A MICROPROCESSOR IMPLEMENTATION OF AN IMAGE ENHANCEMENT/TRANSMISSION SYSTEM

by

RALEIGH CEDRIC GALLINGTON

B.S. Massachusetts Institute of Technology

(1978)

SUBMITTED TO THE DEPARTMENT OF ELECTRICAL ENGINEERING IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREES OF

MASTER OF SCIENCE

and

ELECTRICAL ENGINEER

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

August 1981

Signature of Author _____
Department of Electrical Engineering
August 7, 1981

Certified by _____
William F. Schreiber
Thesis Supervisor

Accepted by _____
Arthur Smith
Chairman, Department Committee

A MICROPROCESSOR IMPLEMENTATION OF AN IMAGE ENHANCEMENT TRANSMISSION SYSTEM

by

RALEIGH CEDRIC GALLINGTON

Submitted to the Department of Electric Engineering on August 7, 1981 in partial fulfillment of the requirements for the Degree of Master of Science and Electrical Engineer.

ABSTRACT

This thesis describes the design (implementation) and performance of an image enhancement/transmission system and involves the use of a standard Intel 8080 or 8085 microprocessor system. Given image data at 8kb/s corresponding to values obtained by raster scanning the image, performs a 2:1 data compression using psychovisual coding techniques and presents this data to a communication channel of 4kb/s capacity. This channel could be, for example, a standard voice communication channel. The receiver portion of the system reconstructs an image from this data that is visually superior to the original image. The transmitter and receiver portions of the system are the subject of this thesis.

Thesis Supervisor: William F. Schreiber

Title: Professor of Electrical Engineering

Acknowledgements

I wish to thank Professor Schreiber for proposing, supervising, and sponsoring this work. I also wish to thank Lakshmi Dasari and James Cyr for their suggestions for and assistance in the construction of the hardware, respectively, and Deloris Butler for her invaluable assistance in the preparation of this paper.

Raleigh Cedric Gallington

TABLE OF CONTENTS

TABLE OF CONTENTS

TABLE OF FIGURES

TABLE OF FIGURES

INTRODUCTION

The image enhancement/transmission system described herein consists of an 8080 microprocessor implemented image enhancer/transmitter (transmitter) and an image receiver/reconstructor (receiver)*. Although a discussion of the performance of one without the other is not very meaningful, their design and operation can be discussed seperately once the system's specifications have been discussed.

The enhancement techniques used in the system have been verified by, and rely heavily on, results from the theses of Curlander (1), Gilkes (2), and Hoover (3). Curlander and Gilkes discussed enhancement techniques while Hoover presented and discussed the performance of a system whose overall approach and operation is similar to that of this system. These results will be summarized in a later section.

The function of this system is to perform a 2:1 data compression of the information from a digitized image and transmit it to a remote receiver without degrading the subjective visual quality of the resultant image and possible even improving it. The basic approach involves first separating the high and low spatial frequency components (content) of the image, then performing the data compression on each independently. Thus it is not necessary to find a single algorithm which can be successfully applied to the entire image content without failing to meet the

---

* The transmitter and receiver could also be described as a psychovisual coder and decoder respectively. see ref. (T.H. Huang)

system objectives.

The low frequency components(lows) can be compressed by simply subsampling the values since, by their very nature, these values vary spatially quite gradually. Just how gradually they vary depends on the frequency response of the filter used to extract them. Thus the lows value for a given pel (pixel) location will not be much different than the corresponding values at nearby pel. Thus, not much error is introduced if a particular pel value is also assumed to be the value at all of the nearby* pels. Since that value would then specify the pel values for that entire area only that one pel value need be sent to the receiver for it to reconstruct the area. The error, although small, would be highly structured spatially (as is typical of quantization noise), and its visibility would tend to be emphasized as spurious contours as the spatial sampling rate is decreased; thus limiting the degree of subsampling allowable. This piecewise constant approximation can be readily improved upon by somewhat smoothing the transition from one value to the next. By assuming the lows to be spatially piecewise linear rather piecewise constant, a greater degree of subsampling can be allowed, and since linear interpolation can be easily reproduced in the receiver this is achieved without any increase in the amount of information to be transmitted. A gaussian interpolation would allow subsampling

---

* "nearby" is relative to filter length

to an even greater degree but would also be considerably more difficult to implement, especially in the receiver. In fact, for the amount of data compression desired for this system it is not worthwhile to go beyond the linear interpolator.

The high frequency components, on the other hand, vary quite rapidly spatially and thus do not lend themselves readily to subsampling. Instead, data compression is achieved by quantizing (elimination of the lower order bits) the original 8-bit high pel values to 3 bits. To reduce degradation of the resultant image, the highs are adaptively enhanced and companded before quantization. Noise processing (dithering) at both the transmitting and receiving ends, using Robert's techniques (7), is also used to reduce the visible effects of this quantization, such as edge contours*. The enhancement procedure involves two scaling operations. The first is a scaling of the highs pel values according to the luminance (value) at that pel location in the original image. This is then followed by a scaling according to some measure of the amount of detail in the area, i.e. the amount of edge information. These two procedures are hereafter referred to as the luminance and detail scaling functions, respectively.

The receiver has the task of reconstructing the image from the transmitted lows and highs. The transmitter carries the brunt of the computational load, largely leaving the receiver to only correctly align and add the highs and lows. It must also expand the highs, complete the noise processing, and inter-

* Note that it is impossible to use an average of four or more bits per highs value, transmit the subsampled lows, and still keep pace with the incoming image data.

polate the lows values, but these operations are relatively straightforward compared to those required in the transmitter. After these are completed, for each given pel location, the 8-bit resultant data is presented to a facsimile reproduction device at a rate of 1-kbyte/s (8kb/s); the same as the data rate from the scanning device to the transmitter.

II. Background

Before further discussion of the actual implementation of the system, a summary of some of the theoretical and empirical bases for the techniques just mentioned is in order. These techniques are used to first achieve image quality improvement by means of increasing the sharpness and decreasing the noise (9), and then to perform data compression with minimal effects on the resultant image quality (11). While the section does not delve much into the details of human visual perception in general, it does make use of some important findings.

II-1 Enhancement

As mentioned earlier, the system relies heavily upon the findings of Gilkes (2), Curlander (1), and Hoover (3). In Gilkes' work digital unsharp masking* was used to optimally sharpen images, an improvement on direct linear amplification of the edge information (highs). In this technique an unsharp (or slightly fuzzy) mask from the original** is combined with the original in such a way as to achieve a form of spatial high pass filtering. A scale factor is determined from the resulting image data at each pel then applied to the corresponding pel of the original image.

From experimental results he found that for optimum edge sharpening this scale factor should (1) be considerably larger in bright areas, (2) be inversely related to edge contrast,*** and

---

* see ref. [9] section IV. E.

** The mask would correspond to the image resulting from the lows data in this system.

*** 1) and 2) correspond to this system's luminance and detail scaling factors, respectively.

(3) have magnitude such that the dynamic range of the optimally sharpened image does not exceed the limitations of the system and lead to edge compression. The measure of edge contrast used here is the edge information itself. This corresponds to the scaling of the edge information (highs) of the original by a factor determined by the same edge information, i.e. a nonlinear scaling and by a factor determined by the luminance of the original. A significantly greater amount of sharpening can be achieved than is possible with linear scaling without causing such visible artifacts as mach bands (9) and increased noise visibility (graininess in the image). These mach bands, which appear as light or dark halos alongside an edge, result from excessive overshoot and/or undershoot in the luminance transition, which corresponds to the edge, and are characteristic of oversharpening. While this approach can achieve optimal sharpening over the entire image, Gilkes found that it also leads to some deformation of the edges. Also, while it can reduce noise visibility, it does not differentiate between this undesired noise and desired texture in the image.

Curlander also dealt with optimal sharpening and, in fact, achieved this without the edge deformation inherent in Gilkes' approach.* One major difference between his approach and the approach in the earlier work by Gilkes was the measure of edge contrast used to determine the corresponding scale factor. Curlander used a detail measure corresponding to the average over an area of the magnitude of the edge information rather than the edge informa-

* Curlander's work was a continuation of the work done by Gilkes.

tion itself. In addition to the guidelines for scale factor determination arrived at by Gilkes, it was determined that the contrast scaling factor should be small for low detail measures to avoid accentuating noise. As before the reduction of noise visibility may also cause the attenuation of some desired texture since they may involve approximately the same amount of detail or edge activity. It is possible, however, to achieve rendition of texture superior to that of Gilkes' approach (9).

As did Gilkes, this detail scaling is combined with edge scaling according to the luminance value of the original, taking advantage of the fact that more edge sharpening can be tolerated in high luminance (bright) areas than in lower luminance (darker) areas. This is because such artifacts as the occurrance of mach bands and noise visibility are inversely related to luminance (7). The optimal scale functions according to detail measure and according to luminance were both determined empirically (1) and are used directly in this system.

II-2 Data Compression

The aim of psychovisual coding* is to achieve data compression by selectively eliminating that information which is relatively unimportant to the visual quality of the image (5) (10) (11). The technique used for the lows data is closely related to the sampling theorem (or Nyquist criterion). Since the lows are clearly bandlimited, the subsampling and reconstruction procedures are directly analogous to

* also referred to as psychophysical coding.

the sampling and reconstruction of, for example, a continuous time waveform. The compression technique used for the highs cannot be so easily related to linear system theory and is dependent rather on knowledge about the visual perception of edge information.

While the human eye is very sensitive to edges it is not very sensitive to the exact size of the actual edge transition (5).

Quantization takes advantage of this insensitivity, with the number of bits kept being determined by the dynamic range and/or resolution necessary to maintain the optimal sharpening achieved by the enhancement procedures*. For example, if not enough resolution were retained the error between the quantized and unquantized edges might be large enough to produce the visual artifacts of oversharpening or, on the other hand, produce unsharpened edges.

A more objectionable effect of the quantization is appearance of artificial contours paralleling the real edges and corresponding to the quantization steps encountered as an edge is approached**. (see Figure 2-1). To minimize the visibility, the highs are compressed such that the quantum levels correspond to subjectively equal increments of brightness (9). The receiver expands this quantized information to restore the highs. The function used for the compressor and expander (compander) are thus based on empirical facts concerning the human eye's brightness perception (3).

---

* Hoover (3) obtained excellent results with three bits.

** Huang (4) found that 100 level (~7 bits) were needed to present the occurance of visible quantization noise.

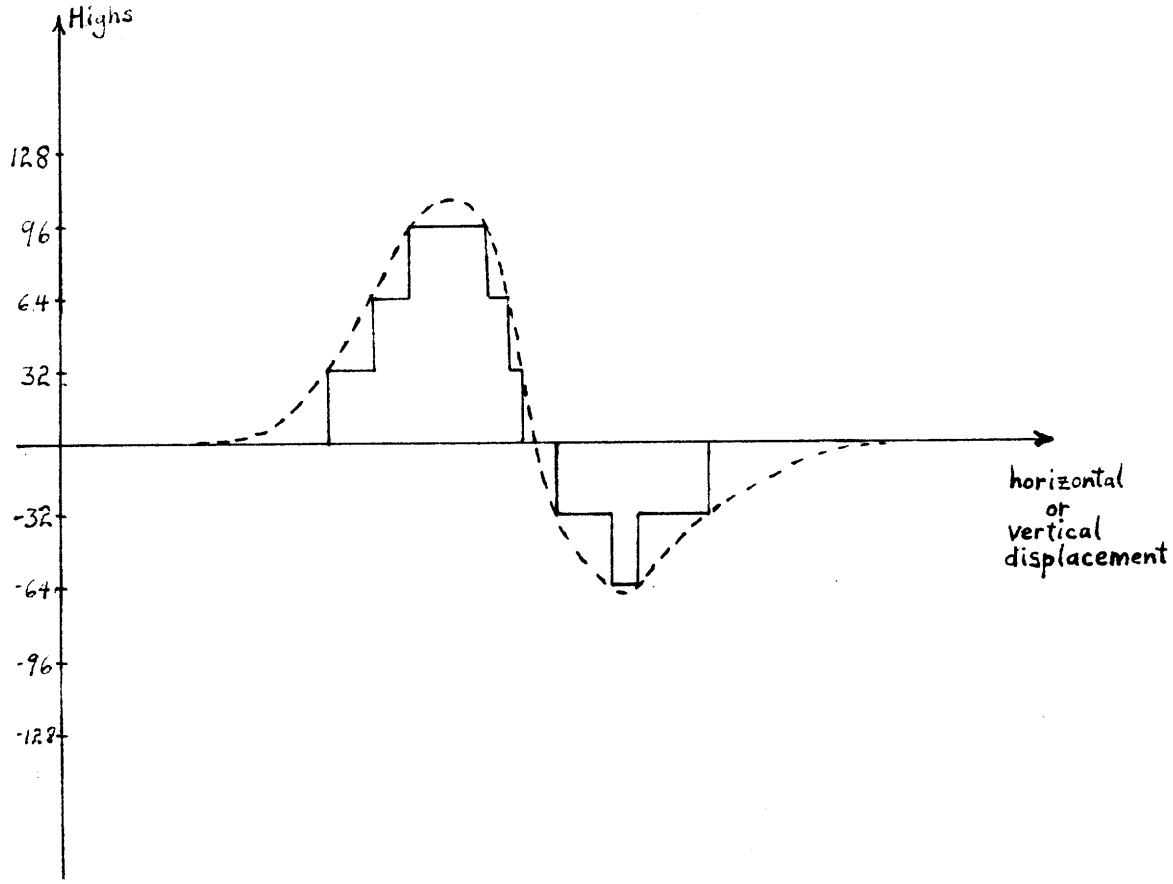


FIGURE 2-1 Edge with Q-steps

$$\text{compressed}^{*}_{\text{highs}} = \frac{127\ (\ |\text{highs}|^{.5} - 1)}{127^{.5} - 1} + 1$$

$$\text{expanded highs} = \left[\frac{(127^{.5} - 1)\ (\ \left|\frac{\text{compressed}}{\text{highs}}\right| - 1\ )}{127} + 1\right]^{2}$$

The visibility of the quantization noise also tends to be emphasized because of the eye's greater sensitivity to strongly structured noise, as opposed to unstructured (random) noise. Roberts' (7) noise processing scheme exchanges this strongly structured quantization noise for spatially random noise, thus greatly reducing its visible effects. In this technique, pseudorandom noise is added to the highs. After quantization and transmission to the receiver, the same noise** is subtracted, giving data with random noise of noise power equal to that of the quantization noise in the absence of the noise processing.

Hoover's work demonstrated the performance of an enhancement/transmission system utilizing the techniques just discussed, verifying that the combined effect of the enhancement and data compression techniques, does indeed correspond to an overall improvement in the visual quality of an image.

---

* Note that the number, 127, is the maximum positive or negative value of the highs data.

** Since the noise is pseudorandom, it can be duplicated in the receiver.

III. Functional Block Diagram

This section is concerned with the discussion of the functional block diagram. Although in the actual implementation some of the functions are distributed somewhat differently.* The block diagram described in this section is more useful in explaining the operation of the system. Figure 3-1 shows how the system would be utilized (set up) for an actual image transmission process. The functional block digrams for the transmitter and the receiver are shown in figures 3-2 and 3-3 respectively. The necessary delays have not been included here but will be discussed in the section(s) dealing with implementation. These diagrams do, however, show the flow of data through the major blocks of the system; blocks which correspond directly to operations descussed in earlier sections. The operation of (design of) each of the blocks and their interaction with one another are discussed after a brief discussion of of the overall system as shown in Figure 3-1.

The transmitter accepts 8-bit data values from the scanning device at a rate of one word per second, i.e. 8kb/s, and presents data to the transmission channel at a rate of 4kb/s. The data from the scanner to the transmitter must be accompanied by, or contain, appropriate framing pulses** to facilitate successful transfer without necessarily having prior knowledge of the size of the original image. Given the scanner's rate of 8kb/s, the trans-

---

* This will be discussed in a later section.
** new page (image) and new line indications

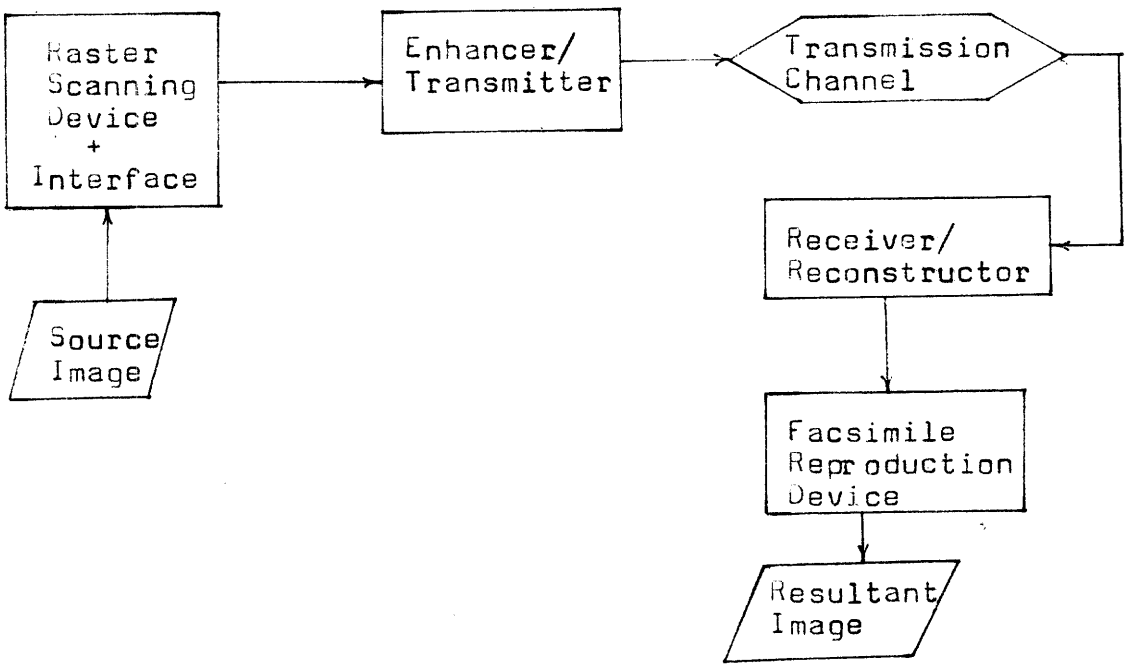
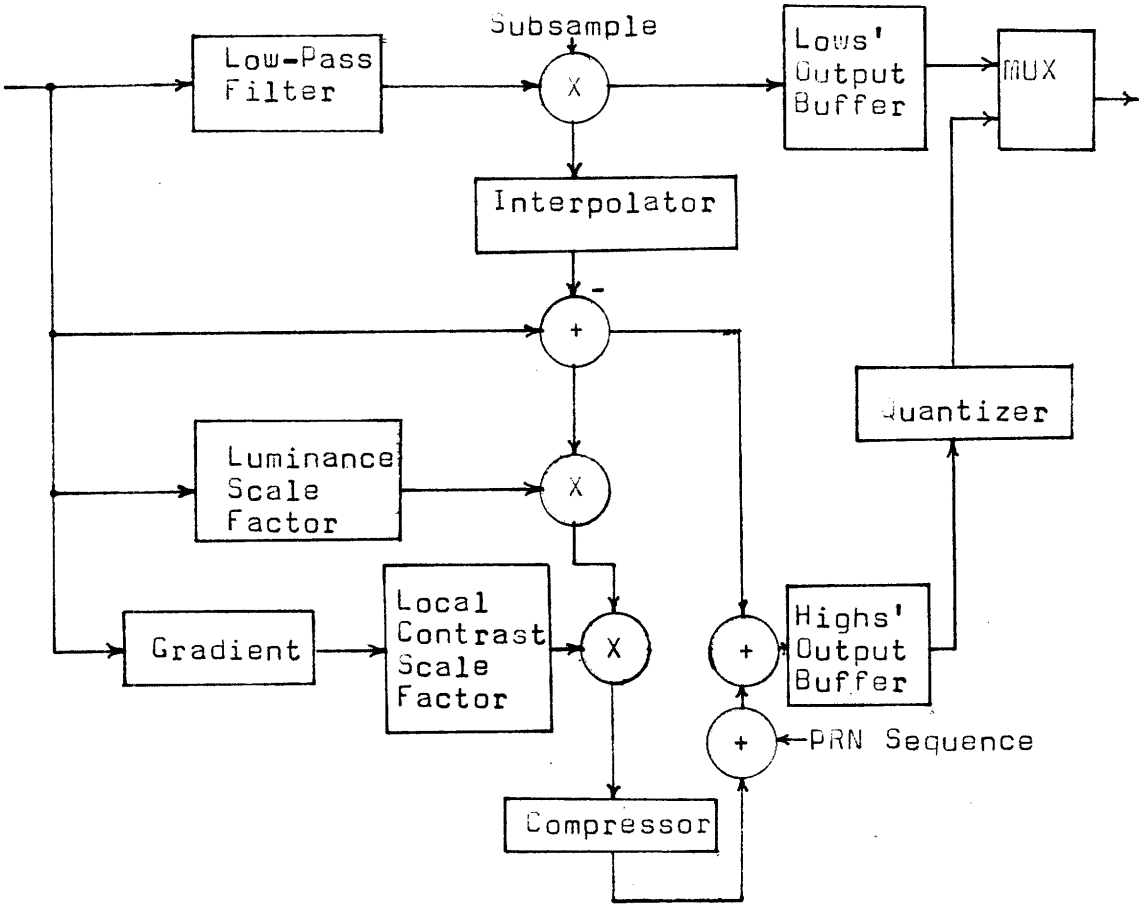
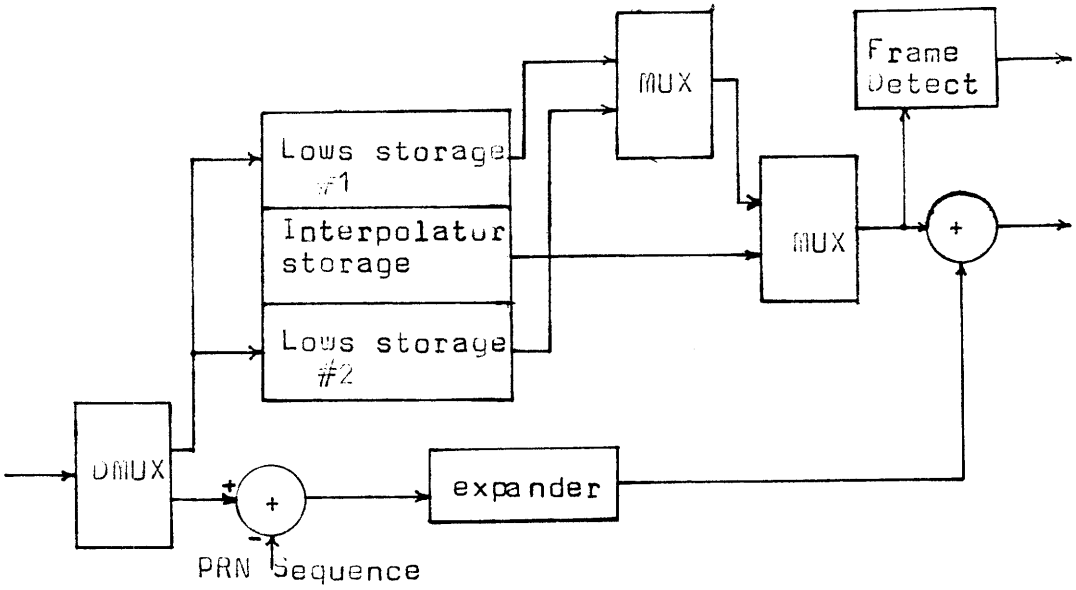


FIGURE 3-1 System Orientation



Transmitter Functional Block Diagram

FIGURE 3-2



Receiver Functional Block Diagram

FIGURE 3-3

k	-4	-3	-2	-1	0	1	2	3	4
a_k	1	3	13	28	37	28	13	3	1

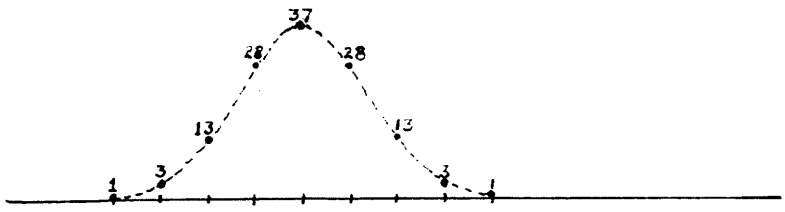


FIGURE 3-4 Filter Coefficients

mitter's 2:1 data compression and the transmission channels 4kb/s capacity there is no channel capacity left in which to send these necessary framing signals to the receiver.

Two alternative methods to circumvent this problem were considered: 1) Delay the beginning of the transmission and use this time to send a count of the number of pels per line. 2) Have the framing pulses (indications), when they occur, override the data value being transmitted.

The delay of the first method would not pose a problem and has the desirable feature of leaving the actual image data unaffected. However, any framing errors that might occur* would tend to accumulate line to line since the system would have no means of realigning itself.

The second method does affect the data, but is immune to the type of cumulative framing errors of the first. In addition, not that the only data affected is the first or last pel value on each line and corresponds only to the outer boundary of the image. Since this does not affect visual quality, this method is chosen over the first.

The first block to be discussed is the low-pass filter, the major tool in the separation of the lows and highs data. It is a two dimensional, circularly symmetric Gaussian digital filter. It is implemented as two cascaded one-dimensional filters; one for hori-

---

* An error in the line length assumed by the receiver could result from a miscount in the transmitter or even just a noisy transmission.

zontal, one for vertical. This is a valid procedure for any multi-dimensional filter whose impulse response is a separable sequence (5)* Each of the filters is implemented as a weighted sum with nine non-zero coefficients

$$= \sum_{k=-\infty}^{\infty} a_k \cdot x[n-k] = \sum_{k=-4}^{4} a_k \cdot x[n-k]$$

where x[n] is the original data along either a horizontal or vertical line, depending on the direction of the filter. Thus the separation into two filters implies that

$$\sum_{k=-4}^{4} \sum_{l=-4}^{4} b_{kl} \cdot x[i-k, j-l]$$

$$= \sum_{k=-4}^{4} a_k \cdot \sum_{l=-4}^{4} a_l \cdot x[i-k, j-l]$$

$$\text{i.e. } - b_{kl} = a_k \cdot a_l$$

where x[i,j] is the original image data's 2-D representation.

Each of the filters requires nine 8 x 8 - bit multiplications and the summation of these 16-bit products. Thus the representation

---

* For example, a 2-D filter whose coefficients correspond to the function $\exp[-(x^2 + y^2)]$ can be separated into two cascaded 1-D filters [$\exp(-x^2)$ and $\exp(-y^2)$]. However, if the coefficients were to correspond to a function such as $\exp[-(x+y)^2]$, separation into two 1-D filters is not valid since there is no way to separate such a function into two functions, each of just one variable.

of the output of the filter would require 20 bits. The microprocessor used has only 8-bits/word so that this would require multiple precision arithmetic, for which there is neither enough computation time or memory space available. Limiting the filters to single precision arithmetic, the multiplication operation rounds off to an 8-bit product and the filter coefficients are constrained to

$$\sum_{k=-4}^{4} a_k \leq 128.$$

This guarantees that the nine 8-bit products will not lead to a more than 8-bit sum (3). (see Fig. 3-4).

The algorithm for the multiplication operation is to shift and add, much as would be done by hand. The individual bits of the multiplier are used to determine whether or not to add the next left-shifted version of the multiplicand to the accumulated sum. In actual implementation, beginning with the LSB of the multiplier, the multiplicand is multiplied by either one or zero (depending on the multiplier bit), added to the existing sum, and the result shifted right one bit. This is repeated eight times for an 8x8 multiplication and gives a full 16-bit product. However, if only the eight MSB's of the sum are kept at any time and after the last shift the bit shifted out* is added to the sum, the desired product, rounded to eight, bits results. As an example of this algorithm, consider a 4x4 bit multiplication with a desired 4-bit

* This bit being one implies that the lower order byte of the product would have been one-half or more of the upper byte LSB (i.e. $\geq$ 128).

rounded product. (example shows 11(1011) times 5(0101))

Step #1.	multiply by LSB	0101 • 1 ⇒ 0101
#2.	add to sum	0000 + 0101 ⇒ 0101
#3.	shift sum right	0101 ⇒ 0010
#4.	multiply by 2nd LSB	0101 • 1 ⇒ 0101
#5.	add to sum	0010 + 0101 ⇒ 0111
#6.	shift sum right	0111 ⇒ 0011
#7.	multiply by 2nd MSB	0101 • 0 ⇒ 0000
#8.	add to sum	0011 + 0000 ⇒ 0011
#9.	shift sum right	0011 ⇒ 0001
#10.	multiply by MSB	0101 • 1 ⇒ 0101
#11.	add to sum	0001 + 0101 ⇒ 0110
#12.	add bit last shifted out	0110 + 1 ⇒ 0111; which is 56 if one kept track of the decimal point.

The error inherent in this algorithm is common to any fixed point multiplication algorithm using finite word lengths (clearly $11 \cdot 5 = 55$, not 56). The round off improves the resolution by one half bit (i.e. a factor of $\sqrt{2}$), therefore, the 8x8 multiplication is accurate to one part in 362 ($2^{8.5}$).

The results from the filter are subsampled by a factor of four

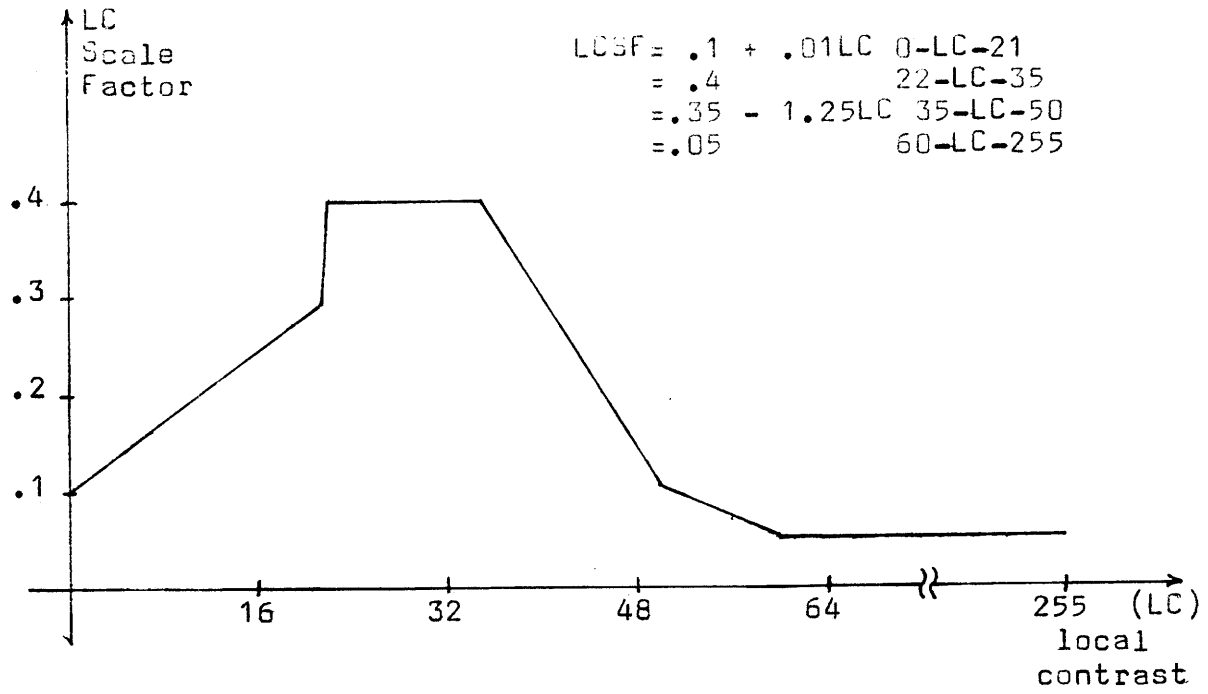
horizontally and a factor of two vertically to obtain the lows data to actually be transmitted. This subsampled data is also linearly interpolated in two dimensions and subtracted from the corresponding unfiltered original data to obtain the highs. Since the receiver has the same interpolation scheme, it agrees with the transmitter on the lows data. This insures that the transmitter's enhancement of exactly the information that the receiver lacks after having received and interpolated the lows data, i.e. the receiver's highs data.

The enhancement of the highs involves the determination of two scale factors. The first, the luminance scale factor, is obtained from a lookup table (Fig. 3-5) using the corresponding original data value as the index. The determination of the detail scaling factor is more involved in that the index to be used is the local contrast. This is computed as average of the magnitude of the gradient over a 15x15 pel area centered at the pel to be scaled (Figure 3-6).

Computation of the gradient at a given pel involves the four adjacent pels (Right, Left, Above and Below) as shown below.

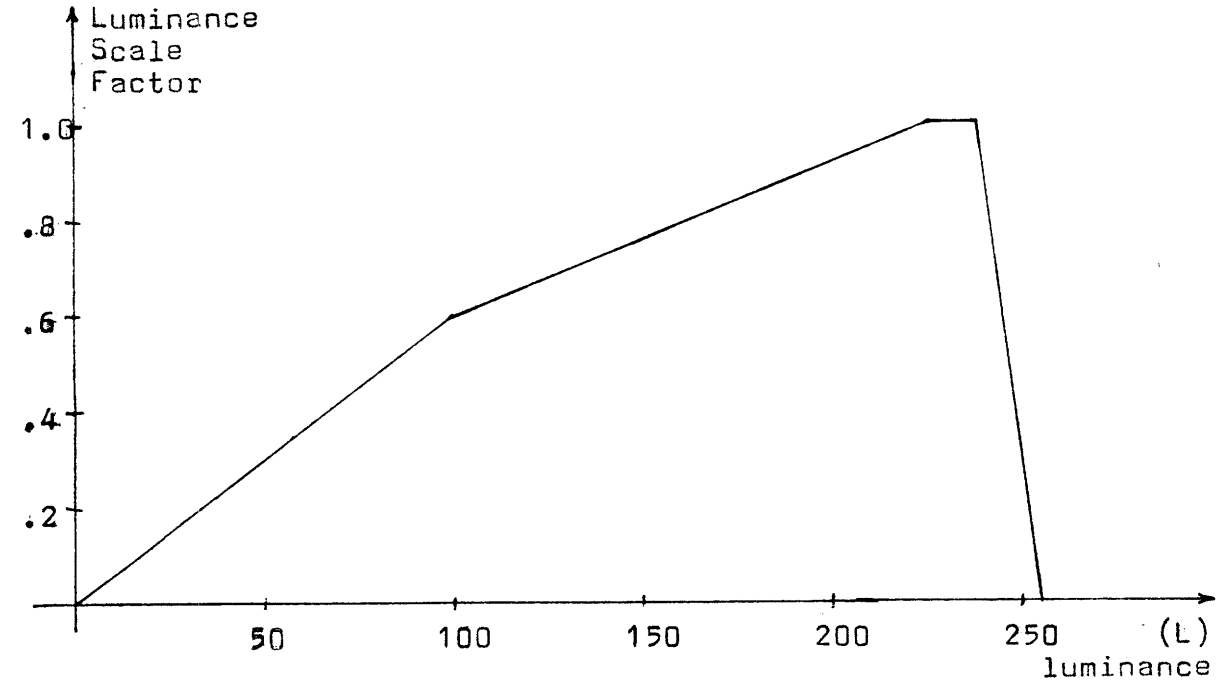
$$\text{gradient (pel)} = \left(\frac{|\text{pel} - L|}{2} + \frac{|R - \text{pel}|}{2} \right)_{\text{horizontal}} + \left(\frac{|\text{pel} - B|}{2} + \frac{|A - \text{pel}|}{2} \right)_{\text{vertical}}$$

The remaining blocks perform the data compression of the highs. The first block is the amplitude compressor which in effect makes the quantization steps smaller for smaller inputs or, equivalently,



Local Contrast Scale Factor

FIGURE 3-6



Luminance Scale Factor

FIGURE 3-5

makes them larger for larger inputs. This is done in such a way that the steps are equally visible over the whole dynamic range (9). The function, about midway between logrithmic and linear, follows the equation presented toward the end of section II.

$$\text{compressed highs} = 127 \cdot (|\text{highs}|^{.5} - 1)/(127^{.5} - 1) + 1$$

Roberts' (7) noise processing is used in order to exchange the strongly structured noise from quantization for unstructured noise and thus reduce the visibility of spurious contours. The three MSB's of each highs value are then presented to the transmission channel.

The receiver (Fig. 3-3) is not required to perform any involved operations such as filtering, etc. It accepts lows data and linearly interpolates the missing values. It also accepts quantized highs data from which it subtracts PRN noise values corresponding to those added to it in the transmitter. It then expands* the result and adds it to the corresponding interpolated lows value giving the data for the reconstructed image.

---

* The expander is the inverse function of the transmitter's compressor and is presented in section II.

IV. System Implementation

This chapter discussed the actual techniques used to implement the the functions of the blocks discussed in chapter III. Each of these operations must be done each time a new byte of image data is received. Some redistribution of the blocks was necessary to reduce the amount of computation, and thus minimize the time required.

Associated delays and memory allocation are also discussed for each block and the section concludes with a discussion of the incorporation of framing, system initialization, and Input/Output capability.

Note that in all of the discussions to follow it has been assumed that the image is raster scanned horizontally from left to right, moving vertically from top to bottom. There are also references to two-dimensional locations with the location of the most recent pel considered as the origin as shown below.

$$\begin{matrix}\text{most recently} \\ \text{received pel}\end{matrix} = \text{(original) data } [0,0]$$

horizontal ↑ vertical ↖

Note that this definition implies that locations with negative horizontal and vertical components are to the left of and above the current pel, (most recently received) respectively.

IV-1 Gradient and Local Contrast

As described in chapter III, the gradient is computed as the sum of the magnitude of the differences between the pel of interest and the four adjacent pels. It is implemented in software and the results are stored in memory for later use in the computation of the local contrast.

Note that the gradient uses data from three different scan lines, therefore requiring the storage of three lines of original image data.

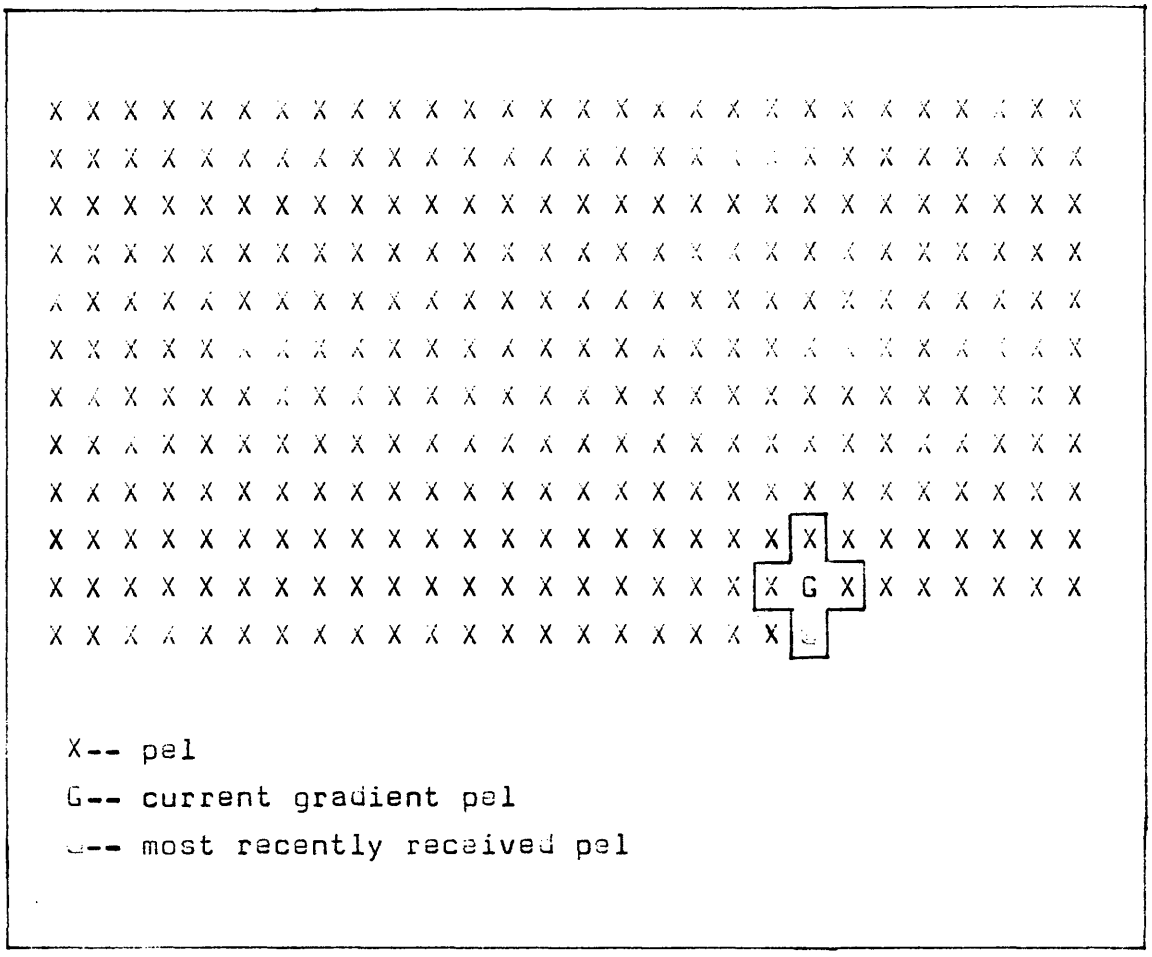
The result of a gradient operation will therefore, be delayed from the incoming data by at least one scan line. The minimum delay minimized both storage requirements and the delay of the local contrast operation. Figure 4-1 shows the pels involved in the gradient and their orientation relative to the incoming data. From this, the gradient is observed to correspond to the equation:

$$\text{gradient } (0,-1) = \tfrac{1}{2}[\ \underset{\text{most recently received value}}{\text{data}(0,0)}-\text{data}(0,-1)+\text{data}(0,-2)-\text{data}(0,-1)$$
$$+\ \text{data}(-1,-1)-\text{data}(0,-1)+\text{data}(1,-1)-\text{data}(0,-1)]$$

The local contrast is the average of the magnitude of the gradient values over a 15x15 pel area centered at location (-8,-9) relative to the location of the most recently received pel (see Figure 4-2). A straightforward implementation of this would require 224 additions, probably taking more than one millisecond to run*. A great reduction in the amount of computation required is achieved by taking advantage of the fact that the areas used for consequetive local contrast computations have all but 30 pel locations in common. The area moves along with the incoming pels so that when a new pel is received the area picks up a new column and discards its oldest (leftmost) one. Therefore, this implementation requires the continuous storage of the local contrast and then fifteen additions and subtractions to update this value. Care must taken in handling the way in which this value is affected by edges of the image. The affects at the edges themselves are not important, but for the second technique these edge effects

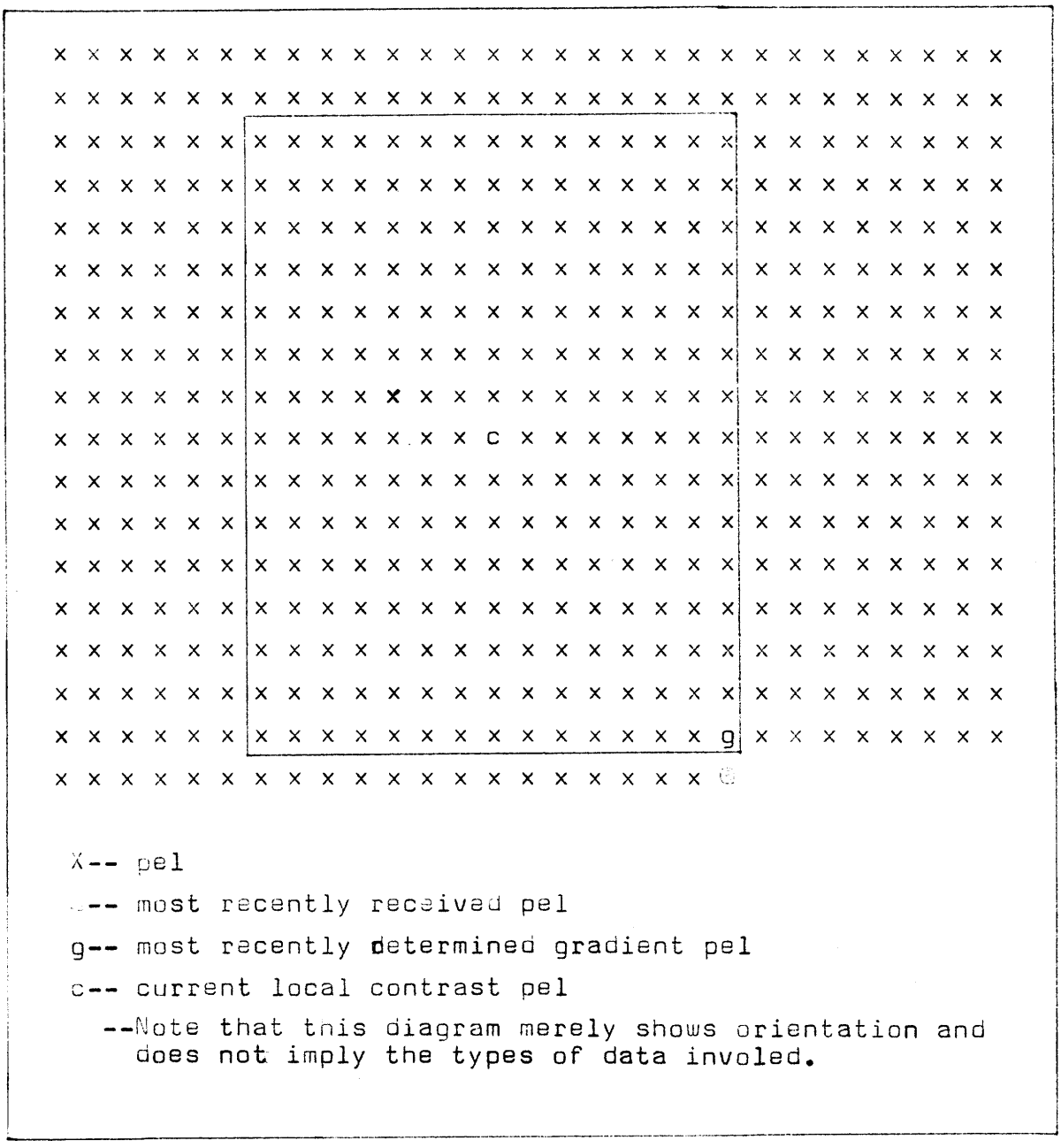
---

* the minimum instruction cycle time of the type of system proposed (Intel 8085) is .8 usecs.



Gradient Area

FIGURE 4-1

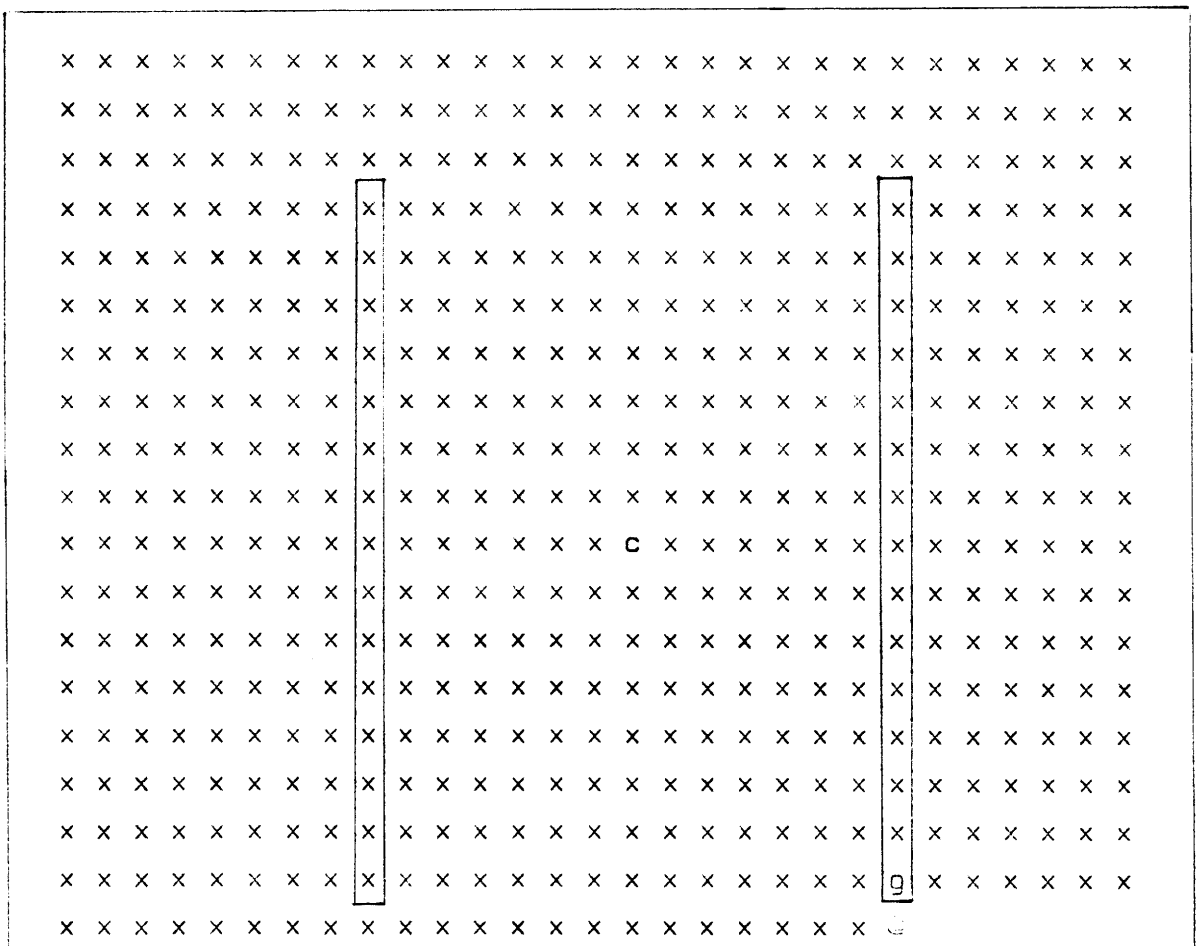


Local Contrast Area

FIGURE 4-2

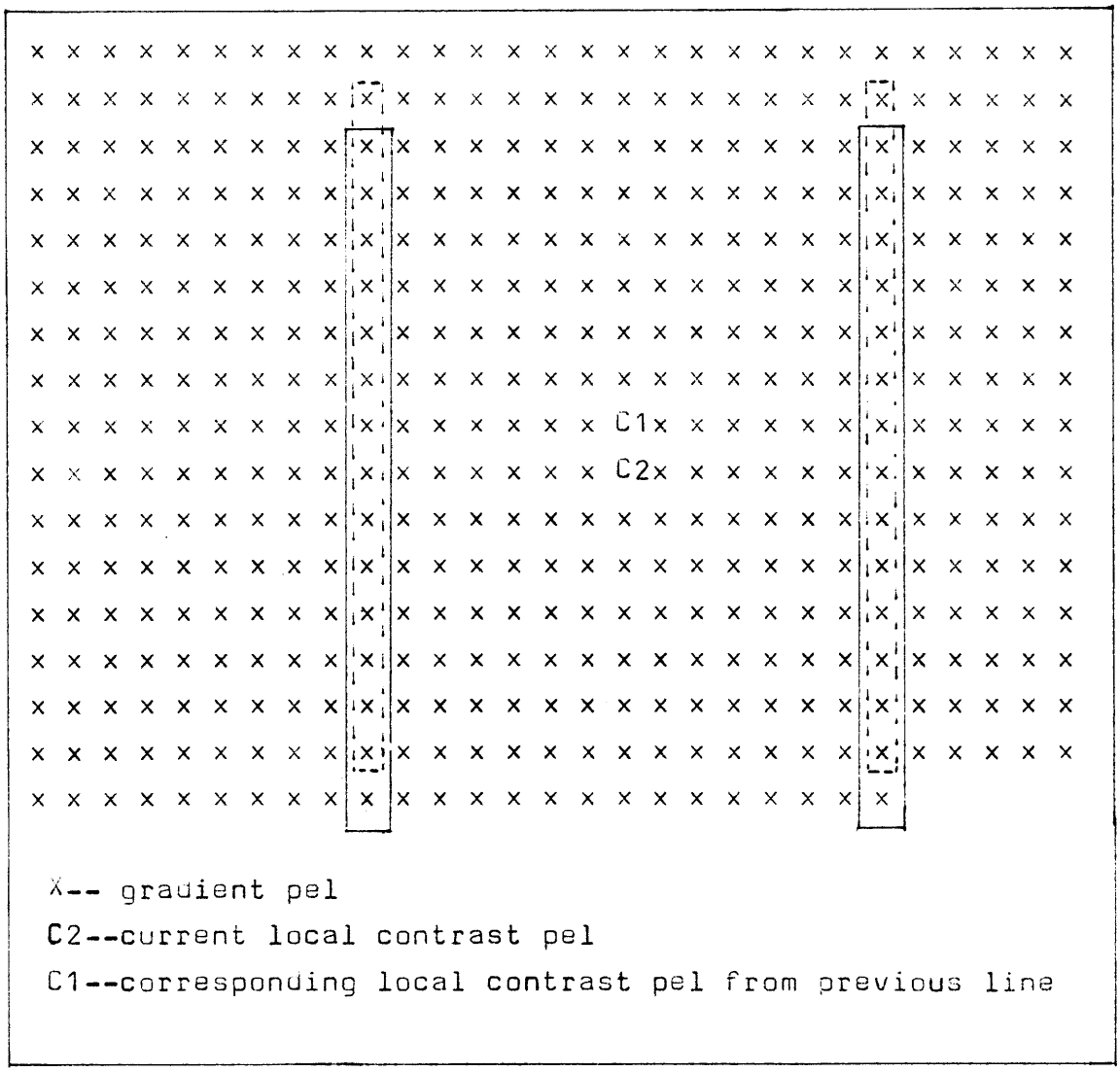
accumulate and persist throughout the interior of the image. To reduce these affects, any portion of the averaging area that falls outside of the image boundaries is filled in with zeroes. In addition, to keep whatever effect there may be from accumulating from line to line, the local contrast value is reset to zero at the beginning of each new line of data. Notice that with this scheme, the first fifteen local contrast computations for each line do not involve any subtractions since the columns to be subtracted fall outside of the image boundaries. Thus the averaging area is allowed to grow to the full fifteen columns before any columns begin to be discarded. Figure 4-3 shows the averaging area and columns most recently added and subtracted, respectively, and their orientation relative to the most recently received pel and the corresponding local contrast value location.

An even greater reduction in computation can be achieved by taking the above approach and apply it in the vertical direction as well. Figure 4-4 shows two averaging areas with vertically adjacent centers and the corresponding columns to be subtracted and added. Notice each of the columns has all but two pels in common with its counterpart from vertically adjacent area. Therefore, only two additions and subtractions are required to update the columns or, equivalently, to directly update the local contrast value from one line to the next. This reduction in computation is achieved at the expense of increased storage requirements, since, in addition to continuous storage of the local contrast value, the contribution from the corresponding columns for each update of the local contrast value must be stored until the following line, at which time it will be updated. This means an additional line of storage of value; in fact, a line of double-precision values since each is the sum



Local Contrast with Column Update

FIGURE 4-3



Local Contrast Areas with Vertically Adjacent Centers and using Column Update

FIGURE 4-4

of fifteen 8-bit values.

To summarize, the storage requirements of these two operations include sixteen lines of gradient magnitude values, one line of local contrast update values, and one local contrast value. The results of these are gradient [0,-1] and local contrast [-7,-8] and correspond to one and eight lines of delay respectively. Since the highs are scaled according to the local contrast value, the corresponding highs value must also be stored long enough to still be available when needed.

As was alluded to briefly, the local computation requires double precision arithmetic. The value before scaling is the sum of 225 8-bit numbers, thus requiring sixteen bits for full representation. The high order byte is used as the index for the scale factor lookup table, the values of which correspond to the curve presented in chapter II.

IV-2 Low Pass Filter

As discussed in chapter III, the 2-D low pass filter is implemented as two cascaded 1-D filters. Each filter has nine nonzero coefficients, therefore, the horizontal filter requires the storage of nine lines to give valid results. The output of this filter pair corresponds to lows (-4,-4) regardless of the order in which they are implemented and note that the highs cannot be obtained until the corresponding lows value has been determined. Therefore the original data must be stored for at least five lines*. Even after the highs have been separated, the lows must be stored until all of the scaling

---

* This actually only requires four lines plus four pels, but is impractical to implement as such.

operations have been completed for the corresponding highs value so that they can be transmitted together. The corresponding local contrast scale factor is delayed by eight lines from the incoming data, requiring that the lows and highs be stored for five* lines after they are first obtained. Also note that the vertical filter requires nine lines of either original or horizontally filtered data, depending on whether it is implemented before or after the horizontal filter, respectively. To keep from having to store the highs, separation of the highs from the original data can be delayed until the corresponding local contrast scaling operation can be carried out. This, in itself, does not reduce storage requirements, since now the luminance scaling must also be delayed; bringing the storage requirements for the original data up from five to nine lines. This means storage has actually only been reduced by one line using this scheme. Note, however, that these nine lines of original data are the same used by the vertical filter when it precedes the horizontal filter. Thus there is a net savings of 5 lines, since no additional storage of original data is implied by delaying the highs separation.

With the vertical and horizontal filters ordered as just discussed the vertical filter operates on the data from the most recent nine lines of original data. Their locations correspond to (0,0), (0,-1), (0,-2), (0,-3), (0,-4), (0,-5), (0.-6), (0,-7) and (0,-8) with the results of the operation corresponding to locations (0,-4); all relative to the most recently received pel. The horizontal filter then

---

* Once again, the actual requirement is four lines and three pel but is impractical to implement(complexity, computation time).

operates on the nine most recent vertical filtering results. These correspond to locations (0,-4), (-1,-4), (-2,-4), (-3,-4), (-4,-4), (-5,-4), (-6,-4), (-7,-4) and (-8,-4) with the result corresponding to location (-4,-4).[see Fig. 4-5]

All of the computations involved in the filtering are single-precision. The multiplier used for coefficient multiplication is accurate to 8.5 bits*, as discussed earlier, and the constraint on the filter coefficients insures that one byte will be sufficient to represent the result.

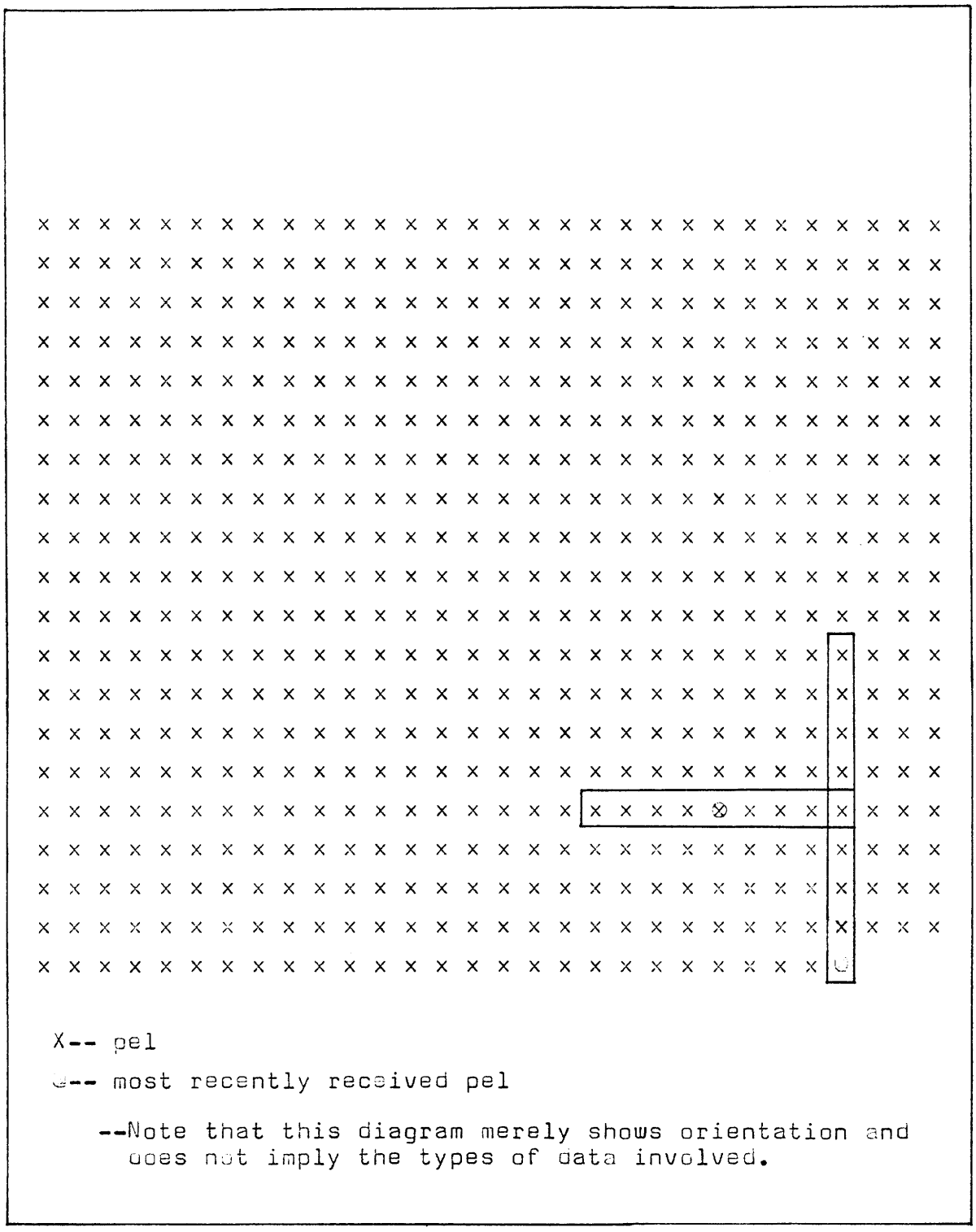
IV-3 Subsampler and Interpolator

Only one in every eight locations is to be a sample point for the lows. This is arranged such that every fourth location on every other line corresponds to a lows sample point. The value in between are to be filled in by the interpolator to achieve lows actually used for the highs separation. Only the sample points, however, will be transmitted. Rather than filter at each location, pick out the sample points, and then interpolate over the other points, it makes sense (and greatly reduces computation time) to determine the sample points before filtering and only compute the lows values for those locations. Thus horizontal filtering need be done only for every fourth location on every other line. Vertical filtering, while it must be done for every point on the line, need be done only on every other line.

The interpolator is split into two one-dimensional linear inter-

---

* the result is rounded off to eight bits for an accuracy of one part in 362.



X-- pel

⊗-- most recently received pel

--Note that this diagram merely shows orientation and does not imply the types of data involved.

Low-Pass Filter Areas

FIGURE 4-5

polator. The horizontal interpolator operates on the lows sample point it is given and the one previous to it. It simply takes their difference, divides it by four, and repeatedly adds it to earlier sample value to fill in the three undetermined values at the locations in between. Similarly the vertical interpolator operates on the sample point given and the corresponding point on the previously sampled line to fill the undetermined value at the location in between. This value and location filled are then given to horizontal interpolator so that the three locations between the last two filled by the vertical interpolator, can be filled in.

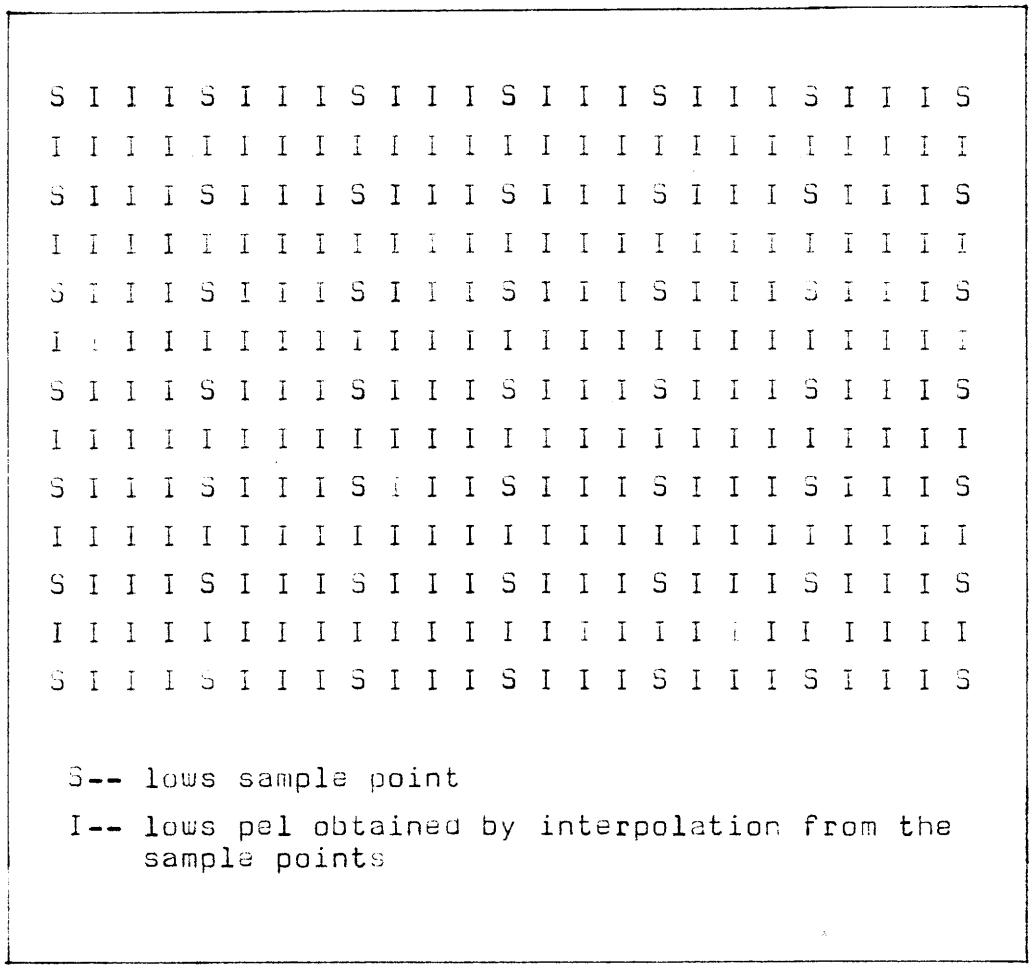
Given that the result of the low pass filtering, at the times when it is to be done*, corresponds to location (-4,-4) relative to the most recently received pel, the horizontal interpolator results first correspond to locations (-7,-4), (-6,-4), and (-5,-4). The vertical interpolator result corresponds to location (-4,-5) and the second set of horizontal interpolator results correspond to locations (-7,-5), (-6,-5), and (-5,-5).[see Fig.4-6]

IV-4 Compander

As with the other scaling operations, the compression just before the PRN processing is achieved by using the highs value as the index for a lookup table. The same goes for the expander following the noise processing in the receiver. The equations used to obtain the values were presented in chapter II.

---

* location (-4,-4) must be a lows sample point



Subsampled and Interpolated Lows

FIGURE 4-6

IV-5 Overall System

There has been considerable discussion thus far of the various image processing techniques being implemented in the system, but so far not much has been said about the overall system and how it controls and cycles through these operations. Since it is the goal of this work to present first a system implementation to be operable with a microprocessor simulator and later with an actual microcomputer system, it is necessary to discuss somewhat how this system actually controls the flow of data through it. This includes not only input from the scanner and output to the facsimile device, but also matters such as the formatting of the data for transmission from the transmitter to the receiver, the flow of data in and out of the various storage area for the various operations, and the interfacing to I/O devices.

IV-6A Transmitter controller

The transmitter is responsible for receiving and storing input data, controlling the functions discussed earlier, determining when and on what data they operate, and formatting the data to be transmitted. The overall operation is the same for both the simulator and microcomputer implementations. The microcomputer implementation has several added features which are necessary both because it must interact with external devices and because of constraints within the system itself. These features will be discussed in detail in a later section devoted to the microcomputer implementation.

The basic system objectives were for it to be capable of enhancing, transmitting, and reconstructing images with scan lines of arbitrary length up to 1024 pels. It is to receive data at up to 8kbytes/sec and

transmit data at up to 4kbytes/sec, thus the 1 millisecond constraint on total computation time. The controller maintains and updates the appropriate storage areas for each of the functions. It also formats the highs, lows and appropriate framing pulses to be presented to the transmission channel. It is designed to expect framing indications nested within the image data corresponding to the beginning of the image('new page') and the end of each scan line ('new line'). The allocation of memory storage areas for the various functions was discussed earlier. There must also be some storage of completely processed highs and lows values so that it can be formatted for transmission. The format calls for transmission of one line of lows followed by two lines of highs values. This transmission occurs in the same time it took to receive the corresponding two lines of original data. There are two output storage areas, each comprised of one line of lows and two lines of highs. While the data from one area is being transmitted, the other area is being loaded with new values. Note that the two highs lines correspond to the highs from the line located above and on the line from which the lows sampler were determined.

In summary,the memory requirements are 9kbytes of original data, 16kbytes of gradient magnitude values, 2kbytes of local contrast update values, 5kbytes of lows values, and 4.5kbytes for the output buffer areas. There are also about 64 bytes devoted to flags and pointers for the various operations and storage areas. The controller initializes all pointers and flags and zeroes out the storage areas for the gradient magnitude and local contrast update values to proper startup for the local contrast operation.

The controller must insert additional framing information into the data as it is being transmitted. The 'new line' and 'new page' indications are already nested in with lows data, but the line mismatch value must be inserted into the first lows line right after the 'new page' indication. Since the mismatch value is important to image reconstruction, it is transmitted four times to reduce to probability of an error due to noisy transmission.

IV-6B Receiver controller

The receiver receives the formatted data from the transmitter and reconstructs the original image from it to be output to some device. It must also extract the necessary framing information so that the device can properly align the image. As mentioned earlier, this output data corresponds to the digitized version of raster scanned image, therefore the new line and new page indications are sufficient for proper alignment. Involved in the reconstruction process are a 2-D linear interpolation of the low similar to that done in the transmitter, a magnitude scaling*, and the remaining portion of the PRN processing. The controller must also make use of the line mismatch information to insure that the highs and the lows lines are of the same length.

In the format discussed in the previous section a line of lows is transmitted, then is followed by two lines of highs. When the line of lows is received, the interpolator is used to fill in the values which were not transmitted. By maintaining the two most recently

---

* This scaling is the expander portion of the compander pair.

received lows lines, it is possible to fill in not only the missing values on the received line, but also all of the values for the line between these two for which no low values were transmitted. Thus the highs lines are to align with the most recently received lows line and the untransmitted line that would have preceded it were every lows line transmitted. All of these are transmitted in the order in which they were determined in the transmitted; thus the first highs line alligns with the untransmitted lows line, and the second with the lows line that was actually transmitted.

As in the transmitter, the scaling operation is done via a lookup table. The scaling is preceded by the noise processing, where the PRN values are also obtained from a lookup table. The same table is used for both the receiver and transmitter and the indices are derived from the horizontal and vertical position of the pel being processed relative to upper left hand corner of the image. The table contains 64 1-byte values corresponding to an 8 x 8 noise mask. Thus, it is sufficient to have two 3-bit counters, one for horizontal and one for vertical, in each the transmitter and the receiver.

The scaling and noise processing are performed on the highs values immediately as each is received. The result is then added to the corresponding lows value and output to some sort of facsimile reproduction device. There is almost certainly a need to convert this digitized data to a form compatible with the device and, time permiting, some such converter will be implemented. For the simulated version, this is not a factor in testing. The results will, in this case, be stored in a file and viewed later on of the systems* television monitors.

---

* the simulator is implemented on the Cognitive Information Processing Group's Unix system.

V. Simulator Implementation

In this chapter, the details of the implementation of the system on an 8080 microprocessor simulator are presented. The discussion of these details includes the software written to implement the functions and achieve the overall system behavior described in the preceding chapter. Before dealing with software, it seems appropriate to first discuss briefly some of the programs be used in conjunction with the simulator. The format of the images available for testing and that of the resultant images is also to be considered.

V-1 Support Programs

There are two programs available on the Unix system that facilitate the use of the simulator. The micro-assembler, MICAL, assembler files written in either the Motorola 6800 or the Intel 8080 instruction set, converting them to the appropriate machine codes. A second function, RELDLD, converts this code to a form that can be loaded directly into the simulator for execution. This section briefly presents the logic format for using to the degree need for this project. For more details on the functioning and use of these the reader should refer to the UNIX system manual.

The assembler will attempt to convert any file with the suffix .8080 into Intel 8080 machine code.

V-2 TRANSMITTER CONTROLLER

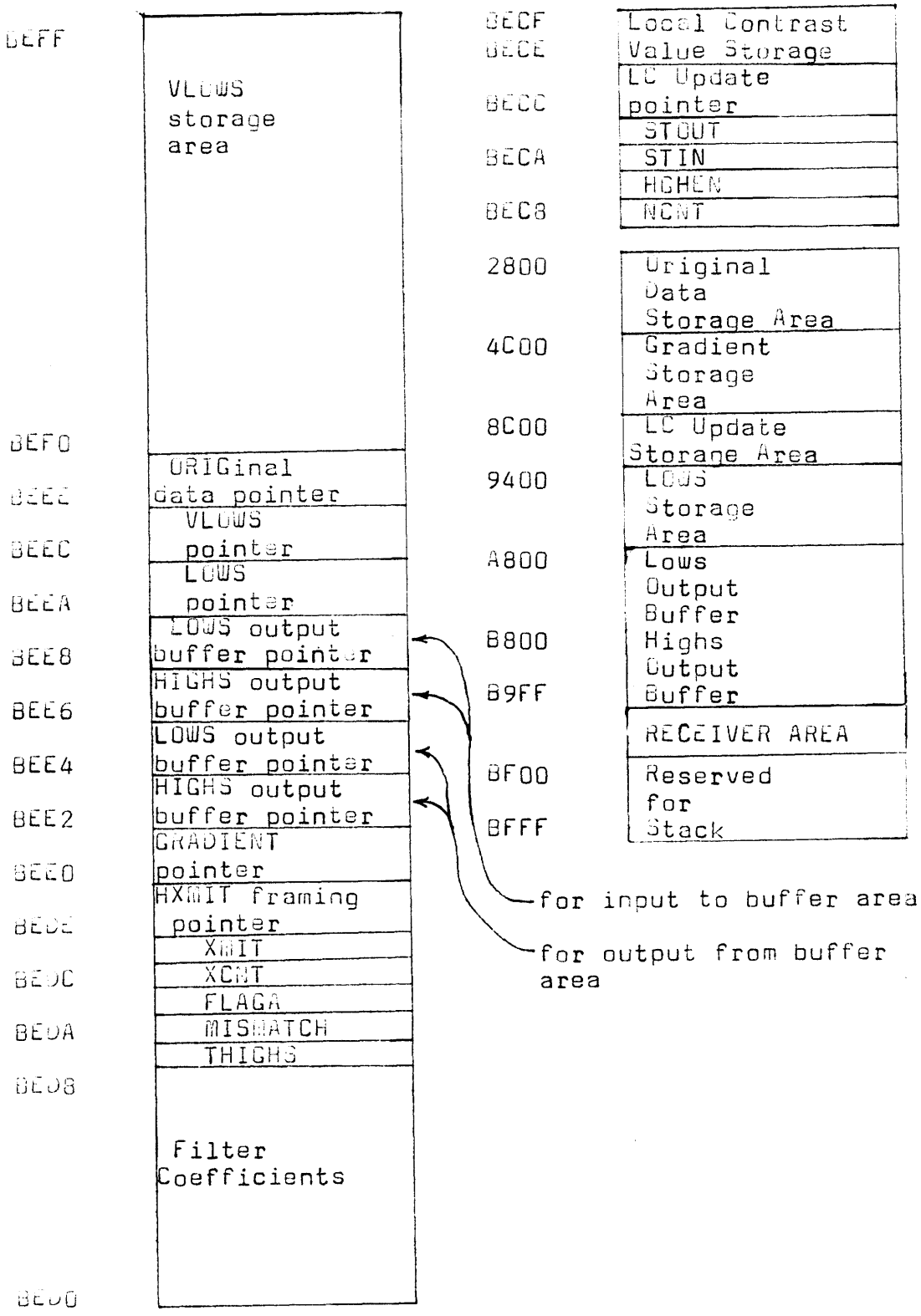
This section is mainly concerned with the various control and data flow operations carried out by the controller. All of the other transmitter functions are implemented as subroutines, called from this program. It also carries out the scaling operations, i.e.- it fetches the appropriate values from the various lookup tables.

The first section of the controller program initializes the data storage areas for the various subroutines. It also sets all of the locations to zero; the gradient and local contrast update storage areas* as was specified in chapter VI. Appendix A contains a complete listing of the program plus all of the subroutines for the simulator. The initialization portion is quite straightforward. It begins by loading the filter coefficients. It next initializes all of the control flags and the pointers for the various functions. Finally, it zeroes out the areas mentioned and proceeds to the rest of the program.

Figure 5-1 show the memory allocation for storage of the various pointers and flags. The pointers are used both in loading new data and as references in accessing previous data. In fact, the majority of the computation in the controller itself is devoted to these pointer manipulations. The corresponding data storage areas are also shown, with the address indicated by the pointers corresponding to the locations in which data has most recently been stored or, for the transmitter output pointers, the locations from which data was most recently taken. That the proper manipulation of these pointers is sufficient to achieve the desired data flow structure, will be demonstrated later in this section. After initialization has been completed each of the pointers contains the address of the lowest address of the corresponding storage area. The control flags are set such that all functions pertaining to obtaining the lows and loading the transmitter output buffer area are disabled. There are five such flags labelled STATUS, STIN, STOUT, HGHEN and FLAGA. These too will be discussed in the next several paragraphs.

---

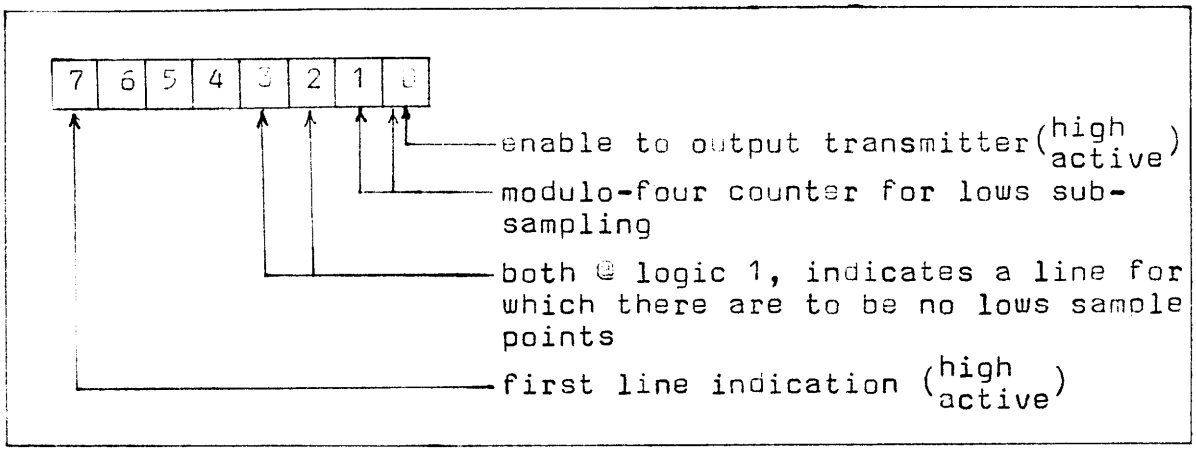
* A total of 18 kbytes.



TRANSMITTER-Flags, Pointers, and Memory Allocation

FIGURE 5-1

The flag word STATUS is used both to totally disable the functions pertaining to the determination of the lows and later, to implement the subsampling scheme. As mentioned, it is initially set such that the filters and interpolators are totally disabled. It is first modified when it is detected that the pel at the center of the vertical low pass filter area (see Fig. 4-5) contains a 'new page' indication. It should also be mentioned that this flag is also used to signal when the line mismatch value can be determined. Figure 5-2 shows what each of its bits indicates. When the 'new page' indication is detected, bit 7 is set and bits Ø,1, and 2 are reset. Since for vertical filtering to be inhibited both bit 2 and bit 3 must be set, vertical filtering is done on for each subsequent pass through the program. The filter is designed to pass any 'new line' or 'new page' indication that is detected at its center, directly to its output, thus preserving the framing information in the lows. Each 'new line' indication detected at the vertical filter causes the controller to toggle bit 2 of STATUS toggle. Similarly, when a 'new page' indication is detected at the pel just after the horizontal filter's center, the controller sets bit 3 of STATUS. When a 'new line' indication is detected at this location, bit 3 of STATUS is toggled. Note that bit 3 should therefore get toggled five passes after bit 2. Thus filtering is enabled from the time the vertical filter detects a 'new page' or 'new line' indication until the time that the horizontal filter detects the following 'new line' indication. Thus bit 3 is used to, in effect, extends the filter's enable times just long enough for the horizontal filter to pass on the new line indication.



STATUS Flag

FIGURE 5-2a

```
Filter input--x  np x  x  x  x  x  x  x  x  x  x  x  x  x  nl x
STATUS-0c 80 81 82 83 80 81 82 83 80 81 82 83 80 81 04 05
       x  x  x  x  x  x  x  x  x  x  x  x  x  nl x  x  x
       06 07 04 0c 0d 0c 0d 0c 0d 0c 0d 0c 0d 08 09 0a 0b
       x  x  x  x  x  x  x  x  x  x  x  nl x  x  x  x  x
       00 00 01 02 03 00 01 02 03 00 01 04 05 06 07 04 0c
       x  x  x  x  x  x  ...
       0d 0c 0d 0c 0d 0c ...


       x-- data input to filter
       np- 'new page' indication input to filter
       nl- 'new line' indication input to filter
         -- Note that the state of the STATUS flag is
            indicated in hexidecimal notation.
         -- The sequence shown corresponds to a line
            length of fifteen pels.
```

STATUS Flag Sequence

FIGURE 5-2b

While the filters are enabled, i.e. on every odd line, bits $\emptyset$ and 1 are made to carry out a modulo-four binary count of the number of pels on the line. Each time the count is zero, the horizontal filtering is done. So that the lows sample points will correspond to pels $\emptyset$, 4, 8, etc. the count is set to three each time the horizontal filter passes a 'new line' indication to its output. Thus if the indication occurs during the enable time, the horizontal filtering will be done again on the next pass, rather than skipping the next three, which corresponds to the first pel of the line. The next 'new line' indication passed on by the horizontal filter will affect bits $\emptyset$ and 1 in the same way, but since the filters will be disabled by the next pass, it doesn't cause a problem. The horizontal filter itself is allowed to set these bits since, as was just shown, it can be done rather blindly.

There is one remaining issue concerning the use of the STATUS flag for controlling the filters. Since it is necessary that the framing indications be present in the lows data, it is necessary that the pels from the original image that contain 'new line' indications, in fact, correspond to sample points for the lows. From the discussion thus far, this would only be true for images with line lengths that were multiples of four. However, setting the modulo-four count to zero whenever the vertical filter detects the indication, guarantees that the pel will indeed be a sample point. It does cause the last two sample points to correspond to locations other than multiples of four on the line, however, this is acceptable since it is only an edge effect and not at all cumulative.

Finally, STATUS is also used to enable the transmitter to present

one byte of data to the transmission channel*. Since the transmission rate is one-half of the input rate and since bit Ø and 1 are already performing a modulo-four count of pels when the filters are enabled, bit Ø can be used to enable or disable the output. When the filters are disabled the controller can simply toggle the bit on each pass.

The next three flags mentioned are almost inconsquential compared to STATUS. The flag STIN (Set Transmitter INput) is used to disable the routine that loads the lows area of the output buffer, thus preventing it from loading erroneous data until valid low data is available. This routine scans the lows values shortly before they are used in highs separation. When a 'new page' is detected, the flag is set**the routine begins loading the lows sample points into the transmitter output buffer.

The flag STOUT (Set Transmitter OUTput) is used when reset, to totally disable any output from the transmitter. It is set by the routine that loads the lows output buffer when it loads the first 'new line' indication.

The flag HGHEN disable the routine that loads the highs storage area in the output buffer. After STIN has been set, it will be set when it detects the 'new page' indication in the original data pel

---

* Although data is presented to the channel in parallel bytes at a time, there are no constraints on the actual form used by the channel as long as appropriate converters are available at both its transmitter and receiver ends.

** STIN = FF when set

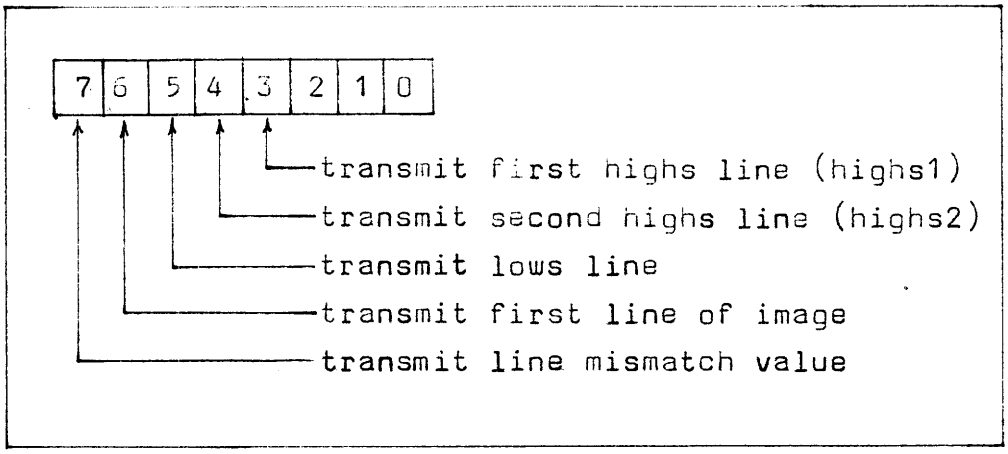
being used in highs separation. Remember that the highs separation is achieved by subtracting the lows values from the corresponding original data value. Since the 'new page' indication is the first valid input pel, this flag guarantees that no invalid data will be loaded into the output buffer.

Clearly these three flags have not been as fully utilized as the STATUS flag, and it would have been more efficient to combine these into one flag. In development, however, it was desireable to keep them separate and facilitate debugging.

The last flag mentioned (FLAGA) is used by the routine that actually takes the data from the appropriate output buffer areas and presents it to the transmission channel. Figure 5-3 shows what each of the bits in this flag indicates. It is initialized to indicate lows and controls the type of data transmitted and the word length. Once the routines are enabled, the routines begin taking data from the lows area of the output buffer. The very first value it gets should be the 'new page' indication. This value will cause the first line indicator (bit 5) to be set and also set FLAGA so that it will next transmit the line mismatch value* (bit 7 set). After this value is transmitted, FLAGA is set such that the routine will go back to transmitting the lows values. When a 'new line' indication is transmitted, FLAGA is checked for the 'first line' indication. If it is present then FLAGA is set such that one line of highs will be transmitted, i.e. bit 3 is set. Otherwise it will be set such that two lines of highs will be transmitted, i.e. bit 4 is set. At the end of

---

* Remember that the mismatch value is transmitted four times.



FLAGA Flag

FIGURE 5-3a

--The sequence shown would correspond to a line length of fifteen pels.

initially set to 20 -- 20 20 20 20 e0 60 60 60 60 10 10 10 10 10
('new page' at 20 20 20 20; 'new line' at 60 60)

10 10 10 10 10 10 10 10 10 10 20 20 20 20
('new line' at 10 10; 'new line' at 20 20 20 20)

08 08 08 08 08 08 08 08 08 08 08 08 08 08

08 10 10 10 10 10 10 10 10 10 10 10 10 10
('new line' at 08)

10 10 20 20 20 20 08 08 08 08 08 08 08 08
('new line' at 10 10; 'new line' at 20 20 20 20)

08 08 08 08 08 08 08 10 10 10 10 10 10...
('new line' at 08 08 08 08 08 08 08)

FLAGA Flag Sequence

FIGURE 5-3b

each line of highs FLAGA is shifted left once such that FLAGA is eventually set for lows again. Notice that manipulation of this and the other flags discussed is dependent on data, especially the framing indications. The proper manipulation of the pointers is, therefore, crucial to the operation of the controller and will be discussed next.

As was mentioned earlier, much computation is devoted to pointer manipulation. There are even some special subroutines for incrementing the pointers when framing indication are received. Starting just after the initialization process is completed, the first operation involves getting the new pel value and storing it in the original data storage area at the address indicated by the pointer*, ORIG. The pel is then checked for a 'new line' or a 'new page' indication. For a 'new page' indication two subroutines** are called which set the ORIG and GRADIENT pointers to the first location of the next line in their respective storage areas. Note that since each line is allocated 1kbyte, it is necessary to increment the high order byte of the pointer by four to move to the next line*** These routines also take care of wrapping the pointers to

---

* Note that all of the pointers, except those for the output buffers, are incremented further down in the program.

** The subroutines are named INCDAT and INCGR, respectively.

*** The six MSB's of the pointer determine the line number, and the ten LSB's the location on that line.

guarantee that they remain within the bounds of their respective storage areas. The 'new page' indication is then removed from its original location and put at the new one indicated by the pointer. For a 'new line' indication, the same two routines are called to move the ORIG and GRADIENT pointers to the first locations of the next line in their respective storage areas. They are each then decremented once, such that the point to the last location allocated for the previous line. Thus it is guaranteed that on the next pass the pointers will point to the first location of the desired line. The 'new line' indication is left in its original location to mark the end of valid data on that line, but is also put at the new location indicated by the pointer. This serves as a marker for when some reference from the pointer crosses the boundary between lines and is important in that it makes it unnecessary to recheck the pointer when making references to other pels on the same line.

The gradient is then computed for the area centered at the pel just above the pel indicated by the ORIG pointer. The GRADIENT pointer indicates the address where the results will be stored. The inner working of the Gradient subroutine are discussed in appendix C.

The next section of the controller is concerned with checking for framing indications at the pel locations which would correspond to the center of the vertical filter area and the location just after, i.e. to the left of, the center of the horizontal filter area, setting the STATUS flag accordingly. Starting with a copy of the ORIG pointer store in the cpu's memory pointer*, this pointer is first moved back four lines, to the center of vertical filter area, then back (left)

---

* this is register pair HL in the Intel 8080 or 8085 assembly code

five pels to the desired location in the horizontal filter area. It then either calls or doesn't the filter subroutines, depending on the state of the STATUS flag and the results of the checks just mentioned. Whether or not filtering is enabled, or has even begun, an attempt will be made to separate out the highs from the original data. Therefore it moves the pointer back four additional lines and also back three more pels, so that the pointer indicates ORIG (-7,-8). Since the low pointer LOWS* indicates the address of the results of filtering an area centered at ORIG (-4,-4), it must be moved back three lines and then back three pels. The value at LOWS (-7,-8) is then subtracted from the value at ORIG (-7,-8) to obtain the highs value. The pointer ORIG (-7,-8), the value at ORIG (-7,-8), and the highs value are all saved on the stack for later use and the controller proceeds with the edge (highs) enhancement. The magnitude of the highs value is determined and stored temporarily in the memory location labelled THIGHS (Temporary HIGHS buffer). The original data value is put in the lower byte of the memory pointer and the address of the luminance scale factor lookup table is put in the high order byte. The highs magnitude is then multiplied by the resultant value** and the product stored in THIGHS. Next the routine LOCCON (LOCal CONtrast) is called returns the appropriate value in reg H. This value is moved to the low order byte of the memory pointer and the address of the local contrast

---

* From chapter4, the lows pointer corresponds to LOWS (-4,-4) relative to the more recently received pel. That convention is followed here also.

** The value is obtained from the table

lookup table is put in the high order byte. The value in THIGHS is the multiplied by the resultant value and the product is combined with the original highs value from the stack; the result is the fully enhanced highs value. This is then moved to the lower byte of the memory pointer and the address of the compressor lookup table is put in the high order byte. The result is then passed on to the noise processor, NOISEP, which will add in pseudo-random noise. It uses as an index the three LSB's of ORIG (-7,-8) and the three LSB's of the count of the number of lines of highs that have been enhanced*. This count is set to zero when the 'new page' indication occurs at ORIG(-7,-8) and is incremented each time a 'new line' indication occurs there.

When the filters are enabled, the vertical filter is given three pointers; one for accessing the original data, ORIG (0,0), one for accessing the filter coefficients, and one for the storage address of the result, VLOWS (0,-4). The sixteen most recent results are kept, although only nine are needed by the horizontal filter, in order to simplify the updating of and referencing from the VLOWS pointer. The horizontal filter given three pointers; one for accessing data, VLOWS (0,-4), one for accessing the filter coefficients, and one for the storage address of the result, LOWS (-4,-4). The horizontal, when given the pointer LOWS (x,y), will fill in the locations LOWS (x-3,y), LOWS (x-2,y), and LOWS (x-1,y)** The vertical interpolator, given the same pointer, will fill in location LOWS (x, y-1). Thus to fill in the area around the sample point the controller calls

---

* The count is referred to in the program as NCNT

** The horizontal interpolator will not leave the current line to fill in values.

the vertical interpolator with pointer LOWS (-4,-4), i.e. the most recently determined lows sample point. It then calls the horizontal interpolator twice, first with pointer LOWS (-4,-4) and then with pointer LOWS (-4,-5).

Whenever a lows sample point occurs at LOWS (-7,-8), it is loaded into the output buffer area. The input and output pointers start at the same location, but, as was mentioned in discussing the flag STOUT, the transmitter output is disabled until after a full line of lows sample points has been loaded.

The last part of the controller could be considered the actual transmitter. It consist of two routines; one which gathers the next eight bits to be transmitted together into one byte and another which takes data from the output buffer and gives it to the first routine, accompanied by an indication of how many bits were to be transmitted from each value*. The first routine takes one bit from the location referred to as XMIT. It then updates the location XCNT by shifting it once to the left. If the shift results a carry bit being set, the second routine is called to reset XCNT and to put a new value in XMIT. It repeats this sequence seven more times, i.e. until it fills up the byte to be transmitted. Note that by setting XCNT appropriately, the second routine can, in effect, truncate any desired number of bits from the data words as they are transmitted; thus there is no need for a separate quantizer for the highs. The control aspects of these routines were discussed previously, with the exception of one issue. While highs are being transmitted there

---

* The two routines are referred to as TXOUT and UPDATE, respectively in the program.

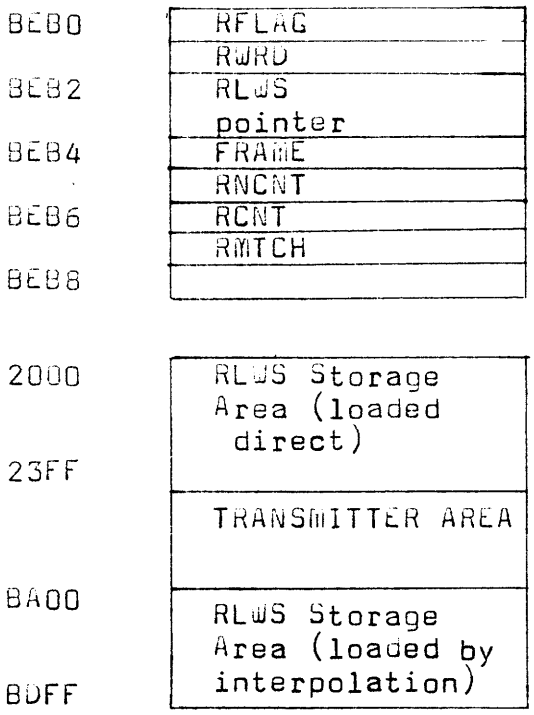
must be some means of detecting when the end of each line has been reached. Since there are no framing indications in the highs area of the output buffer and since there was no static relationship between the corresponding output pointer and any other existing pointer, another pointer was created. Its location on a given line* is kept in agreement with that of the highs output buffer's output pointer, but its line number** is such that it falls within the original data storage area. Thus this pointer can be used by the transmitter to monitor the original data storage area for framing indications. There is a separate subroutine, INCHBF, which is very similar to INCDAT and INCGR, which is used to for either of the pointers for the highs output buffer. More details on the other subroutines can be found in the appendices.

V-3 Receiver Controller

The receiver controller is much less complex than the transmitter, thanks largely to the pains taken in designing the transmitter to present the data in a simple format. The receiver does not have much to do in the way pointer manipulation and, in fact, only requires a little over 3kbytes of data storage. The memory allocation is shown in Figure 5-4. There is only one data pointer and one control flag. The flag, RFLAG, determines what type of data the receiver will be expecting. Figure 5-5 shows this flag and what each of its bits indicates. The receiver is initially in the 'off'

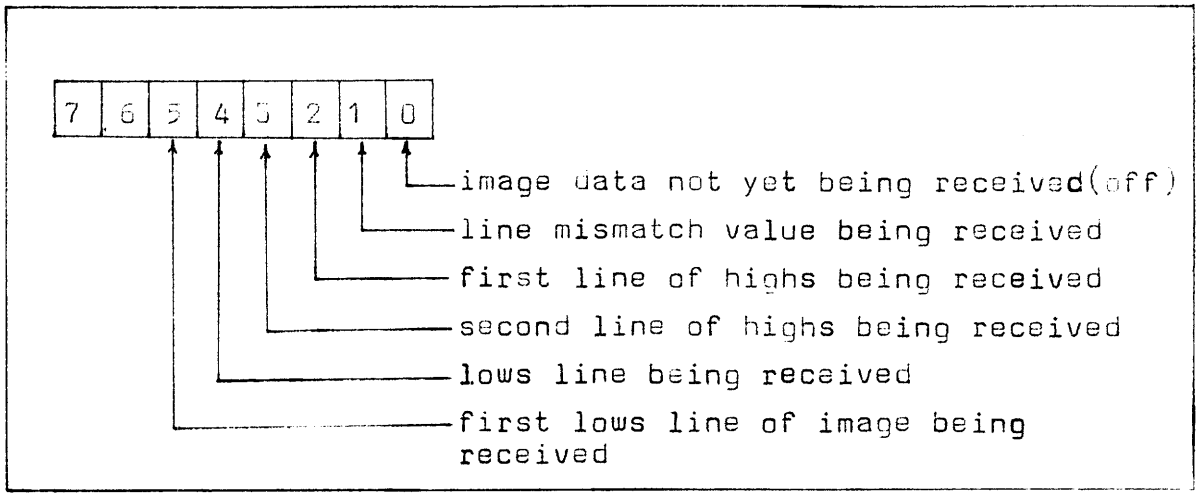
---

* 1Ø LSB's of the pointer
** 6 MSB's of the pointer



RECEIVER-Flags, Pointers, and Memory Allocation

FIGURE 5-4



RFLAG Flag

FIGURE 5-5

-- The sequence shown would correspond to a line length of sixteen pels. One state is shown for each byte received.

```
initially
set to 01 -- 01 01 01 01 02 20 20 08 08 08 08 01 01 01 01
                  'new      'new                      'new
                  page'     line'                     line'

             04 04 04 04 08 08 08 08 10 10 10 10 04 04 04
                                                'new
                                                line

             04 08 08 08 08 10 10 10 10 04 04 04 04 08 08
                                       'new
                                       line'

             06 08 10 10 10 10 04 04 04 04 08 08 08 08...
                           'new
                           line'
```

RFLAG Flag Sequence

FIGURE 5-6

state, (bit Ø set), and remains there until a 'new page' indication is detected. It then expects the line mismatch value, (bit 1 set), followed by the rest of the lows line (bit 5 set). At the end of this line, indicated by a 'new line' indication being received, RFLAG is shifted right twice such that the second line of highs is indicated, (bit 3 set). The end of the highs line is determined by monitoring the corresponding lows values for framing indications as the highs are being received. At the end of each of these lines, RFLAG is shifted left once. Since bit 4 indicates lows, only one line of highs will be expected to follow the first line of lows. Since, however, all subsequent lows lines are indicated by bit 4 rather than bit 5, two lines of highs will be expected after each of these lows lines. Figure 5-6 shows the sequence described and it can be seen to agree with the transmitter's output format.

The program to implement this controller is listed in appendix I. For test purposes it is called from the transmitter when it presents data to the transmitter. This is necessary in the simulator version since there is no way to interrupt the processor and hence the transmitter routine. In a real world application the transmitter and receiver would be implemented on separate processors. The next chapter deals somewhat with such issues. Since the same processor is used here, however, the transmitter must be both initialized before any processing can begin. The initialization routine for the receiver is thus inserted just before the transmitter initialization.

When the receiver routine, RCVR, is called it stores the value given to it in the accumulator in memory location RWRD and sets RCNT, the number of bits left in RWRD, to eight. As each bit is read, this value is decremented such that control returns to the transmitter when the bits are exhausted. In all of the various receiver modes the location, FRAME, is used to collect the bits as they are read. These bits are shifted in from the right as they are read and one of the bits is in FRAME set initially such that a carry bit will occur when it is full. For example, when the 3-bit highs values are being received, bit 5 of FRAME is initially set. When the 8-bit lows values are being received, bit Ø of FRAME is set, so that carry bit occurs when the eighth bit is loaded.

The receiver consists of four sections which are executed almost totally independently of each other and a small section that determine which of these will be executed for a given RCVR call. These sections correspond to the four types of data expected by the receiver and begin at the points labelled ROFF, RMSMTH, RLOWS, and RHIGHS.

The receiver is initially in the 'off' state and in this mode the section beginning at ROFF is executed. It continually checks loads new bits into FRAME until it either runs out of bits from RWRD or until it detects a 'new page' indication in FRAME, i.e. the eight most recently loaded bits were all 1's. The first case, simply results in a return from subroutine. For the second case, however, the pointer RLWS is moved to point to the

first location of the other directly loadable line and the new page indication is stored there. Note that although there are actually 3 lines for lows storage, one of these is reserved to be filled in exclusively by interpolation from the values on the other two lines. FRAME is then set to accept eight bit, i.e. bit Ø is set, and RFLAG is set to indicate that the line mismatch value is to be expected next. Similar to the transmitter, the receiver has a line counter that is used by the PRN processor, that is reset to zero at this time. Control is then transferred to the section that receives the line mismatch value, i.e. the section that begins a RMSMTH, so that whatever bits that may yet remain in RWRD can be loaded into FRAME before returning from the subroutine.

When the receiver is in the mode that executes the program section that expects the mismatch value, the next eight bits* received will be loaded into FRAME. These eight bit should correspond to four identical 2-bit line mismatch values. These values are separated and added together. The result should be the line mismatch value shifted right twice, and if either of the two LSB's in nonzero some error must have occurred. The value is then rounded off to the nearest multiple of four and the result shifted right twice to obtain the actual line mismatch value and

---

* These eight bits will come from more than one input byte, unless the 'new page' indication that was received came from a single input byte without overlapping.

stored in RMTCH. The flag word, RFLAG, is then set to indicate the first lows line, bit 5 set, the bit accumulator, FRAME, is set to accept eight bits, and control is transferred to the lows receiver, which begins at RLOWS, so taht the remaining bits in RWRD can be loaded into FRAME.

The section used to receive the lows, not only collects incoming bits into the 8-bit lows values, but also spaces them appropriately and calls the two interpolators to fill in the missing values. The horizontal interpolator is identical to the one in the transmitter, except in the way it deals with 'new line' indicators. The vertical interpolator fills in the corresponding location on the line reserved for indirect loading. the horizontal interpolator is then again used to fill in the three locations between this and the previous location filled by the vertical interpolator.

If the value loaded was a 'new page' indication, the section starting at RPAGE is executed. It switches the pointer, RLWS, to point to the first location of the other line and stores the indication there. RFLAG is set such that the line mismatch value is expected next. FRAME is set up to accept eight bits and control is transferred to the mismatch receiver section. The line counter is also reset to zero. Thus images can be transmitted one after another if the transmitter maintains the format expected by the receiver.

If the value loaded was a 'new line' indication, the section starting at ENDLW is executed. It, too, switches the lows pointer

to point to the first location of the other line. This insures that the pointer will indicate the most recently loaded lows line when the highs are being received. Before this is done, it is necessary to augment the current line of lows values according to the line mismatch value. Zero is loaded into the current location and into each subsequent location until the one corresponding to the current pointer plus the mismatch value. The same is done to the corresponding locations on the line reserved for interpolated values.

The flag word, RFLAG, is then set to receive either one, bit 3 set, or two, bit 2 set, lines of highs values, depending on the current state of the flag. Remember that for the first lows line, bit 5 is set, while all for subsequent low lines, bit 4 is set. The flag word is shifted right twice when the end of the lows line is reached, thus setting the receiver to expect the appropriate number of highs lines according to the transmitter's data format. The bit accumulator, FRAME, is then set to accept three bits, i.e. bit 5 is set, and control is transferred to the section beginning at RHIGHS so that any remaining bits in RLWS can be read into FRAME.

The section that receives highs values also generates the system output and framing indications. After a three bit highs value has been loaded into FRAME's three LSB positions, it is shifted over until it fills the three MSB positions. The pointer for the lows is then put into the memory pointer register and, if the values are for the first line of highs, i.e. if bit 2 is

set, the six MSB's of the high order byte are switched to indicate the line of lows that was loaded indirectly by means of interpolation. Then the PRN value is obtained from the lookup table, using the three LSB's of the line counter, RNCNT, and the three LSB's of the lows pointer as indices. The routine for obtaining the PRN value is identical to the one used by the transmitter and it was only made a separate routine so that the transmitter and receiver would be totally independent of each other. The result is subtracted from the highs value and this value is expanded, again via a lookup table.

The scaled highs value is then added to the corresponding lows value and the result is ouput as the enhanced image data. The lows values are checked for framing indications and if any occur a prompt is sent to some external device to generate the necessary framing pulses for the facsimile reproduction device. When the 'new line' indication is detected RFLAG is shifted left once and if the second highs line is indicated, i.e. if bit 3 is set, then FRAME is set to accept three bits and control remains with the section starting at RHIGHS. For a 'new page' indication, FRAME is set to accept eight bits and control is transferred back to the section which begins at RLOWS. In the absence of any framing indications, FRAME is set to accept three bits and control remains with the highs receiver section.

VI. Microcomputer Implementation

The hardware implementation of the image enhancement/transmission system discussed in the preceding chapters is achieved using a microcomputer based on the Intel 8085 microprocessor. This system should be capable not only of carrying out the enhancement and transmission procedures, but also of completing them in less than one millisecond, as the initial specification stated. This does not include the time that is devoted to the receiver since it would be implemented separately, or at least would not run concurrently with the transmitter, in any real world application. This chapter discusses the design of this system, beginning first with the overall system requirements, then going into its detailed design, and the I/O structure.

VI-1 System Requirements

The microcomputer requirements discussed in this section pertain mainly to requirements on data storage capability, software and parameter storage capability, and CPU requirements. These are based on and/or derived from the discussion in previous chapters.

In discussing the transmitter portion of the system, it was mentioned* that much of the effort involved manipulation of the pointers for the various data storage areas. The storage requirements there were 9k bytes for the original image data, 16k bytes for the gradient magnitude data, 2k bytes for the local contrast

---

* chapter V

update data, 5k bytes for the lows data, 4k bytes for the highs output buffer, and 512 bytes for the lows output buffer. In the receiver, the only storage required mentioned was 3k bytes for the lows data. Thus the implementation of both the transmitter and the receiver in the same microcomputer system requires 39.5k bytes of data storage capability. To this must be added the data storage needed for storage of filter coefficients and the intermediate lows values for the transmitter plus the flags, pointers, and stack area for both the transmitter and receiver.

The software requirements for the transmitter and receiver implementations discussed in the previous chapter are approximately 15ØØ and 512 bytes, respectively. Later in this chapter, I/O considerations will be discussed which will increase the overall software image requirement somewhat. It should not, however, be large enough an increase to cause the overall requirement to exceed 2k bytes.

The requirements for parameter storage were not discussed directly in the previous chapters, but can be determined from the inferred sizes of the various lookup tables. First mentioned were the two scale factor lookup tables and since the indices of each can range from Ø to 255*, 256 bytes of storage is required for each. Observing the behaviour of the curve for the local contrast scale factor, however, reveals that it is a constant for any input value greater than sixty. Thus by adding a few

---

* i.e. each is accessed with a one byte index

steps to the part of the transmitter controller program that does the lookup, it was possible to decrease the storage requirement for the corresponding table by 75%. The compander pair each also have to cover the full range of an 8-bit index and, therefore, require 256 bytes of storage each for their respective lookup table. The lookup table for the PRN sequence uses 6-bit indices and, therefore, requires 64 bytes of storage. Thus the requirement for parameter storage comes to a total of 896 bytes.

The CPU requirements are not so well defined as the storage requirements. The main consideration was in making sure that it would be capable of running at a speed sufficient to execute either the transmitter or the receiver program in less than one millisecond per cycle. Each of the filters uses nine multiplications and the adaptive scaling operations each also require a multiplication. There is very little chance of any microcomputer executing twenty multiplications in less than one millisecond, using a software multiplier*. Therefore, in a real world application it is assumed that this software multiplier would be replaced by some sort of external hardware mulitplier**. Using a subroutine to drive this external multiplier, multiplication can be done in under twenty instruction cycles. If time permits, this replacement will be made. Even so, there is a good deal of other

---

* see the appendices

** This is discussed again in chapter VII and the appendices.

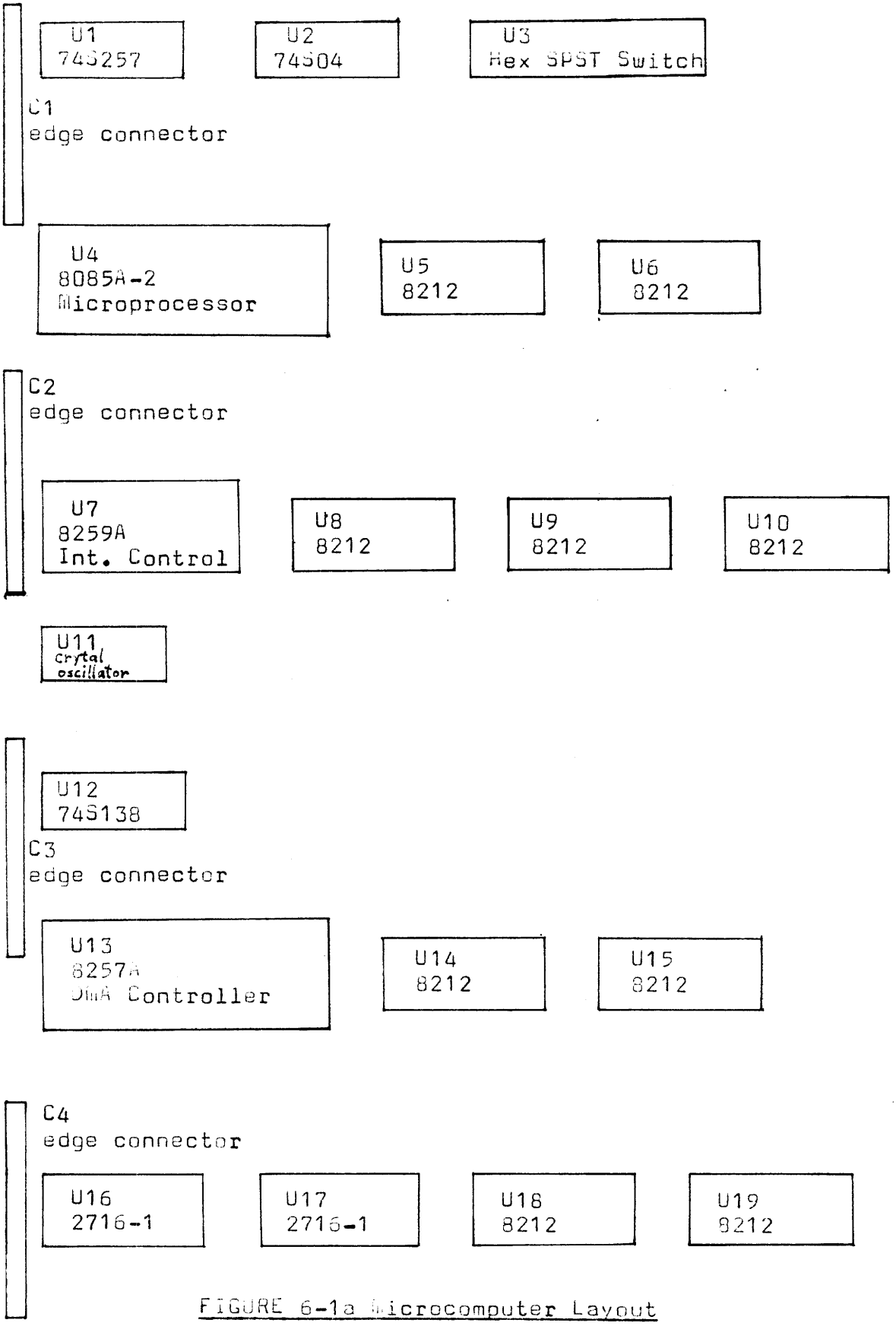


FIGURE 6-1a Microcomputer Layout

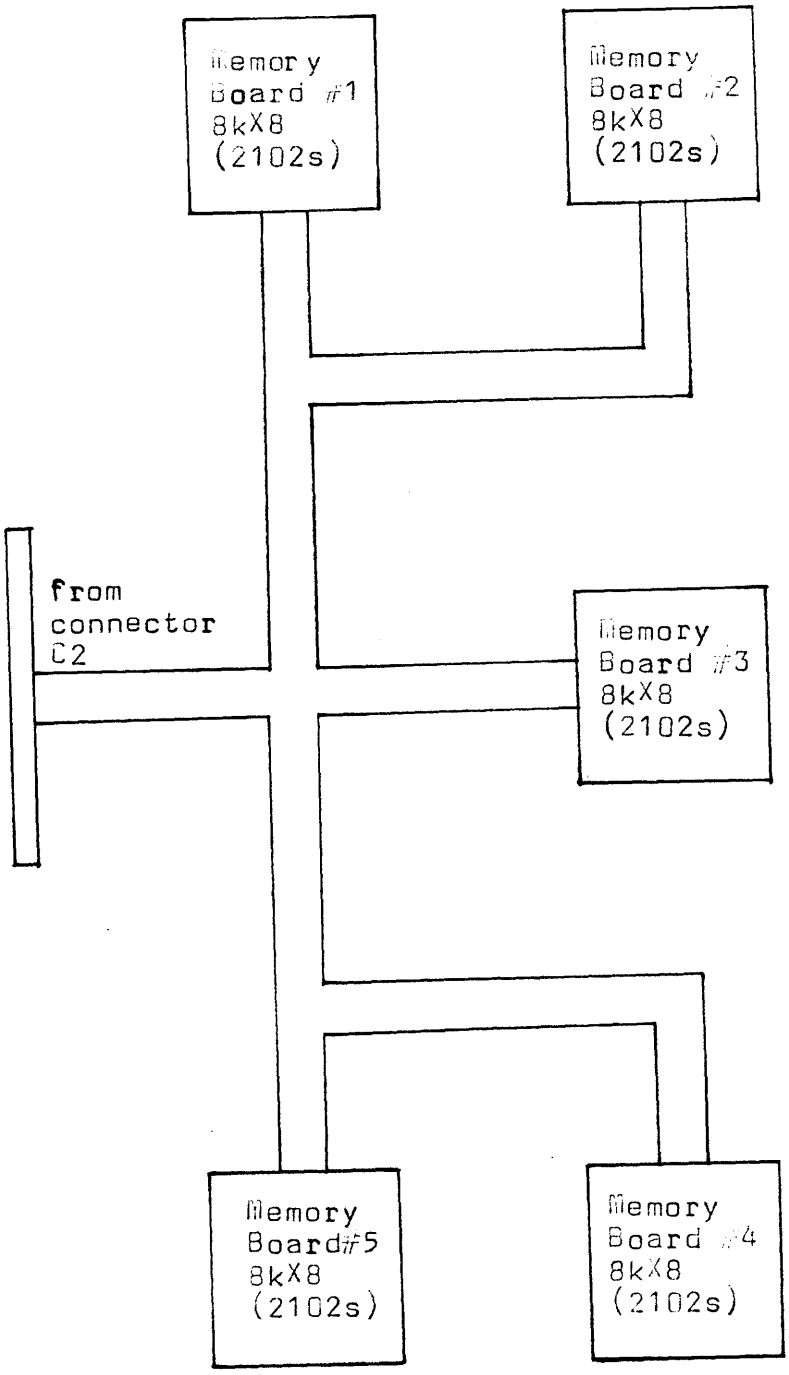


FIGURE 5-1b Microcomputer Layout

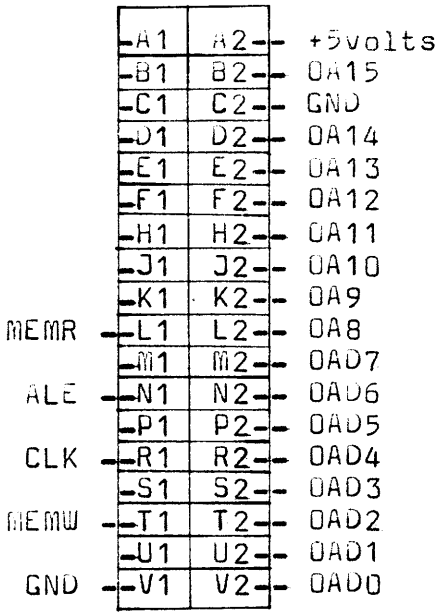


FIGURE 6-1c Microcomputer Layout
(Edge Connector C2)

computation to be done and the processor must be able to in a short enough period to at least allow time for multiplication using the hardware device.

VI-2 Microcomputer System Design

The section presents the detailed design of a microcomputer system capable of performing the functions of the transmitter and receiver as discussed in the previous chapters. The Intel 8085, rather than the 8080, is chosen as the central processing unit for this system, mainly because of its greater speed* and its total compatability with the 8080 software used in the simulator implementation discussed. It also requires fewer support devices for the basic system. In the following sections, this system is divided into five major groups: 1) the CPU group, 2) the ROM group for program and parameter storage, 3) the RAM group for data storage, 4) the I/O group, and 5) the DMA group. The information for the devices incorporated in this design was obtained from various microprocessor and TTL data books (see references 12 through 17).

A. CPU Group

The 8085 microprocessor requires very little assistance from support devices to perform the tasks of the CPU for this system. It has an 8-bit data bus and a 16-bit address bus, the low-order byte of which is multiplexed with the data bus. Thus to have the

---

* The 8085A-2 has a minimum instruction cycle time of 800nsec, while the 8080A-1 has a minimum instruction cycle time of 1300nsec.

full address available continuously, it is necessary to latch the low-order address byte. In addition to insure that its output drive capabilities are not exceeded by the loading of the other devices on the bus, buffers are inserted between the 8Ø85 and the data and address buses. The data bus requires a bidirection buffer and the address bus, since the low-order byte is already buffered, only requires a buffer for the high-order byte. Notice that the low-order address byte is latched by the signal ALE (Address Latch Enable) and the direction of the bidirectional data buffer is determined by the 'read' strobe from the microprocessor, -RD. A multiplexer, 74S257, with tri-state outputs, is used to generate the signals - MEMW,-MEMR,-I/OW, and -I/OR from the 8Ø85 signals -RD,-WR, and IO/-M and the various input which are not used are either tied to GND or 'pulled up' to V_{cc}, whichever corresponds to the particular input's inactive state.

Figure 6-2 shows the detailed circuit diagram for the CPU group. Note that although the 8Ø85 has an internal clock generator, whose frequency is set by component(s) placed across inputs X1 and X2, in this implementation the X1 input is driven by an external 5 MHZ crystal oscillator. This was done entirely because of availability and for convenience. The 5 MHZ clock frequency at this input corresponds to a 2.5 MHZ frequency at the CLK output of the 8Ø85, only half the maximum clock rate.

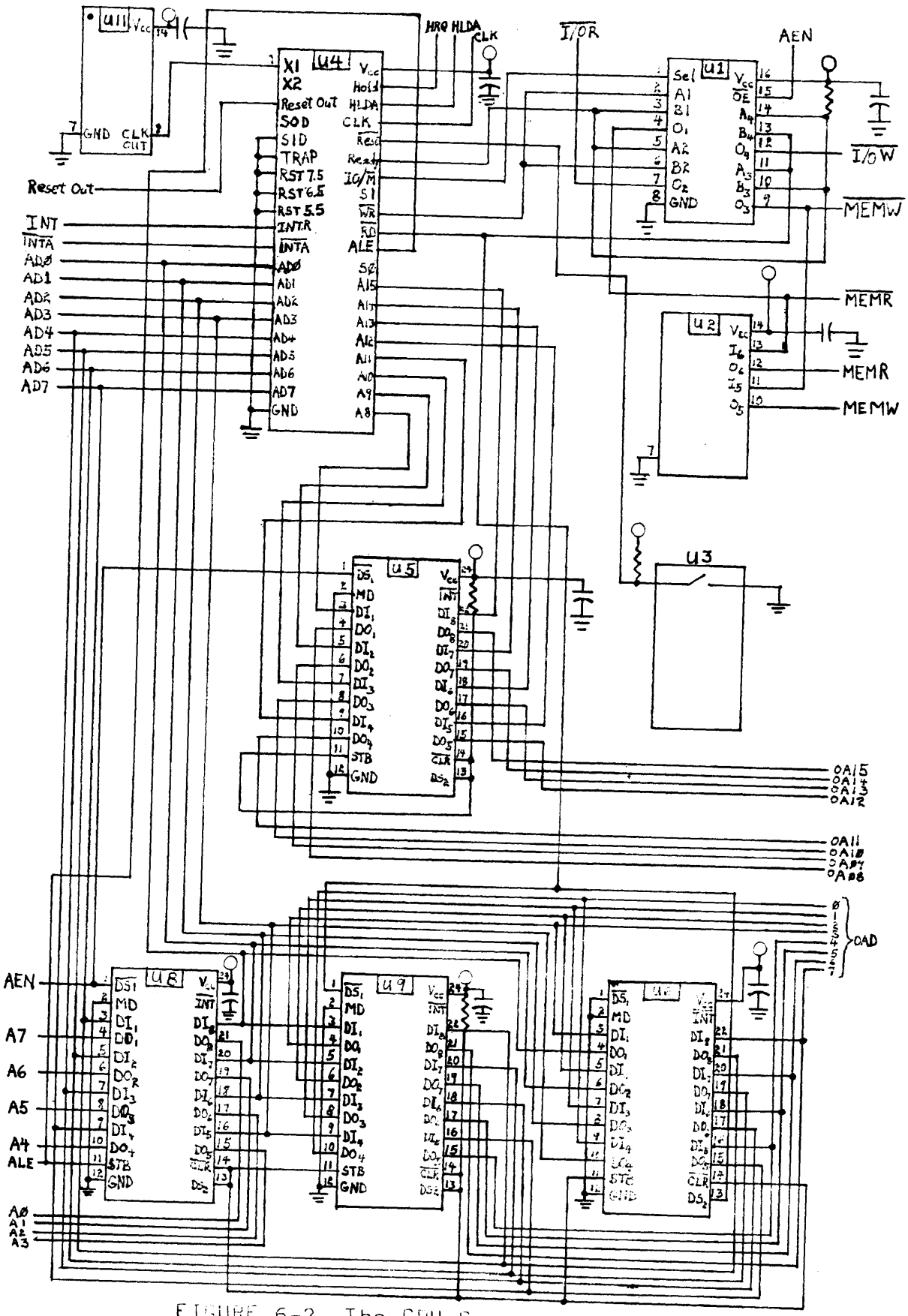


FIGURE 6-2 The CPU Group

B. ROM Group

The ROM device chosen for this system is the Intel 2716-1 and has access times short enough to allow its use with the 8085A-2 at the full 5 MHZ clock rate. It is a 2k byte device and and has two enable inputs, -OE (Output Enable) and -CE (Chip Enable). The delay from the address being stable or from -CE being enabled to the corresponding output is 350 nsec maximum and the corresponding delay from the -OE is only 120 nsec maximum. Therefore, the -CE input is connected to the enable from the address decoder* and the -OE input is connected to -MEMR signal, which occurs somewhat later in the machine cycle. Fig. 6-3 shows the detailed circuit diagram for the ROM group.

C. RAM Group

The RAM group is implemented with five 64k bits memory boards built around the 2102**memory device. The boards were designed for another system and their control circuitry had to be modified somewhat in order to convert them from 4kx16 to 8kx8 and to make them totally compatible with the CPU's control signals. In addition, many of the 2102's used were seen to of a variety that would not be able to operate properly were the CPU operated at its maximum rate. Thus the overall system is not capable of performing for the ROM group its functions in the specified time,

* The 74S138 is used to determine which of the 2k byte memory segments is indicated by bits 11-15 of the address bus.

** The 2102 is a 1k x 1 bit random access memory device with an access times ranging from 250 to 1000nsec, depending on the version used. See ref. 16 pp 282.

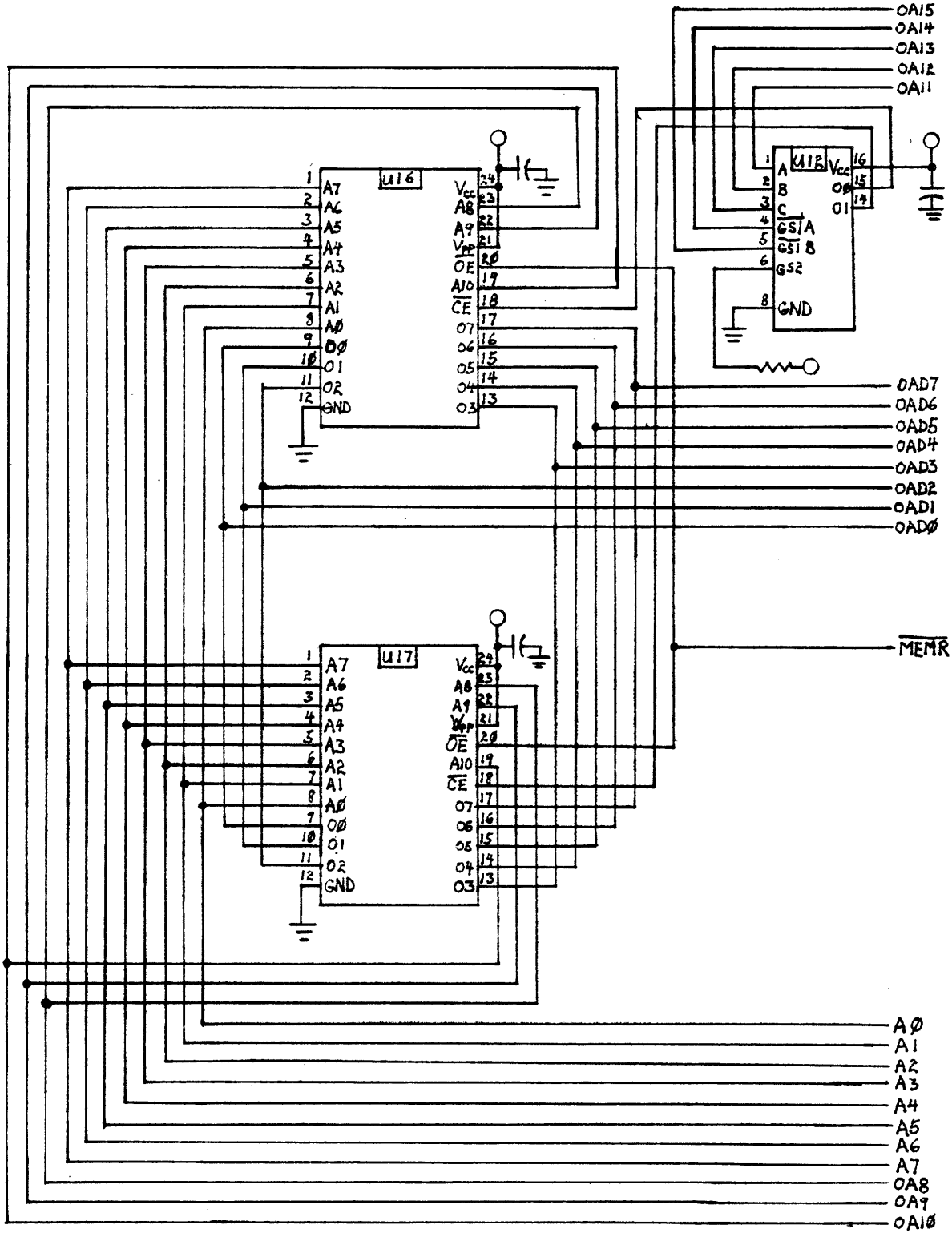


FIGURE 6-3 The ROM Group

unless the slower memory devices are replaced. Figure 6-4 shows the detailed circuit diagram.

D. I/O Group

The I/O capabilities of the system include receiving one byte data value on either of two input ports and transmitting one byte data on either of two output ports. For the system implementation discussed in chapter V, these ports are simply polled by the transmitter and receiver routines. To give the system presented here a more flexible I/O structure, it was decided to use an interrupt driven I/O scheme. This relaxes the timing constraints which would have to have been imposed upon the image data source, the transmission channel, and the facsimile reproduction device.

To implement this scheme, four I/O ports, 8212's, are used in conjunction with a programable priority interrupt controller.* The detailed circuit diagram is shown in Figure 6-5. The interrupt controller must be initialized by the CPU before any I/O operations can begin and special interrupt routines must be added to the transmitter and receiver software to achieve the desired I/O features. To relax the timing constraints each of the four interrupt routines uses a sixteen byte FIFO buffer as either a data source for the output routines or as a storage area for the input routines. The transmitter then gets data from or writes data

* The interrupt controller was chosen over the interrupt scheme incorporated into the 8085 itself because it allows more flexibility in positioning the interrupt vectors.

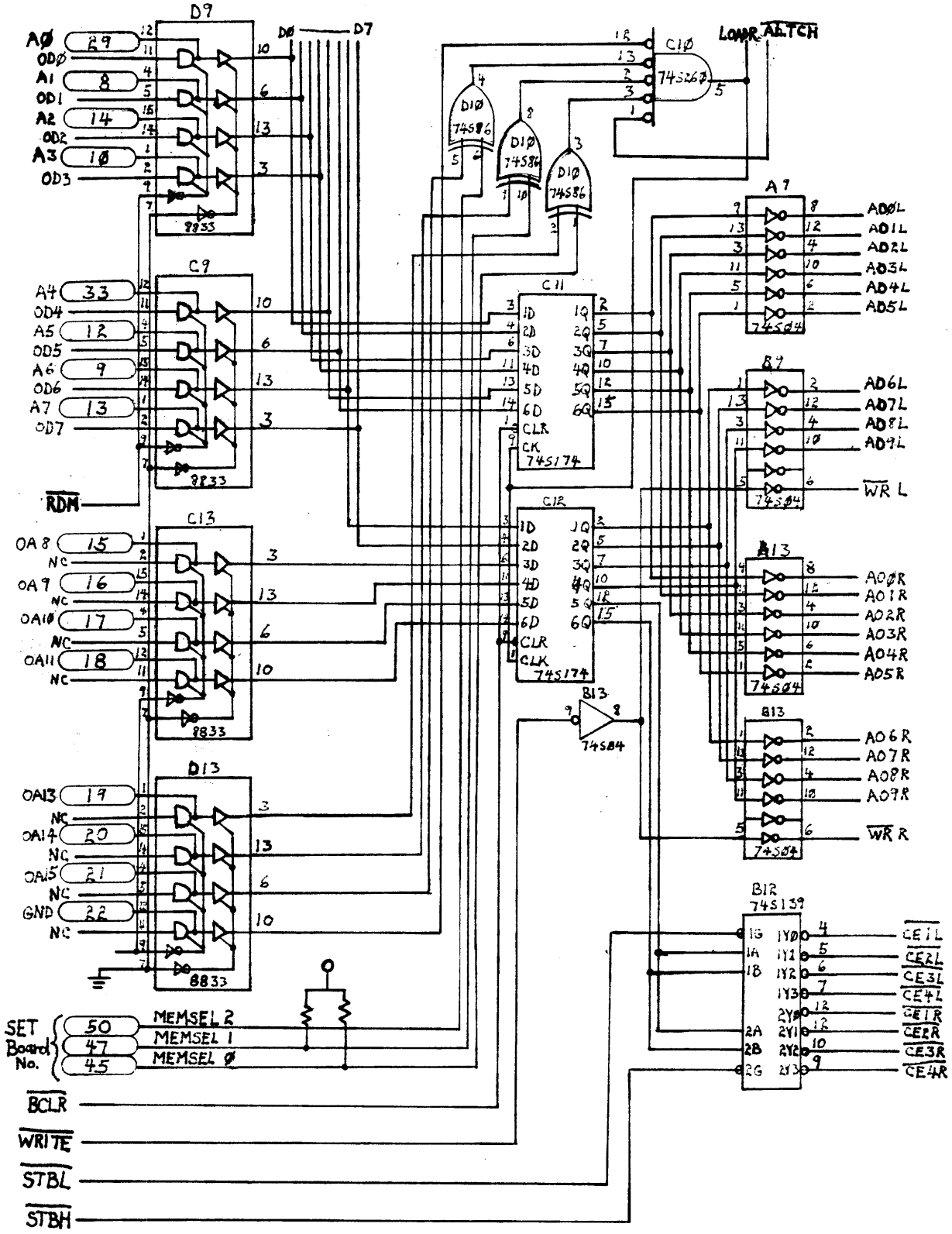


FIGURE 6-4a The RAM Group

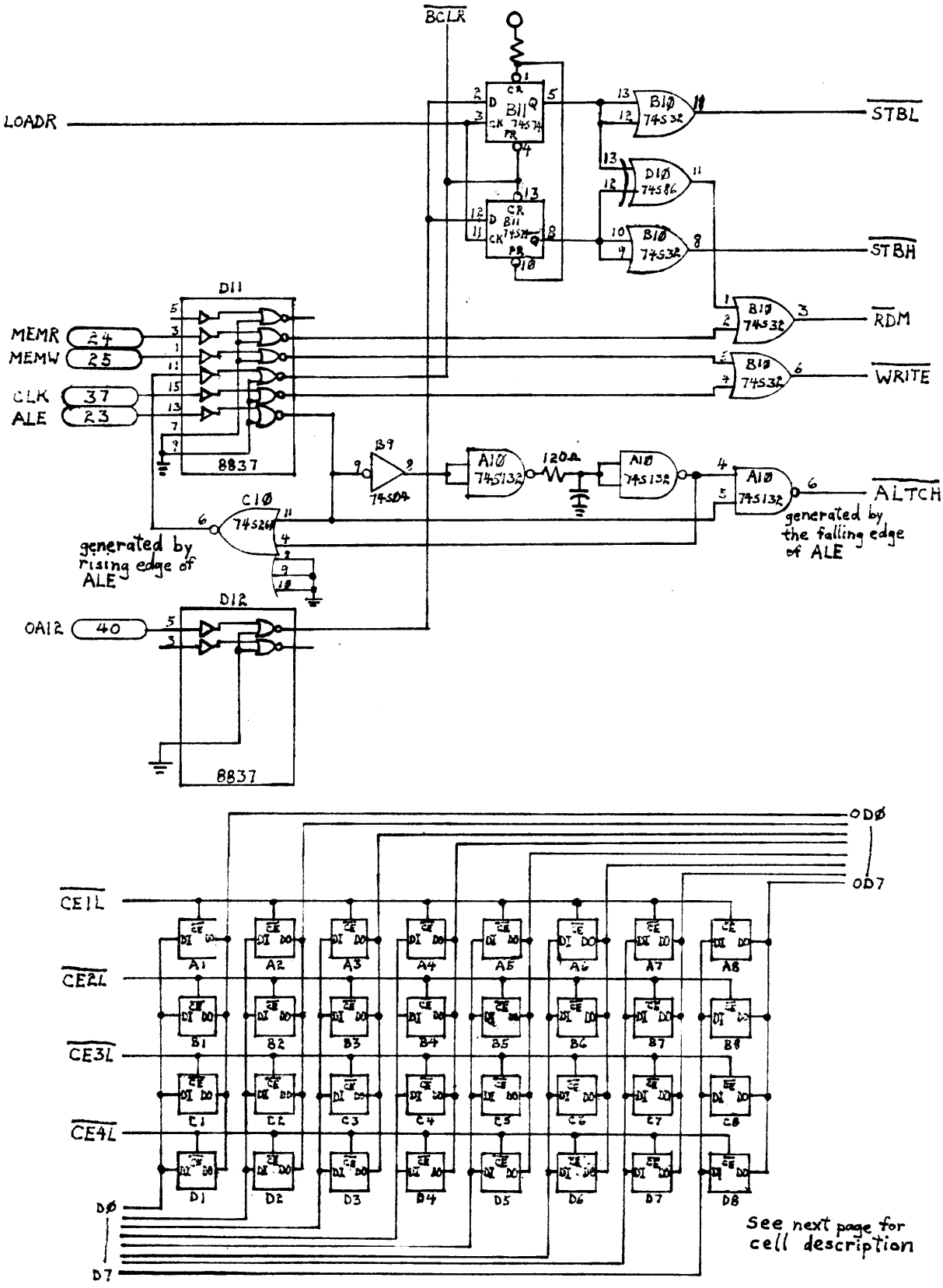
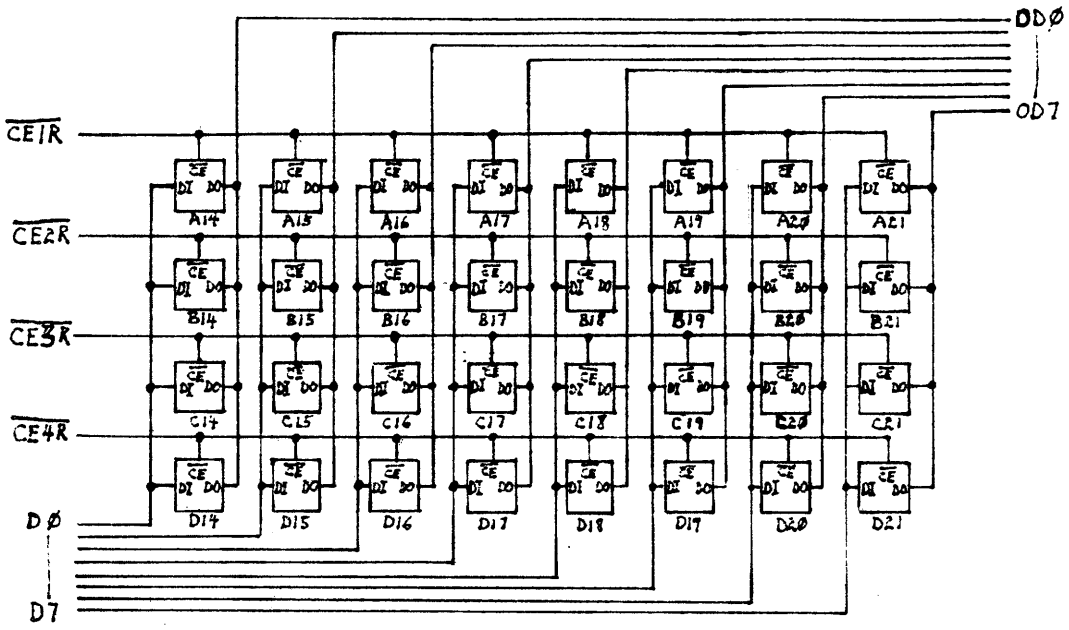
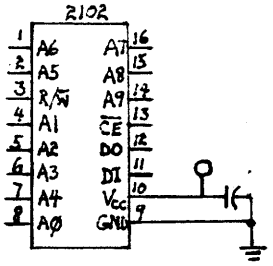


FIGURE 6-4b The RAM Group



each cell



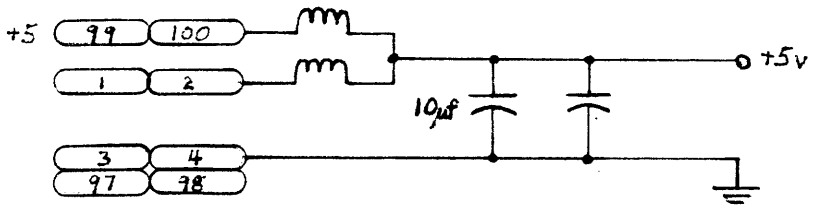


FIGURE 6-4c The RAM Group

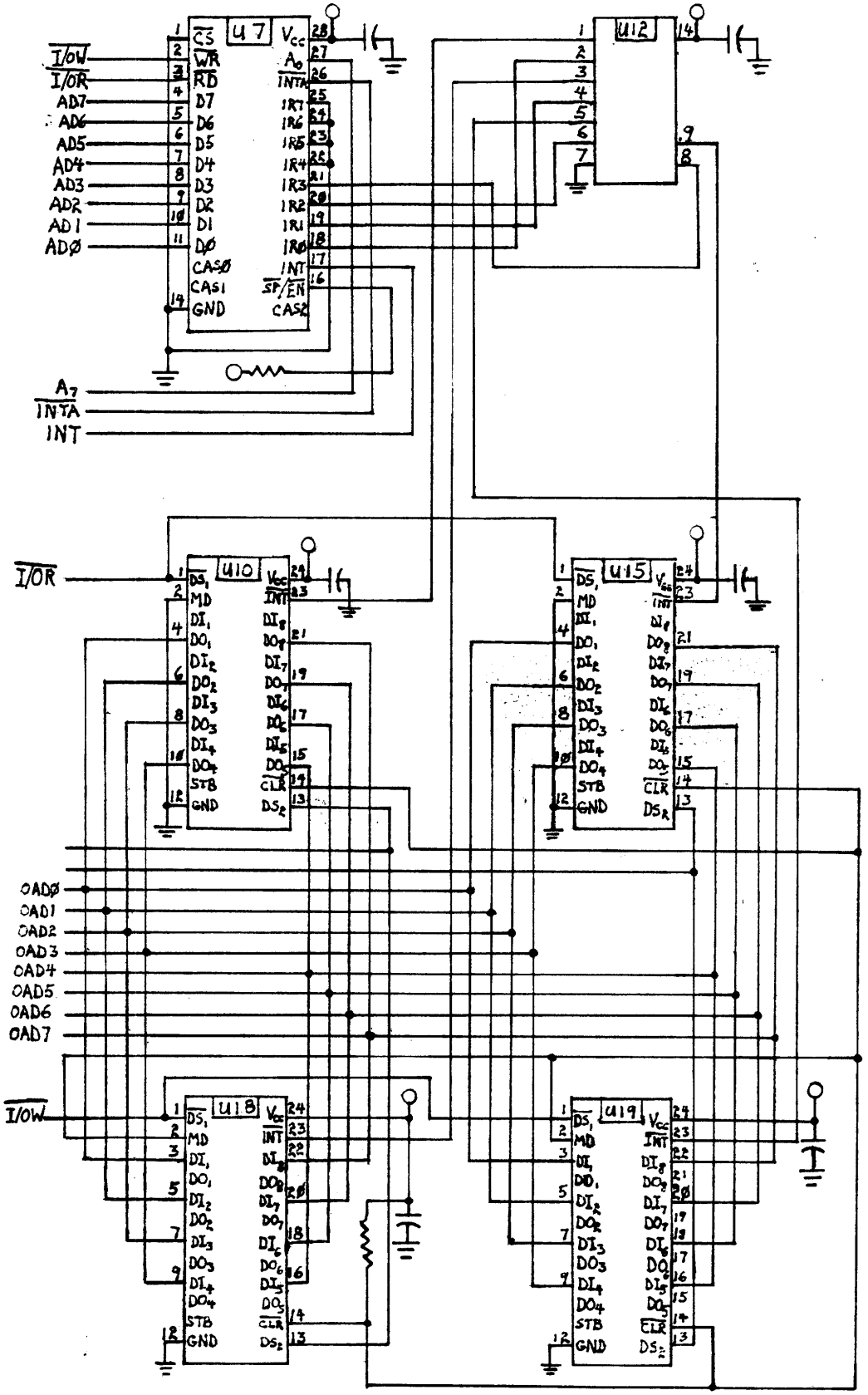


FIGURE 6-5 The I/O Group

into these buffer areas, rather than the I/O devices directly. The receiver and transmitter are also made to wait if, during a transfer, it is detected that either the appropriate output buffer is full or the appropriate input buffer is empty. Thus the only timing constraints are that the receiver and transmitter be able to run at least as fast as the scanning and reproduction devices and that these two devices operate at comparable speeds.

The interrupt controller is designed to send a 'call' instruction to the CPU, sending it to one of eight different locations that contain the corresponding interrupt vectors. These vectors are merly 'jump' statements to the appropriate interrupt routines which are used to actually update the storage buffers.

These interrupt vectors and the corresponding routines are appended to the routines presented in chapter V after the last lookup table and the receiver initialization routine. A few changes to transmitter and receiver routines are also made, so that data transfers are made only through the appropriate I/O buffer.

These interrupt routines and the initialization routine for the interrupt controller are listed in appendices. The controller can be programmed for either a static or a rotating priority scheme and requires two or three bytes for this programming. These bytes and their interpretation are shown Fig 6-6. Note that for the first byte, the input AØ is a logic Ø and that for the following bytes is a logic '1'. Any -I/OW or -I/OR signal that occurs, with AØ a logic 'Ø', can be taken by the controller as an attempt

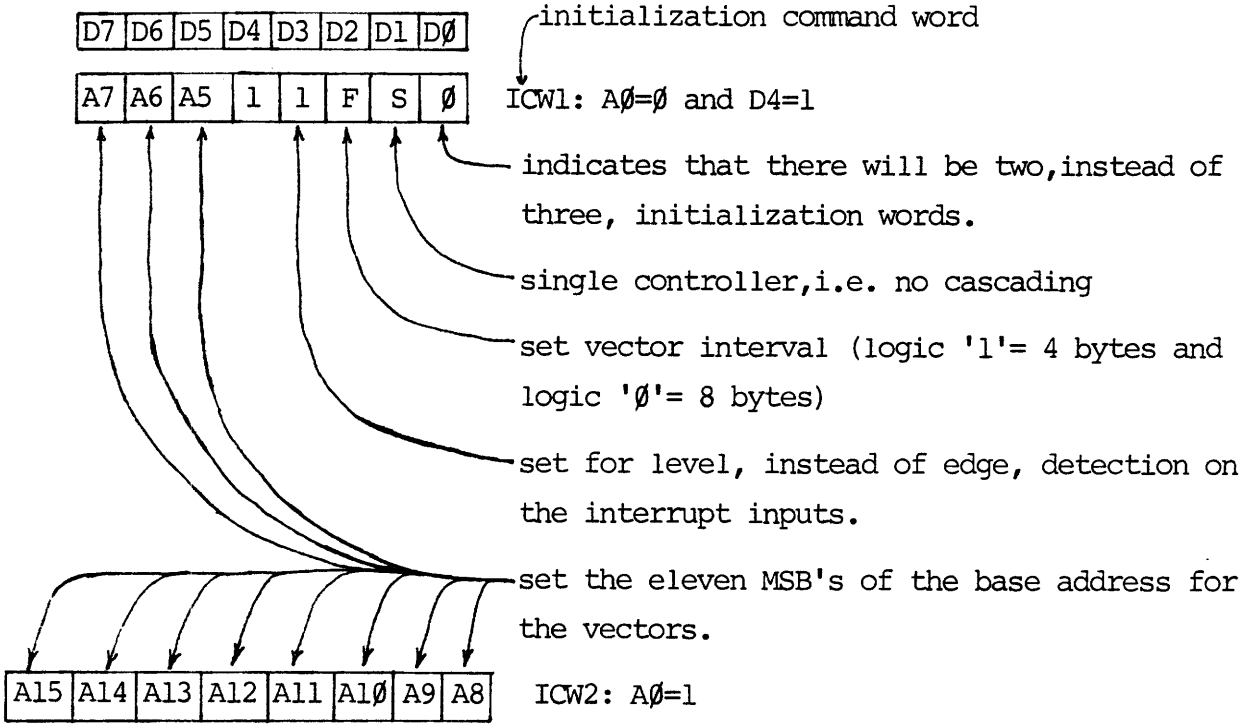


FIGURE 6-6. Interrupt Controller Program Bytes

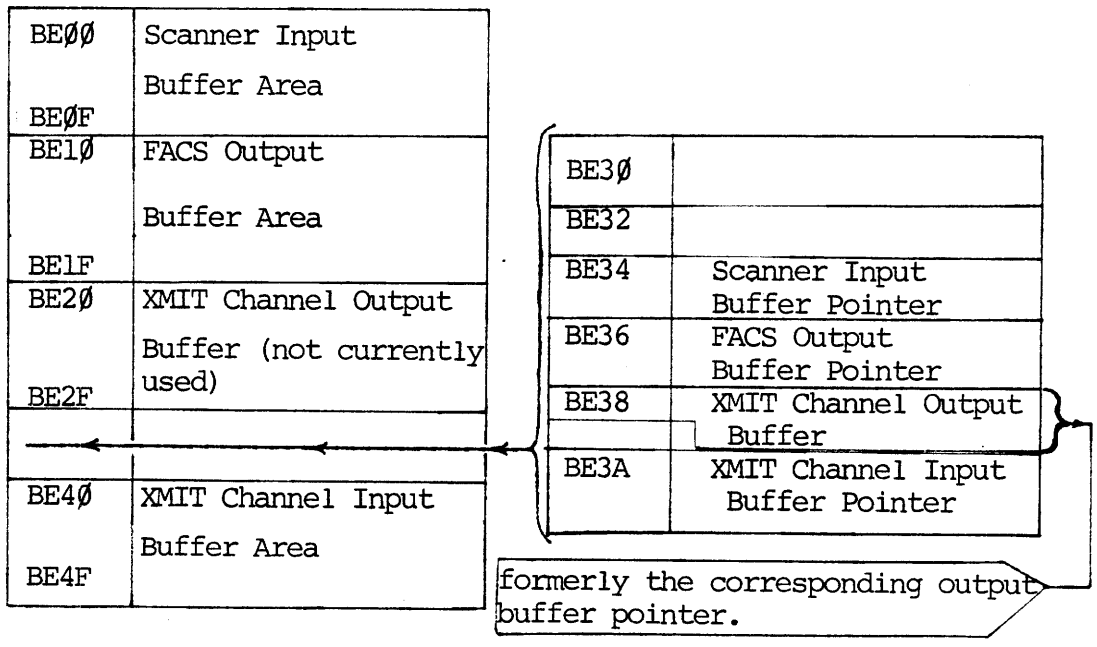


FIGURE 6-7 I/O Buffers and Pointers

to either begin programming it or to read its status or mask registers. Thus care must be taken in accessing the other I/O devices, to maintain AØ at logic '1'. Note that since there are only four I/O ports, no decoding of the address lines was required to select them. Each port is simply selected by a distinct address line and the appropriate I/O read or write signal. The port assignments, based on the state of the address bus during the transfer, are indicated in the detailed circuit diagram of Fig. 6-5.

There are two similar sets of interrupt routines for the various ports. The routines for the output ports use the pointers for the appropriate data buffer areas in order to access the oldest value stored there. There are no checks made to insure that the buffers are not empty since the loading routines, i.e. the transmitter and receiver routines are designed to be able to load these buffers at least as rapidly as they can be depleted. The loading routines from the transmitter and receiver make sure that they don't write over data not yet removed from the buffers by comparing their pointers to the corresponding pointers from the interrupt routines. The routines for the input ports use pointers to their respective buffer areas in order to load new data values over the oldest remaining data values. There are, again, no checks made to insure that the buffer areas are not already full since it is assumed that data is being removed from these areas, by the transmitter and the receiver, at least as fast as they can be loaded. The transmitter and receiver compare their pointers to

the corresponding pointer from the interrupt routines in order to insure that they do not attempt to remove data from empty buffer areas. Figure 6-7 shows the memory allocation for the various pointers and buffer areas discussed, including the additional pointers for the transmitter and receiver.

External devices which are to be connected to the I/O ports should conform to the following specifications for data transfers. Those used with input ports should present data, one byte at a time, accompanied by positive pulse* to strobe the data into the port and not attempt to input another byte until the interrupt line has returned to the high state. This will occur when the corresponding interrupt routine takes the data byte from the port, i.e. executes an 'in' instruction referenced to the port; thus the interrupt line is used also as a data acknowledge signal from the CPU to the external device. For devices requesting data from an output port, the request should be a positive pulse similar to the strobe used for input ports. This will cause the port to interrupt the CPU. When the corresponding interrupt routine loads the port, i.e. executes an 'out' instruction referenced to the port, the interrupt line returns to the high state, acknowledging the devices request and signalling that the data is available. Figure 6-8 shows these configurations.

E. DMA Group

The ability to directly load or read from memory was initially incorporated into the system with the intent of downloading most

---

* This pulse must be at least 3Ønsecs wide.

of the software and parameters from the UNIX system directly into random access memory devices. This would make it possible to load, test, and readily modify the routines on the microcomputer itself. The main device is the Intel 8257, a programmable DMA controller, and the 8212 is present only to latch the high-order address byte, which is time multiplexed with the data byte*. They are connected directly to the system data and address buses and the signal AEN (Address ENable) is used to give it control of these for a data transfer. The controller has four channels which can be used both to load and to read from the system memory or the I/O ports and can be programmed by the CPU to transfer any desired number of bytes, starting from any desired location. When it receives a request for a data transfer, DRQ, it sends a hold request, HRQ, to the CPU. Upon receiving the hold acknowledge, HLDA, signal from the CPU, it takes control of the system buses and sends an acknowledge signal, -DACK, to the device requesting the transfer. It then maintains control of the buses until the transfer is completed, at which time it removes the hold request to the CPU. Completion of the transfer is determined by the programming of the controller and not by the device being serviced.

In programming the 8257 the four lsb's of the address bus select the mode or any one of the channels. The 'mode' is program-

---

* The signal ADSTB is used to strobe this latch.

med by one byte and determines which channels are enabled, their priorities and the manner in which transfers are terminated, reinitiated, or carried out in general. The channels are programmed with two bytes for each the beginning address of the transfer and the type of, read or write, and number of bytes in the transfer.

Figure 6-8 shows the detail diagram for the DMA group*, as it presently exists. It was decided to suspend the further development of this portion of the system, since time did not permit the development of the necessary interface device between the UNIX system and because the downloading program for UNIX sends the data in a format that would not appear to be appropriate for direct loading into memory for execution. It would also be necessary to design some circuitry or modify the ROM structure to initialize the controller. The main reason, however, was lack of time, since the latter problems could be worked out given sufficient time. Thus, although this section was built, it is not utilized; there is no software that pertains to it.

It is important to reiterate that this transfer was intended to load the system's software and parameters into memory. This feature would be used only with the ROM devices removed and appropriate RAM storage added to the microcomputer to replace them. As mentioned, this would be done mainly for test purposes and the ultimate goal would still have been to put the software in ROM.

---

* Note that the chip selest, -CS, is attached to one of the address lines and that this line must remain high for any I/O transfers not involving this device.

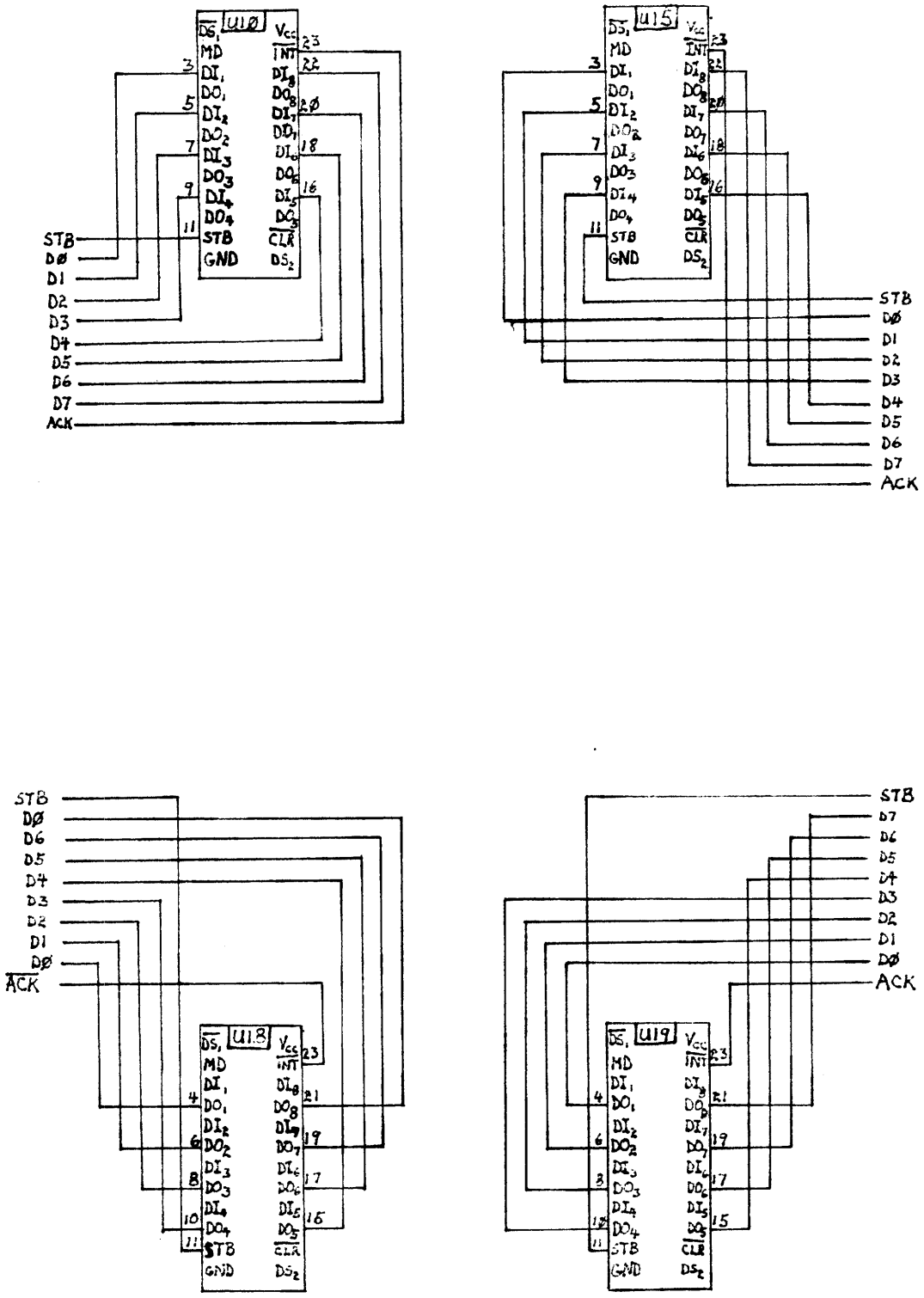


FIGURE 6-8 I/O Group--External Configuration

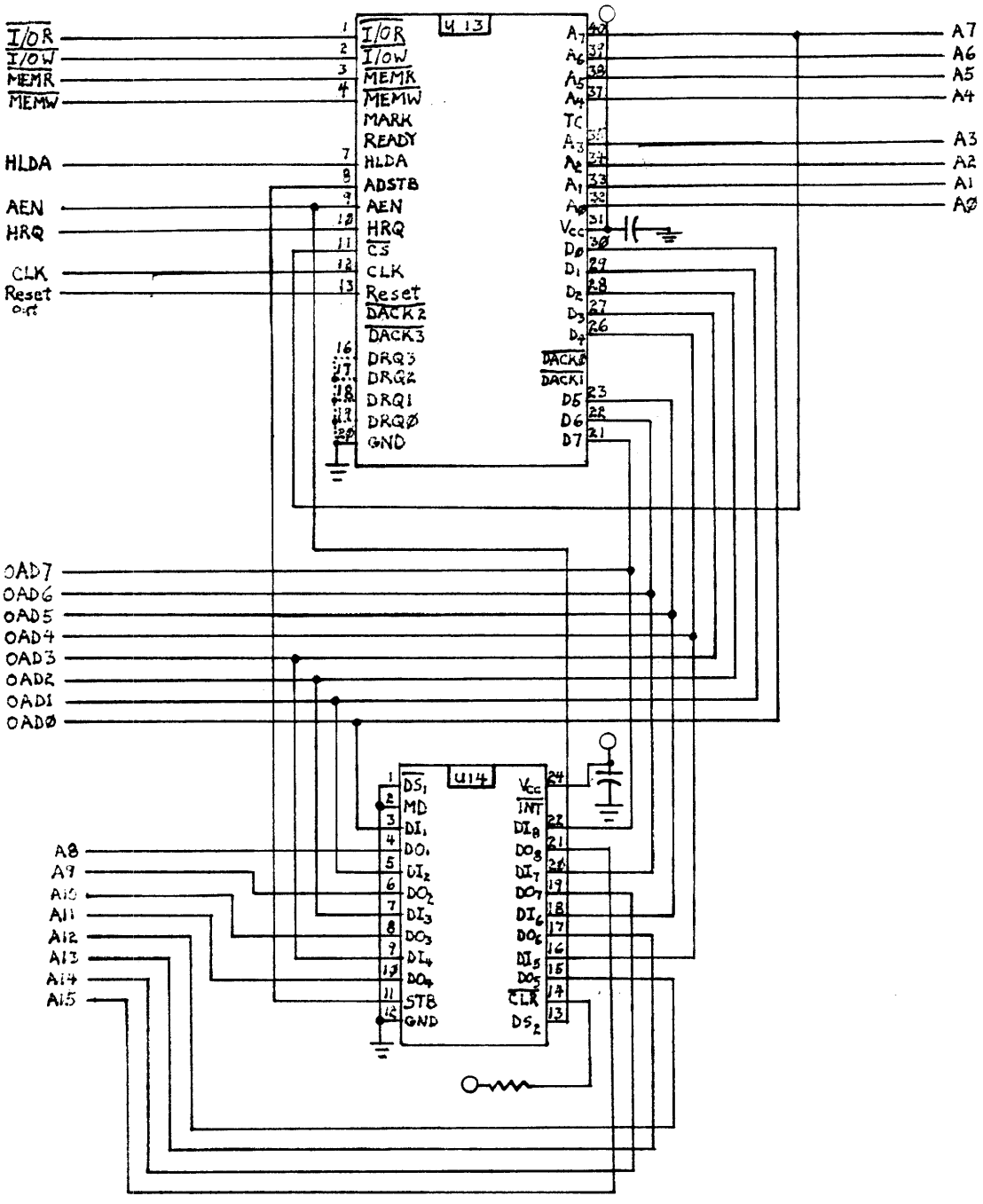


FIGURE 6-9 The DMA Group

VII. Conclusion and Suggestions for Further Work

A. Simulator Implementation

The routines presented in chapter V were all tested and the data paths verified on the 8080 simulator, using as input an internally generated test pattern. The test pattern was then modified and enlarged to 256x256 pels, which is the maximum size obtainable with the pattern generator used, and the result loaded into a file. For this image to be displayed on the IPS system, it was necessary to put it into standard image format. The test image was appended to the header from another image of the same dimensions and then loaded from the UNIX system onto a magnetic tape for transfer to and display on the IPS. Next the pattern generator was modified so that it would, instead, read data from an image* and simply insert the 'new line' and 'new page' indications where appropriate. In addition to not containing the necessary framing indications, the images stored in UNIX also have a thousand byte header of image parameters. A short routine was written which would read this header and copy it into some destination file. The rest of the image file was then the real image data and was passed on to the transmitter. The result out of the receiver was then loaded into the destination file right after the header and the resultant file copied also onto magnetic tape for transfer to and display on the IPS.

Throughout the course of this work, improvements were

---

* These images are stored as raw data in the /pic directory. See /pic/pic. doc The one mentioned here was 256 X 256 pels.

almost constantly being made to the various routines as the author became more fluent with the 8080 assembly language and as the various algorithms were developed. Admittedly, some liberties were taken, with the local contrast determination especially, in the interest of reducing the amount of computation required. Indeed the estimated computation time* almost exactly meets the initial specification of a 1 millisecond cycle time for the transmitter routine. It may be of interest to carefully examine the images resulting from this implementation in comparison to images resulting from Hoover's[3] system and to determine possibly which features of this system give rise to whatever differences are observed. Time-efficient methods of improving performance might also be investigated, although it would probably require an intimate knowledge of the inner workings of at least the transmitter.

B. Microcomputer Implementation

The microcomputer system discussed in chapter VI was built and tested to verify that it functioned properly as a computer. In other words, verify that it properly executes whatever 8080-type instructions were loaded into its ROM devices. Once functional, the memory boards were also tested, using the microcomputer to load and verify the contents of the storage locations. Time did not permit, however, the construction of a test system suitable for verifying its performance with an actual input image

* See the appendices.

from a scanning device and with its resultant image sent to a facsimile reproduction device. Developing such a system would be a major undertaking in and of itself and, along with a model for the transmission channel, could be incorporated into a further project. The microcomputer was only operated at half of its maximum speed, as mentioned earlier, in order to accomodate the slower RAM devices. It would be necessary to upgrade these devices in order to make the system even be able to approach the time specification of a 1millisecond cycle time. Appendices A and J show that, even with the upgraded devices and with the CPU at its maximum 5 MHZ lock rate, it is necessary to limit the transmitter output to the transmission channel to being direct loaded* rather than interrupt driven, and seems to be a reasonable limitation for most real world applications.

The last major drawback to fully testing the microcomputer's performance with a real image was the very real problem of loading approximately 3k bytes of data into the PROM by hand, one address at a time. There is an existing system**to which entire assembly program files can be copied and used to completely program the PROMs, but it proved impossible, within the allotted time, to gain access to it; thus the test program had to be loaded in machine code a single address at a time.

The system has been tested on several shorter programs and

---

* In the simulator version, all I/O is done by direct loading.
** The system is in the EE & CS departments microcomputer laboratory. (bldg. 38, 6th floor)

performed as specified, demonstrating, at least to the author, that it indeed functions properly as a computer and should, therefore, be able to handle the transmitter and the receiver programs as well. The main concern is that there may yet be some defective RAM devices on the memory boards since they were purchased from a surplus vendor and contain some rather ancient devices. This would be resolved with the aforementioned component upgrading. The initial intent was to merely expand an existing 8080 microcomputer system, rather than to build one from scratch. Unfortunately, the system became unavailable when the full scope of the modifications to be made were made known to the owner. The cost of purchasing a suitable board was also prohibitive, leaving no recourse but to build the system from scratch.

Appendices

The following appendices contain listings and discuss briefly the operation of the various programs that comprise the system. The average computation time is also discussed and is determined from the number of machine states, i.e. clock cycles, for each instruction and the number of times that it is executed in each cycle. Included in this computation is a scaling based on the percentage of cycles for which the instruction is executed at all. For example, the estimated computation time for the low-pass filter subroutine is scaled by .125 when determining its contribution to the system's cycle time, since it is only called for one out of eight pels. Note also that for clarity an instruction cycle is considered to be four machine states rather than as defined in the technical manuals, where it can range from four to six machine states.

Appendix A: Transmitter Controller

This section contains the 8Ø8Ø assembly language program listing for the transmitter controller discussed in chapter V. Since much effort was devoted to its explanation there, it is presented here without further explanation of the main functions. Note that this is the version used in the simulator implementation and treats the receiver and its initialization routine as subroutines. The pattern generator is also used as a subroutine and is presented in appendix B.

The routine cycle takes an average of about 455 instruction cycles, excluding those functions external to it such as gradient, local contrast filters, and interpolators. This cycle time corresponds to approximately 364 microseconds for the 8Ø85A-2 operating at its maximum rate.

When requirements for these other functions are included, except for the multiplication routine, the average number of instruction cycles increases to 854. This would require approximately 683 microseconds, i.e. under 19 instruction cycles, to complete its operations. This can only be achieved with some sort of hardware multiplier. For example, the routine, MLTPLY, listed in Appendix J takes only 18 instruction cycles, under 14 microseconds, to complete its operation and is completely compatible with the multiplication routine, MULTIPLY, used in the simulator implementation. The total time for multiplication would come to 272 microseconds, leaving a approximate 45 microseconds for the I/O routine.

```
startup: ei
      call rinit  ¶initialize receiver
         lxi h,0bed0 ¶store the filter coefficients
         mvi m,1 ¶starting with location 0bed0
         inx h  ¶and ending with location 0bed8
         mvi m,3.
         inx h
         mvi m,13.
         inx h
         mvi m,28.
         inx h
         mvi m,37.
         inx h
         mvi m,28.
         inx h
         mvi m,13.
         inx h
         mvi m,3
         inx h
         mvi m,1
         inx h   ¶set HL register to 0bed9
         mvi a,0 ¶set accumulator to zero
         mov m,a ¶set location 0bed9 to zero (temporary highs buffer--THIGHS)
         inx h
         mov m,a ¶set location 0beda(MISMATCH) to zero
         inx h
         mvi m,20 ¶set location 0bedb(FLAGA)
         inx h
         mvi m,1 ¶set location 0bedc(XCNT) to one
         inx h
         mov m,a ¶set location 0bedd(XMIT) to zero
         lxi h,0bec8 ¶set HL to point to 0bec8
         mov m,a   ¶initialize it to zero
         inx h   ¶set HL to point to 0bec9
         mov m,a   ¶initialize it to zero
         inx h ¶set HL reg to point to STIN
         mov m,a ¶set memory location to zero
         inx h   ¶set HL reg to point to STOUT
         mov m,a ¶and initialize to zero
        lxi h,0bef0 ¶VLPF output(VLOWS) storage area extends from 0BEF0 to 0BEFF
         shld 0beec ¶initialize associated pointer
         lxi h,8c02 ¶DELTA Local Contrast storage area extends from
                    ¶8C00 to 93FF (double precision values)
         shld 0becc ¶initialize associated pointer
         mov h,a ¶set reg H to zero
         mov l,a ¶set reg L to zero
         shld 0bece ¶initialize value of LOCAL CONTRAST(2-bytes)
         mvi h,28 ¶ORIGinal data storage area extends from 2800 to 4BFF
         shld 0beee ¶initialize associated pointer
         shld 0bede  ¶initialize pointer for framing indications for trans-
                   ¶mission of highs
```

FIGURE A-A. Transmitter Controller Routine

```
        mvi h,94 ¶HLPF output(LOWS) storage area extends from 9400 to A7FF
        shld 0beea ¶initialize associated pointer
       mvi h,0b8 ¶lows transmitter storage areas extend from B800 to B8FF(L1)
                 ¶land from B900 to B9FF(L2)
        shld 0bee8 ¶initialize associated input pointer to area L1
        shld 0bee4 ¶initialize associated output pointer to area L2
        mvi h,0a8 ¶highs transmitter storage areas extend from A800 to AFFF
                  ¶(H1) and from B000 to B7FF(H2)
        shld 0bee6 ¶initialize associated input pointer to area H1
        mvi h,0ac
        shld 0bee2 ¶initialize associated output pointer to area H2
        mvi h,4c ¶GRADIENT storage area extends from 4C00 to 8BFF
        shld 0bee0 ¶initialize associated pointer
        mvi b,0 ¶prepare to zero-out the memory storage areas for
        mvi a,94 ¶GRADIENT and DELTA LC
zerolc: mov m,b ¶this extends from the current value of the HL reg(4C00)
        inx h ¶to 93FF
        cmp h ¶does H=94?
        jnz zerolc ¶if not then iterate
        lxi sp,0C000 ¶initialize stack pointer
        mvi b,45 ¶this value to be used by the pattern generator
        mvi c,1  ¶initialize associated counter
        push b   ¶store both on the stack
        mvi a,0c
        push psw ¶this word(STATUS) is used in subsampling the lows
        mvi a,0ff ¶set first input value to 'page'
        jmp ftime
resume: call gptrn
ftime:  lhld 0beee ¶prepare to store the data in storage area ORIG
        mov m,a
        cma
        cpi 01  ¶check for framing indications i.e. new line and new page
        cma
        jc pnew ¶if new page, jmp
        jnz lntnw ¶if no framing indication, jmp
        call incgr ¶else increment GRADIENT pointer to the next line
        dcx h ¶set pointer to the last possible address of the previous line
        shld 0bee0 ¶and store
        call incdat ¶increment ORIGial data pointer to the next line
        dcx h ¶set pointer to the last possible address of the previous line
        mvi m,0fe ¶put 'new line' indication at pointer address
        shld 0beee ¶store pointer
        mov a,m
        jmp frnwln
pnew:   mvi m,0
        call incgr ¶increment GRADIENT pointer to the next line
        shld 0bee0 ¶and store
        call incdat ¶increment ORIGinal data pointer to the next line
        mvi m,0ff ¶put 'new page' indication at pointer address
        shld 0beee ¶store pointer
```

FIGURE A-A (continued)

```
lntnw:  mov a,m
        call gradient ¶this subroutine both computes and stores the magnitude
                      ¶of the gradient for the pel area centered above the
                      ¶current pel.
        lhld 0beee   ¶restore the ORIG pointer
frnwln: mov a,h ¶put high order byte of ORIG pointer into accumulator
        sui 38 ¶move back four lines (offset of 28 since ORIG begins at 2800)
        jp good ¶if within storage area bounds, jmp
        adi 24 ¶else wrap around by adding the size of the storage area
good:   adi 28 ¶add back the offset
        mov h,a ¶put value into reg H
        mov d,h
        mov e,l ¶save pointer in reg DE
        dcx h
        dcx h
        dcx h
        dcx h
        dcx h
        mov a,m ¶check for framing indication at location (-4,-4)
        cma
        cpi 01
        jc lnwpg
        jnz nlnwln ¶if no framing indications then jmp
        pop psw    ¶else new line. get STATUS from stack
        xri 08     ¶toggle bit 3
        push psw   ¶and store new value on the stack
        jmp nlnwln
lnwpg:  pop psw    ¶get STATUS from the stack
        ani 0f7     ¶set bit 3

        push psw   ¶store new value on the stack
nlnwln: xchg ¶restore ORIG pointer for location (0,-4) to reg HL
        mov a,m ¶get corresponding data value
        ¶The next few lines involve getting the pointers needed for the
        ¶vertical low-pass filtering operation.
        ¶reg BC gets the pointer for the filter coefficients
        ¶reg DE gets the pointer for the data to be filtered
        ¶reg HL gets the pointer for the storage address of the result
        lhld 0beee ¶get ORIG pointer
        xchg ¶and put it into reg DE
        lhld 0beec ¶put VLOWS pointer in reg HL
        lxi b,0bed8 ¶put address of end filter coefficient into reg BC
        cma
        cpi 01 ¶check for framing indications in the corresponding ORIG value
        cma ¶i.e. ORIG(0,-4), where ORIG(0,0) is the most recently received pel
        jc pnwpg ¶if new page, jmp
        jz pnwln ¶if new line, jmp
        pop psw ¶get STATUS word from stack
        dcx sp
        dcx sp  ¶reset stack pointer
        ani 0c ¶isolate bits 2 and 3 (toggle at each line change to
        cpi 0c ¶control vertical subsampling)
```

FIGURE A-A. (continued)

```
        jz offln ¶if bits 2 and 3 are nonzero(line # is odd), jmp
        call vlpf ¶else filter
        pop psw  ¶get full status word from stack again
        mov d,a  ¶and save it in reg D
        dcr a
        ani 3 ¶isolate bits 0 and 1 (these control horizontal subsampling)
        jnz nsmpl ¶if bit 2 or bit 3 is nonzero, then jmp
        mov a,d     ¶else get the original STATUS word
        dcr a
        cmp a
        jmp hflt  ¶prepare for horizontal filtering
nsmpl:  mov e,a
        mov a,d
        ani 0fc     ¶mask off the two lsb's of the STATUS word
        ora e       ¶and replace them with the new values
        jmp fltskip ¶and skip filtering and interpolating operations
        ¶This section is used if and only if the current line or pel
        ¶is not to be sampled (for lows only)
offln:  pop psw ¶get STATUS word from stack
        xri 1 ¶toggle LSB (controls output transmitter)
fltskip: push psw ¶put STATUS word back on stack
        lhld 0beea ¶put LOWS pointer in reg HL
        mov a,h
        sui 10 ¶move LOWS pointer back four lines
        cpi 94 ¶check to insure pointer still within storage area
        jnc olskip ¶if so ,jmp
        adi 14  ¶if not, then wrap around before proceeding
olskip: mov h,a ¶before proceeding
        dcx h
        dcx h
        dcx h  ¶move LOWS pointer back(left) three pels
        mov b,m ¶get LOWS value used in separating next highs value-LOWS(-7,-8)
        jmp nwln
pnwpg:  mov m,a ¶put 'new page' indication at VLOWS pointer address
        pop psw ¶get STATUS word from the stack (discard)
        sub a ¶set accumulator to zero
        sta 0beda ¶reset MISMATCH value
        mvi a,80 ¶set accumulator to 83(STATUS bit 7=1, indicates the first
           cpi 0 ¶line of an image)
        jmp hflt
pnwln:  mov m,a ¶put 'new line' indication at VLOWS pointer address
        pop psw ¶get STATUS word
        cpi 80 ¶check for first line indication

        jc noframe ¶if none,jmp
        ¶MISMATCH is the value of the modulo four count of the line length.
        ¶Since only the subsampled version of the LOWS data is to be trans-
        ¶mitted, this value is needed to properly allign the image data for
        ¶reconstruction.
        dcx sp
        dcx sp
```

FIGURE A-A. (continued)

```
          dcr a
          ani 03
          sta 0beda ¶store MISMATCH
          pop psw  ¶restore status word to the accumulator
noframe:ani 0c ¶reset firstline and hor. subsampling bits(bit 7 and bits 1and 0)
          xri 4 ¶toggle vertical subsampling bit(bit 2)
          cpi 4 ¶check to see if all other bits of the STATUS word were zero
hflt:     push psw ¶put STATUS word back on stack
          jnz nwln ¶if a sampled line , jmp
          ¶The appropriate pointers for the
          ¶horizontal low-pass filtering operation are:
          ¶reg BC still contains the address for the filter coefficients
          ¶reg DE gets the pointer for the data to be filtered i.e.VLOWS
          ¶reg HL get the pointer for the storage address of the result
          call hlpf
          call vint
          call hint
          dcr h
          dcr h
          dcr h
          dcr h ¶move LOWS pointer back one line
          mov a,h
          cpi 94 ¶check to insure LOWS pointer still within storage area
          jnc liskip ¶if so , then jmp
          adi 14 ¶if not, then wrap around
          mov h,a
liskip:   call hint
          mov a,h
          sui 0c ¶move LOWS pointer back three more lines
          mov h,a
          cpi 94 ¶check to insure pointer still within storage area
          jnc lskip ¶if so, then jmp
          adi 14 ¶if not, then wrap around
          mov h,a
lskip:    push h
          call ldlbuf
          pop h
          dcx h
          dcx h
          dcx h ¶move LOWS pointer back(left) three pels
          mov b,m ¶get LOWS value used in next highs separation--LOWS(-7,-8)
nwln:     lhld 0beee ¶get pointer for ORIG data
          mov a,l
          sui 7 ¶move back(left) seven pels
          mov l,a
          jnc nofix ¶check for borrow from subtraction. jmp if none
          dcr h ¶else take borrow from pointer's high order byte
nofix:    mov a,h
          sui 20 ¶move pointer back(up) eight lines
          mov h,a
```

FIGURE A-A. (continued)

```
        cpi 28 ¶check to see if pointer is still within storage area
        jnc skip ¶if so , jmp
        adi 24 ¶else wrap-around pointer
        mov h,a
skip:   mov a,m ¶get corresponding ORIGinal data value i.e. ORIG(-7,-8)
        push psw ¶store temporarily on stack
        push h ¶store ORIG pointer on stack also
        mov l,a ¶save ORIG value in reg L


        sub b ¶obtain highs by subtracting LOWS value from ORIG value
        push psw ¶save the highs value on the stack
        jp nhance  ¶compute the magnitude of the highs value
        cma        ¶since the multiplier is only for unsigned numbers
        inr a      ¶i.e. a result that should have been negative will
                   ¶no longer be in two's complement form
nhance: lxi b,0bed9 ¶get pointer for Temporary HIGHS(THIGHS) buffer
        stax b ¶store highs value in THIGHS
        mvi h,0a ¶set high order byte of HL reg to address of lookup table for
                 ¶luminance scale factor
        xchg ¶use ORIG value in reg L as the index and move pointer to reg DE
        call multiply ¶computes product of values pointed to by the BC and DE
        stax b ¶store value in THIGHS
        call loccon ¶computes local contrast(double-precision)--loccon(-7,-8)
        mov a,h ¶put high order byte in reg A (ignore low order byte)
        cpi 40  ¶see if index exceeds limit
        jc snext ¶if not , jmp
        mvi a,3f ¶else set index to its upper limit
snext:  mov l,a  ¶put index into reg L
        mvi h,0b ¶put address of lookup table for detail scale factor in reg H
        xchg ¶put address into reg DE using the loccon value as the index
        call multiply ¶scale the value in THIGHS
        mov d,a ¶save value in reg D
        pop psw ¶get highs value from the stack
        jp postv   ¶and combine with the result of the scaling operations
        sub d
        jmp comprss
postv:  add d
comprss: mvi h,08
        mov l,a ¶get pointer for highs compression lookup table
        mov a,m ¶get compressed highs value
        mov d,a ¶and save in reg D
        pop h ¶retrieve the ORIG pointer
        call noisep ¶this generates a PRN value
        add d ¶add the PRN and THIGHS values
        mov d,a ¶and save in reg D
ldhbuf: lhld 0bee6 ¶get input pointer for the HIGHS output buffer
        mov m,d ¶and store the scaled highs value there
        pop psw ¶get ORIG value from the stack
        cma
        cpi 01 ¶check for framing indications
        cma
```

FIGURE A-A. (continued)

```
          jc npage ¶if 'new page' indication, then jmp
          push psw
          lda 0bec9 ¶get HGHEN from memory
          cpi 0ff ¶and see if the enable is set
          inx sp
          inx sp ¶restore the stack pointer
          jnz xmitout
          dcx sp
          dcx sp ¶set stack pointer to retreive value
          pop psw
          jnz sline ¶if no framing indications, jmp
          call inchbf ¶else 'new line'.increment the highs output buffer's input
                      ¶pointer to the next line
          dcx h ¶mov pointer back to the last possible value of the previous line
          lda 0bec8  ¶get NCNT and increment it (output line counter)
          inr a
          sta 0bec8
          jmp sline
npage:    call inchbf ¶increment the H.O.B.'s input pointer to the next line
          mov m,d ¶insert the new page indication at the beginning of the line
          mvi a,0ff
          sta 0bec9 ¶set HGHEN

          mvi a,0
          sta 0bec8 ¶reset the line counter, NCNT
sline:    inx h ¶increment the HOB input pointer
          shld 0bee6 ¶and store
xmitout:  pop psw ¶get STATUS word from stack
          pusn psw ¶but put it back unchanged
          rrc ¶put LSB into carry (this bit toggles each time the program cycles)
          cnc txout ¶if zero, present one byte to the transmit channel
mfinish:  lhld 0bee0
          inx h ¶increment GRADIENT pointer
          shld 0bee0
          lhld 0beee
          inx h ¶increment ORIGinal data pointer
          shld 0beee
          lhld 0beec
          inx h ¶increment VLOWS pointer
          mov a,l ¶check to insure that pointer is within storage area
          ani 0f
          jnz vwrap ¶if so, jmp
          lxi h,0bef0
vwrap:    shld 0beec
          lhld 0beea
          inx h ¶increment LOWS pointer
          mov a,h
          cpi 0a8
          jc alrght
          sui 14
          mov h,a
```

FIGURE A-A. (continued)

```
alrght:  shld 0beea
         jmp resume ¶cycle
     ¶This is the end of the main program. The rest are the subroutines
¶referred to in the body of the main. Those referred to in main, but not
¶listed below include the vertical low-pass filter(vlpf), the horizontal
¶low-pass filter(hlpf), the gradient routine(gradient), the local contrast
¶routine(loccon), the vertical interpolator(vint), the horizontal inter-
¶polator(hint), and the multiplication routine(multiply). They are stored
¶individually in other files (see /usr/gallin and type dir).
incgr:   lxi h,0bece
         sub a
         mov m,a
         inr l
         mov m,a  ¶reset the local contrast value to zero
         lxi h,8c00 ¶reset delta local contrast pointer for the next line
         shld 0becc
         lhld 0bee0 ¶get GRADIENT pointer
         mov a,h ¶put high order byte of pointer into accumulator
         adi 4 ¶increment pointer to the next line
         ani 0fc ¶mask 2 LSB's(note: the 10 LSB's of pointer must be set to 0)
         mov h,a
         cpi 8c ¶check to see if upper of the storage exceeded
         mvi l,0 ¶set low order byte of pointer to zero
         rnz ¶return if pointer within storage area
         mvi h,4c ¶else wrap pointer to lower bound of the storage
         ret
¶This is the end of this rountine. It is very similar to a couple of the
¶routines to follow, but it was undesirable to combine them into a more
¶general routine because of the constraints on computation time.
update:  lxi h,0bedb ¶load flaga, which indicates the type of word to
         mov a,m ¶be transmitted next
         ani 80 ¶bit 7 indicates that the two-bit line mismatch is
         jnz xmtmsmat ¶to be transmitted
         mov a,m ¶if not, restore flaga to the accumulator
         ani 20 ¶bit 5 indicates that an 8-bit lows word is
         jnz updlow ¶to be transmitted
         lhld 0bede  ¶else 3-bit highs.--get pointer for framing indications
         mov a,m   ¶get value from ORIG storage area and check it for framing
         cma       ¶indications
         inx h     ¶increment pointer
         shld 0bede  ¶and store it
         cpi 01
         jc nwhpg  ¶if new page,then jmp
         jnz hout  ¶if no framing indications,then jmp
         mvi l,0   ¶else 'new line' indication.
         mov a,h   ¶set pointer to the first possible value of the
         ani 0fc   ¶line being observed for framing
         mov h,a   ¶indications  i.e. wrap pointer
         shld 0bede  ¶and store
         lhld 0bee2 ¶get pointer for next highs value to be transmitted
```

FIGURE A-A. (continued)

```
        mov b,m   ¶get value and save in reg B
        mvi l,0   ¶set low order byte of pointer to zero
        mov a,h   ¶move pointer to the last possible location of the current
        ani 0fc   ¶line , so that the next increment will move it to the first
        adi 04    ¶location on the next line.
        cpi 0b8   ¶check to insure that pointer is still within the bounds
        jc txhwrp ¶of the HOB storage area
        mvi a,0a8 ¶else wrap pointer
txhwrp: mov h,a
        dcx h
        lda 0bedb  ¶get and
        rlc        ¶update FLAGA
        sta 0bedb  ¶and store
        mov a,b
        jmp hgout
nwhpg:  mvi l,1
        mov a,h
        ani 0fc
        mov h,a
        shld 0bede ¶reset pointer to beginning of line and store it
        lhld 0bee2 ¶get HOB pointer
        mov b,m    ¶and the corresponding highs value
        mvi l,0
        mov a,h
        ani 0fc
        mov h,a    ¶move pointer to the beginning of the current line
        mov a,b
        jmp hgout
hout:   lhld 0bee2 ¶get next highs value to be transmitted
        mov a,m ¶from the highs output buffer
hgout:  inx h ¶update the pointer
        shld 0bee2 ¶and store it.
        lxi h,0bedd ¶store this value in the transmitter output
        mov m,a ¶buffer location(XMIT)
        lxi h,0bedc ¶set transmit output bit counter for
        mvi m,20 ¶three bit words(XCNT)
        ret
updlow: lda 0becb  ¶get the output enable indicator(STOUT)
        cpi 0ff    ¶and check it
        rnz        ¶if not enabled, then return
        lhld 0bee4 ¶get value from the lows output buffer and
        mov a,m ¶put it in the accumulator
        cma
        cpi 01 ¶compare it with the 'new line' indication
        cma
        jc unwp ¶if 'new page' indication , then jmp
        jnz pgbr ¶if no framing indication , then jmp
msmatch: mov b,a ¶save value in reg b while switching output buffer segment
        mov a,h ¶the switch between the two output buffer segments is achieved
        xri 1 ¶by complementing the LSB of the higher order byte of the
        mov h,a ¶pointer each time a new line indication is encountered
mtch:   mvi l,0 ¶set lower order byte to zero.(corresponds to rastor retrace)
```

FIGURE A-A. (continued)

```
          shld 0bee4 ¶store the updated pointer
          lxi h,0bedd ¶store the value from reg b in the transmitter
          mov m,b ¶output buffer location(XMIT)
          lxi h,0bedc ¶set the transmitter output bit counter for 8-bit
          mvi m,1 ¶words and store(XCNT)
          lxi h,0bedb ¶get FLAGA and check to see if this the end of the
          mov a,m ¶first line of the image to be transmitted
          xri 28 ¶set FLAGA to indicate highs
          mov m,a ¶and store
          ani 40 ¶this is the actual check for the first line(as mentioned
          rz   ¶above) with bit 6 as the indicator
          mvi m,10 ¶if so set FLAGA to 01(bit 0=1) ,store
          ret ¶and return
unwp:     xchg
          lda 0bedb ¶set FLAGA for lows first line(bit 5)
                    ¶and to transmit the MISMATCH value(bit 7)
          ori 0c0
          sta 0bedb
          xchg
          mvi a,0ff ¶restore data value to the accumulator
pgbr:     inr l ¶update the lows output buffer address pointer
          shld 0bee4 ¶and store
          sta 0bedd ¶store value in transmit output buffer location(XMIT)
          mvi a,1
          sta 0bedc ¶set transmit output bit counter for 8-bit words
          ret
xmtmsmat:lxi h,0beda ¶get value of line mismatch
          mov a,m ¶and put it in the accumulator
          rlc
          rlc
          ora m
          rlc
          rlc
          ora m
          rlc
          rlc
          ora m
          lxi h,0bedd ¶store this value in the transmitter output
          mov m,a ¶buffer location(XMIT)
          lxi h,0bedc ¶set the transmit output bit counter(XCNT) for
          mvi m,01 ¶8-bit words
          lxi h,0bedb ¶get FLAGA and put it in the accumulator
          mov a,m
          ani 7f ¶set bit 7 to zero(no more mismatch)
          mov m,a ¶and store
          ret ¶the end
¶This is the end of the routine which provides the actual transmitter
¶routine with both data values in the desired format and information on how
¶many bit of each of these values is to be transmitted.
```

FIGURE A-A. (continued)

```
txout:  mvi d,7
txt:    lda 0bedd ¶get value from transmit output buffer location(XMIT)
        rlc       ¶move LSB into the carry bit
        sta 0bedd ¶store rotated version of value in XMIT
        mov a,c
        ral       ¶shift bit into reg C from the left
        mov c,a
        lda 0bedc ¶get transmit output bit counter(XCNT),
        rlc       ¶update,
        sta 0bedc ¶and store it.
        cc update ¶if valid bits of XMIT exhausted,get next values for X
                  ¶XMIT and XCNT
        dcr d     ¶decrement counter for the number of iterations remaining
        mov a,c   ¶get value to be transmitted
        jp txt    ¶if count greater thanor equal to zero then iterate
        out 06 ¶TRANSMIT CHANNEL or direct buffer

    call rcvr  ¶send output to the receiver for test
        ret       ¶end routine
    ¶This is the end of the routine. It has not yet been decided whether
¶to send its output directly to the transmit channel(which would be sufficient
¶for test purposes) or to send it to some FIFO buffer such as on the input
¶to allow for more realistically flexible timing.
noisep: mov a,l ¶this routine fecthes values from the PRN lookup table
        ani 07 ¶using the three lsb's of the pointer given it,
        mov l,a ¶and the three lsb's of the line counter ,NCNT
        lda 0bec8
        ani 07
        rlc
        rlc
        rlc
        ora l
        ori 40
        mov l,a
        mvi h,0b
        mov a,m
        ret
inchbf: mov a,h ¶high order byte of HOB pointer into accumulator
                ¶(pointer was already in HL reg)
        adi 4 ¶increment pointer to next line
        ani 0fc ¶mask 2 LSB's (Note: the 10 LSB's of pointer must be zeroed)
        mov h,a ¶put new value into reg H
        cpi 0b8 ¶check to see if upper bound of storage area exceeded
        mvi l,0 ¶set low order byte of pointer to zero
        rc ¶return if pointer within storage area
        mvi h,0a8 ¶else wrap pointer to lower bound of storage area
        ret ¶end of routine
    ¶This is the end of the routine. It can be used to increment either
¶of HOB's pointers since no pointer is specified and the storage area
¶specified is appropriate for both.
```

FIGURE A-A. (continued)

```
incdat:  lhld 0beee ¶get pointer for ORIGinal data
         mov a,h ¶high order byte of pointer into accumulator
         adi 4 ¶increment pointer to the next line
         ani 0fc ¶mask 2 LSB's (Note: the 10 LSB's of pointer must be zeroed)
         mov h,a ¶put new value in reg H
         cpi 4c ¶check to see if upper bound of storage area exceeded
         mvi l,0 ¶set low order byte of pointer to zero
         rc      ¶return if pointer within storage area
         dcx h
         mvi m,0fe ¶set the last location of line before wrapping pointer
         inx h     ¶this resets reg L to zero
         mvi h,28 ¶else wrap pointer to lower bound of storage area
         ret ¶end of routine
      ¶This is the end of the routine. There is'nt really anything more
¶to be said, since it is almost identical to inchbf and even moreso
¶to incgr.
ldlbuf:  mov a,m ¶get value to be loaded--LOWS(-8,-4)
         mov c,a
         cpi 0ff ¶check for 'new page' indication
         lhld 0bee8 ¶get Lows Output Buffer(LOB) pointer
         jz ldpg ¶if 'new page' , then jmp
         lda 0beca
         cpi 0ff
         rnz
         mov a,c
         cpi 0fe
         jnz ldwd ¶if no framing indications, then jmp
         pop psw
         pop psw
         pop psw
         push psw
         dcx sp

         dcx sp
         dcx sp
         dcx sp
         ani 04
         jnz ldlows
         mvi a,0ff
         sta 0becb  ¶set transmitter output enabled, STOUT
         ret
ldlows:  dcr l ¶else 'new line'. move pointer back(left) one pel
         mov m,c ¶store value at LOB pointer address
         mov a,h
         xri 1 ¶switch LOB areas
         mov h,a
         mvi l,0 ¶set low order byte of pointer to zero
         shld 0bee8 ¶store pointer
         ret
```

FIGURE A-A. (continued)

```
ldpg:   mvi l,0 ¶set low order byte of pointer to zero
        sta 0beca ¶set enable(STIN) for output buffer's input routine
ldwd:   mov m,c ¶store value at pointer address
        lda 0beca ¶check enable (STIN)
        cpi 0ff
        rnz     ¶if not enabled then return
        inx h ¶increment LOB input pointer
        shld 0bee8 ¶store pointer
        ret ¶end of routine
    ¶This is the end of the routine. It has one feature which may need
¶to be explained. It will always write whatever 'new line' indications
¶it gets over the last value it stored.
    ¶The reason for this is that the subsampled lows lines cannot be more
¶than one fourth of the total line length before subsampling. For a line
¶with a nonzero MISMATCH (i.e. line length not a multiple of four), this
¶feature prevents the transmitted line length from exceeding this limit.
¶Since the 'new line' indication must be transmitted, it will cause this
¶limit to be exceeded unless it happens to occur at a pel that was to be
¶sampled (and transmitted). This corresponds to the line length being a
¶multiple of four i.e. MISMATCH=0
```

FIGURE A-A. (continued)

Appendix B: Pattern Generator

There were three different versions of the pattern generator used in testing the system's simulator implementation. The first generates a checker test pattern with a line length of sixty-four pel and was used mainly to test the data paths and various subjunctions. The second generates test pattern of vertical bars with the appropriate framing indications and with a line length of 256 pels. This pattern was to be put through the system and displayed in order to check the receiver's reallignment procedure. The third version generates framing indications for images* input to the system and otherwise passes the image data directly through.

*The line length cannot exceed 256 pels.

```
gptrn:  pop b
        pop b
        inr c
        jz i4    ¶checker height = 4
        mvi a,3f ¶line length = 64
        ana c
        jz i3
        mvi a,0f ¶checker width = 16
        ana c
        mov a,b
i1:     jnz i2
        cma
i2:     mov b,a
        jmp endit
i3:     mov a,b
        cma
        mov b,a
i4:     mvi a,0fe
endit:  push b
        dcx sp
        dcx sp
        ret
```

Version 1

```
¶This is the pattern generator used in testing the transmitter and receiver.
gptrn:  pop b
        pop b
        pop b
        inr c
        jz i3    ¶line length = 256
        mvi a,0f ¶stripe width = 16
        ana c
        mov a,b
i1:     jnz i2
        cma
i2:     mov b,a
        jmp endit
i3:     mov a,b
        cma
        mov b,a
        mvi a,0fe
endit:  push b
        dcx sp
        dcx sp
        dcx sp
        dcx sp
        ret
```

Version 2

FIGURE A-B. Pattern Generator Subroutine

```
gptrn:  pop b
        pop b
        inr c
        in 01 ¶get input value
        jz i4    ¶if end of line reached, then jmp
        jmp endit ¶else return with input value
i4:     mvi a,0fe ¶if end of line, generate 'new line' indication
endit:  push b
        dcx sp
        dcx sp
        ret
```

Version 3

FIGURE A-B. (continued)

Appendix C: The Gradient Subroutine

The gradient subroutine is used to determine the magnitude of the gradient at the pel location just above the most recently receive- pel. This would correspond to location (0,-1) relative to that pel. The routine follows the procedure outlined in chapter IV and begins by getting the original data pointer, ORIG*. It gets the correspond pel value, location (0,0), then uses this pointer as a reference in accessing the values for the pels at location (0,-2), 0,-1), (1,-1) and (-1,-1), in that order. It then computes the magnitude of the difference between the value for location (0,-1) and the other four locations and sums them. The results are divided by two before summing and the sum is stored at the location indicated by the gradient pointer.

The routine takes about 71 instruction cycles on the average, which corresponds to approximately 57 microseconds for an 8085A-2 at its maximum rate.

*This containes the storage address for the most recently stored pel See the section on the transmitter controller.

```
gradient:lhld 0beee ;The gradient routine both computes the magnitude of the
        mov b,m ;gradient for the area centered at the pel just above the
        mov a,h ;most recently received pel and stores it for later use by the
        sui 8   ;local contrast routine
        cpi 28
        jnc nowrap
        adi 24
nowrap: mov h,a
        mov c,m
        adi 4
        cpi 4c
        jc centered
        sui 24
centered:mov h,a
        mov a,m
        inx h
        mov e,m
        dcx h
        dcx h
        mov d,m
        mov h,a
        sub e
        jnc egrd
        cma
        inr a
egrd:   rar
        mov e,a
        mov a,h
        sub d

        jnc dgrd
        cma
        inr a
dgrd:   rar
        add e
        mov e,a
        mov a,h
        sub c
        jnc cgrd
        cma
        inr a
cgrd:   rar
        add e
        mov e,a
        mov a,h
        sub b
        jnc bgrd
        cma
        inr a
bgrd:   rar
        add e
storeg: lhld 0beeU
        mov m,a
        ret
```

FIGURE A-C. Gradient Subroutine

Appendix D: The Local Contrast Subroutine

The Local contrast routine follows the procedure outlined in chapter IV and its results corresponds to location (-1,-8) relative to the current pel. It first saves the contents of the BC register pair on the stack, since at the time this routine is called they contain the temporary storage address for the highs value being scaled. It then gets the GRADIENT pointer and manipulates it to access the gradient values corresponding to locations (0,-1), (0,-16), (-14,-16), and (-14,-1), in that order. The latter two values are only accessed if the back reference from the pointer doesn't cross a line boundary. These four values are combined in the manner discussed in chapter IV; the first and third values are added and the second and fourth are subtracted. The result in then added to the corresponding local contrast update value, which is obtained using the appropriate pointer and is a double precision value. The new update is stored and then added to the previous local contrast value. This result is then stored in memory and the HL register pair.

On the average about 97 instruction cycles are required to complete this operation, which would take approximately 78 microseconds for an 8Ø85A-2 microprocessor at its maximum rate.

```
Toccon:  push b ¶This is the local contrast routine. It computes the average of
         lhld 0bee0 ¶the gradient over a 15X15 pel area centered at (-7,-8),
         mov c,m ¶relative to the current pel
         mov a,h
         sui 3c
         cpi 4c
         jnc lowrap
         adi 40
lowrap:  mov h,a
         mov e,m
         mov a,c
         sub e
         jnc edged
         dcr b
edged:   mov a,l
         cpi 0f
         jnc noedge
         mov a,h
         rar
         jc noedge
         rar
         jnc contfin
noedge:  mov a,l ¶this section is skipped if the averaging area overlaps the
         sui 0f  ¶edge of the image
         mov l,a
         mov a,c
         mov e,m
         add e
         jnc down
         inr b
down:    mov c,a
         mov a,h
         adi 3c
         cpi 8c
         jc local
         sui 40
local:   mov h,a
         mov e,m
         mov a,c
         sub e
         jnc contfin

         dcr b
contfin: mov c,a
         lhld 0becc
         mov a,m
         inx h
         mov e,m
         add c
         mov c,a
         mov a,e
```

FIGURE A-D. Local Contrast Subroutine

```
adc b
mov b,a
mov m,a
dcx h
mov m,c
inx h
inx h
shld 0becc
lhld 0bece
dad b
shld 0bece
pop b
ret
```

FIGURE A-D. (continued)

Appendix E: The Horizontal Low-Pass Filter Subroutine

The Horizontal filter routine computes the weighted sum of the nine most recent vertical results and its output corresponds to location (-4,-4) relative to the most recently received pel. It expects the low end filter coefficient address and the vertical filter result pointer to be in the register pairs BC and HL, respectively, and begins by checking the fifth most recent vertical filter results for framing indications. It then moves to VLOWS pointer to register pair DE and loads the lows pointer into register pair HL. For a "new page" indication, it moves the lows pointer to the first location of the next line and simply stores the indication there. For a "new line" indication it immediately puts the indication at the new location indicated by the lows pointer. It also puts this "new line" indication at the corresponding location on the previous line * and then at the last possible location on both lines. It leaves the lows pointer pointing to the location just before the first one of the next line, thus the next location loaded will be the first one of that line.** It also modifies the controller's STATUS flag word such the first location will be a sample point. The marking at the end of the line are necessary to signal when a back reference from the lows pointer has crossed a line boundary.

---

*Since the lows pointer is being moved, the interpolators would not be able to fill in these locations.

**The transmitter controller increments this pointer at the end of each cycle.

Appendix E (cont.)

In the absence of any framing indications this sets the counter, stored on the stack, for nine iterations. It then successively gets the next filter coefficient and vertical filter result, multiplies them together, and accumulates the products in the location indicated by the lows pointer.

It takes an average of about 260 instruction cycles for this routine, corresponding to about 208 microseconds for the 8Ø85A-2 operating at its maximum rate.

```
hlpf:  mov e,l ¶This is the horizontal low-pass filter.
       mov a,l ¶Its results correspond to location (-4,-4), relative
       sui 4  ¶to the most recently received pel
       cpi 0f0
       jnc okayed
       adi 10
okayed:mov l,a
       mov a,m
       cma
       cpi 01
       cma
       mov l,e
       xchg
       lhld 0beea
       jc nwfrm
       jz frame
       mvi c,0d0
hstart:mvi a,9
       push psw
       mvi m,0
cont:  call multiply
       add m
       mov m,a
       pop psw
       dcr a
       rz
       push psw
       inr c
       dcr e
       mvi a,0ef
       cmp e
       jc cont
       mov a,e
       adi 10
       mov e,a
       jmp cont
frame: mov m,a
       mov a,h
       sui 04
       cpi 94
       jnc endint
       adi 14
endint:mov h,a
       mvi m,0fe
       mvi l,0ff
       ori 3
       mov h,a
       mvi m,0fe
       adi 4
       cpi 0a8
       jc nhwrp
       sui 14
```

FIGURE A-E. Horizontal Low-Pass Filter Subroutine

```
nhwrp: mov h,a
       mvi m,0fe
       shld 0beea
       pop psw
       pop psw  ¶get STATUS word from stack
       inr a    ¶and set it such that the first pel of the next line
       push psw ¶will also be a sample point.
       dcx sp
       dcx sp
       ret
nwfrm: mov a,h
       adi 4
       cpi 0a8
       jc nhwp
       sui 14
nhwp:  ani 0fc
       mov h,a
       mvi l,0
       mvi m,0ff
       shld 0beea
       ret
```

FIGURE A-E. (continued)

Appendix F: The Vertical Low-Pass Filter Subroutine

The vertical filter routine computes the weighted sum of the most recently received pel and the corresponding pels from the eight preceding lines. It expects the high and filter coefficient address, the ORIG data pointer, and the VLOWS pointer to be present in the register pairs BC, DE, and HL, respectively.* It begins by setting a counter stored on the stack, for nine iterations. It then successively gets the next filter coefficient and appropriate pel value, multiplies them together and accumulates the products in the location indicated by the VLOWS pointer.

Excluding the time required for the multiplication routine, this routine takes an average of about 240 instruction cycles, corresponding to about 192 microseconds for the 8Ø85A-2 at its maximum rate.

---

*Framing indications are handled by the transmitter controller, which will store these at the address indicated by the VLOWS pointer without even calling this routine.

```
vlpf:   mvi a,9 ;This is the vertical low-pass filter routine.
        push psw ;Its results correspond to location (0,-4), relative
        mvi m,0 ;to the current pel
        mov a,d
        cpi 48
        jnc roll
        adi 0
roll:   mov d,a
roll2:  call multiply
        add m
        mov m,a
        pop psw
        dcr a
        rz
        push psw
        dcx b
        dcr d
        dcr d
        dcr d
        dcr d
        mov a,d
        cpi 28
        jnc roll2
        adi 24
        mov d,a
        jmp roll2
```

FIGURE A-F. Vertical Low-Pass Filter Subroutine

Appendix G: The Vertical Interpolator Routine

The vertical Interpolator computes the average of the most recently determined lows value and the correspond value from the second line above it, corresponding to locations (-4,-4) and (-4,-6) relative to the most recently received pel, and puts the result in location (-4,-5). It expects the lows pointer to be in register pair HL and, before the average is computed, the lows value is checked for a "new line" indication. If it is present, it is copied directly into location (-4,-5) instead of the average.

It takes about 34 instructions cycles on the average, which corresponds to 27 microseconds for the 8Ø85A-2 at its maximum rate.

```
;This is the vertical interpolator routine.
vint:   mov c,m
        mov a,h
        sui 8
        cpi 94
        jnc noroll
        adi 14
noroll: mov b,n
        mov h,a
        mov a,c
        cpi 0fe
        jz lnedge
        mov a,m
        sub c
        rar
        add c
        mov c,a
lnedge: mov a,b
        sui 4
        cpi 94
        jnc rlln
        adi 14
rlln:   mov h,a
        mov m,c
        mov h,b
        ret
```

FIGURE A-G. Vertical Interpolator Subroutine

Appendix H: The Horizontal Interpolator Subroutine

The horizontal interpolator takes the difference of the most recently determined lows value, corresponding the location (-4,-4) and (-8,-4), and linearly interpolates the values for locations (-7,-4), (-6,-4), and (-5,-4). If however, the attempt to reference back to location (-8, -4) crosses a line boundary, no action is taken, preventing the interpolator from writing over the end of line markers.

The routine takes about 45 instruction cycles on the average, which corresponds to about 36 microseconds for the 8Ø85A-2 at its maximum rate.

```
;This is the horizontal interpolator routine.
hint:   mov a,l ;check for first pel of the line. If so, don't interpolate.

        ora a
        jnz hnt
        mov a,h
        rrc
        jc hnt
        rrc
        rnc
hnt:    mvi d,2 ;else proceed
       mov a,m
       dcx h
       dcx h
       dcx h
       dcx h
       mov b,m
       sub b
       jc neg
       rar
       stc
       cmc
       rar
       jmp intrplt
neg:   rar
       stc
       rar
intrplt:mov e,a
       mov a,b
eoae:  inx h
       add e
       mov m,a
       dcr d
       jp eoae
       inx h  ;return pointer to its original state
       ret
```

FIGURE A-H. Horizontal Interpolator Subroutine

Appendix I: The Receiver

The section included in the program listing for the receiver controller and all of its subfunctions. These subfunctions are almost identical to some of those already presented and the controller routine was discussed in chapter V, therefore further explanation is not included here. Also included is the machine code listing with memory address assignments as they were in the simulator implementation.

```
rcvr: sta 0beb1  ¶store input byte in RWRD
mvi a,9  ¶set to keep track of the number of bits left in the present word
sta 0beb6 ¶and store the value
        lda 0beb0  ¶get flag which indicates the type of data that is currently
                   ¶expected from the transmitter
        rrc
        jc roff  ¶if no transmission in progress then jmp

        rrc
        jc rmsmth ¶else if line mismatch value is expected then jmp
        rrc
        jc rhighs ¶else if first line of highs then jmp
        rrc
        jc rhighs ¶else if second line of highs then jmp
rlows:  lda 0beb6 ¶else lows expected; get bit counter value for input word
        dcr a    ¶and decrement it
        rz ¶if all bits exhausted then return
        sta 0beb6  ¶else store its new value
        lda 0beb1  ¶most recently received byte from transmitter
        rlc       ¶get the next bit from the LSB position
        sta 0beb1  ¶and store the modified word
        lda 0beb4 ¶get byte used for accumulating the input bits
        ral       ¶shift in the bit just taken from the received word
        sta 0beb4 ¶and store the result
        jnc rlows ¶if the byte is still not full then iterate
        lhld 0beb2  ¶else full. get address for storage
        inx h
        inx h
        inx h
        inx h
        shld 0beb2
        mov m,a
        call rhint ¶ horizontal interpolation
        call rvint ¶ vertical interpolation
        mov a,h
        ani 03    ¶mask off the upper 6 bits of the version in the accumulator
        adi 0ba ¶and replace them with the pointer for the middle line
        mov h,a  ¶use this new pointer
        call rhint ¶to iterpolate the corresponding values on the middle line
        xchg
        lhld 0beb2 ¶restore the orignal pointer
        mov a,m   ¶and the value just stored there, to check for framing indicat
        cma       ¶indications
        cpi 01
        mvi a,1
        sta 0beb4 ¶set the bit accumulator to accept 8 bits next time also
        jz endlw  ¶if new line indication then jmp
        jnc rlows ¶if no framing indications then jmp
```

FIGURE A-I. Receiver Routines

```
rpage:  mvi l,0   ¶else 'new page' indication
        mov a,h   ¶switch the address pointer
        xri 04    ¶to the beginning of the other line
        ani 0fc
        mov h,a
        mvi m,0ff ¶also put the 'new page' indication there
        shld 0beb2  ¶ store the pointer
        mvi a,02
        sta 0beb0 ¶set the flag such that the line mismatch value is expected
        rrc
        sta 0beb4 ¶and set the bit accumulator to accept 8 bits
        jmp rmsmth
endlw:  lda 0beb7 ¶get the line mismatch value
        add l   ¶the value is added to the present value of the
                ¶address pointer
fixup:  mvi m,0 ¶zeroes are loaded into all location in this interval
        inx h ¶move pointer to the next address
        xchg
        mvi m,0
        inx h
        xchg
        cmp e   ¶comparison with the upper limit
        jnz fixup
        mvi m,0fe ¶when finished, load last address with 'new line'
        xchg

        mvi m,0fe
        xchg
        mov a,h
        ani 0fc  ¶set pointer back to beginning of line so that the values
                 ¶can be added to the incoming highs values
        mov h,a
        mvi l,0
        dcx h ¶move pointer back such that the first value taken
        shld 0beb2 ¶will come from the first location of the line
        lda 0beb0
        rrc
        rrc
        sta 0beb0 ¶set flag such that one or two lines of highs are expected
        mvi a,20
        sta 0beb4 ¶set input bit accumulator to accept 3 bits
        jmp rhighs
rmsmth: lda 0beb6 ¶get value from input bit counter
        dcr a
        rz ¶if bits exhausted, then return
        sta 0beb6
        lda 0beb1  ¶get the current input data word
        rlc      ¶take the bit from the LSB position
        sta 0beb1 ¶and put back the remainder
```

FIGURE A-I. (continued)

```
        lda 0beb4 ¶get input bit accumulator
        ral       ¶and load in the new bit
        sta 0beb4
        jnc rmsmth ¶if not yet full then iterate
        mov b,a  ¶else  full. save value in reg B
        mvi l,0  ¶set reg L=0
¶The line mismatch value is transmitted four consecutive times to reduce the
¶probability of an error due to noise in the transmission channel.
¶ The rest of this program separates and adds these values together.If they are
¶all the same, the two LSB's of the result will be zero. If not, then some error
¶has definitely occurred and the value is rounded to the nearest valid value.
        ani 3  ¶set upper 6 bits to zero
        add l
        mov l,a
        mov a,b
        rrc
        rrc
        ani 3
        add l
        mov l,a
        mov a,b
        rlc
        rlc
        mov b,a
        ani 3
        add l
        mov l,a
        mov a,b
        rlc
        rlc
        ani 3
        add l
        rrc
        rrc
        jnc noerr
        inr a
noerr:  sui 04 ¶convert it to a more convenient form
        cma
        inr a
        ani 3f
        sta 0beb7
        mvi a,20

        sta 0beb0 ¶set flag to continue receiving the first lows line of the
                  ¶image
        mvi a,1   ¶set input bit accumulator to accept 8 bits
        sta 0beb4
        jmp rlows
```

FIGURE A-I. (continued)

```
roff:   lda 0beb6
        dcr a
        rz
        sta 0beb6
        lda 0beb1
        rlc
        sta 0beb1
        lda 0beb4
        ral
        sta 0beb4
        cpi 0ff
        jnz roff
        lhld 0beb2
        mvi l,0
        mov a,h
        ani 0fc
        xri 04
        mov h,a
        mvi m,0ff
        shld 0beb2
        mvi a,01
        sta 0beb4 ¶SET INPUT BIT ACCUMULATOR TO ACCEPT 8 BITS
        rlc
        sta 0beb0 ¶set flag to expect the line misatch value
        jmp rmsmth
rhighs: lda 0beb6
        dcr a
        rz
        sta 0beb6
        lda 0beb1
        rlc
        sta 0beb1
        lda 0beb4
        ral
        sta 0beb4
        jnc rhighs
        cmc
        rrc
        rrc
        rrc ¶rotate value into the three msb's
        sta 0beb4 ¶store temporarily back in the bit accumulator
        lhld 0beb2 ¶get lows pointer
        inx h
        shld 0beb2 ¶and update it
        lda 0beb0
        cpi 04    ¶check for highs1 indication
        jnz hgh2  ¶if none, then jmp
hgh1:   mov a,h
        ani 03
       adi 0ba ¶switch six msb's of the lows pointer to the interpolated line
       mov h,a
```

FIGURE A-I. (continued)

```
hgh2:  push h  ¶save pointer on the stack
       call rnoisep
       lda 0beb4 ¶get hghs value
       sub b     ¶and subtract the PRN value from it
       mov l,a   ¶put result in low order byte of pointer for
       mvi h,09 ¶expander lookup table
       mov a,m   ¶and get the scaled value
       pop h     ¶restore the lows pointer
       mov b,m   ¶and get the corresponding lows value.

       add b
       out 07 ¶this should be the completely enhanced image data
       mvi a,20
       sta 0beb4  ¶reset FRAME in anticipation of next highs value
       mov a,b
       cpi 0fe   ¶check the lows value for framing indications
       jc rhighs  ¶if none, then jmp
       out 08   ¶else generate output framing pulse
       jnz rhighs ¶if not the end of the line, then jmp
       lhld 0beb2 ¶else reset the lows pointer, RLWS
       mvi l,0
       mov a,h
       ani 0fc
       mov h,a
       dcx h
       shld 0beb2
       lda 0beb0 ¶get and update RFLAG
       rlc
       sta 0beb0
       cpi 08   ¶check for highs2 indication
       mvi a,20
       sta 0beb4 ¶set FRAME for 3-bit words
       jz rhighs ¶if highs2 indication,then jmp
       mvi a,01  ¶else prepare to receive lows
       sta 0beb4 ¶by setting FRAME for 8-bit words
       mov a,h   ¶and by setting RLWS for storage of the next lows line
       xri 08   ¶switch lows pointer to just before the first location of the
       mov h,a  ¶other line
       mvi l,0fc ¶decrement it three more times, so that the next value loaded
        ¶will indeed be loaded into the first location of the line
       shld 0beb2
       jmp rlows
¶This is the end of the receiver's main program (i.e. the receiver controller)
¶The following are the subroutines used by this program
rnoisep:  mov a,l
          ani 07
          mov l,a
          lda 0beb5
          ani 07
```

FIGURE A-I. (continued)

```
          rlc
          rlc
          rlc
          ora l
          ori 40
          mov l,a
          mvi h,0b
          mov a,m
          ret
¶This is the end of the routine which fetches the pseudo-random noise values
rhint:    mov a,l ¶check for the first pel of the line
          ora a
          jnz rhnt
          mov a,h
          rrc
          jc rhnt
          rrc
          rnc    ¶return if the first location of the line
rhnt:     mvi d,2 ¶else proceed
          mov a,m
          dcx h
          dcx h
          dcx h
          dcx h
          mov b,m
          sub b

          jc rneg
          rar
          stc
          cmc
          rar
          jmp rintrplt
rneg:     rar
          stc
          rar
rintrplt:mov e,a
          mov a,b
reoae:    inx h
          add e
          mov m,a
          dcr d
          jp reoae
          inx h   ¶return pointr to its original state
          ret
¶This is the end of the horizontal interpolator routine for the receiver, and is
¶exactly the same as the one used in the transmitter.
rvint:    mov c,m
          mov b,h  ¶save upper byte of lows pointer
          mov a,h
          xri 04
```

FIGURE A-I. (continued)

```
        mov h,a
        mov a,m
        sub c
        rar
        add c
        mov c,a
        mov a,h
        ani 03
        adi 0ba
        mov h,a
        mov m,c
        mov h,b ¶restore pointer to its original state
        ret
¶This is the end of the vertical interpolator for the receiver, and is identical
¶to the one used in the transmitter except for the way it treats 'new line'
¶indications and the pointer manipulations required
rinit:  lxi h,0beb0 ¶HL reg points to RFLAG (←Receiver Initialization)
        mvi m,1
        inx h       ¶HL reg points to RWRD
        mvi m,0
        inx h
        inx h
        inx h       ¶HL reg points to FRAME
        mvi m,0
        inx h    ¶HL reg points to RNCNT, the receiver's line counter
        mvi m,0
        inx h    ¶HL reg points to RCNT, the receiver's input bit counter
        mvi m,0
        inx h    ¶HL reg points to RMTCH, the line mismatch value
        mvi m,0
        lxi h,2000
        shld 0beb2  ¶initialize receiver's lows pointer, RLWS
      ret
```

FIGURE A-I. (continued)

Appendix J: I/O routines

The I/O scheme discussed in chapter VI requires the addition of I/O initialization and interrupt routines to the software used for the simulator implementation. There also need to be some additions made for the proper maintenance of the input output buffer areas. These later additions necessitate making some minor changes to the existing system so that these buffer areas are accessed rather than the ports themselves. These were discussed in chapter VI and are listed here in a manner that will hopefully be understandable without listing the entire systems software.

Note that a tentative routine to drive an external hardware multiplier (MLTPLY) has been included. This routine requires little under 14 microseconds, assuming the CPU is run at its maximum rate, or in this case 2Ø instruction cycles. It replaces the current multiplication routine, MULTIPLY. The external multiplier would be expected to convert in at most 1.6 microseconds, a specification that can be readily met with even a sequential-add type of device. In fact, this specification can be halved using several standard TTL components as shown in the diagram (FIG.J2). The associated I/O ports do not require any additional interrupt and I/O routines since they are not interrupting devices, but rather slave devices responding to the CPU. They would be located, most likely, on a separate board with the multiplier.

```
init:   di
        mvi a,36 ¶program the interrupt controller
        out 040 ¶for vectors to start at 64 and for a
        mvi a,00 ¶four byte interval--for this example,at least.
        out 0c0
        ret ¶transfer control back to the main program
¶This initialization routine would be called from the main
¶program. The starting address for the vectors would
¶actually be set such that it did not conflict with the
¶storage space for the transmitter and receiver routines,
¶i.e. somewhere between 3 and 4k.
       di ¶This is the location of the first interrupt vector
       jmp int0
        di
       jmp int1
        di
       jmp int2
        di
       jmp int3
¶The interrupt routines themselves follow
int0:   push h
        push psw
        lhld 0be34
        in 0c1 ¶from scanner (AB0)
        mov m,a ¶store value in input buffer
        inr l
        mvi a,0f ¶the buffer area extends from 0be00 to 0be0f
        cmp l
        jnc fin0
       mvi l,0 ¶wrap pointer
fin0:   shld 0be34
        pop psw
        pop h
        ret
int1:   push h
        push psw
        lhld 0be36
        mov a,m
        inr l
        out 0c1 ¶ to FACS (AB0)
       mvi a,1f ¶the buffer extends from 0be10 to 0be1f
        cmp l
        jnc fin1
        mvi l,10  ¶wrap pointer
fin1:   shld 0be36
        pop psw
        pop h
        ret
```

FIGURE A-J1. I/O Routines

```
int2:   push psw
        lda 0be38 ¶the output to the transmission channel
                  ¶is direct, due to time constraints
        out 0c2 ¶to channel (AB1)
fin2:   pop psw
        ret
int3:   push h
        push psw
        lhld 0be3a
        in 0c2 ¶from channel (AB1)
        mov m,a
        inr l
        mvi a,4f ¶the buffer area extends from 0be40 to 0be4f
        cmp l
        jnc fin3
        mvi l,40 ¶wrap pointer
fin3:   pop psw
        pop h
        ret
```

```
¶This is the subroutine for driving an extrnal multiplier.
mltply: ldax b
        out 0c4
        ldax d
        out 0c8
        nop
        in 0c8
        mov m,a
        ret
```

FIGURE A-J1. I/O Routines(cont'd)

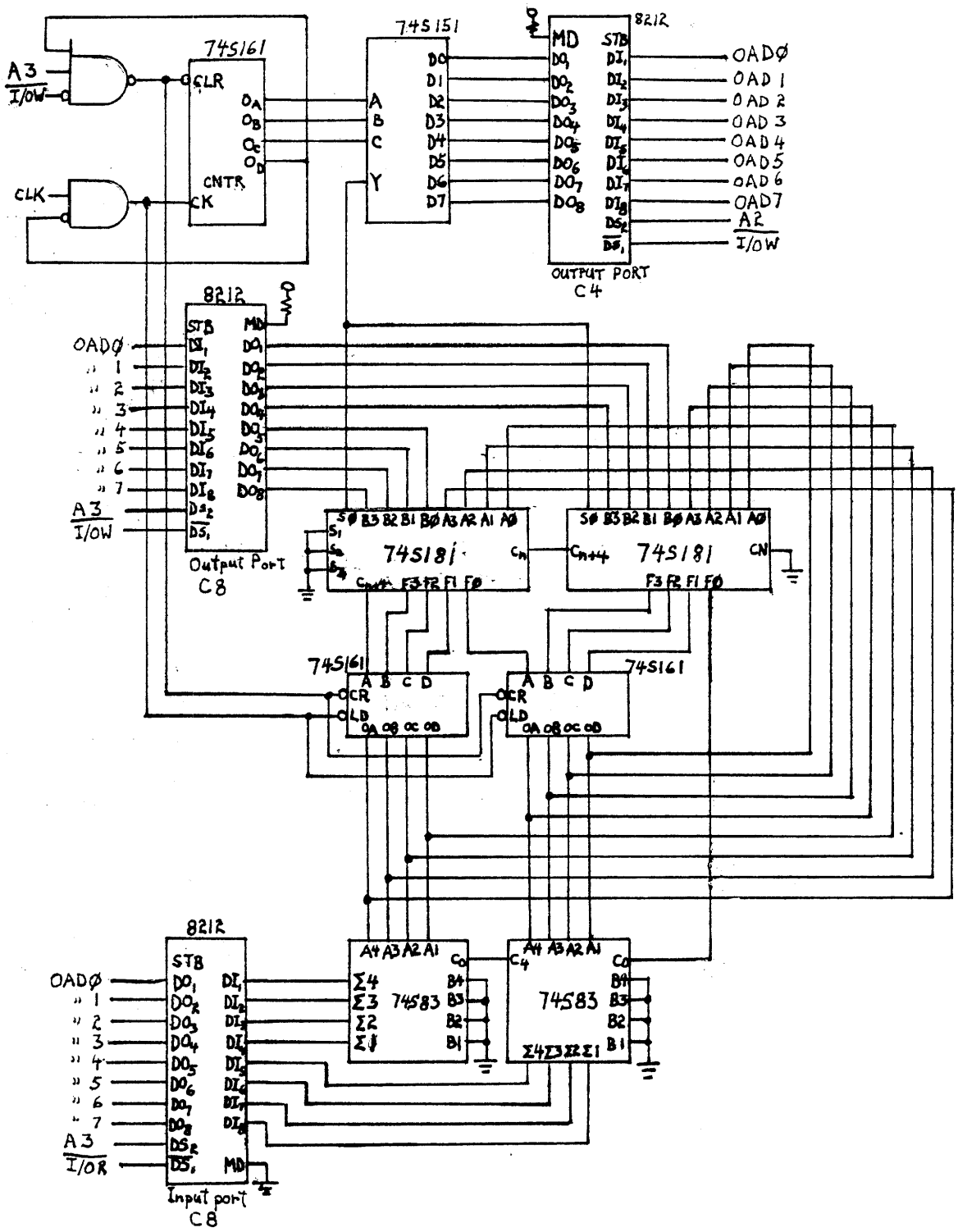


FIGURE A-J2. MSI Multiplier

Appendix K: The Multiplication Subroutine

The program listing is for the multiplier used by the transmitter in the simulator implementation and performs according to the technique presented in chapter III. It expects the register pairs BC and DE to containg pointers to the two avalues to be multiplied and returns the result in register A.

It takes about 142 instructions on the average, which would correspond to about 114 microseconds at the 8Ø85A-2's maximum rate.

```
;This is the software multiplier routine, i.e. no external devices
;are involved. It multiplies the bytes indicated by the DE register
;pair and by the BC register pair and put the result into the
;accumulator. Its result is rounded to one byte.
multiply: push h
        mov h,b

        mov l,c
        mov a,m
        xchg
        mov d,m
        xchg
        mvi l,8
        push d
        push b
        mvi e,0
        stc
        cmc
mult:   rar
        mov d,a
        mov a,h
        jnc mnext
        cmc
        add e
        mov e,a
mnext:  mov a,e
        dcr l
        jz finim
        rar
        mov e,a
        mov a,d
        mvi b,0
        jnc mult
        mvi b,1
        jmp mult
finim:  dcr b
        jm fmult
        inr a
fmult:  pop b
        pop d
        pop h
        ret
```

FIGURE A-K. Multiplication Subroutine

Appendix L: Parameter Storage

The data storage for the various lookup tables and the noise mask is listed below*.

```
¶The first 256 bytes are for the compressor
 0000   00 01 06 0A 0D 10
        13 15 18 1A 1C 1E
        1F 21 23 25
 0010   26 28 29 2B 2C 2D
        2F 30 31 32 34 35
        36 37 38 39
 0020   3B 3C 3D 3E 3F 40
        41 42 43 44 45 46
        47 48 49 49
 0030   4A 4B 4C 4D 4E 4F
        50 50 51 52 53 54
        54 55 56 57
 0040   58 58 59 5A 5B 5B
        5C 5D 5E 5E 5F 60
        60 61 62 63
 0050   63 64 65 65 66 67
        67 68 69 69 6A 6B
        6B 6C 6D 6D
 0060   6E 6E 6F 70 70 71
        72 72 73 73 74 75
        75 76 76 77
 0070   78 78 79 79 7A 7A
        7B 7C 7C 7D 7D 7E
        7E 7F 7F 80
¶this is the end of the positive half of the table
 0080   80 81 81 82 82 83
        83 84 84 85 86 86
 008C   87 87 88 88 89 8A
        8A 8B 8B 8C 8D 8D
 0098   8E 8E 8F 90 90 91
        92 92 93 93 94 95
 00A4   95 96 97 97 98 99
        99 9A 9B 9B 9C 9D
 00B0   9D 9E 9F A0 A0 A1
        A2 A2 A3 A4 A5 A5
        A6 A7
 00BE   A8 A8 A9 AA AB AC
        AC AD AE AF B0 B0
        B1 B2
 00CC   B3 B4 B5 B6 B7 B7
        B8 B9 BA BB BC BD
        BE BF
```

---

* The #'s in the left column are the start addresses for each group of values(0000 maps to memory location 8000).

```
00DA   C0 C1 C2 C3 C4 C5
       C7 C8 C9 CA CB CC
       CE CF
00E8   D0 D1 D3 D4 D5 D7
       D8 DA DB DD DF E1
       E2 E4
00F6   E6 E8 EB ED F0 F3
       F6 FA FF 00
¶The next 256 bytes are for the expander
0100   00 01 01 01 02 02
       02 02 02 03 03 03
       04 04 04 04
0110   05 05 06 06 06 07
       07 08 08 09 09 0A
       0A 0B 0B 0C
0120   0C 0D 0D 0E 0F 0F
       10 11 11 12 13 13
       14 15 16 16
0130   17 18 19 19 1A 1B
       1C 1D 1E 1E 1F 20
       21 22 23 24
0140   25 26 27 28 29 2A
       2B 2C 2D 2F 30 31
       32 33 34 35
0150   37 38 39 3A 3B 3D
       54 34 24 14 F3 E3
       46 47 49 4A
0160   4B 4D 4E 50 51 53
       54 56 57 59 5A 5C
       5D 5F 60 62
0170   64 65 67 68 6A 6C
       6D 6F 71 73 74 76
       78 7A 7B 7F
0180   81 85 86 88 8A 8C
       8D 8F 91 93 94 96
018C   98 99 9B 9C 9E A0
       A1 A3 A4 A6 A7 A9
       AA AC
019A   AD AF B0 B2 B3 B5
       B6 B7 B9 BA BB BD
       BE BF
01A8   C1 C2 C3 C5 C6 C7
       C8 C9 CB CC CD CE
       CF D0
01B6   D1 D3 D4 D5 D6 D7
       D8 D9 DA DB DC DD
       DE DF
01C4   E0 E1 E1 E2 E3 E4
       E5 E6 E7 E7 E8 E9
       EA EA
01D2   EB EC ED ED EE EF
       EF F0 F1 F1 F2 F3
       F3 F4
01E0   F4 F5 F5 F6 F6 F7
       F7 F8 F8 F9 F9 FA
       FA FA FB FB FC
01F1   FC FC FC FD FD FD
       FE FE FE FE FE FF
       FF FF 00
¶the next 256 bytes are for
the luminance scale factor.
0200   00 02 03 05 06 08
       09 0B 0C 0E 0F 11
       12 14 15 17
0210   18 1A 1C 1D 1F 20
       22 23 25 26 28 29
       2B 2C 2E 2F
0220   31 33 34 36 37 39
       3A 3C 3D 3F 40 42
       43 45 46 48
0230   49 4B 4D 4E 50 51
       53 54 56 57 59 5A
       5C 5D 5F 60
0240   62 64 65 67 68 6A
       6B 6D 6E 70 71 73
       74 76 77 79
0250   7A 7C 7E 7F 81 82
       84 85 87 88 8A 8B
       8D 8E 90 91
0260   93 94 96 98 99 9A
       9B 9C 9C 9D 9E 9F
       A0 A1 A2 A2
0270   A3 A4 A5 A6 A7 A8
       A8 A9 AA AB AC AD
       AD AE AF B0
0280   B1 B2 B3 B3 B4 B5
       B6 B7 B8 B8 B9 BA
       BB BC BD BE
0290   BE BF C0 C1 C2 C3
       C4 C4 C5 C6 C7 C8
       C9 CA CA CB
02A0   CC CD CE CF CF D0
       D1 D2 D3 D4 D5 D5
       D6 D7 D8 D9
```

```
02B0   DA DB DB DC DD DE
       DF E0 E0 E1 E2 E3
       E4 E5 E6 E6
02C0   E7 E8 E9 EA EB EC
       EC ED EE EF F0 F1
       F1 F2 F3 F4
02D0   F5 F6 F7 F7 F8 F9
       FA FB FC FC FD FE
       FF FF FF FF
02E0   FF FF FF FF FF FF
       FF FF FF FF FF FF
       FF FF FF FF
02F0   EF DF CF BF AF 9F
       8F 7E 70 60 4E 40
       30 20 10 00
```

¶The next 64 bytes are for the local contrast scale factor

```
0300   1A 1C 1F 21 24 26
       29 2B 2E 30 33 36
       38 3B 3D 40
0310   42 45 47 4A 4D 4F
       66 66 66 66 66 66
       66 66 66 66
0320   66 66 66 66 61 5C
       57 52 4D 47 42 3D
       38 33 2E 29
0330   24 1F 1A 18 17 16
       14 13 12 11 0F 0E
       0D 0D 0D 0D
```

¶The next 64 bytes are the values for the psuedorandom noise

```
0340   EF 00 F3 04 F0 02
       F4 05 08 F7 0C FB
       09 F8 0D FC
0350   F5 06 F1 02 F6 07
       F2 03 0E FD 0A F9
       0F FE 0B FA
0360   F0 01 F4 05 EF 00
       0C 05 0A F8 0D FD
       09 F7 0C FC
0370   F7 08 F3 03 F5 07
       F2 03 10 00 0C FB
       0E FD 0A F9
0380
```

REFERENCES

1. Curlander, P.J., "Image Enhancement Using Digital Adaptive Filtering", SM Thesis, MIT Electrical Engineering and Computer Science Department, August 1977.

2. Gilkes, A.M., "Photographic Enhancement by Adaptive Digital Unsharp Masking", SM Thesis, MIT Electrical Engineering and Computer Science Department, 1977.

3. Hoover, G.L., "An Image Enhancement/Transmission System", SM Thesis, MIT Electrical Engineering and Computer Science Department, May 1978.

4. Huang, T.S., Tretiak, O.J., Prasada, B., and Yamaguchi, Y., "Design Considerations in PCM Transmission of Low-Resolution Monochrome Still Pictures.", Proceedins of the IEEE, Vol. 55, No. 3, pp. 331-335, March 1967.

5. Huang, T.S., "Digital Picture Coding", Proceeding of the National Electronics Conference, Vol. XXLL, 1966.

6. Oppenheim, A.V., and Schafer, R.W., Digital Signal Processing, Prentice Hall, Inc., Englewood Cliffs, New Jersey, 1975.

7. Roberts, L.G., "Picture Coding Using Pseudo-Random Noise", IRE Transaction on Information Theory, Vol. IT-8, pp. 145-154, February 1962.

8. Schreiber, W.F., "Aspects of Image Processing", unpublished, MIT, Electrical Engineering and Computer Science Department, September 1979.

9. Schreiber, W.F., "Image Processing for Quality Improvement", Proceedings of the IEEE, Vol. 66, No. 12, pp 1640-1651, December 1978.

10. Schreiber, W.F., "Picture Coding", Proceeding of the IEE, Vol. 55, No. 3, pp 320-330, March 1967.

11. Troxel, D.E., Schreiber, W.F., Grass, R., Hoover, G.L., Sharpe, R., "Bandwidth Compression of High Quality Images", ICC, 1980.

Technical Manuals:

12. INTEL, "MCS-8Ø/85 Family User's Manual", Intel Corp., October 1979.

13. INTEL, "Memory Design Handbook", Intel Corp., 1977.

14. INTEL, "8Ø8Ø Microcomputer Peripherals User's Manual", Intel Corp., 1975.

15. INTEL, "8Ø8Ø Microcomputer Systems User's Manual", Intel Corp., 1975.

16. Signetics, "Signetic Bipolar/MOS Microprocessor Data Manuel", Signetics Corp., 1977.

17. Texas Instruments, "TTL Data Book, for Design Engineers", Texas Instruments Inc., 1973.