total.

Nonlinear operations have also been introduced to enhance the edges before thresholding. Most techniques process each pixel in a very local way. Roberts [7] has introduced a simple nonlinear cross operation as a two-dimensional differencing method for edge sharpening:

$$v_{i,j} = \left(\left[u_{i,j} - u_{i+1,j+1} \right]^2 + \left[u_{i,j+1} - u_{i+1,j} \right]^2 \right)^{1/2}$$

He also introduced a computationally simpler operation:

$$v_{i,j} = \left| u_{i,j} - u_{i+1,j+1} \right| + \left| u_{i,j+1} - u_{i+1,j} \right|$$

Extensions to this idea have been proposed for the case of looking at the eight surrounding

neighbors of the pixel being analyzed. Masks with three or five levels, allowing estimation of the position and the orientation of an edge, have been introduced by Sobel [8] and Prewitt [9].

2.6 Edge Following

Once the edges have been found, it is desirable to develop a structural description of the image. As a first step, we might like to know whether or not the edges define convex regions, called *blobs*. A straightforward way of attacking this problem is by following the edge.

Perhaps the simplest contour following algorithm is the commonly called the *bug following procedure*. A conceptual bug begins marching from the white margin to the black pixel region. When the bug crosses into a black pixel, it makes a left turn and proceeds. If the next pixel is also black, the bug again turns left; if the pixel is white, the bug turns right. This procedure continues until the bug returns to its starting point. It is easy to show [3] that this procedure is dependent on the starting point and that if the object has holes, the bug might get lost.

Other schemes have been developed: Rosenfeld, et al. [10, 11] applied a connexity-approach to the problem; Horowitz and Pavlidis [12] have developed split-and-merge methods. Finally, Ashkar and Modestino [13] introduce prototype edges to be followed, by using a dynamic programming algorithm.

2.7 Segmentation

In this section we will discuss some algorithms that are used to find objects in a given image field. The idea is to classify the various regions that appear in the picture. To do this, a *global* approach is used in which the classification takes place while keeping all the other regions in mind. A number of *local* algorithms were discussed in section 2.5.

In the past, people have tried several methods of doing the classification. Most of them have two phases. In the initial phase, a *clustering* algorithm is employed to define classes with similar levels of brightness. After the clustering has provided a set of regions, a *relaxation* algorithm is used to test each pixel and see whether it belongs to a particular region (**blob**) or not. Typically, the relaxation phase is iterative in nature since the blobs are modified with each pass. This relaxation phase can, in most cases, be thought of as a specialized filtering operation similar to those discussed in section 2.3. For example, Basseville et al. [14] utilized a Kalman filter in their adaptive segmentation algorithm (see section 2.8).

It is difficult to talk about the number of operations needed for doing segmentation in an N x N image because clustering may be done in various ways. Consider the following simple procedure: We first find out which pixels have a brightness ζ_k. If we assume that each pixel is represented by eight bits, then k can take values in the interval [0,255]. Once we have this information (in fact, we obtained a histogram,) we want to know whether two pixels, say p_i and p_j, belong to the same blob. A number of criteria may be used. For example, we can define a *connected* region as that enclosed part of the image whose pixels have the same ζ_k, where the "trick" here is to define **enclosed.** We might, for example, analyze a pixel by looking at its eight neighbors. Then it is said that two pixels belong to the same enclosed region if their brightness is the same *and* if there is a path of 8-connected pixels between them. [1]

It is easy to see that in order to obtain the enclosed regions (blobs), a number of very different algorithms might be used. This is why it is difficult to determine the computational complexity of the clustering phase. The relaxation phase might be done by moving a mask through the image. Each time the L x L mask is used in the image L^2N^2 multiplications and $(L^2 - 1)N^2$ additions are needed.

2.8 Adaptive Image Modeling

In the previous sections we have reviewed several algorithms that apply to the image in the same way, independently of what it actually contains. Several attempts have been made to model the image which has led to another set of algorithms called *adaptive algorithms.*

There are several ways to model an image. Two-dimensional models have been proposed but they generally require too much computation. One-dimensional models, on the other hand, tend not to show two-dimensional relational information. Nevertheless, one-dimensional models are very appealing in terms of computational loads and, if carefully developed, yield two-dimensional information.

In the early stages of image processing, the one-dimensional signal taken from raster-scanning the image was modeled using stationary second-order models. These models were later found to be inadequate. So, in the past few years, several authors have tried nonstationary statistical models. For example, Hunt and Cannon [15] have used models in which the mean is not assumed constant and the covariance statistics are described as stationary fluctuations about a spatially nonstationary mean vector. Jain [16] has decomposed pictures into a stationary process and a boundary process; this structure has been used in the area of picture coding.

It is important to note that most natural scenes can be thought of as a composition of locally stationary regions (**blobs**) separated by edges. In the global nonstationary model, the edges are the areas where abrupt changes occur in the first- and second-order statistics. They separate homogeneous areas in which these statistical properties, or *textures*, are constant or slowly varying.

The problem of detecting the changes in the statistical models is not a new one and several approaches have been tried [17]. Basseville et al. [14, 18] have shown that edge detection is possible by using algorithms that detect a sudden change in the mean of the brightness function. They present a recursive procedure to find the edge elements on each line. The edge-following problem is solved by a Kalman filter, the state model corresponding to a noisy straight line.

In general, the approach of segmenting a signal by detecting abrupt changes in its statistics have been used in more complex one-dimensional signals. Sanderson et al. [19] have applied an adaptive autoregressive (AR) model to the EEG signal. Segen and Sanderson [20] showed that it is possible to detect changes in a piece-wise stationary signal by detecting the points where the cumulative prediction error of the AR model exceeds a threshold. At this point, the change is recorded and a new model is generated for the next segment. Sanderson and Segen [21] also showed that their method is applicable to the image segmentation problem where the prediction is the mean value (similar to a zeroeth-order AR model).

The execution time of this *adaptive segmentation* algorithm depends on the number of changes detected since a new model must be generated for each change. For each pixel, however, we must test the current model and decide whether a new model should be generated or not. This means that we must execute the following loop N^2 times:

1. Fetch the pixel $u_{i,j}$.
2. Calculate the new cumulative error ε_j:

   $$\varepsilon_j = \varepsilon_j + (u_{i,j} - \mu_j),$$

   where predicted μ_j is the mean value.
3. Compare ε_j to θ (a threshold dependent on the the model's standard deviation).
4. Increment the index j.
5. If either ε_j is greater than θ or j exceeds 255, record the change and generate a new model. Otherwise return to step 1 above.

This loop will be used as a benchmark in the following chapter where we compare different architectures for image processing algorithms.

Current work is being done to determine whether the AR model should include a recursive term to also take shading into account.

AR models have several properties that are very appealing. First, they may be obtained sequentially from the image, either row by row, or column by column. It is important to realize that the AR representation assumes that the image has additive gaussian white noise on it. Therefore, the stochastic processes theory for discrete random processes applies in full. If we let $\boldsymbol{u}_r$ be the row-ordered vector of our random field **U**, then each element u_i can be represented as:

$$u_i = \tilde{u}_i + \varepsilon_i$$

where

$$\tilde{u}_i = \sum_{n=1}^{p} a_n u_{i-n}$$

$\tilde{u}_i$ is the best mean-square predictor of u_i based on its past p samples. The sequence ε_i is a zero-mean white-noise random process independent of its past outputs. For a discussion of the main properties of AR models, see Jain's paper in mathematical models for image processing [22].

In any AR model, the main difficulty lies in obtaining the autoregressive coefficients a_i. Several methods have been proposed, with various degrees of success. One problem is that since the AR representation is recursive, it can lead to instabilities. In order to avoid this, some methods rely on the positive-definite characteristics of the covariance matrix of any given random process. A matrix inversion is typically needed with the size of the matrix being no less than $p \times p$ (p is the order of the model.) There are other methods that don't require the matrix inversion at the expense of potential instability. A newer method, based on a Lattice filter realization, has been found to yield stable models without matrix inversion. Rabiner and Schafer [23] showed that the Lattice method requires almost as many computations as the covariance method *on a general purpose computer.* The only apparent advantage is that it can be performed sequentially, an important advantage if we are talking about a signal that is raster scanned from a TV camera. In the next chapter we will compare several architectures performing the Lattice algorithm to obtain the AR coefficients of a tenth-order model.

3. Architectures for image processing

In this chapter we will analyze several architectures that have or could have been used for image processing. It should be noted that this is not an exhaustive survey of possible architectures.

As discussed in chapter one, we would like to perform **near real-time** analysis on the image. Since image acquisition takes ~33 ms., it is desired to start the analysis *before* the whole image is in memory. This constrains us to use **sequential** algorithms; the following algorithms are illustrative of the processing we wish to consider:

- A filtering operation utilizing the low-pass filter mask described in section 2.3. Note that this is a convolution of the whole image with a 3 x 3 matrix. As such, the execution time for this algorithm should be representative of several other ones.
- A 1024-point, one-dimensional FFT. This is a typical benchmark for signal processing architectures. Even though it is one-dimensional, repeated application of the algorithm is used to achieve a two-dimensional FFT.
- The computation of the coefficients for a 10th order AR model from 100 points of data. AR modeling is used extensively in signal modeling, and may be used to describe texture. We will deal with the recursive Lattice method, which is related to more complex recursive algorithms.
- The adaptive segmentation procedure presented in section 2.8. The part discussed will be the inner loop where each pixel is tested against the mean value and the cumulative error is compared to a threshold.

The first three of these algorithms are of general interest for evaluation of image processing architectures, while the fourth is of direct value for real-time image feature extraction. We will present **approximate** execution times of these four algorithms for each architecture analyzed. First, let us define the necessary operations for each algorithm.†

1. **Low-pass filtering.** In this case we are performing the following operation:

$$u_{i,j} = \sum_{k=1}^{2} \sum_{l=1}^{2} a_{k,l}\, u_{i+k-1,\, j+l-1}$$

   Note that 9 multiplications and 8 additions are needed per pixel. The $a_{k,l}$ coefficients are assumed to be 16 bits wide.

2. **FFT.** Since a 1024-point FFT is computed, we have 10 stages with 512 butterflies per

---

† In our discussions, we will assume a 256 x 256 x 8 image. In other words, the TV frame will have 256 lines, each having 256 pixels, and each pixel will have an 8-bit brightness value.

stage. Then, a total of 5120 butterflies will be computed. Each butterfly requires one complex multiplication and two complex additions (or four real multiplications and four real additions.) The data is assumed to be 16 bits wide.

3. **AR coefficients.** In order to obtain the 10 coefficients in a sequential manner, the Lattice method [23] is used, where two error sequences are computed for each pixel: the *forward prediction error* $e^{(i)}(m)$ and the *backward prediction error* $b^{(i)}(m)$. They are given by the equations:

$$e^{(i)}(m) = e^{(i-1)}(m) - k_i\, b^{(i-1)}(m-1) \tag{3.1}$$

and

$$b^{(i)}(m) = b^{(i-1)}(m-1) - k_i\, e^{(i-1)}(m) \tag{3.2}$$

where

$$k_i = \frac{n_i}{d_i} \tag{3.3}$$

$$n_i = 2\sum_{m=0}^{N-1} \left[e^{(i-1)}(m)\, b^{(i-1)}(m-1)\right]$$

and

$$d_i = \sum_{m=0}^{N-1} \left[e^{(i-1)}(m)\right]^2 + \sum_{m=0}^{N-1} \left[b^{(i-1)}(m-1)\right]^2$$

The AR coefficients are obtained from the following formulae:

$$a_j^{(i)} = a_j^{(i-1)} - k_i\, a_{i-j}^{(i-1)} \qquad \forall\, j = 1,2,\ldots,i-1$$

and

$$a_i^{(i)} = k_i \tag{3.4}$$

The procedure for obtaining the 10 coefficients $a_1,\ldots,a_{10}$ consists of the following steps:

a. Set $e^{(0)}(m) = b^{(0)}(m) = u_m$, the pixel value, $\forall\, m = 0,1,\ldots,99$.

b. Compute $k_1 = a_1^{(1)}$ from eq. (3.3).

c. Determine $e^{(1)}(m)$ and $b^{(1)}(m)$ from eqs. (3.1) and (3.2).

d. Set $i = 2$.

e. Compute $k_i = a_i^{(i)}$ from eq. (3.3).

f. Obtain $a_j^{(i)}$ for $j = 1,\ldots,i-1$ from eq. (3.4).

g. Determine $e^{(i)}(m)$ and $b^{(i)}(m)$ from eqs. (3.1) and (3.2).

h. Let $i = i + 1$.

i. If i is less than 11 go to step d. above.

j. The 10 AR coefficients are $a_j^{(10)}$ for $j = 1,2,...,10$.

Summarizing, we need 201 multiplications and 199 additions for step b., 200 additions for steps c. and g., 301 multiplications and 298 additions for step e., one addition for step h., and one comparison for step i. Taking into account that steps e., g., h. and i. are repeated 9 times and adding 45 additions (total) for step f., we have a grand total of 2909 multiplications, 4935 additions and 10 comparisons. The data is assumed to be 16 bits wide.

4. **Adaptive segmentation.** Since we are only concerned with the inner loop, 3 additions and one comparison are needed per pixel. All operations are byte-wide. **We would like to execute this algorithm in less than 33 ms., to achieve near real-time image segmentation.**

3.1 Single-processor architectures

A number of general purpose (GP) computers have been used for digital signal processing and image processing. In most cases, their use has been limited to *off-line* types of algorithms. This is partially due to the fact that if the GP computer is very fast it is also expensive; slow computers don't lend themselves to *real-time* signal processing except for a few rare cases (image processing is not among them).

We will analyze the MC68000 and the PERQ as *conventional* and *microprogrammable* processors, respectively.

3.1.1 Conventional processors - The MC68000

The term *conventional processors* describes a computer designed to perform many different tasks, one that hasn't been optimized for performing mathematical operations, string manipulations, etc.; it is, in this sense, a *very-general purpose* computer. A characteristic of conventional processors is that the user sees a virtual machine (also called *programming model*,) and doesn't have direct control of the hardware.

The MC68000 is a VLSI 16/32-bit microprocessor manufactured by Motorola Semiconductors, Inc. It has a very regular 32-bit internal architecture with an instruction set designed for structured high-level languages. This processor offers several advantages for digital image processing:

- 16 MB addressing space. Enough for several images plus whatever software is needed to process them.
- Post-increment addressing mode and extensive indexing capabilities. Useful for keeping several pointers on the image.
- Long-word (32 bits) operations which enable the processor to obtain up to four pixels at a time.
- Multiply instructions. Although not as fast as dedicated hardware, these instructions may be used when implementing signal processing algorithms.

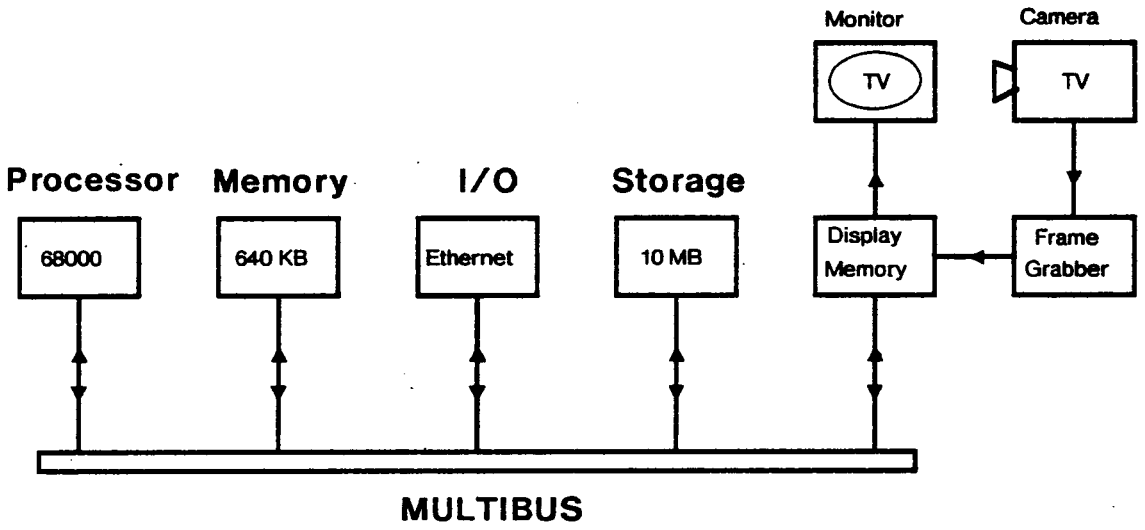

Figure 3-1: MC68000-based image processing system

Figure 3-1 shows a typical image processing system built around the MC68000. This system has been implemented and it is in use in the Image Processing Laboratory at Carnegie-Mellon University. Table 3-1 gives the approximate execution times for this system. It is easy to see that the MC68000 is far from our goal of obtaining the adaptive segmentation in ~33 ms. In general, such conventional processors do not have the speed required. It should be emphasized that, although the MC68000 is a microprocessor, it compares very well with most minicomputers. It is slowed down for some algorithms by the lack of a hardware multiplier.

Low-pass filtering	46,268,416 cycles ⇒ **5.784 s.**
1024 -point FFT	2,191,360 cycles ⇒ **273.92 ms.**
AR coefficients	348,954 cycles ⇒ **43.619 ms.**
Adaptive segmentation	2,097,152 cycles ⇒ **262.144 ms.**

Table 3-1: MC68000 execution times

3.1.2 Microprogrammable processors - The PERQ

Another type of GP computer, where the user is allowed to interact with the hardware, are the *microprogrammable* machines. By writing a program in microcode, the user is allowed to control the data flow inside the CPU.

The PERQ, manufactured by Three Rivers Computers Corp., is a stand-alone computer that uses the AM2910 microprogram sequencer with a standard ALU. It is intended primarily for text processing and graphics but, since it is possible to write microcode for it, it performs quite well for simple algorithms. Table 3-2 shows the approximate times for the four benchmarks using simple microcode routines. Note that the PERQ is 4 to 5 times faster than the MC68000.

Low-pass filtering	10,223,616 cycles ⇒ **1.704 s.**
1024-point FFT	399,360 cycles ⇒ **66.56 ms.**
AR coefficients	~63,925 cycles ⇒ **10.654 ms.**
Adaptive segmentation	294,912 cycles ⇒ **49.152 ms.**

Table 3-2: PERQ execution times

At first, it may seem that the PERQ is good enough for our application (~49 ms for the adaptive segmentation algorithm,) there are two major problems in using the PERQ however: first, it also lacks a hardware multiplier, which is why it doesn't perform as well in the other algorithms, and second,

loading the memory with the image would slow down the PERQ. The latter problem is due to the fact that DMA in the PERQ is done by *cycle-stealing*†. Let's look at what this means in terms of execution time: The data rate is one pixel every 250 ns, on the average. Since the PERQ's microcycle takes 166.67 ns, it may execute 3 μcycles in the same time it receives 2 pixels (500 ns). If we have to steal a cycle for each DMA transfer, this means that the PERQ is slowed down 66% during the 16.67 ms that takes to load an image! During that time, the PERQ is merely 1¼ – 1½ times faster than the MC68000. This timing assumes that the memory bandwidth is utilized at maximum, which is the case of the adaptive segmentation algorithm. In order to overcome these problems, the PERQ would have to be substantially modified.

3.2 Multi-processor architectures

The solution of using more than one processor to speed-up computation has long been contemplated by computer architects. Ideally, if we have N processors performing the same task, they should finish N-times faster than a single processor would. In reality, however, there is always some overhead due to the interprocessor communication.

3.2.1 Asynchronous communication

The most straightforward way to design a multiprocessor system is by connecting several processors to a common bus. Bus arbitration may be done on a cycle-by-cycle basis or the bus might be assigned to a processor for a certain period of time. In any case, each processor is running at its own speed, performing its own task. It is this "independence" what introduces problems when we want the processors to perform a small part of a complicated task; whenever a processor has to know what some other processor is doing, it is forced to go through some protocol. If they are sequentially using the same data (as in a pipeline,) the software is responsible for synchronizing the data transfers; this is typically done via *mailboxes* and *semaphores* [24].

In a multiprocessor system, the less the communication overhead, the closer it gets to the ideal system. For a system like the one just described, this means that we should minimize the number of times a processor has to know the status of another processor. This can be done, and very effectively, when all the processors are running separate tasks or when the tasks are loosely coupled. Unfortunately, if we want to utilize a multiprocessor for digital signal (or image) processing, we may want each processor to run a **section** of a particular algorithm. The tasks are, therefore, very tightly coupled.

---

† In the MC68000 system, the display memory is *dual-ported* so it can be read and loaded simultaneously

3.2.2 Synchronous communication - The RAPID bus

One way to overcome the synchronization problem is to **always** keep the processors informed of what the others are doing. As an example, consider two processors which clocks are supplied from the same source; one of them could know what the other one is doing, at any given time, by knowing each other's programs. This system works provided the two processors were in a known state once (reset).

If we have such a synchronous system, it is possible to create *pipelines*, a very useful structure in signal processing, where the result of one processor is used as input to the next one down the pipe. No software overhead is introduced since, once synchronization is achieved, every *n* cycles processor **A** finishes a computation and puts the result in the mailbox $\mathbf{m_{AB}}$. Processor **B** may use whatever is there ($[\mathbf{m_{AB}}]$) knowing that it becomes valid every *n* cycles.

The **RAPID** bus (**R**otary **A**ccess **P**arallel **I**mplementing **D**igital bus,) first conceived by Sanderson and Zoccoli [25, 26] and lately refined by Bracho, et al. [27], is a time-shared bus that can be modeled as a special case of a synchronous system since the processors' clocks are synchronized to a master clock. A processor connected to the bus may make a transfer whenever needed since each processor believes it always owns the bus. This "effect" is accomplished by a very fast latching interface that makes the switching between processors transparent to them. In actuality, each bus-master has a *window-address* associated with it and the bus is switched between masters at a high speed. Master/slave transfers occur during a very narrow *window* in which the bus-master selected (via the window-address bus) is allowed to make a quick transfer.

Figure 3-2 shows the timing relationships between the master clock, driving the window addresses, and the processors' clock. The figure assumes that the system has eight masters and that a processor receives the bus on alternate cycles of its clock. The figure also shows when the processor with window-address six would communicate with a slave: A master/slave link is established every other processor cycle (every 250 ns. if the frequency of the processor's clock is 8 MHz,) it is during this link that the data transfer takes place. Three types of master/slave communications are recognized:

- **Initial.** Initially, a master specifies to which slave it wants to be connected. The slave recognizes that is being addressed by comparing the contents of the address bus with its slave-address(es). Then the slave latches the type of request (whether is a read/write, etc.) *and the master's window-address*. Hence, the slave becomes *selected* by the master.

- **Intermediate.** A number of transfers, from zero to infinity, occur in which both the master and the slave are looking at the window-address bus. When the master's window-

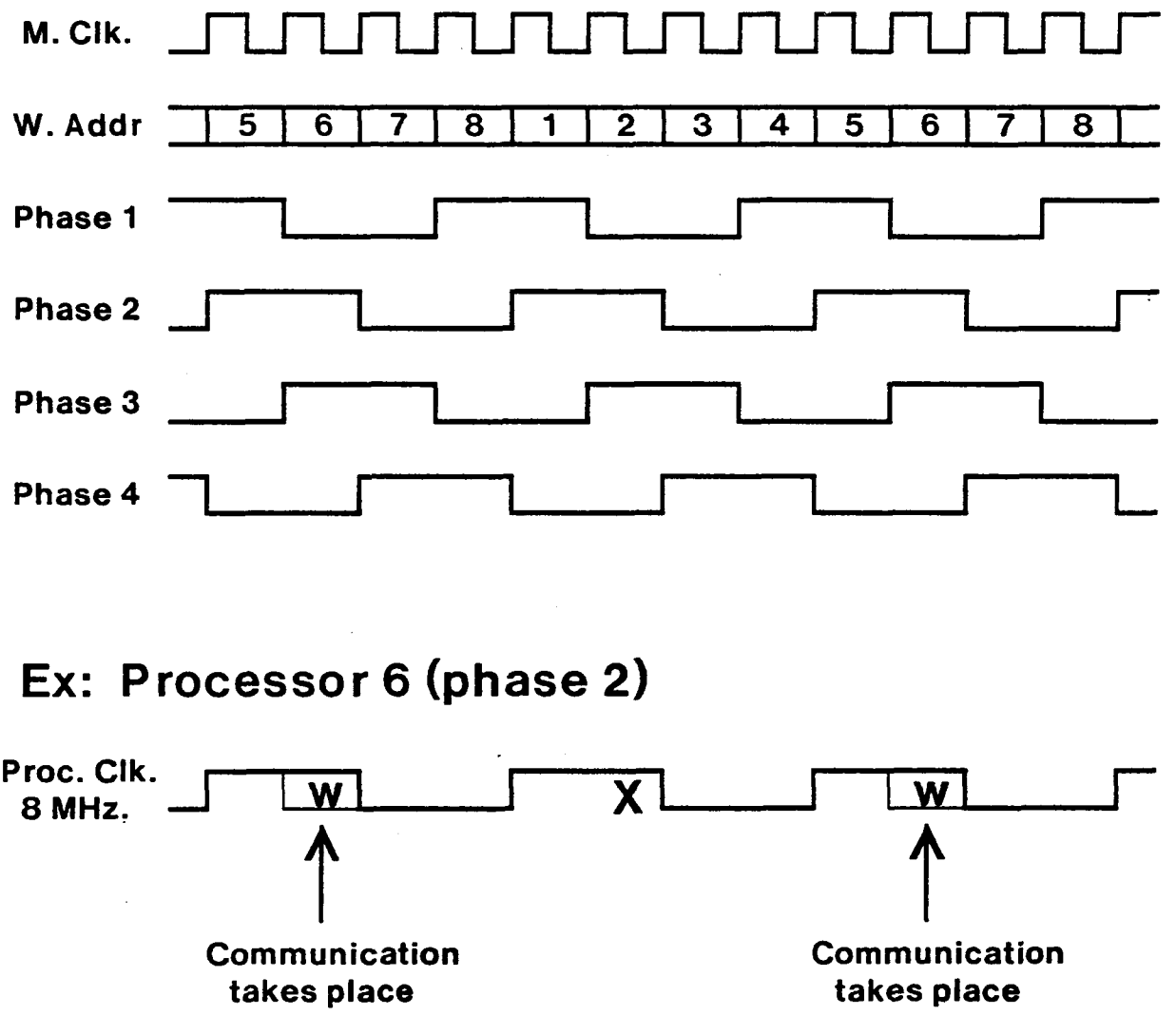

Figure 3-2: Timing waveforms in an 8-processor RAPID-Bus system

address comes along, their latching interfaces open and a communication link is established between the master and the selected slave.

- **Final.** During the last transfer, it is the master's responsibility to *release* the slave. From then on, the slave no longer looks at the window-address bus but, instead, at the address bus. Therefore, the slave can recognize an initial transfer again.

Figure 3-3 shows a system with three masters. The boxes labeled "FAST interfaces" are responsible for all the protocols needed in establishing proper master-slave communication. Included

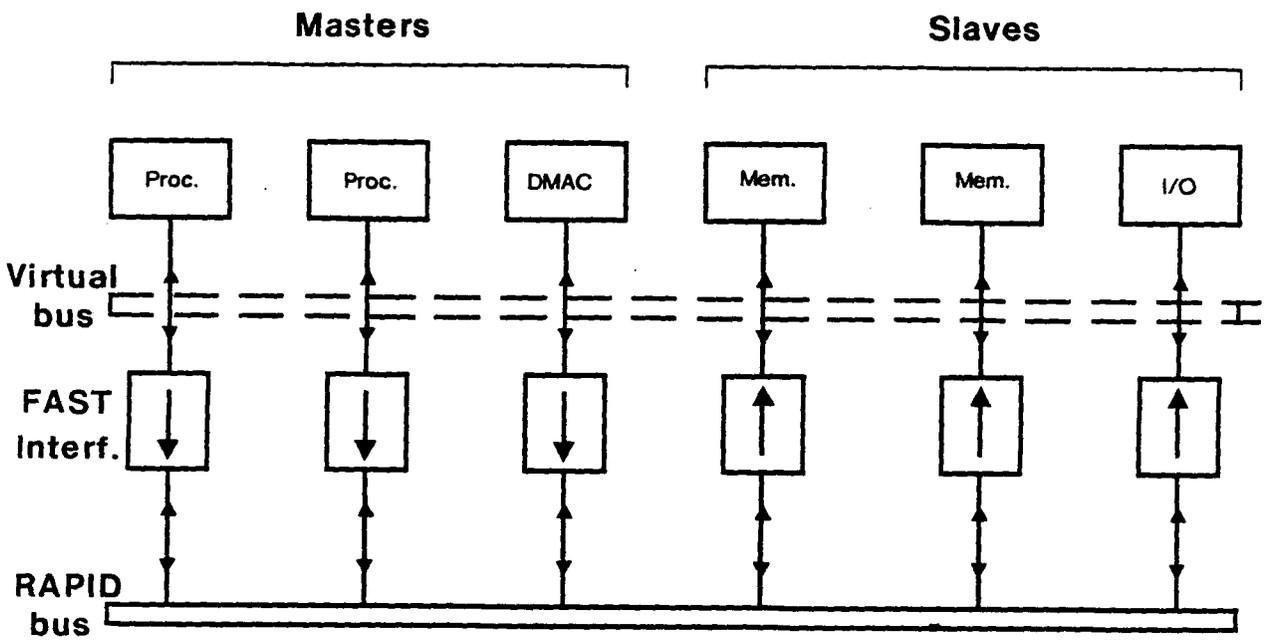

Figure 3-3: Two-processor RAPID-bus system

in them are the latching interfaces and the window-address specification-recognition systems. Two types of interfaces exist; they are pictorially represented by the direction of the arrows. A down-pointing arrow means a **master's interface** while an up-pointing arrow is a **slave's interface**. Not shown in the figure are the main clock and the window-address generator. Note that there is a *virtual* bus between the interfaces and the masters/slaves, which enables us to use off-the-shelf units.

The current implementation uses the Versabus protocol for the *virtual* bus. The prototype under construction has two monoboard microcomputers (Motorola's VM02,) which serve as both masters and slaves simultaneously (the VM02 has 128KB of dual-ported memory). This situation has posed several restrictions on the interface cards since they must deal with the problem of the local master wanting to make a RAPID-bus access while the slave is also being accessed externally. Figure 3-4 shows the block diagram of the prototype just described; note that the contention is resolved at the Versabus level since the Versabus is shared by both the master and slave portions of the interface (single path in the figure). A problem, which could lead to a deadlock situation, arises when both processors try to access each other's slave at the same time. If both interfaces are allocated to their

respective masters (via the normal Versabus arbitration logic,) the accesses will not be completed. Currently the cycles simply time out and the retry facility of the MC68000 is used. This hazard will be less present in later implementations where there will be *global slaves*, then programs may be written in which the number of accesses to other masters' local memory is greatly reduced.

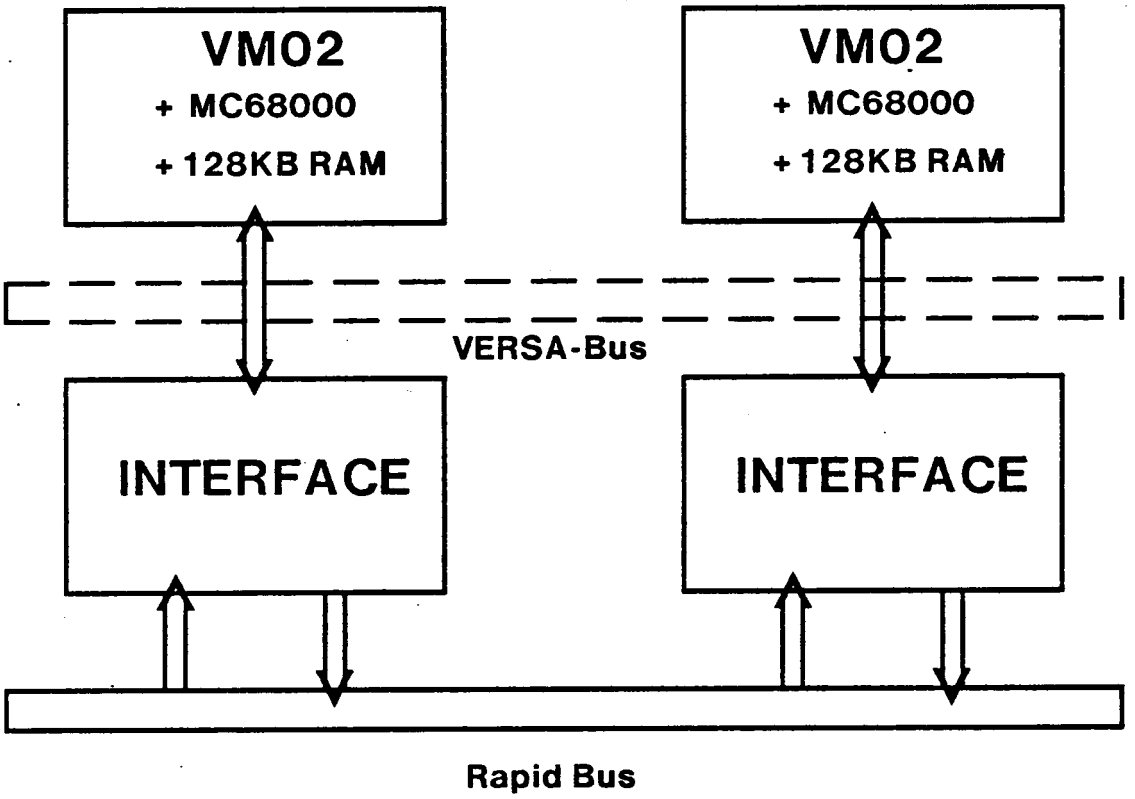

Figure 3-4: RAPID-bus prototype (block diagram)

The interface supports the full Versabus protocol, as defined by Motorola Semiconductors, Inc. [28]. It may be configured as a master, slave or master/slave interface. Two key features of the interface are:

- **Memory management.** The interface includes an MC68451 Memory Management Unit (MMU) that enables the user to define 32 variable length segments to be mapped anywhere within the 16MB physical address space. This MMU also provides the user with write protection to the segments as well as enough information to implement a virtual memory system. For high speed applications the MMU, which poses an extra 130ns series

delay to any memory access, may be circumvented by setting a bit in the interface's control register.

- **Address Generator.** A special mode of master/slave communication which enables one master to send data to several slaves has been provided within the RAPID bus specification. This mode, called *broadcast mode*, requires the slave to be able to generate the address where the data is to be stored. The inteface has an AMD2940 address generator for this purpose.

Figure 3-5 shows the block diagram of the interface. It is completely housed in a single Versabus board.

If we want to use a RAPID-bus configuration for image processing, we would have to load the image in global memory via a DMA channel. Then all the processors could access the same picture frame; figure 3-6 shows such a system. The approximate execution times for it are given in table 3-3. In order to obtain those times, the following assumptions were made:

- Six processors, MC68000s, are used.†

- The communication burden between processors is negligible. This is probably the most difficult assumption to justify and it should be noted that, in an actual case, execution times could vary quite a bit.

- All processors are used to perform the same algorithm on 1/6 of the data. This is possible but not without creating overhead since, for some algorithms, partial results would have to be stored in global memory.

† Due to technological limitations, current RAPID-bus implementations have no more than eight masters

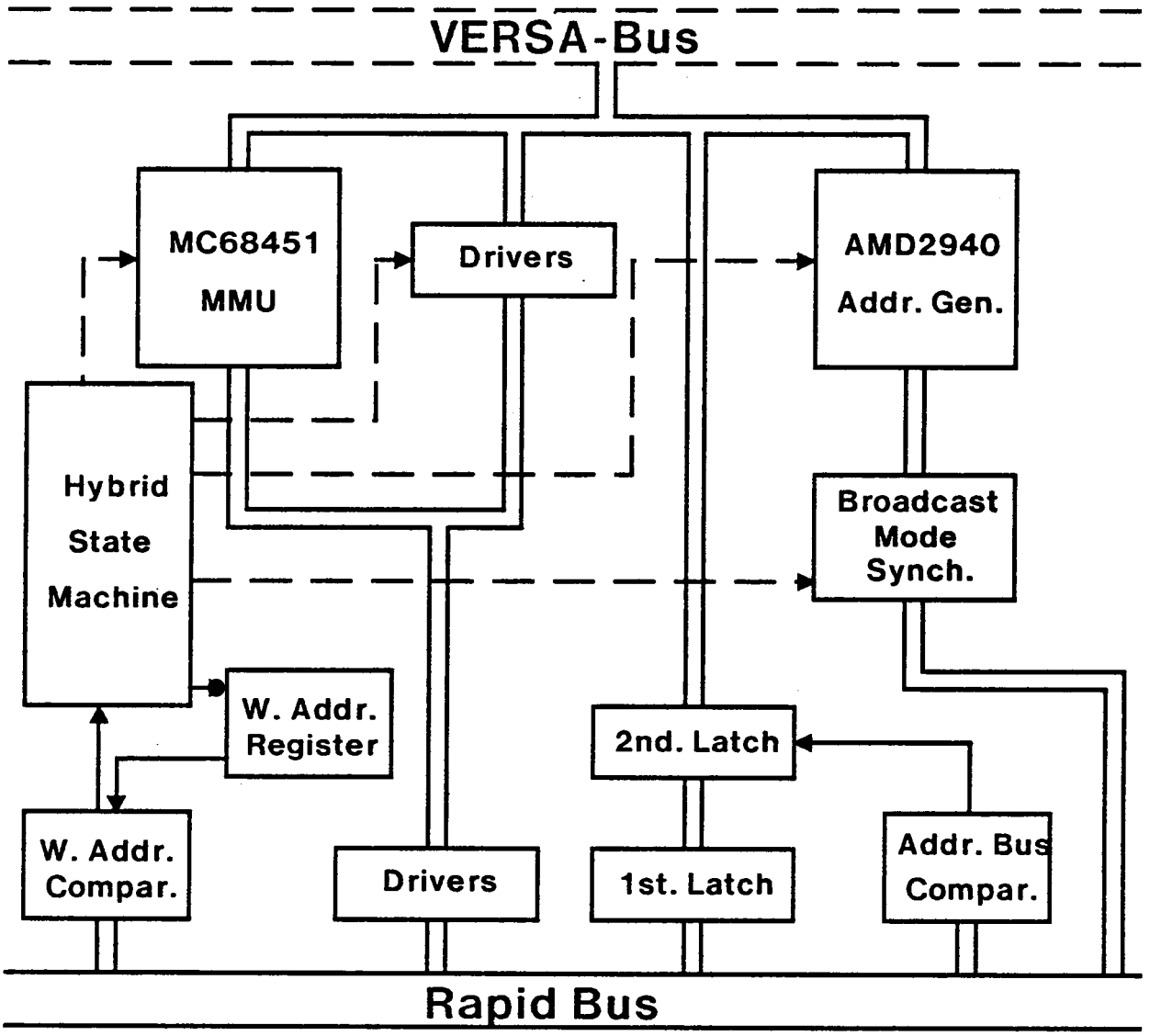

Figure 3-5: Prototype's interface (block diagram)

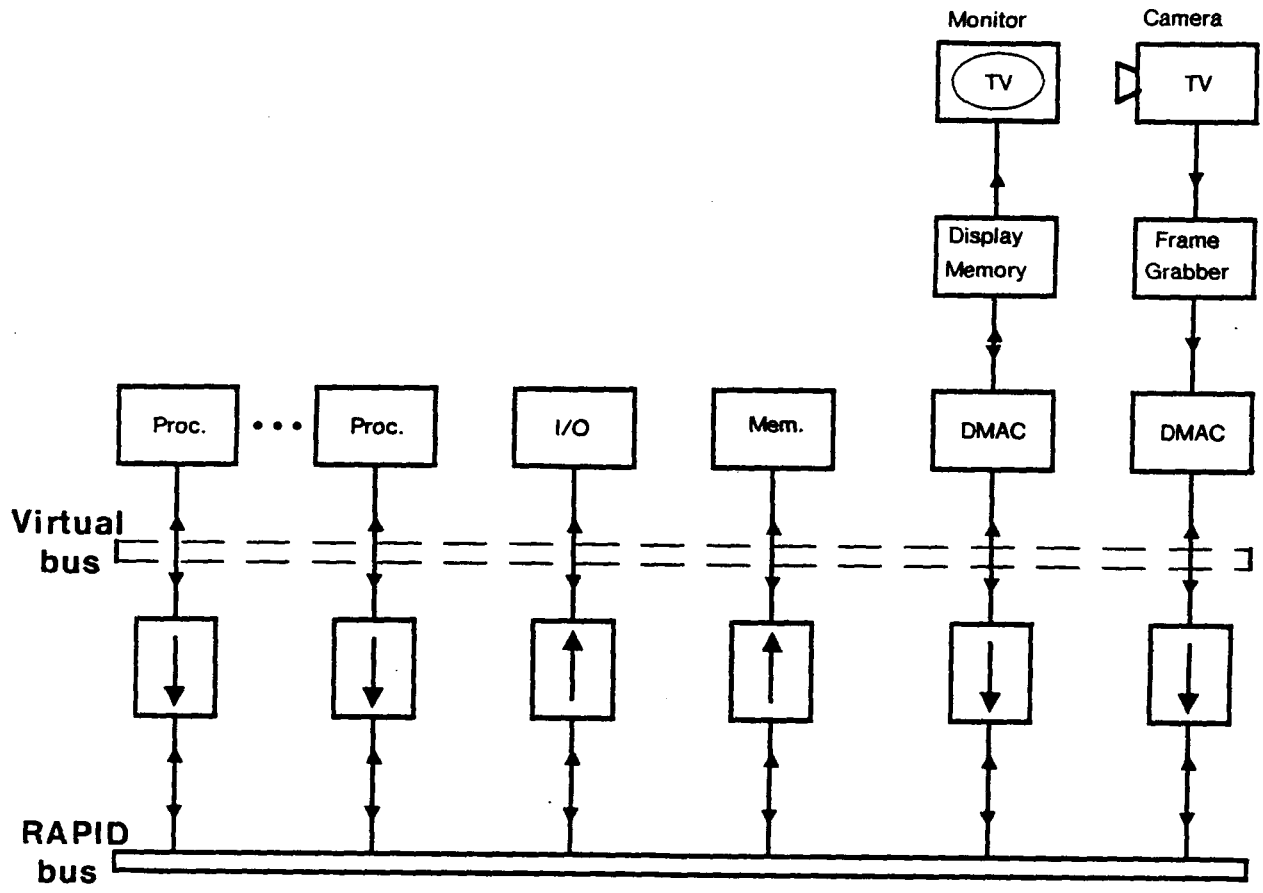

Figure 3-6: RAPID-bus configuration for image processing

Low-pass filtering	7,711,403 cycles ⇒ **963.93 ms.**
1024-point FFT	426,667 cycles ⇒ **53.33 ms.**
AR coefficients	59,359 cycles ⇒ **7.42 ms.**
Adaptive segmentation	349,536 cycles ⇒ **43.69 ms.**

Table 3-3: RAPID-bus system execution times

3.3 Array processors

Array processors are special purpose computers designed for "number crunching". They can perform a number of mathematical functions very fast; they usually have a hardware multiplier unit. In a typical array processor, the data and address computations are done separately and in parallel. This makes them a little clumsy as far as conditional branching but speeds up the data manipulation. Array processors may not be well suited for input/output; they usually depend on the host computer to perform the I/O operations.

Several commercial array processors are available. Honeywell's **HAP** which has a 16-bit, fixed point, data processing section and a 12-bit address generator. It runs at 5.6 MHz (177 ns/μcycle) and can be either microprogrammed (ROM) or used via pre-defined *macros* such as FFT, threshold, log/log^{-1}, magnitude-squared, etc.

Floating Point Systems' **AP-120B** has a 38-bit, floating point, data processing section and a 16-bit address generator. Its clock runs at 6 MHz (166.67 ns/μcycle) and, with the aid of pipelined adder (2-level) and multiplier (3-level,) it can effectively perform a floating-point addition or multiplication in a single microcycle. The user may program the AP-120B in assembly language or via Fortran-callable subroutines. It is very fast in executing pre-defined instructions but special algorithms require several instructions.

Somewhere in between is IBM's **ASP** which has a 16-bit, fixed point, data processing section and a 10-bit address generator. The data processor can have up to four highly pipelined *Arithmetic Elements (AEs)* which, in a six-level pipeline, contain a 16 x 16 multiplier, a 32-bit adder and other circuitry to output a properly-scaled sum-of-products term every 100 ns. A microprogrammed *Arithmetic Elements Controller (AEC)* coordinates the AEs. The data-flow control, however, is done by the *Control Processor (CP)* which resembles, very strongly, an IBM 360 GP computer. In fact, the user programs the ASP in a language called *Signal Processing Language*, which is the 360/370 assembly language with the addition of two instructions.

The table 3-4 gives rough approximations of the execution times for an ASP with one AE. Note that even though the array processor is very fast, it is hard to program an adaptive segmentation algorithm that uses the pipeline of the AEs effectively. Also we should stress that the ASP comes with a GP computer (360-like) as a control processor, and such a system cost at least $500,000 DIIs.

Low-pass filtering	589,824 cycles ⇒ **58.98 ms.**
1024-point FFT	20,543 cycles ⇒ **2.054 ms.**
AR coefficients	4999 cycles ⇒ **499 μs.**
Adaptive segmentation	196,608 cycles ⇒ **19.66 ms.**

Table 3-4: ASP execution times

3.4 The Robotics Institute Signal Processor (RISP)

With the advent of bit-slice microprocessors, it became possible to design processors with speeds approaching that of the array processor at a much lower cost. The Robotics Institute Signal Processor (RISP) is a microprogrammable bit-slice computer specifically designed for digital signal processing. The RISP is a proposed design and a prototype is under consideration; extensive simulations are being done using ISPS. [29, 30]

Figure 3-7 shows the block diagram of RISP. The image memory is a two-port, very fast (55 ns. access time,) byte-addressable RAM array with 64 KB. This memory may be read or written by the CPU or the Bus Interface (BI) in a single microcycle. Both the CPU and the BI are microprogrammable as explained below. RISP may also be programmed in assembly language; instructions include: solve AR model, compute FFT, filter a data array (FIR or IIR), window data, etc.

Figure 3-8 show the programming model when assembly language is used. All the general purpose registers (R0 – R31) may be used as index registers or user stack pointers, by using the pre-decrement and post-increment addressing modes. The user and supervisor stack pointers are used by RISP to store return addresses in the stack, only one being active at any given time. The status register's *S* bit determines whether RISP is working in the user or supervisor mode. The status register contains, besides the aforementioned *S* bit, the usual machine flags (*C, V, Z, N*) as well as three user flags (*U0, U1, U2*) that may be set, cleared and tested by the user in the supervisor mode.

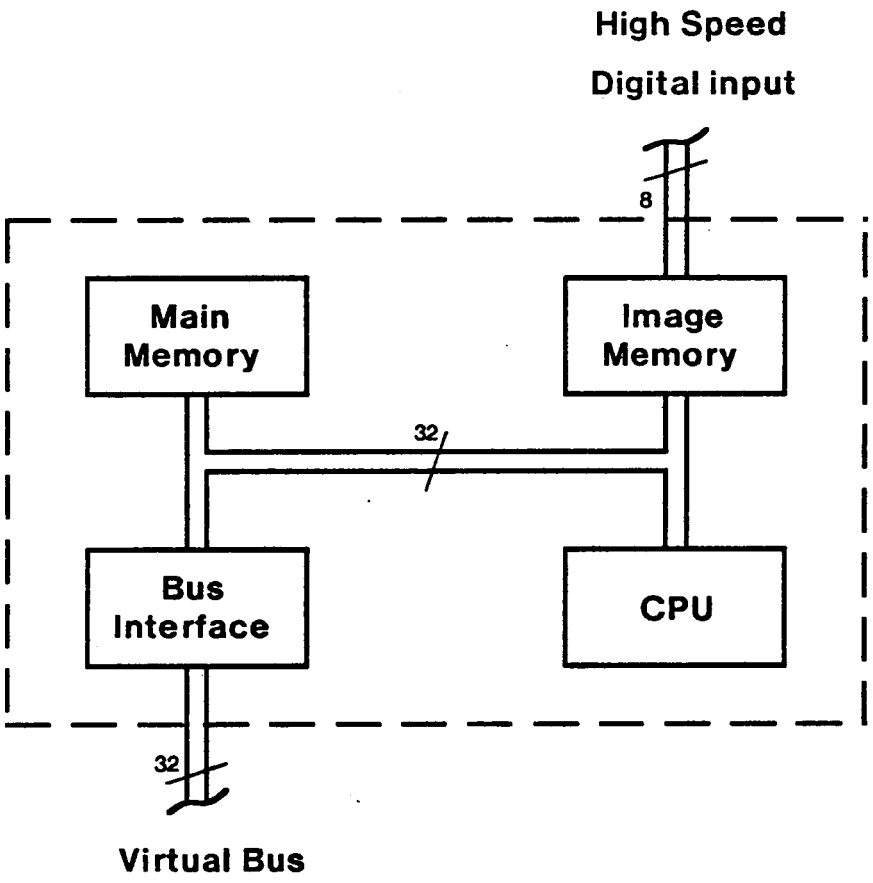

Figure 3-7: Block diagram of RISP

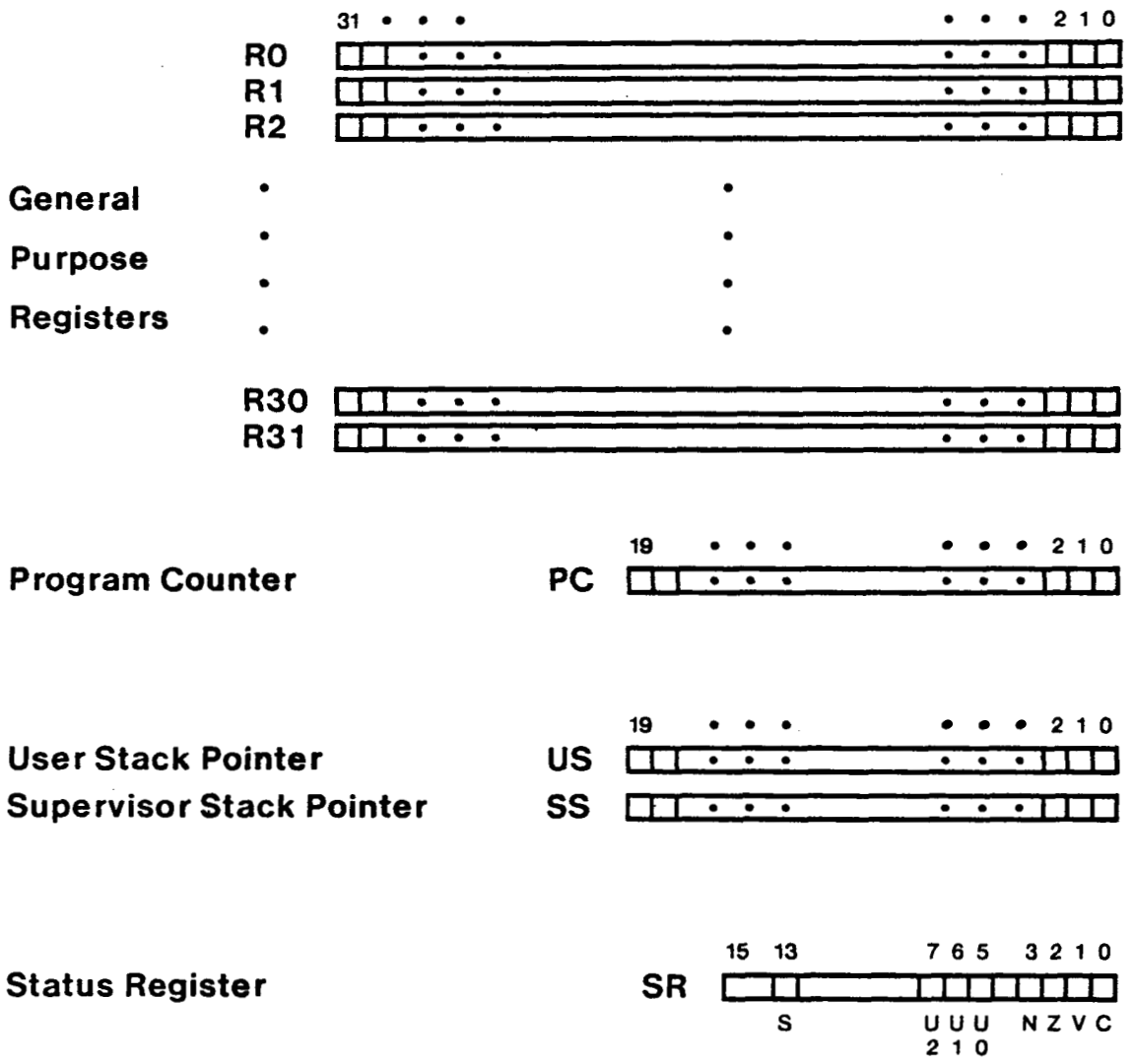

Figure 3-8: RISP programming model

3.4.1 Central Processing Unit

3.4.1.1 General description

Figure 3-9 shows a block diagram of RISP's Central Processing Unit (CPU). It consists of a high-performance, 32-bit Processing Unit (PU) and a microprogrammable (horizontal) Control Unit (CU).

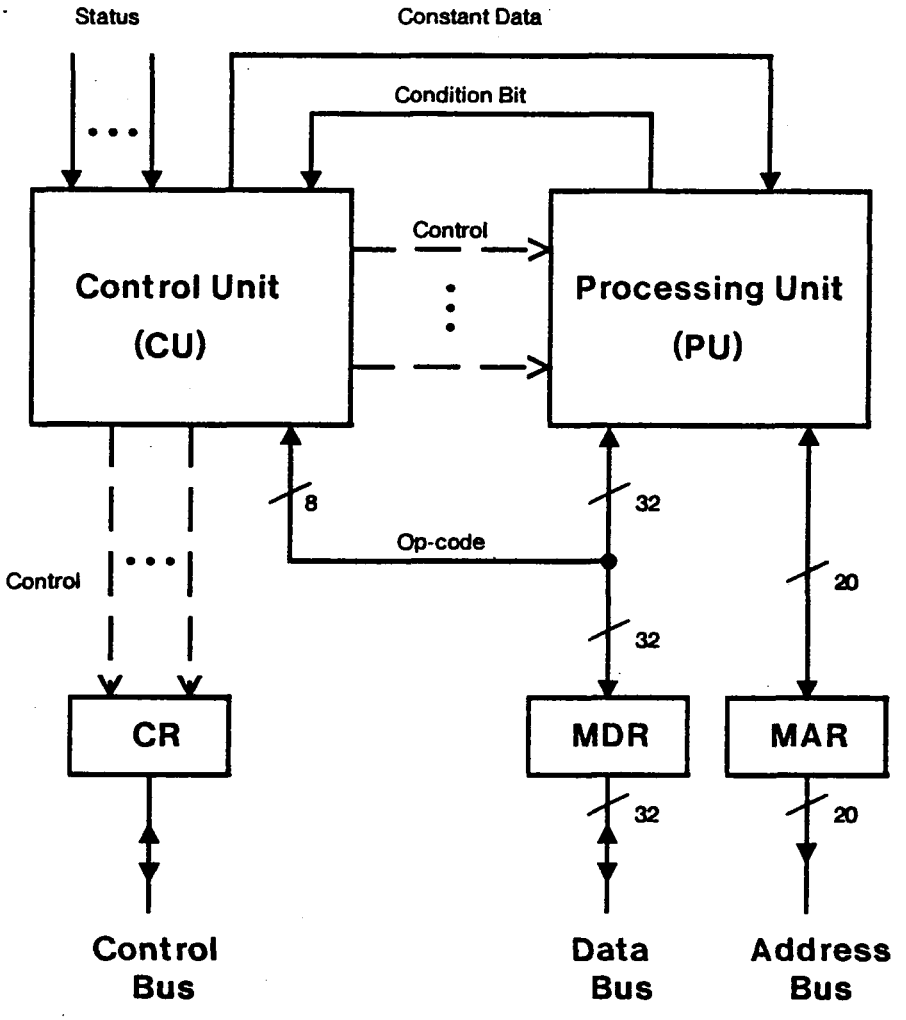

Figure 3-9: RISP's Central Processing Unit

The CU decodes the instruction's op-code (out of a possible 256), is in charge of the microsequencing, including branches and subroutines, and controls the near-150 control points of the PU. The PU, in turn, generates addresses and processes data. In the PU there is a 20-bit address generator, capable of addressing 1MB; and a 32-bit, fixed point, data processing section that can operate on bits, 8-bit bytes, 16-bit words and 32-bit long-words. It has been shown that floating-point PUs are easier to program for signal processing applications (the user doesn't have to scale up/down intermediate results to avoid under/overflow) however, fixed-point machines are generally faster and most of the image processing algorithms deal with fixed-point quantities. Nevertheless, RISP has

floating point routines microcoded, available to the user as assembly-language instructions. Also a barrel shifter, to speed up floating-point normalization, is included.

3.4.1.2 Control Unit

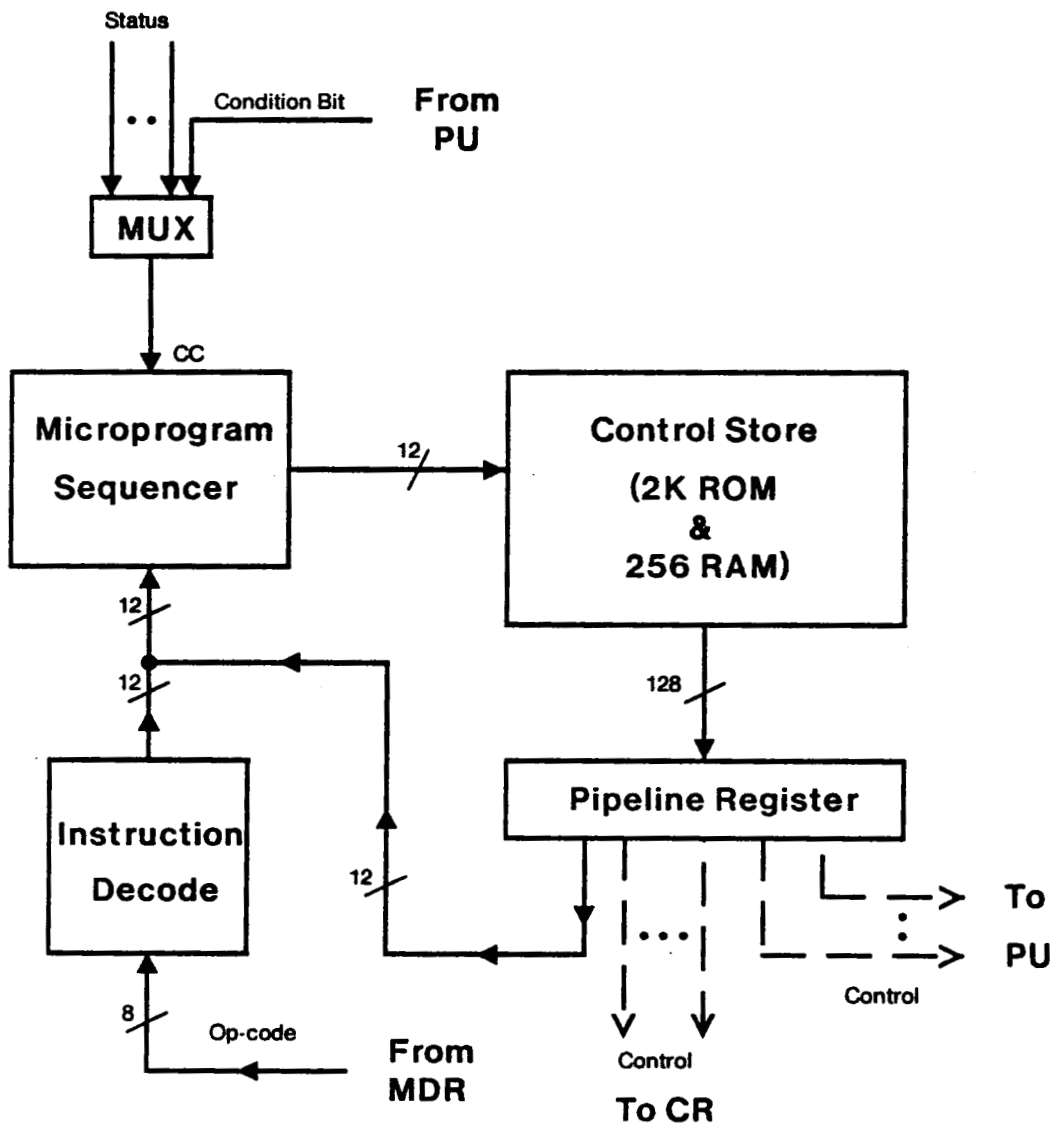

Figure 3-10: RISP's Control Unit

The CU of RISP is based on AMD's AM2910 microprogram sequencer. This 12-bit sequencer is capable of addressing 4K words of control store. The design has been kept as horizontal as possible to improve control of the hardware and enhance speed. Currently the word length is close to 128 bits.

The CU operates in a pipeline fashion with the PU, i.e. the CU is fetching the jth word from the control store while the PU is executing the (j – 1)th word. The microprogrammer has to be aware of this when branching and looping.

The design of the CU is as straight-forward and conventional as it can be, figure 3-10 shows its block diagram.

3.4.1.3 Processing Unit

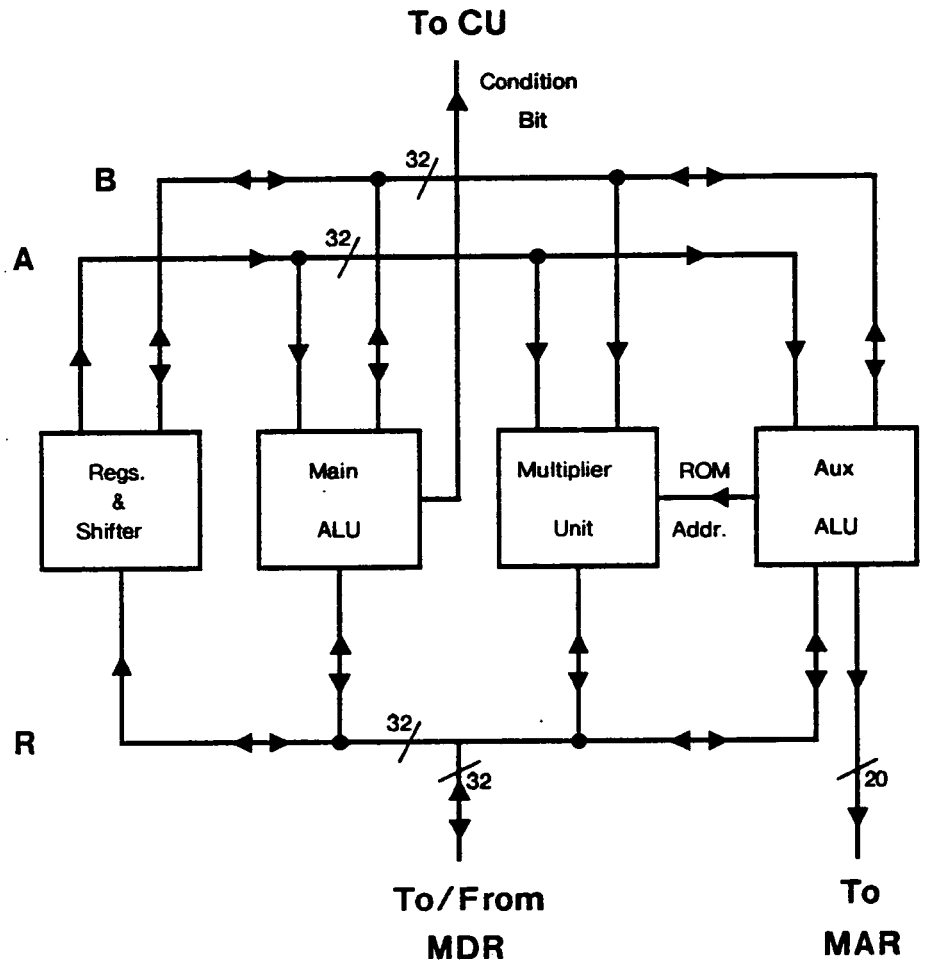

Figure 3-11: RISP's Processing Unit

The PU of RISP resembles that of Lincoln Laboratories' LSP 2 [31]. Both are designed with a three-

bus architecture where units are in parallel, figure 3-11 shows the block diagram of RISP's processing unit (PU). Each unit is able to complete an operation within a single μcycle. A big advantage of this design is its regularity; hence, it is easy to both simulate and expand it. These characteristics imply that we are able to use hardware design aids available at C-MU (such as ISPS) as well as to bring the system up slowly. For example, we could build an early prototype without the multiplier unit and with only one ALU, to be added after this version is fully debugged.

Figures 3-12 through 3-15 show the block diagrams of the PU elements which are described in the following paragraphs.

Registers and Shifter

This element contains all 32 general purpose registers available to the assembly language programmer. A barrel shifter is included which is capable of performing any logical or arithmetic shift in one μcycle which is very useful when normalizing floating-point numbers. Note that the register file is two-ported; hence, in the same cycle, a register can be shifted by a number of bits determined from a previous operation while the A bus is being loaded with the contents of another register. It is worth mentioning that the register file has three-address capabilities where two registers might be read while yet another one is written.

Main ALU

There are two ALUs in the PU. Both are 32-bit wide and both are designed around AMD's AM2903 bit-slice; figure 3-13 shows the *main ALU*. The main ALU may be thought as part of the data processing section of the PU, it keeps both flag registers (RISP's and the microengine's) therefore being responsible of sending the condition bit to the CU. The main ALU also performs the bit manipulations (set, clear, test and toggle).

Note that the main ALU has 16 registers used to hold partial results; as a matter of fact, all the elements of the PU have a register file, this is to avoid excessive contentions for the A, B and R buses. Both the main and the auxiliary ALU's also operate in a three-address fashion where operations of the type **A.op.B→C** are permitted. For more information about how this is done or about the AM2903/2910 parts, consult the AM2900 Family Data Book [32] or chapters II, III and IV of the book Bit-Slice Microprocessor Design [33].

Multiplier Unit

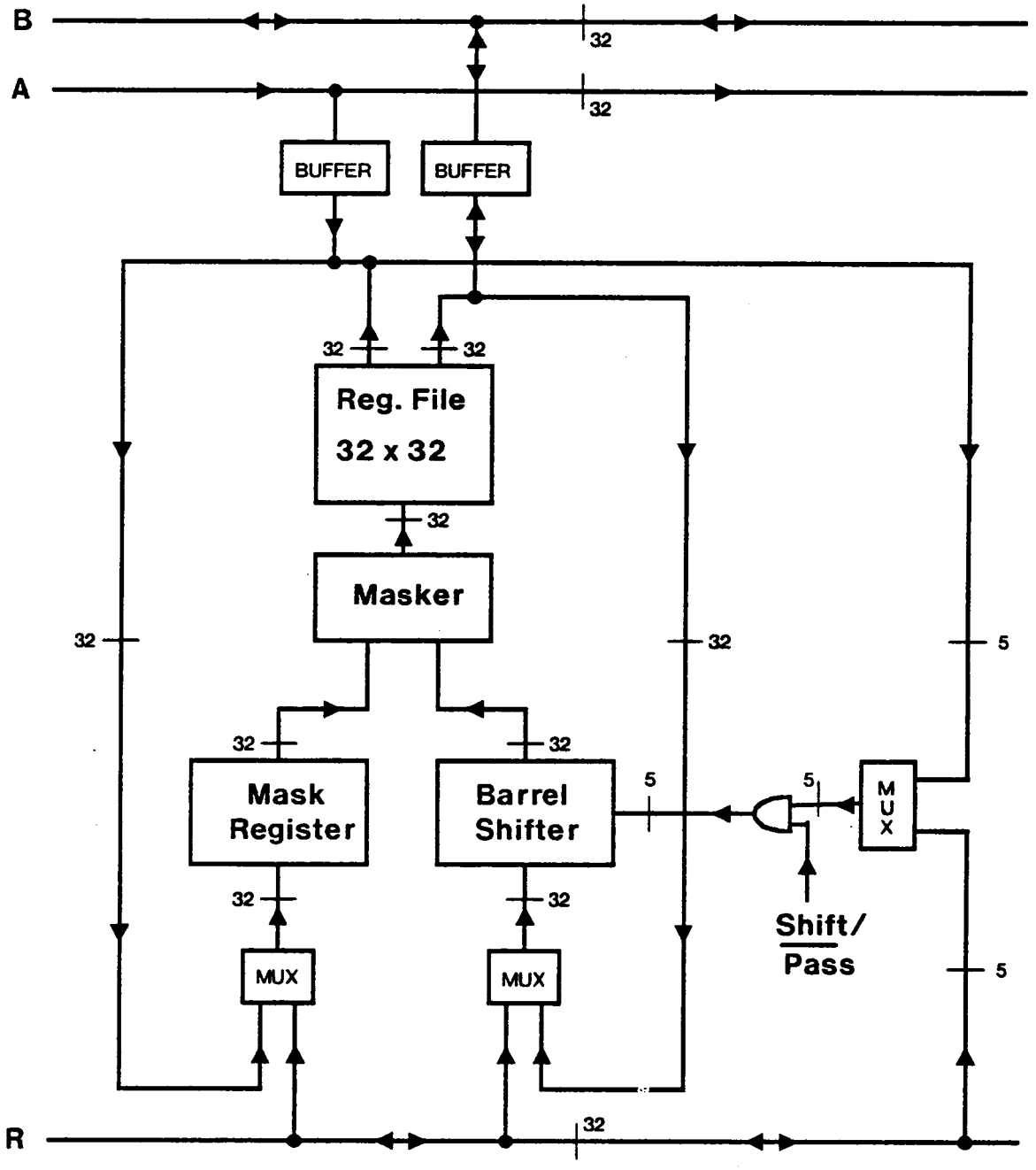

Figure 3-12: Registers and Shifter

RISP has a high-speed multiplier unit built around TRW's TDC-1010J multiplier-accumulator. This

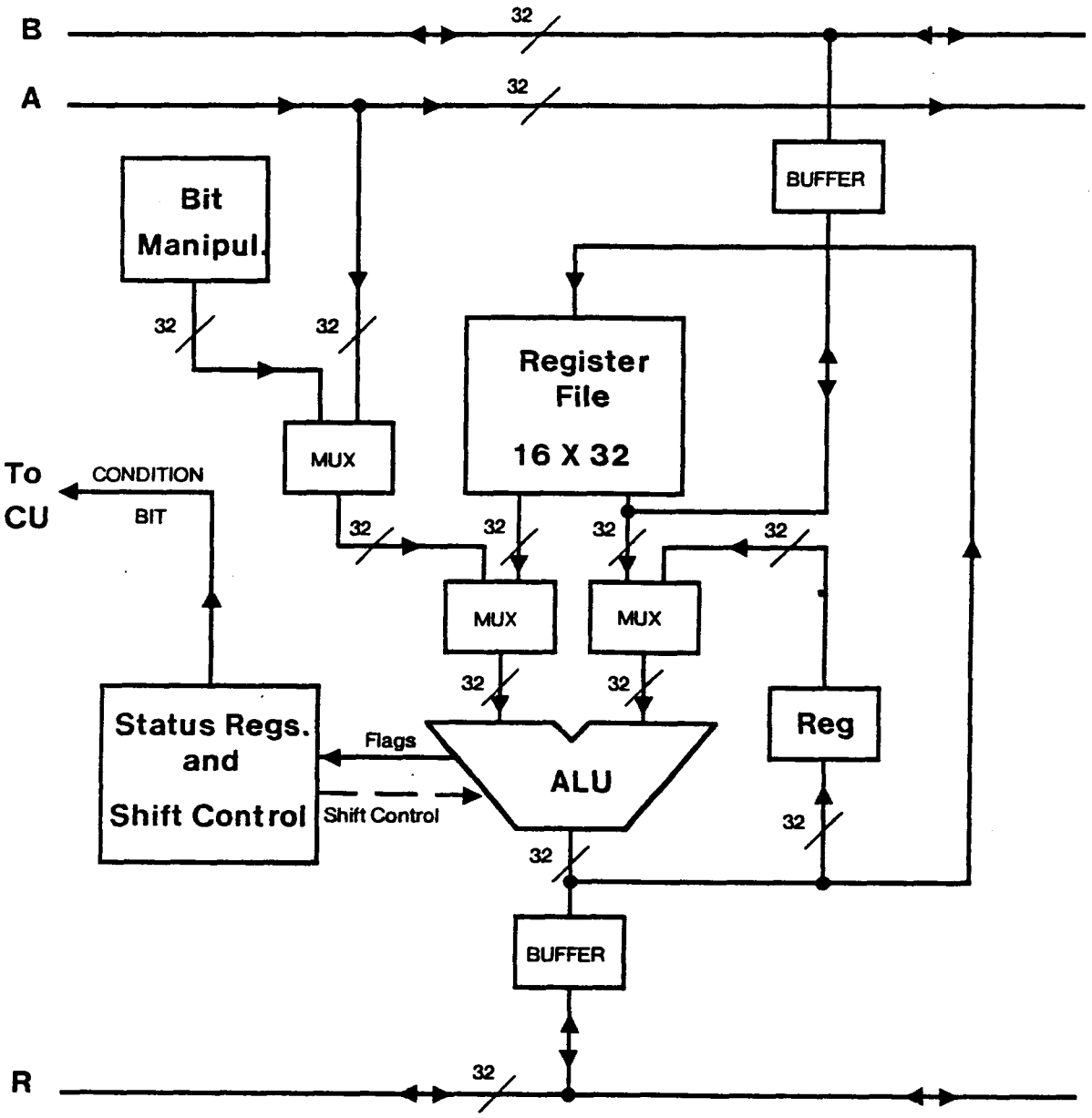

Figure 3-13: Main ALU

unit is capable of performing a 16 x 16 multiplication and 35-bit addition in a single μcycle. It also has a ROM with trigonometric and other important constants which is addressed by the auxiliary ALU. Figure 3-14 shows the block diagram of this unit.

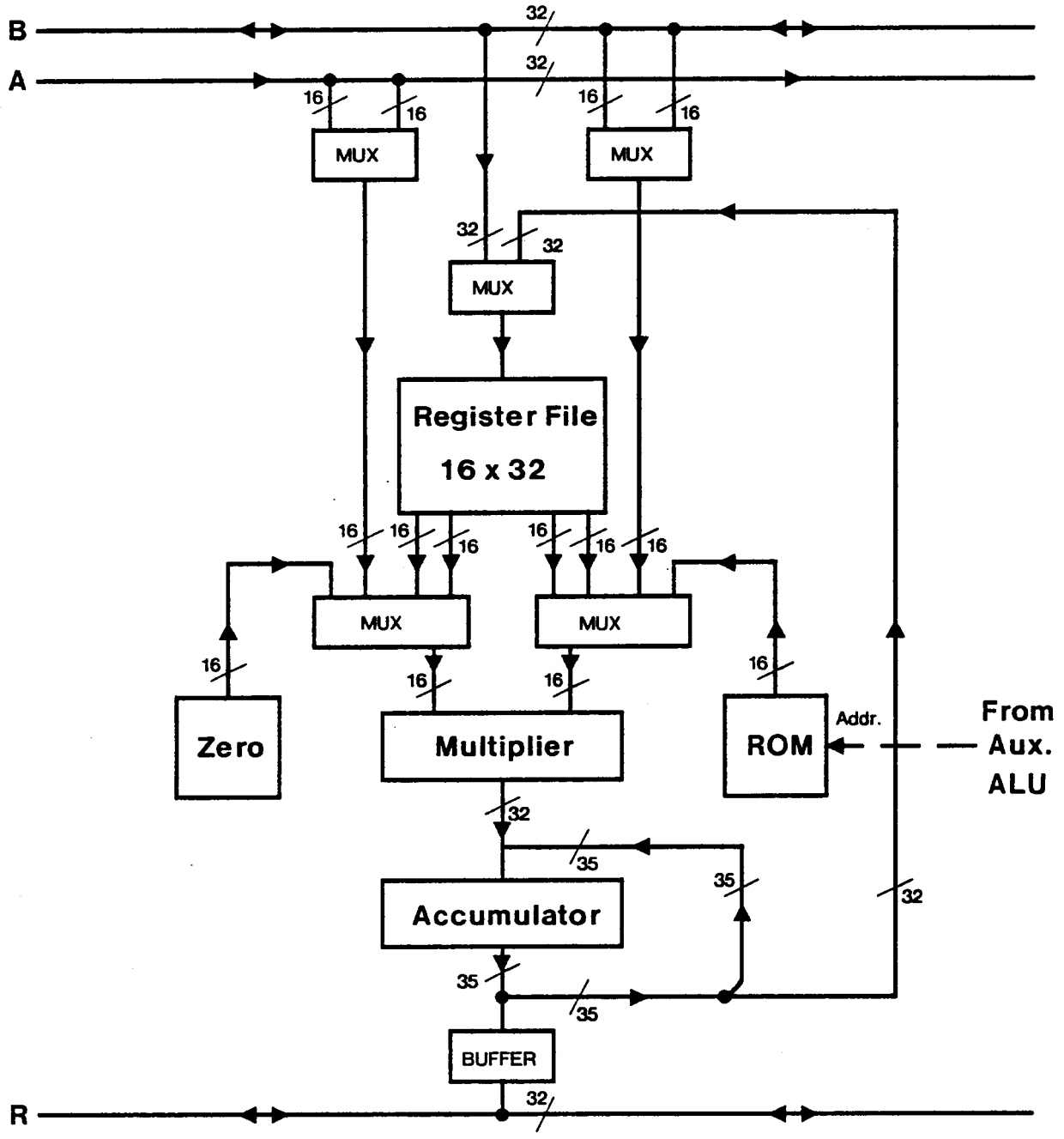

Figure 3-14: Multiplier Unit

Auxiliary ALU

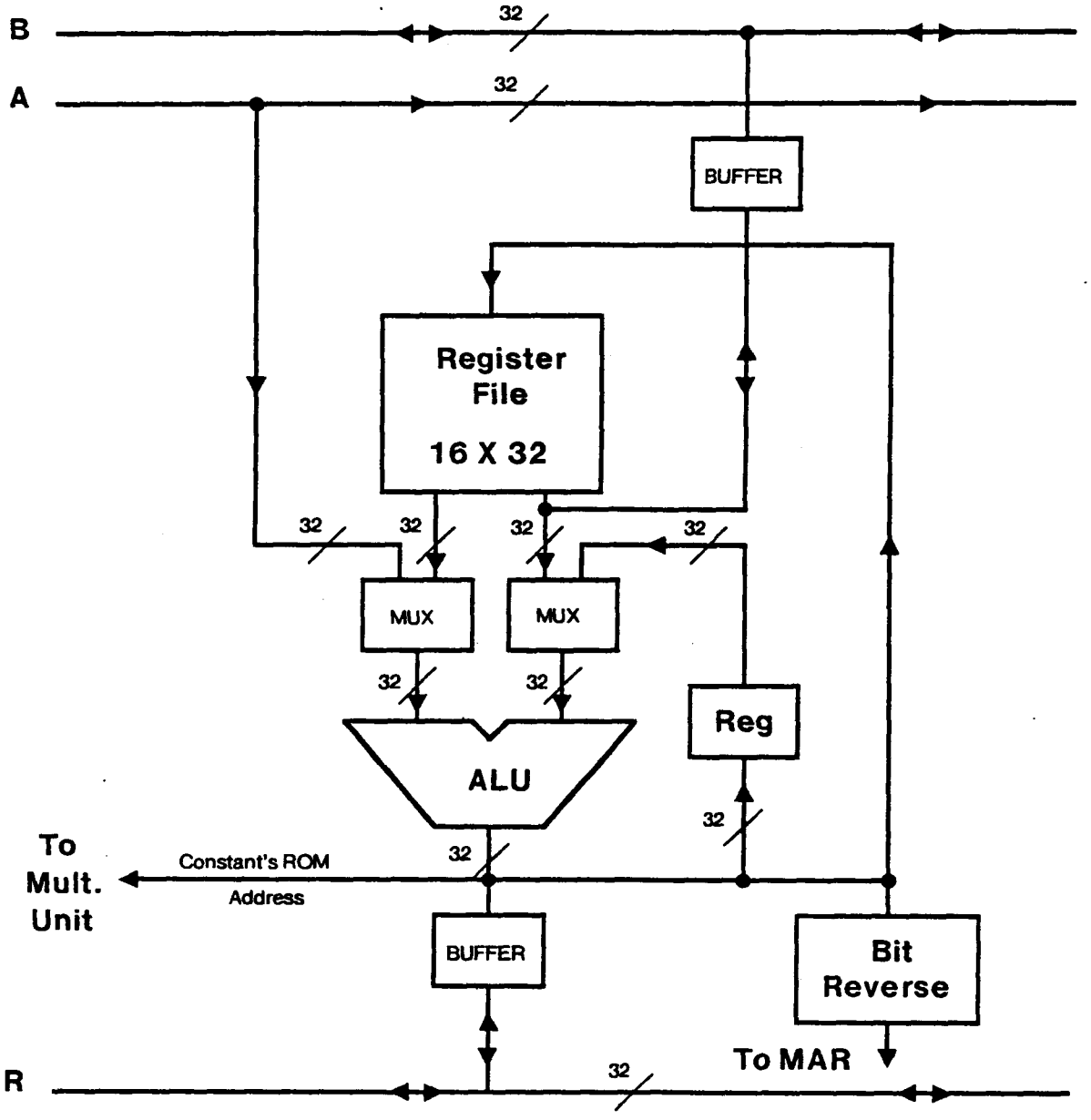

Figure 3-15: Auxiliary ALU

Finally, the fourth element of the PU is another 32-bit ALU, the *auxiliary ALU*, its block diagram is shown in figure 3-15. This ALU keeps RISP's program counter and stack pointers. In general, the auxiliary ALU is in charge to generating the address of the next fetch. The Memory Address Register (MAR) is loaded by this element.

Two other features about the auxiliary ALU should be mentioned: First, it generates the addresses of the constants' ROM of the Multiplier Unit and second, it has a bit-reverse unit before the MAR register. This unit, implemented via multiplexer chips can reverse any or all of the 16 low-order bits of the data being sent to the MAR. Hence, an up-to-64K-point FFT may be calculated without introducing extra cycles for bit-reversing the address.

3.4.2 Bus Interface

3.4.2.1 General description

RISP was conceived with several on-going projects in mind. Research in hierarchically modeling the EEG signal [19] requires complex computations on tremendous amounts of data; fortunately, this data comes at a very low speed. Speech analysis and recognition demands high computational speed in order to be performed in real-time. The robotics image processing problem described in section 1.2 requires simple computation on data with very high bandwidth. A considerable degree of flexibility is required so RISP can function in a variety of task environments. Part of this flexibility was achieved by making RISP able to communicate to any machine via a microprogrammable *bus interface* (BI).

Figure 3-16 shows the block diagram of the BI. Data is always stored (or retrieved) by DMA; this means that the CPU doesn't waste time moving data to/from global memory, I/O devices, etc. The BI, which timing is dependent on the host's (virtual) bus and not on RISP's timing, can execute up to 64 "macro" instructions like block moves, memory fill, etc. Parameters and flags are passed to/from the BI via two uni-directional register files.

3.4.2.2 An example - Interfacing to MULTIBUS

The bus interface makes it possible for RISP to enhance the computing capabilities of machines like the PDP-11 (Unibus), LSI-11/23 (Q-bus), PERQ, etc. As an example let's consider the addition of RISP to the system depicted in figure 3-1 which is currently being utilized at C-MU's Image Processing Laboratory and uses MULTIBUS boards.

The MULTIBUS was developed by Intel Corp. for their SBC series of computers; nowadays, it has become a *de-facto* standard for microcomputers and is in the process of standardization by the IEEE. MULTIBUS has a 16-bit bi-directional data bus and a 20-bit address bus. Transfers between the MC68000 and RISP would be done in fixed-length blocks of 16-bit words. The address generators of the BI can increment by one, two or four so, for byte addressable memory, it can transfer bytes, words or long-words.

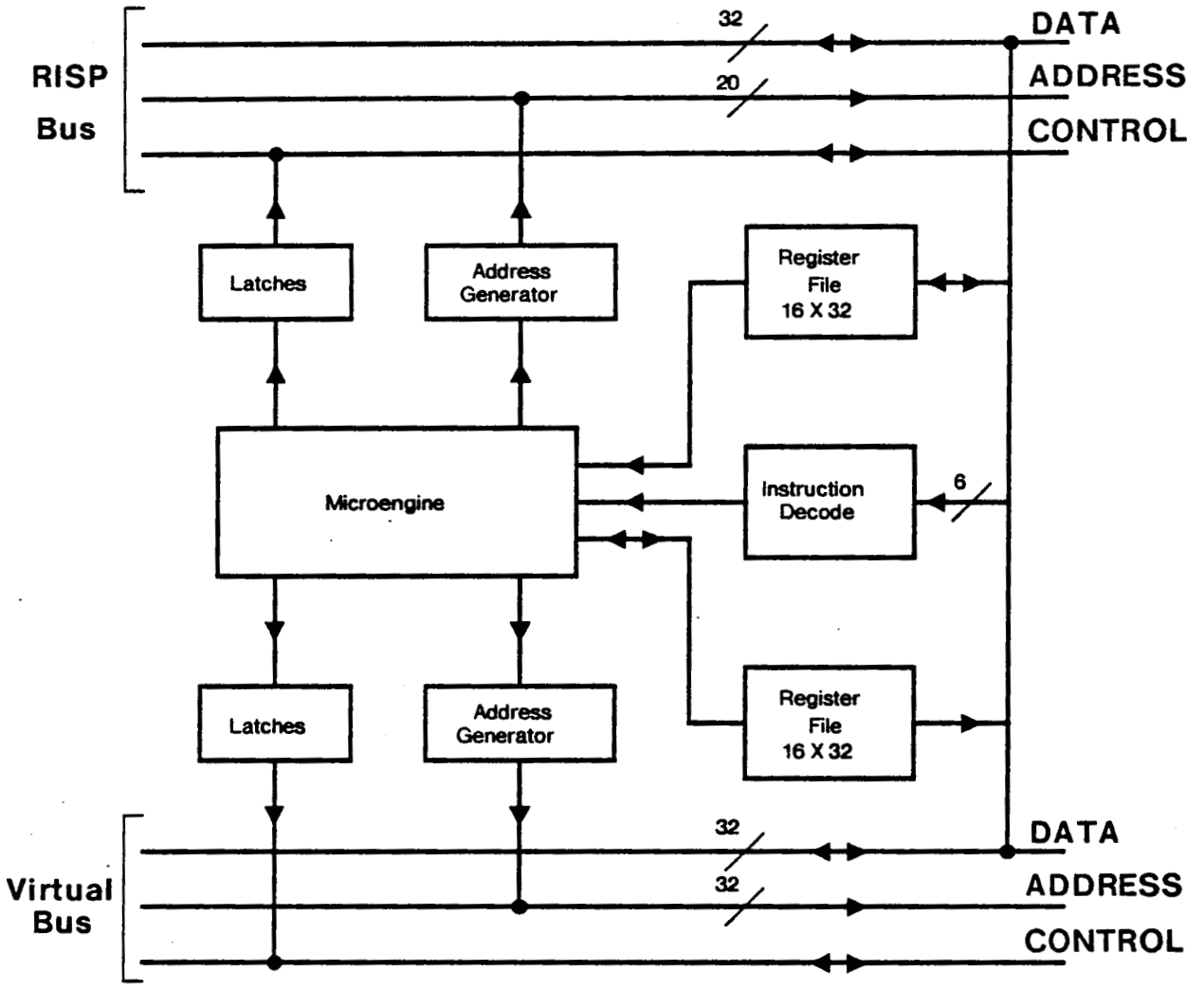

Figure 3-16: RISP's Bus Interface

The important signals that will be generated by the BI are†:

- **MRDC (MWTC).** Memory read (write) commands. These signal the memory whether the operation is a read or a write.
- **IORC (IOWC).** I/O read (write) commands. Similar to the commands discussed above.

† All signals are negative-true, i.e. they are asserted low

- **BPRO.** Bus priority out. Asserted when BPRN is also asserted (see below) and RISP doesn't need to make a bus transfer.

- **BUSY.** Bus busy. Asserted by the master that controls the bus.

- **BREQ.** Bus request. Asserted by RISP when it needs to make a transfer and BPRN is also asserted. It is synchronized with BCLK (below).

The BI would be monitoring the following signals to decide when to assume mastership:

- **BPRN.** Bus priority in. Asserted by the master that has the next higher priority than RISP if it doesn't need the bus.

- **BUSY.** Bus busy. See above.

- **XACK.** Transfer acknowledge. Asserted by the slave when the transfer is completed. RISP wouldn't initiate another transfer until the previous one is properly terminated by a XACK unless AACK is asserted (see below).

- **AACK.** Advanced acknowledge. The assertion of this line by a slave will cause RISP not to wait for XACK.

- **BLCK.** Bus clock. Used to synchronized bus events (like the assertion of BREQ,) it may be stretched, single-stepped or halted.

3.4.3 RISP's performance

Low-pass filtering	589,824 cycles ⇒ **110.592 ms**
1024-point FFT	40,960 cycles ⇒ **7.68 ms.**
AR coefficients	4,999 cycles ⇒ **937.31 μs.**
Adaptive segmentation	131,072 cycles ⇒ **24.576 ms**

Table 3-5: RISP execution times

Table 3-5 gives RISP execution times for the same four algorithms we've been using. Note that these times are the result of careful analysis of the RISP architecture. The actual timing of RISP is implemented with AMD's AM2925 clock generator which enables us to have different microinstruction execution times. For the times shown in table 3-5 however, a *fixed* μcycle time of 187.5 ns was assumed. This seems to be a reasonable value according to the architecture's propagation delays.

	Single 68000	PERQ	Six 68000 (RAPID)	ASP	RISP
Low-pass filter	52.300	15.408	8.716	0.533	1.000
1024 point FFT	35.667	8.667	6.944	0.267	1.000
AR Coeffs.	46.537	11.367	7.916	0.532	1.000
Adapt. Segm.	10.667	2.000	1.778	0.800	1.000

Table 3-6: Relative execution times

Finally, table 3-6 shows the relative execution times of the five architectures evaluated. There, a value of one was arbitrarily assigned to RISP's performance. It should be noted that RISP performs extremely well for complex calculations. The last measure, the adaptive segmentation algorithm, is a very good comparison of the "raw" speed of the processor since the algorithm consists of simple operations (additions and subtractions).

4. The proposed multiprocessor system - RIP 1

As was discussed briefly in chapter 1, there are, in the robot problem, four stages of computation: feature extraction, image modeling, pattern recognition and control (figure 1-1). In this chapter, a multiprocessor architecture, the **RIP 1**, is presented. We will show how it can be used to solve all those tasks and therefore be used as a robot's *"brain"*.

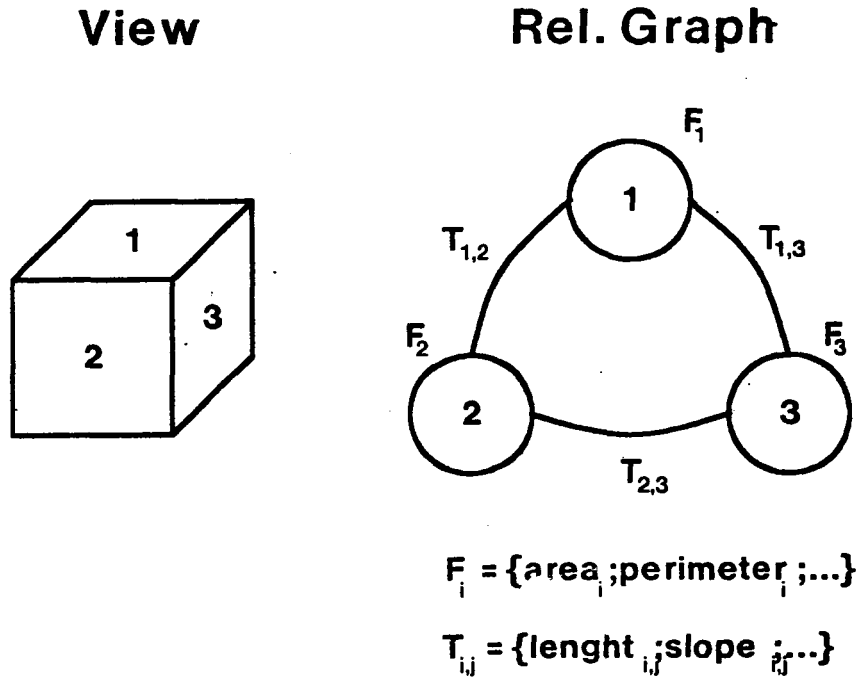

Figure 4-1: View and graph of a simple cube

The first two stages (feature extraction and image modeling) could be performed by using the adaptive segmentation procedure discussed in section 2.8. This procedure identifies the *blobs* within the image at the same time it gives statistical properties, or *attributes*, for each one of them. It has been shown [1, 3] that when an image is segmented, the blobs tend to be elongated in the direction of the segmentation. To avoid this problem, we would run the adaptive segmentation algorithm in both row and column directions *simultaneously*. This has the added advantage of providing us with two independent sets of attributes for each blob. However, the segmentation becomes a two-step procedure: in the first step the models are obtained for each row (column) independently, while in the

second step the models are clustered to define the blobs. Also, it is possible that the blobs have ragged edges so a relaxation algorithm is included, along with the clustering, in the second step. The image modeling task is completed by obtaining a relational graph representation of the image where nodes represent blobs and arcs represent edges. A set of attributes is associated with each node or arc of the graph (see figure 4-1). It is important to note that such image regions do not necessarily correspond to physical object boundaries, but may depend on light, shading, texture, etc. The recognition algorithms are based on training sets of images for a variety of such conditions rather than detailed physical interpretation of image content.

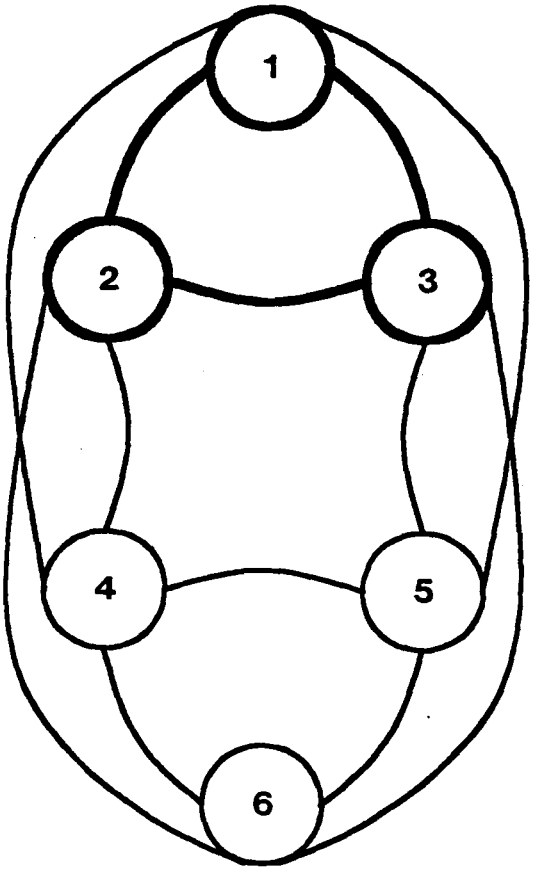

Figure 4-2: World graph for the cube's graph (thicker trace)

The third stage, the pattern recognition, involves classifying the previously obtained graph as a *subgraph* of a member of a graph-dictionary stored in memory (figure 4-2.) Each member of the

dictionary, called a *world graph*, represents an object; it defines clearly its structure for *every possible view* of the object. Then the aforementioned graph obtained from the image should be a part of the world graph of that object. Some nodes of the world graph will not appear in the view graph due to the 3-D to 2-D mapping inherent in the image acquisition. Note that the problem becomes more complex when the image contains several objects since the view graph is not a subgraph of a world graph but, instead, a *subgraph of the view graph* is. Obviously, the world graph is a "normalized" version in terms of node/arc attributes, that way the same graph may represent objects that only vary in color, size, etc.

The last stage, control, will be done by looking at the attributes of the graph given we know its type. Then, for example, orientation and remoteness might be inferred by looking at the particular subgraph and the size attributes (the farther the object, the smaller it seems.) Sanderson and Weiss [34] have shown that visual servo control can be done by using the graph error signals.

Figure 4-3 shows the logical arrangement of the tasks discussed previously. RIP 1 is a multiprocessor system designed to perform these tasks in real-time, figure 4-4 shows the block diagram of RIP 1. A few characteristics are worth mentioning: it has two RISPs whose image memories are loaded simultaneously by a frame grabber. Each one can perform the adaptive segmentation algorithm on a 256 x 256 image in less than 33 ms so a RISP is dedicated to the row modeling while the other works, on the same image, to obtain the column models. The results are loaded in global memory so an MC68000 processor can perform the clustering/relaxation on them. Two more MC68000s are utilized for the formation and recognition of the graph; this includes selection of the appropriate attributes that will be passed to the last MC68000 which is in charge of the visual servoing. Commands to a robot may be given every 100 ms approximately.

As a final note, we should emphasize that there are two key features that make RIP 1 so effective for this task: The RISP processors that can keep up with the extremely high bandwidth of the raw data, and the RAPID bus which allows us to have the brute speed of two RISPs and the sophistication of four MC68000 in one system.

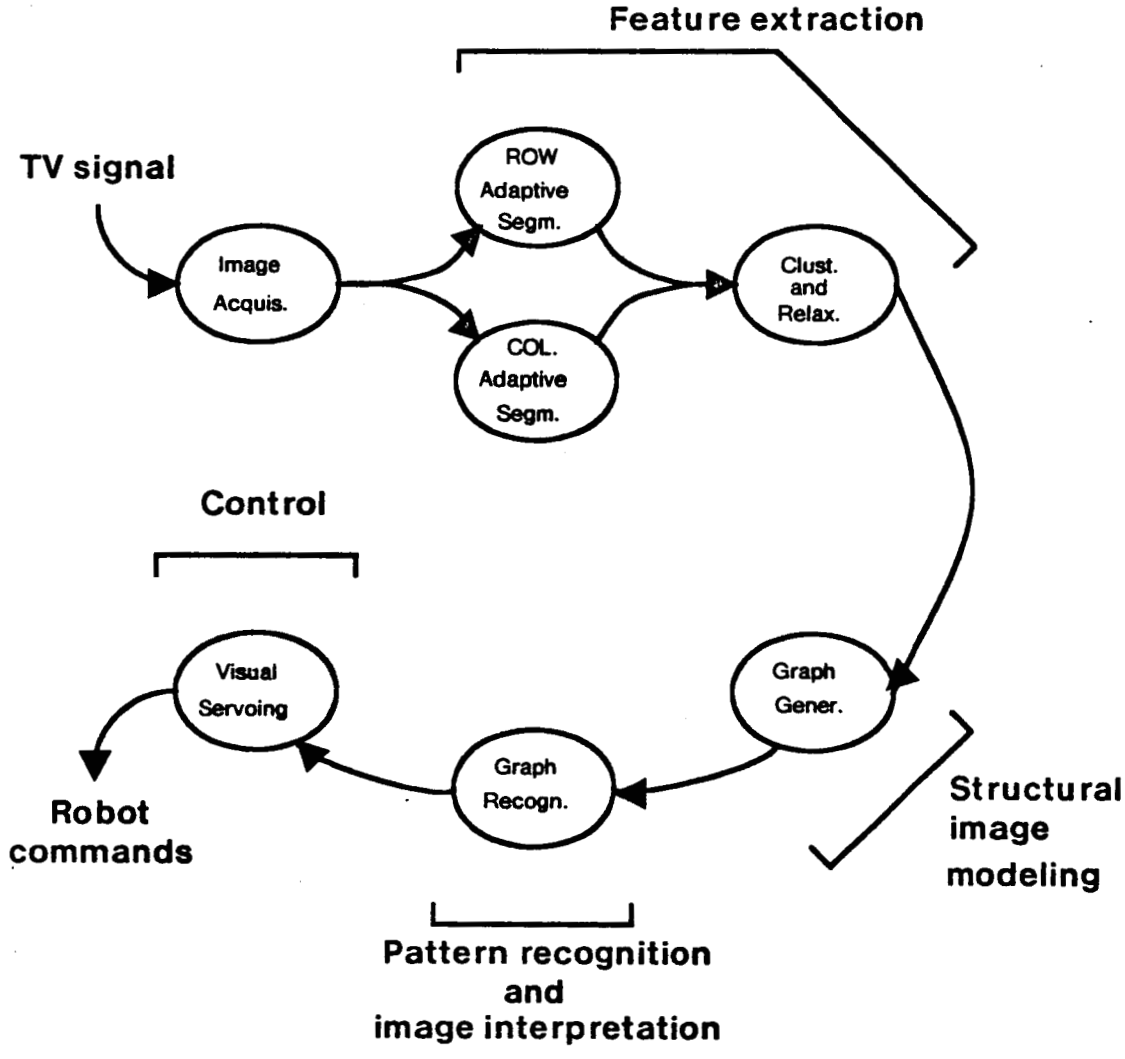

Figure 4-3: Computational tasks in the robot problem (expanded view)

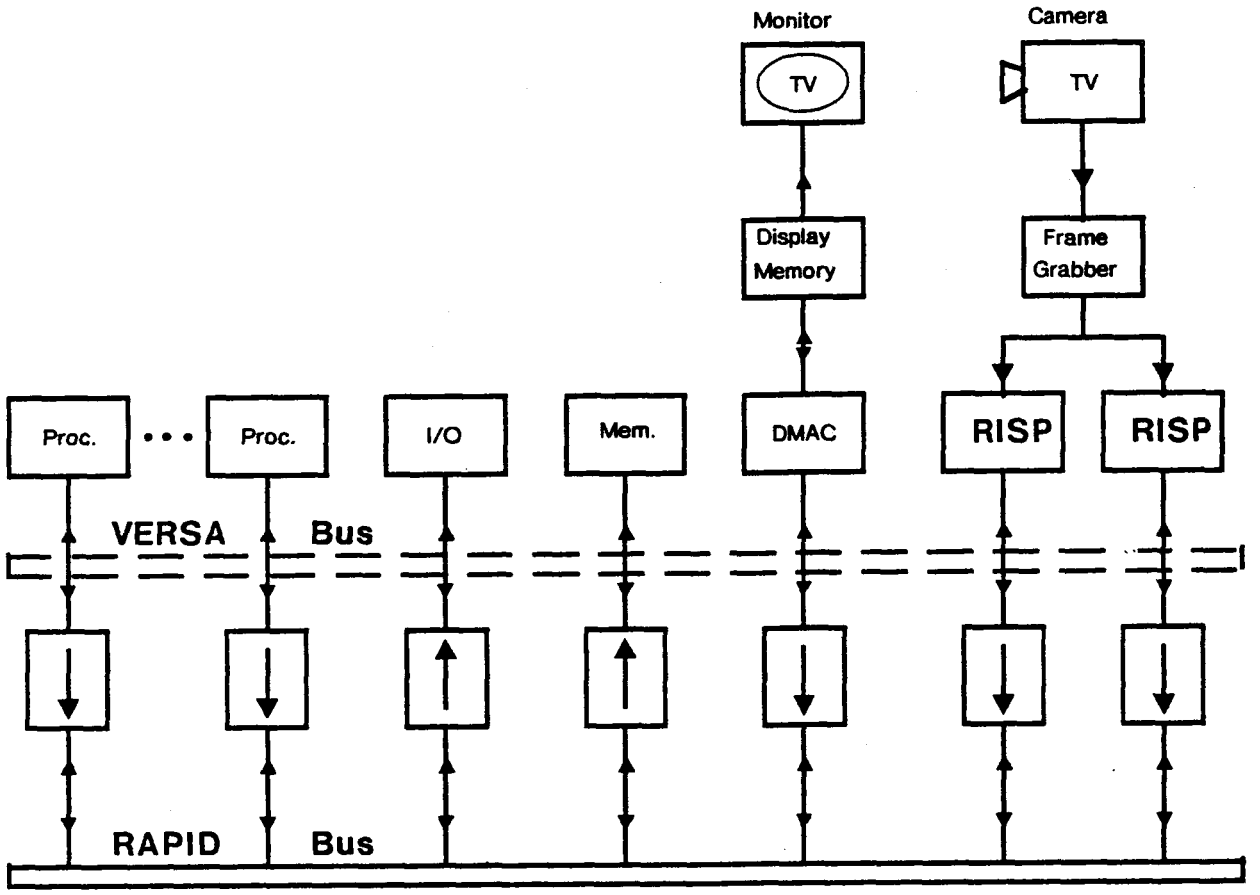

Figure 4-4: The RIP 1 multiprocessor system

5. Summary

In this report, we have discussed computing hardware for vision-based robot control. We have concluded that, in order to solve such a problem in near real-time, computing facilities capable of performing image preprocessing in less than 33 ms are required.

A few traditional image preprocessing algorithms are reviewed; we discuss their computational complexity and their application to the robotics image processing problem. In particular, we suggest that work on adaptive image modeling is well-suited for this application. An algorithm that segments the image by detecting abrupt changes in the mean of the brightness function is discussed. The inner-loop of this algorithm requires three additions and one comparison per pixel.

A few of the more prominent architectures used for image preprocessing have been surveyed. For each one of them we presented approximate execution times of four representative algorithms: a low-pass filtering operation performed by convolving a 256 x 256 image with a 3 x 3 mask; a one-dimensional 1024-point FFT; the generation of a 10th order AR model from 100 data points via the Lattice method; and the execution of the adaptive segmentation's inner-loop on a 256 x 256 image.

The Robotics Institute Signal Processor (RISP) is a bit-slice microprogrammable computer designed as a cost-effective alternative to array processors. It was shown that RISP is 10 to 52 times faster than the MC68000 and only 2 to 2½ times slower than ASP, IBM's array processor. The preliminary architecture of RISP was presented; it was shown that its high-performance CPU and its microprogrammable Bus Interface (BI), give RISP enough power and flexibility to attack a wide variety of problems.

A proposed multiprocessor system, *RIP 1*, consisting of four MC68000 and two RISP processors joined via a RAPID bus, was discussed. Evaluation of RIP 1 suggests that this system may accomplish the goal of coordinating a vision-based robot in real-time although the system would be equally suitable for less-demanding vision problems such as visual inspection problems.

References

1. Rafael C. Gonzalez and Paul Wintz, *Digital Image Processing,* Addison-Wesley Pub. Co., Reading, MASS., Applied Mathematics and Computation, Vol. 13, 1977.

2. J. S. Weszka, et al., "A threshold selection technique," *IEEE Transactions in Computers,* Vol. C-23, No. 12, December 1974, pp. 1322-1326.

3. William K. Pratt, *Digital Image Processing,* John Wiley and sons, New York, N.Y., 1978.

4. John W. Woods and Clark H. Radewan, "Kalman Filtering in Two Dimensions," *IEEE trans. on inf. theory,* Vol. IT-23, No. 4, July 1977, pp. 473-482.

5. John W. Woods, "Correction to: Kalman Filtering in Two Dimensions," *IEEE trans. on inf. theory,* Vol. IT-25, No. 5, September 1979, pp. 628-629.

6. John W. Woods and Vinay K. Ingle, "Kalman Filtering in Two Dimensions: Further Results," *IEEE trans. on Acous., Speech and Sign. Proc.,* Vol. ASSP-29, No. 2, April 1981, pp. 188-197.

7. L. G. Roberts, "Machine perception of three dimensional solids," *Optical and Electro-optical information processing,* J. T. Tippett, et al., eds., MIT press, Cambridge, MASS, 1965, pp. 159-197.

8. I. E. Abdou, "Quantitative methods of edge detection," USCIPI Report 830, University Southern California, 1978.

9. J. M. S. Prewitt, "Line detection by local methods," in *Picture Processing and Psychopictories,* B. S. Lipkin and A. Rosenfeld, eds., Academic Press, New York, N.Y., 1970.

10. A. Rosenfeld and M. Thurston, "Edge and curve detection for visual scene analysis," *IEEE trans. Computers,* Vol. C-20, May 1971, pp. 562-569.

11. A. Rosenfeld and M. Thurston, "Edge and curve detection: Further experiments," *IEEE trans. Computers,* Vol. C-21, July 1972, pp. 677-715.

12. S. L. Horowitz and T. Pavlidis, "Picture Segmentation by a Tree Traversal Algorithm," *J. Ass. Comp. Mach.,* Vol. 23, No. 2, 1976, pp. 368-388.

13. G. P. Ashkar and J. W. Modestino, "The contour extraction problem with biomedical applications," *Comput. Graph. Image Processing,* Vol. 7, No. 3, 1978, pp. 331-355.

14. M. Basseville, et al., "Edge detection using sequential methods for change in level-Part I," *IEEE trans. on Acoust., Speech and Sign. Proc.,* Vol. ASSP-29, No. 1, February 1981, pp. 24-31.

15. B. R. Hunt and T. M. Cannon, "Non stationary assumptions for Gaussian models of images," *IEEE trans. Syst., Man and Cyber.,* Vol. SMC-6, June 1976, pp. 876-882.

16. A. K. Jain, "Some new techniques in image processing," *Proc. Image Sci. Math.,* C. O. Wilde and E. Barrett, eds., November 1976, pp. .

17. M. Basseville and A. Benveniste, "Changes in Statistical Models: Various Approaches in

Automatic Control and Statistics," Rapport de Recherche de l'INRIA 145, Universite de Rennes, March 1981.

18. M. Basseville, "Edge detection using sequential methods for change in level-Part II," *IEEE trans. on Acoust., Speech and Sign. Proc.,* Vol. ASSP-29, No. 1, February 1981, pp. 32-50.

19. A.C. Sanderson, J. Segen and E. Richey, "Hierarchical Modeling of EEG Signals," *IEEE trans. on Pattern Analysis and Machine Intelligence,* Vol. PAMI-2, No. 5, September 1980, pp. 405-414.

20. J. Segen and A. C. Sanderson, "Detecting Change in a Time-Series," *IEEE trans. on inf. theory,* Vol. IT-26, No. 2, March 1980, pp. 249-255.

21. A. C. Sanderson and J. Segen, "A pattern-directed approach to signal processing," *Proceedings of the 5th. International Conference on Pattern Recognition,* Miami, FL, December 1980, pp. 613-617.

22. Anil K. Jain, "Advances in mathematical models for image processing," *Proceedings of IEEE,* Vol. 69, No. 5, May 1981, pp. 502-528.

23. L. R. Rabiner and R. W. Schafer, *Digital Processing of Speech Signals,* Prentice-Hall, Englewood Cliffs, N.J., Signal Processing series, 1978.

24. Peter Freeman, *Software Systems Principles,* SRA, Inc., Chicago, Ill., Computer Science Series, 1975.

25. M. P. Zoccoli, "Design of a Synchronous Multi-microprocessor Computer System for Signal Processing," Master's thesis, Carnegie-Mellon University, December 1979.

26. M. P. Zoccoli and A. C. Sanderson, "Rapid Bus Multiprocessor System," *Computer Design,* November 1981, pp. 189-200.

27. R. Bracho, J. C. Willis and A. C. Sanderson, "RAPID-bus preliminary specification," Tech. report In preparation, Carnegie-Mellon University, 1982.

28. Motorola Semiconductor Products, Inc., *VERSA bus Specification Manual,* Third ed., Phoenix, AZ, 1981.

29. Mario R. Barbacci, "The ISPS Computer Description Language," CSD tech. report, Carnegie-Mellon University, August 1979.

30. Mario R. Barbacci, "An ISPS simulator," CSD tech. report, Carnegie-Mellon University, January 1980.

31. L. R. Rabiner and B. Gold, *Theory and Applications of Digital Signal Processing,* Prentice-Hall, Englewood Cliffs, N.J., Signal Processing series, 1975.

32. Advanced Micro Devices, "The AM2900 Family Data Book,"

33. J. Mick and J. Brick, *Bit-slice Microprocessor Design,* Mc.Graw-Hill, New York, NY, 1980.

34. A. C. Sanderson and L. E. Weiss, "Image-based visual servo control using relational graph error signals," *Conference on Cybernetics and Society,* October 1980, .