RSD-TR-1-84

THE USE OF THE DEANZA IP6400 IMAGE PROCESSOR FOR LOCAL WINDOW OPERATIONS

Edward J. Delp
Nirwan Ansari

Department of Electrical and Computer Engineering

The University of Michigan

Ann Arbor, Michigan 48109-1109

January 1984

CENTER FOR ROBOTICS AND INTEGRATED MANUFACTUR

Robot Systems Division

COLLEGE OF ENGINEERING

THE UNIVERSITY OF MICHIGAN

ANN ARBOR, MICHIGAN 48109-1109

UMR0572

ABSTRACT

This report discusses the use of the DeAnza IP6400* image processor for high speed local window operations. A system overview of the image processor and its important features are first introduced. Algorithms for linear and non-linear window operators are then presented.

*IP6400 is an image processing system manufactured by Gould Inc., DeAnza Imaging and Graphics Division.

TABLE OF CONTENTS

2.1. The DeAnza IP6400 2

2.2. Digital Video Processor (DVP) 3

3.1. 3×3 Window Operation 6

3.2. n×m local window operation 12

3.3. 3×3 Median Filter 13

3.4. n×m median filter 19

1. Introduction

Most image processing operations take a large amount of memory space and cpu time. General purpose computers are not usually equipped with features necessary for high speed image processing operations. Nowadays, high speed image processing is desirable in many industrial and medical applications. The DeAnza IP6400 is used for its special features that make real time processing feasible. This report will first introduce an overview of the IP6400, and its most important option, the digital video processor (DVP), which provides facilities for various processing operations. Algorithms using the IP6400 to perform general 3×3 local window operations, $n \times m$ local window operations, 3×3 median filtering, and $n \times m$ median filtering are then presented.

2. System Overview

2.1. The DeAnza IP6400

The IP6400 Series Image Array Processor contains many features which make it a powerful imaging system. It can be used solely as a display system which provides high resolution color, pseudo color, and multiple monochrome image display. Coupled with other options, real time image processing can be achieved using the IP6400. Among the options acquired in our system are a video signal digitizer and control, alphanumeric overlay generator, graphic overlay channel, trackball cursor generator and control, joystick cursor generator and control, external synchronous source input, video output controller, and, most of all, the digital video processor arithmetic unit which makes window operations feasible. Our system also contains four 512x512 8-bit memory planes and four video cards with 8 bit DAC's. It is interfaced with a VAX 11/780 host computer running UNIX** (4.1bsd). Figure 1 shows an overview of the IP6400 system.

**UNIX is a Trademark of Bell Laboratories.

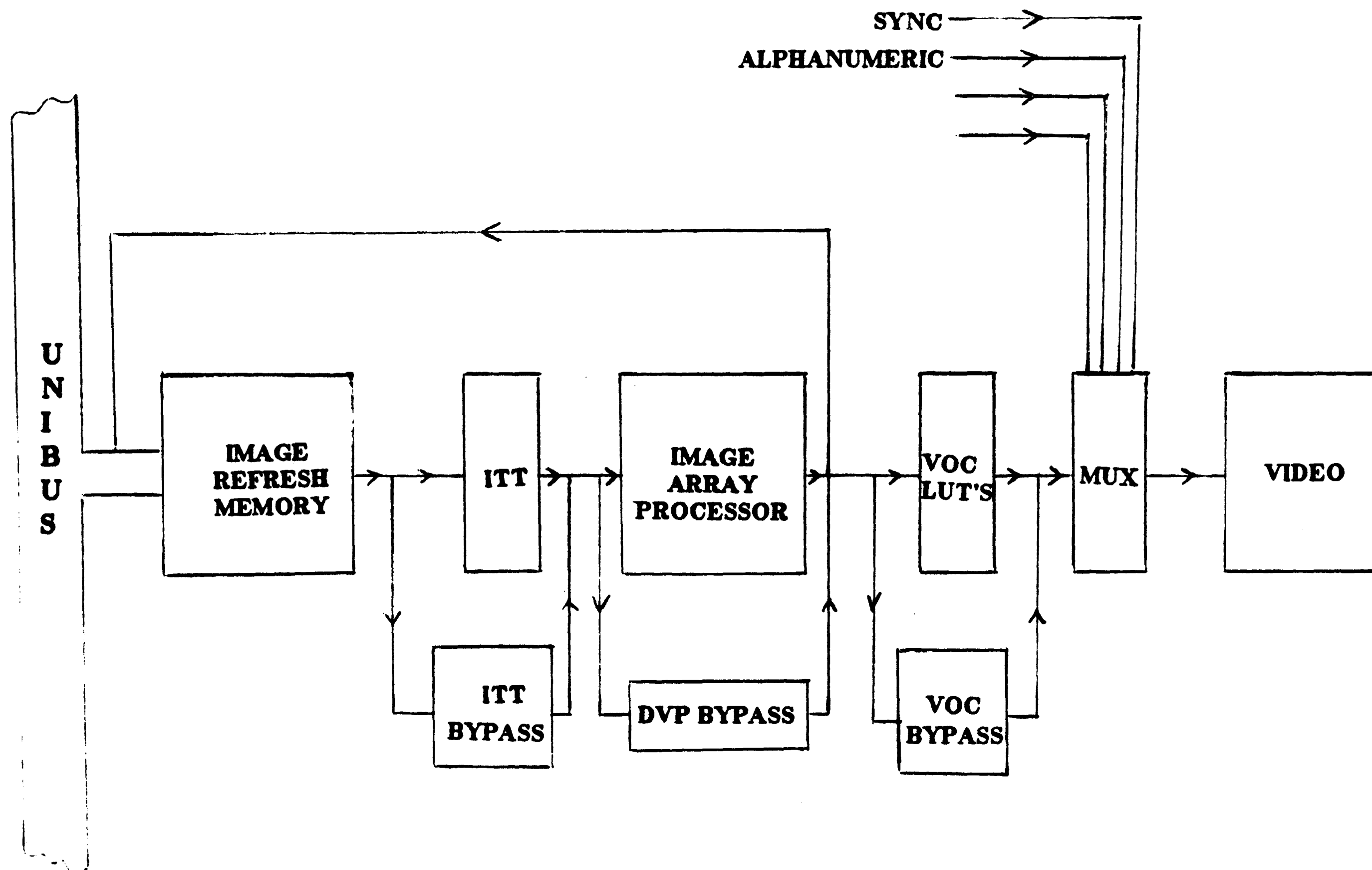


Figure 1. Overview of the DeAnza IP6400***

2.2. Digital Video Processor (DVP)

The importance of the DVP is due to its ability to handle operations on a large amount of image data in one video frame time(33 ms). The data paths of the DVP are shown in Figure 2.

The DVP has two arithmetic/logic units (ALU) which allows the conditional outputs from the test ALU to determine swapping of the two operation codes of the operational ALU. In other words, the results of a test on an individual pixel can be used to determine the operation

***The block diagram is taken from the IP6400 Programmer's Manual[1].

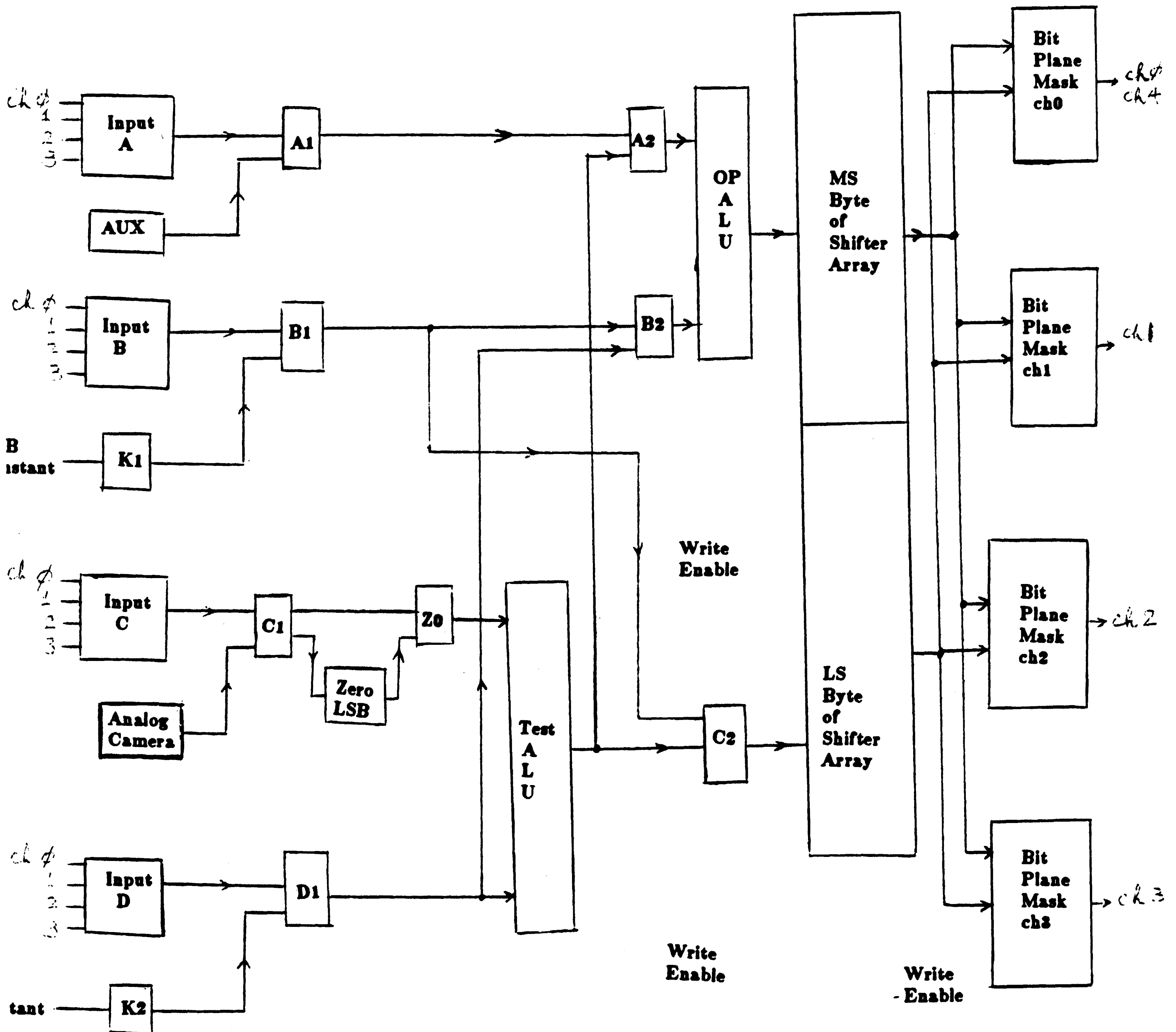


Figure 2. Data Path of the DVP****

on that pixel through the operational ALU.

Inputs to the ALU's are determined by the Input Data Paths Registers (INPDP) by means of multiplexors and selectors. It can select data from any of the four image refresh memory

****The block diagram is taken from the IP6400 Programmer's Manual[1].

channels directly or through the Intensity Transformation Tables (ITT) under control of the Memory Port Control register (MEMPC). The INPDP also allows data acquisition from the camera A/D, and constant values from constant register or auxiliary input. Users can also make use of it to achieve compound operations such as 16-bit arithmetic operations.

Under the control of the Output Data Paths and Shift Control register (ODPSH) , output data from the ALU's can be shifted or rotated as either two independent 8-bit data values or a combined 16-bit data value to each image memory channel including the graphic overlay. The Bit Plane Mask register (BPM) controls the output writing to any single bit or combination of bits of each image memory channel.

The Destination registers (DEST) may be used to scroll data output from the DVP to the target memories in multiples of 16 pixels. This is convenient for arithmetic operations on two memory channels. For arbitrary offsets, rather than restricted by multiples of 16 pixels, there are scroll registers associated with each memory to scroll, zoom, and selectively inhibit data from the source memories. Thus, arithmetic operations can readily be done by appropriate manipulation of these registers.

The rest of the paper will discuss the philosophy of using the IP6400 to perform local window operations including median filtering. The mnemonics, virtual addresses and function of the DVP registers and other registers which are used for window operations are found in Appendix A[1].

3. Local Window Operations

:st image operation is the local window operation. These operations take a great deal of time if done on a general purpose computer because the operations are usually performed pixel by pixel. Using the IP6400 image processor one can speed up the operation tremendously. It takes only one video frame time (33 msec) for one DVP operation on the

whole image (512x512). A DVP operation is a simple arithmetic or logic operation on an image through the DVP. A series of programs have been written to perform operations using the DVP. These programs mainly set up correct communication between the host computer and the IP6400 by loading and synchronizing the registers and operations of the IP6400. Manual entries for the various programs can be found in Appendix B to G.

3.1. 3×3 Window Operation

A 3×3 local linear window operation on an image can be thought of as a convolution of a 3×3 window with the image. Consider the following weighted window:

$$\begin{matrix} w_1 & w_2 & w_3 \\ w_4 & w_5 & w_6 \\ w_7 & w_8 & w_9 \end{matrix}$$

As the window moves across the image, the pixel value in the center of the window at that instant is replaced by the sum of the products of the window coefficients and the corresponding pixel values of the image. In a global sense, the operation can be perceived as the result of the sum of nine image planes, in which each plane is the scrolled image multiplied by one of the window coefficients, i.e. $w_1 \times$ (original image scrolled one pixel left and up) + $w_2 \times$ (original image scrolled one pixel up) + $w_3 \times$ (original image scrolled one pixel right and up) + $w_4 \times$ (original image scrolled one pixel left) + $w_5 \times$ (original image) + $w_6 \times$ (original image scrolled one pixel right) + $w_7 \times$ (original image scrolled one pixel left and down) + $w_8 \times$ (original image scrolled one pixel down) + $w_9 \times$ (original image scrolled one pixel right and down). The versatility of the DVP comes into play for such operation. Each multiplication of one coefficient with the whole scrolled image takes only one video frame time (one DVP operation). The overall operation would take only nine video frame times if no modification is needed after the nine mentioned operations are completed. However, since image gray levels range from 0 to 255, there is always a chance of overflow and underflow if there are both positive and negative window coefficients. It takes another two DVP operations to take care of underflows and

overflows. This timing calculation does not include interface timing between the IP6400 and the host computer, which is usually small. To be exact timing for communication between the IP6400 and the host computer should also be considered. This timing depends on the speed of the host computer and the number of tasks being performed by the host computer. Another aspect that should be considered in using the DVP is that the two ALU's are unsigned 8-bit ALU's which are not capable of performing multiplication or division directly. In order to get around this, the Intensity Transformation Table (ITT) associated with each memory plane must be employed as a look-up table multiplication. Algorithms that require logarithmic or linear tables can also be implemented. The logarithmic table has the advantage of taking care of both positive and negative values, but it loses one bit of accuracy because both positive and negative values are represented by 8-bit data. The linear table is likely overloaded if the window coefficient is large. Hence, a scaling factor is necessary to avoid overloading the Intensity Transformation Table. The linear table is preferred in this operation since it can be implemented easier without the complication of the sign. To multiply a memory plane with a coefficient, the ITT associated with that plane is loaded with 256 bytes of data corresponding to values of the products of the absolute value of the coefficient and gray levels ranging from 0 to 255. The ITT is then used for indexing the pixel intensity. For the 3×3 window operation, the result can be accumulated in one memory plane by summing each of the nine scrolled images (scrolled according to each window position) through its associated ITT which was loaded with indices (products of the absolute value of the coefficient and the gray levels from 0 to 255) corresponding to gray levels from 0 to 255. If a coefficient is negative, the ALU performs subtraction. Synchronization is very crucial for programs that involve DVP operations. Any necessary calculations for the next DVP operation should be done in the host computer as soon as the APGO bit (initiating the DVP operation) of the Control and Status register (CSR, see Appendix A) has been set. Only when the APR bit (ready bit) of the Control and Status register becomes set (indicating the completion of a DVP operation), should the required IP6400 registers be loaded for next DVP operation. In this way, no registers are loaded during the DVP operation. Synchronization is thus ensured.

The 3×3 window operation software is written in the C programming language using a VAX11/780 host computer running UNIX. Three memory planes of the IP6400 are used for the operations. The original image to be processed is assumed to be in memory plane 0, and the result is placed in memory plane 1. Memory plane 2 is used to hold the information of overflows and underflows. Both memory plane 1 and 2 should be cleared before the operation. The algorithm consists of the following steps:

(1) read in nine window coefficients.

(2) do step 3 to step 5 for each window coefficient (i.e., 9 times).

(3) load itt0 with indices corresponding to products of the scaled coefficient (each coefficient is divided by the scaling factor) and image gray levels ranging from 0 to 255.

(4) set memory 2 as the input to the Operational ALU, and memories 1 and 0 as inputs to the test ALU. Note that data from memory 0 is scrolled accordingly before the input to the test ALU is set.

(5) if a coefficient is negative, perform subtraction in the test ALU (memory 1 - memory 0), and place the result in memory 1 (accumulating the result in memory 1). At the same time, decrement the input of the operational ALU (decrement memory 2 which stores underflows and overflows information), and place the result in memory 2. Otherwise, perform addition in the test ALU and increment the input of the operational ALU. Successive results are therefore accumulated in memory 1, while the information about overflows and underflows is accumulated in memory 2.

The result in memory 1 would be correct if there are no underflows or overflows and the scaling factor mentioned before is 1. If the indication that a pixel in memory 2 is an underflow, the corresponding pixel in memory 1 (accumulated result) is actually a negative number. But the ALU's of the DVP are unsigned and image gray level values cannot be negative. It is logical, to set to zero or take the absolute value of those pixels which are underflows. If a pixel is found to be an overflowed value, that pixel is set to 255. >From 2's complement arithmetic, an underflow occurs when a borrow is needed and an overflow when there is a carry bit. Pixels of

memory 2 which holds the overflow and underflow information should have a value of zero if there is no overflows or underflows. They should have values close to ff (hex) if there is underflows and values from 1 to 8 if there is overflows. >From this information overflows and underflows can be corrected using the swapping ability of the DVP's ALU's. The following steps are used for correcting the underflows and overflows:

(6) do step 7 to step 8 twice.

(7) place memory 1 as input to the operational ALU, and memory 2 as input to the test ALU.

(8) perform addition (memory 2 + constant) in the test ALU.

  (a) zero underflows: let the constant be 80 (hex), if there is carry bit from the test ALU, set output of the operational ALU to zero (80 hex plus an underflow indicator of value close to ff hex would produce a carry bit), or otherwise, pass input of the operational ALU as the output (pixels not underflowed are passed unchanged). Place output of the test ALU back into memory 2 which now holds the sum of previous values and the constant. Place output of the operational ALU in memory 1 which now holds the result of having corrected underflows.

  set overflows: let the constant be 7f(hex), if there is a carry bit from the test ALU, set output of the operational ALU to ff(hex) (7f hex plus the sum of an overflow indicator with value from 1 to 8 and 80 hex (previously done) would produce a carry), or otherwise, pass the input of the operational ALU as the output. Place output of the test ALU back in memory 2, and output of the operational ALU in memory 1 which now holds the correct result with both underflows and overflows taken care of.

If complementing underflows (taking absolute values) is desired, step (8b) should be taken instead of (8a):

  (b) underflows: let the constant be 01(hex), perform complementation of the input of the operational ALU if there is a carry bit from the test ALU (this will complement those underflows with actual values between -1 and -255, because the underflow

indicator would have value ff(hex), excessive underflows and overflows (more than once) are taken care of at the same time in the next step). Place output from the test ALU in memory 2, and the output from the operational ALU in memory 1 (which now holds the result with the underflows corrected).

overflows: excessive underflows and overflows can be corrected like step (8a), except that the constant becomes fe(hex). Set output of the operational ALU to 255 if there is a carry bit from the test ALU, or otherwise pass the input unchanged (summing fe(hex), 01(hex) (previous operation) and the overflow indicator would not produce carry bit if and only if the indicator is zero, in other words, there is neither overflows nor underflows). Place output from the operational ALU in memory 1.

Finally, the result in memory 1 should be scaled (multiplied) by the scaling factor. As mentioned earlier, the scaling factor is used to avoid overloading the ITT. Since partial results are accumulating in memory 1, severe truncation inaccuracy will result if the ITT is overloaded when it is loaded as a multiplication table. This truncation error would be accumulated and thus is undesirable. As a rule of thumb, the scaling factor which is determined by the user is usually chosen as the absolute value of the largest window coefficient provided that the

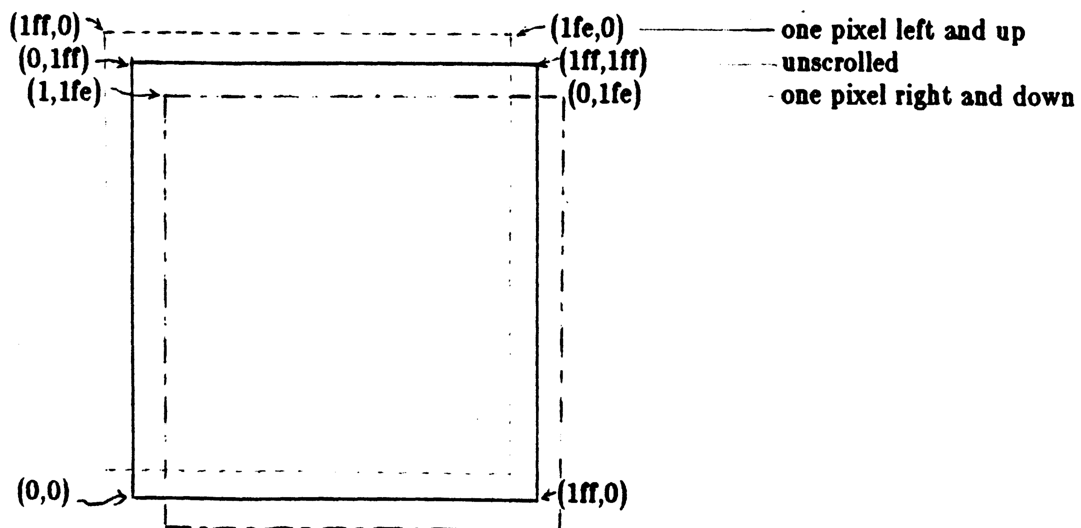


Figure 3. Scroll Position Diagram

magnitude of the chosen coefficient is greater than one. It would be one if magnitudes of all the window coefficients are less than one. Since the result is scaled by the scaling factor, the window coefficients are never affected by this scaling factor. The final step is:

(9) scale the result in memory 1 by the same scaling factor through the ITT of memory 1. Place the result back into memory 1. The final result is then found in memory 1.

The video pixels are organized as a 512x512 array. Columns are numbered in the x-direction (from left to right) from 0 to 511. Rows are numbered in y-direction (from bottom to top) from 0 to 511. Pixel location (0, 511) is then the upper left hand corner of the video display. The 3×3 window operation requires 9 different scrolling positions for the convolution. For example, the scrolling positions for w1 (one pixel left and up) is scrx=1ff(hex) and scry=0(hex) where scrx and scry are the scroll registers corresponding to x and y directions(see Figure 3). For 3×3 window operations the scrolling positions corresponding to the window positions w1 to w9 would be as follows:

window positions	scrx (hex)	scry (hex)
w_1	1ff	0
w_2	0	0
w_3	1	0
w_4	1ff	1ff
w_5	0	1ff
w_6	1	1ff
w_7	1ff	1fe
w_8	0	1fe
w_9	1	1fe

The overall 3×3 window operation takes n+3 DVP operations, where n is the number of non-zero window coefficients. There exists some inaccuracy using the IP6400 due to the round-off errors caused by the Intensity Transformation Table and the fact that the ALU's of the DVP are only 8-bit ALU's. There is however a tremendous gain in speed. There is little visible difference between most linear window operations done on the IP6400 and window operations done on a general purpose computer.

3.2. n×m local window operation

The philosophy of using the IP6400 to perform n×m local linear window operation is very similar to that of 3×3 local window operation. In fact, all the necessary DVP operations are of the same type. It takes nm+3-z (z is the number of zero coefficients) DVP operations for a n×m local window operation, nm-z DVP operations for accumulating the sum of the products of each window coefficient and its correspondingly scrolled image, and 2 DVP operations (exactly the same operations described for 3×3 window operation) for correcting underflows and overflows, and 1 DVP operation to scale the result by the same scaling factor mentioned before. One point that should be mentioned here is the evaluation of the scrolling positions for different window size. Consider the following window:

w(0,0)	w(1,0)	...	w(m,0)
w(0,1)	...	...	w(m,1)
.	...	...	.
.	...	...	.
.	...	...	.
w(0,n)	...	...	w(m,n)

Since the upper left corner of the video display is positioned by scrx=0 and scry=1ff(hex), the scrolling positions (specified by scrx and scry registers associated with each memory channel, see Appendix A) can be related to the window coefficients by the following equations.

For window position w(i,j):

$$scrx[i] = (scrxo - \frac{m}{2} + i) \; \& \; mask$$

$$scry[j] = (scryo + \frac{n}{2} - j) \; \& \; mask$$

where $scrxo = 0$

$scryo = 1ff(hex)$

$mask = 01ff(hex)$

& is the logical operation AND

The mask (1ff, hex) is used to mask out bits that are not used for scrolling position (see Appendix A). Note that the way the scrolling positions is evaluated has restricted m and n to be odd integers.

3.3. 3×3 Median Filter

Using the IP6400 for two-dimensional median filtering is challenging. Median filtering is a non-linear window operator and consists of a great number of sorting operations[2][3]. One of the best known properties of median filtering is that of noise reduction with edge intensity being preserved[3]. For a 3×3 window the median is the 5th largest pixel value within the window. The algorithm to determine whether a given pixel is a median value within a window is to count the number of other pixels within the window having values less than, and the number equal to the specified pixel. If there are four pixels within the window having values less than a particular pixel, that pixel is the median of the window. If there are more than four pixels within the window having values less than the pixel, the pixel cannot be the median. When there are less than four pixels having values less than the particular pixel, that pixel can still be the median depending on how many pixels within the window have gray value equal to the pixel. The conditions for a pixel being the median within the window can be summarized as follows:

Conditions		
no. less	minimum no. equal	median
0	4	yes
1	3	yes
2	2	yes
3	1	yes
4	0	yes
5	-	no
6	-	no
7	-	no
8	-	no

Table 1 - Conditions for a pixel being the median within the window

A program for 3×3 median filtering using the IP6400 was written in the C programming language running UNIX on the host processor. The algorithm requires the four memory planes of the IP6400. The original image to be processed is assumed to be in memory 0, and the final result is placed in memory 2. Memory 1 is used for comparison, and hence contains the same image. Memory 3 is used to accumulate the conditions for each pixel such that it is the possible median of its 3×3 window. The algorithm consists of the following steps:

(1) copy original image in memory 0 to memory 1.

(2) clear memory 2 and 3.

(3) do step 4 to step 8 for each window position (9 times).

(4) find the number of pixels within every 3×3 window that have values less than that specified by the window position. This takes eight comparisons. Accumulate this information in memory 3

(5) mask out the impossible choices that cannot be medians.

(6) find the number of pixels within every 3×3 window that have values equal to that specified by the window position. This takes another eight comparisons. Accumulate this information with that of step 4 in memory 3.

(7) Using the information in memory 3, possible pixels specified by that window position that are medians are determined. The result is accumulated in memory 2.

(8) clear memory 3 so that it is ready to store the conditions for pixels specified by next window position as possible medians.

The DVP must have the capability of comparison to carry out the above steps. Comparisons can be done through the two ALU's by performing one's complement subtraction in the test ALU whose carry output bit in turn determines the swapping of the two operation codes of the operational ALU. Comparisons are done memory plane by memory plane. Consider the following window:

w(0,0)	w(1,0)	w(2,0)
w(0,1)	w(1,1)	w(2,1)
w(0,2)	w(1,2)	w(2,2)

where w(i,j) indicates window position

Comparing pixels specified by w(0,0) with the rest of pixels specified by other window positions is done by comparing properly scrolled images corresponding to their relative positions to w(0,0) with pixels specified by w(0,0). For example, pixels specified by w(0,0) are compared to those of w(1,0) by comparing original image with the original image scrolled one pixel left, to those of w(1,1) with the image scrolled one pixel left and up, and so on. Since the upper left corner of the unscrolled image is positioned by scrx=0 and scry=1ff(hex), the corresponding upper left corner of the image scrolled one pixel left is then positioned by scrx=1ff(hex) and scry=1ff(hex) (see Figure 3). The scrolling positions for comparing window position w(k, l) with respect to w(i, j) are related as follows:

$$scrx = (scrxo + (i-k)) \ \& \ mask$$

$$scry = (scryo + (l-j)) \ \& \ mask$$

where *scrx* and *scry are values* for *scroll registers x* and *y* (*see Appendix A*)

$scrxo = 0$

$scryo = 1ff(hex)$

$mask = 1ff(hex)$

& is the logical operator AND

For the sake of programming, scrolling positions with respect to the center of the window (assume the center positions: scrx=0,scry=1ff) are first evaluated. Then scrolling positions for comparisons with respect to each specified position can be evaluated in a loop without concern for the relative indices of the window. The number of pixels having values less than that speci-

fied by a window position can be found through the operation of the two ALU's as follows:

(1) perform step 2 to step 3 for each window position within the window with respect to the specified window position (there are eight window positions with respect to the specified position, i.e. eight comparisons).

(2) perform one's complement subtraction in the test ALU (C-D-1) with the inputs of the original image represented by D, and the scrolled image (different scrolling for each compared position) represented by C (see Figure 2). The test ALU would produce a carry bit if and only if C is greater than D, i.e. the compared pixel has value less than that specified by the window position within its 3×3 neighborhood.

(3) set memory plane 3, which stores conditions of pixels that are possible medians, as the input to the operational ALU. Set up the operation codes such that the input is incremented if there is a carry bit from the test ALU, otherwise the input is passed unchanged. Accumulate the output in memory 3.

After eight comparisons for each window position, memory plane 3 holds information on the number of compared pixels being less than the center pixel of the window. As discussed above, numbers ranging from 5 to 8 are impossible choices of medians. To mask out those impossible choices, the ALU's are employed again as follows:

(1) add 7b(hex) to memory 3. Memory 3 consists of values ranging from 7b(hex) to 83(hex) in which 80(hex) to 83(hex) become impossible choices.

(2) mask out 80(hex) to 83(hex). Note that adding 80(hex) to a number greater than or equal to 80(hex) would produce a carry bit. Therefore, add memory 3 to a constant (80 hex) in the test ALU. Set memory 3 as the input to the operational ALU. Pass the input unchanged if there is no carry bit from the test ALU. Otherwise, set output to zero.

Memory 3 consists of zeroes and values from 7b to 7f corresponding to pixels in the image that are possible medians. Next, the number of pixels having values equal to that specified by the window position are determined and accumulated in memory 3. This is done in the following steps:

(1) perform step 2 to step 3 for each window position within the window with respect to the specified window position.

(2) perform one's complement subtraction (C-D-1) in the test ALU with the inputs of the scrolled image (compared position) as C, and the unscrolled image (specified window position) as D. Thus, there is a carry out if and only if C is greater than D. Set REQ and RCC of the tstop (test ALU register) to 1. Hence, the conditional operation code of the operational ALU is carried out if and only if there is no carry out and C=D. But, there is no carry out from the test ALU if C is less than or equal to D. Thus, the conditional operation code of the operational ALU is carried if and only if C=D, i.e. pixels compared are equal.

(3) set memory 3 as the input to the operational ALU. Increment the input of the operational ALU if the condition for the conditional operation code of the operational ALU is met, i.e. compared pixels are equal. Otherwise, pass the input unchanged. Place the output of the operational ALU in memory 3.

Memory 3 then contains conditions for corresponding pixels of the specified window position that are medians. As mentioned above, a pixel is a median if it meets the conditions listed in Table 1. Through the above operations, impossible choices have been masked out, and thus a pixel specified by the window position is a median if the corresponding pixel in memory 3 has a value greater than or equal to 7f. Finally, medians can be accumulated in memory 2 for each specified window position as follows:

(1) set the operation code of the test ALU with addition (C+D), and the inputs of memory 3 (scrolled according to window position, see explanation below) as C, and a constant, 81(hex) as D. Hence, a carry bit occurs only if a pixel value in memory 3 is greater than or equal to 7f, in other words, the corresponding pixel in the original image is the median.

(2) set the inputs of the operational ALU with memory 2 (accumulated result) as A, and the original image (scrolled according to window position, see explanation below) as B. Pass B unchanged as output to memory 2 if there is a carry out from the test ALU. Otherwise,

pass A unchanged. Thus, pixels determined to be medians for each window position are accumulated in memory 2.

If a pixel is determined to be the median, the center of the neighborhood should be replaced by this pixel. For a specified window position, say w(0,0), a pixel determined to be the median should be scrolled one pixel right and down replacing the center pixel. That is why memory 3 and the original image mentioned above must be scrolled according to the specified window position as inputs to the ALU's. These scrolling positions are related to their window positions by the following:

For w(i,j):

$scrx[i] = (scrxo{+}1{-}i)\ \&\ mask$

$scry[j] = (scryo{-}1{+}j)\ \&\ mask$

where $scrx[i]$ and $scry[j]$ *are values of scroll register x* and *y*

(*to specify a scrolled position*)

$scrxo = 0$

$scryo = 1ff(hex)$

$mask = 1ff(hex)$

& is the logical operator AND

The mask (1ff) is used for the same reason mentioned earlier.

The 3×3 median filtering takes altogether 182 DVP operations. It requires 1 DVP operation to preblank memory 2 and 3 and 1 DVP operation to copy the original image into memory 1. For each window position (there are nine) it requires 8 DVP operations to find out the number of pixels having values less than a particular pixel for every pixel in the 3×3 window, another 8 DVP operations for finding number of pixels having values equal, 2 DVP operations

to mask out impossible choices of medians, 1 DVP operation to accumulate medians, and 1 DVP operation to clear memory 3. There is a tremendous gain in speed when compared to median filtering done on a general purpose computer. It takes only 144 comparisons between image planes (up to 512x512) for the above operation. If done on a general purpose computer, 8 comparisons for each pixel of a 512x512 image (2097152 comparisons)are required. Since the operation does not involve the ITT's, there is no loss of accuracy such as the truncation inaccuracy mentioned in Section 3.1 and 3.2

3.4. n×m median filter

The algorithm for n×m (n is number of rows, m is number of columns) median filtering follows the same approach as of that 3×3 median filtering. The median of the n×m window is the $(\frac{(m \times n)}{2}+1)$th largest value within the window. Only slight modification has to be done on the evaluation of the scrolling positions.

For window position w(i,j):.

$$scrx[i] = (scrxo + \frac{m}{2} - i) \; \& \; mask$$

$$scry[j] = (scryo - \frac{n}{2} + j) \; \& \; mask$$

where $scrx[i]$ and $scry[j]$ *are values of the scroll registers*

(to specify a scrolled position)

$scrxo = 0$

$scryo = 1ff(hex)$

$mask = 1ff(hex)$

& is the logical operator AND

The mask is used for the same reason mentioned earlier. Note that n and m must be odd numbers. Modification also has to be made to the constant added in memory 3 (step 1 of masking out impossible choices of the previous section) to mask out impossible choices so that conditions for following operations to determine the medians remain the same. Since the constant added for a 3×3 window is 7b(hex), 80(hex)-5(hex), the constant added for n×m window is then 80(hex)-($\frac{(n \times m)}{2}$ +1)(hex). In this way, a pixel of an image is the median of its n×m neighborhood if the corresponding pixel in memory 3, which holds conditions for possible medians, is greater than or equal to 7f(hex), the same conditions for 3×3 median filtering.

The whole operation takes (2nm(nm+1)+2) DVP operations. It requires 2 DVP operations for copying the original image to memory 1, preblanking memory 2 and 3, and, for each window position (there are nm), 2(nm-1)+2 DVP operations for finding conditions(number less, mask out choices, number equal) of possible medians, 1 DVP operation for accumulating the result, and 1 DVP operation clearing memory 3 to set ready to store conditions of the next window position. It takes 2nm(nm-1) comparisons to perform the filtering using the IP6400 and 512x512x(nm-1) comparisons when using a general purpose computer.

4. Discussion

The advantage of using the IP6400 for window operations lies in the capability of processing up to one megabyte of image data in a single video frame time. Various image processing operations can be implemented through the IP6400 to achieve high speed. The sobel operator (edge detector), for example, can be implemented using two 3×3 window operations, and one frame addition, a total of 25 DVP operations as follows (see Appendix B to G):

(1) place original image in memory 0

(2) execute dwin3_3 -1 -2 -1 0 0 0 1 2 1 2 a c (detect vertical edge, result is placed in memory 1)

(3) execute dmove 1 3 (move result into memory 3)

(4) execute dwin3_3 -1 0 1 -2 0 2 -1 0 1 2 a c (detect horizontal edge, result is placed in memory 1)

(5) execute dadd 1 3 2 (add both horizontal and vertical edges and place result in memory 2)

The Laplacian operator for edge enhancement can be performed in 8 DVP operations. Tremendous gain in speed is the major significance in using the IP6400 for various image processing operations.

5. References

[1] *IP6400 Programmer's Manual,* DeAnza Systems, Inc., September, 1981.

[2] T. S. Huang, G. T. Yang, and G. Y. Yang, "A Fast Two-dimensional Median Filtering Algorithm," *IEEE Trans. Acoust., Speech, Signal Processing,* vol. ASSP. 27, pp.13-18, Feb 1979.

[3] T. S. Huang *Two-Dimensional Digital Signal Processing II* New York: Springer, 1981.

Appendix A - IP6400 registers

Appendix A - IP6400 registers

MEMORY PORT CONTROL REGISTER

Mnemonic IMEMPC, Virtual Address 117424 octal

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
NOT USED				I3	I2	I1	I0	P3	P2	P1	P0	V3	V2	V1	V0

V3–V0: Memory thru ITU to video

P3–P0: Memory thru ITU to array processor

I3–I0: Memory thru ITU to optional port

0 disables and 1 enables as described.

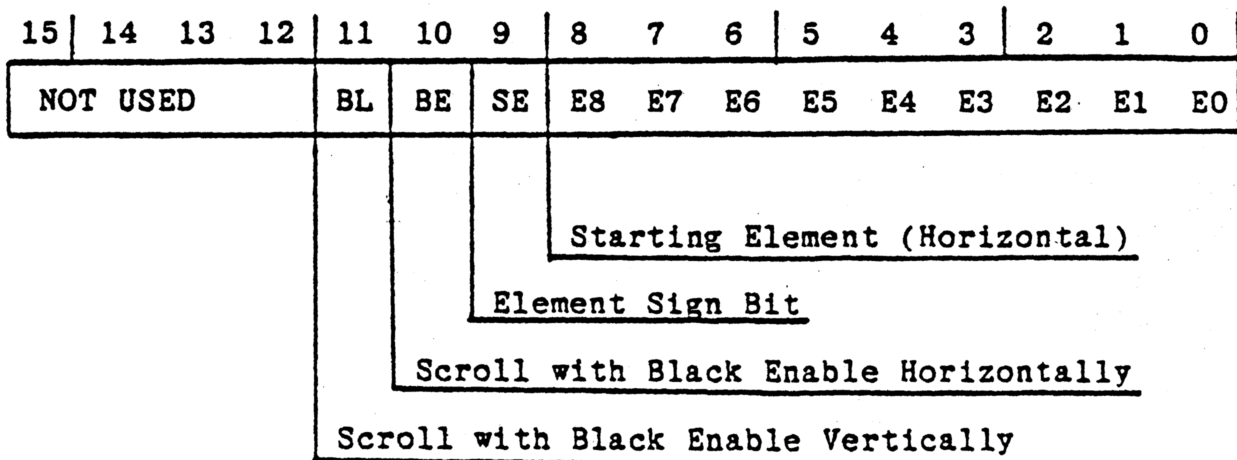
Bit	Label	Function
0-3	V0-V3	Enable data from the memory channels to pass through the ITU to video
V0		Memory Channel 0 thru ITU to video
V1		Memory Channel 1 thru ITU to video
V2		Memory Channel 2 thru ITU to video
V3		Memory Channel 3 thru ITU to video
4-7	P0-P3	Enable data from the memory channels to pass through the ITU to the Array Processor
P0		Memory Channel 0 thru ITU to Array Processor
P1		Memory Channel 1 thru ITU to Array Processor
P2		Memory Channel 2 thru ITU to Array Processor
P3		Memory Channel 3 thru ITU to Array Processor
8-11	I0-I3	Enable optional port
I0		Memory Channel 0 thru ITU to Optional Port
I1		Memory Channel 1 thru ITU to Optional Port
I2		Memory Channel 2 thru ITU to Optional Port
I3		Memory Channel 3 thru ITU to Optional Port

0 disables and 1 enables as described.

Image Scroll and Zoom Registers - 12 Bits

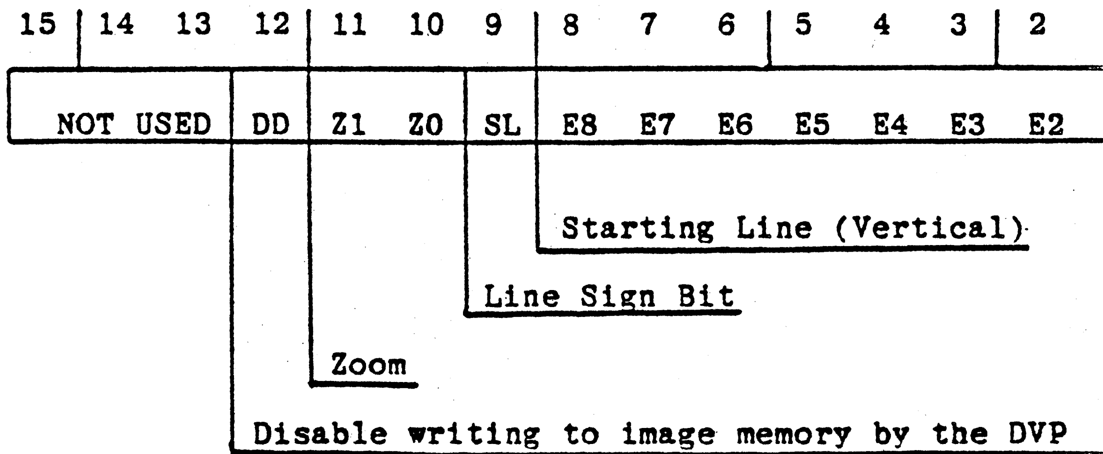
MNEMONIC	VIRTUAL ADDRESS	CHANNEL
ISCRX0, ISCRY0	117460, 117462 octal	0
ISCRX1, ISCRY1	117464, 117466 octal	1
ISCRX2, ISCRY2	117470, 117472 octal	2
ISCRX3, ISCRY3	117474, 117476 octal	3
ISCRX4, ISCRY4	117514, 117516 octal	4 overlay

X Scroll and Zoom Register



BIT	LABEL	FUNCTION
0-8	E0-E8	Specify starting element in horizontal direction
9	SE	Element sign bit 0 Data zeroed after horizontal wrap around if BE is 1 1 Data zeroed before horizontal wrap around if BE is 1
10	BE	Scroll with Black Enable Horizontally - Element 0 No data is zeroed 1 Enable data to be zeroed with horizontal wrap around
11	BL	Scroll with Black Enable Vertically - Line 0 No data zeroed 1 Enable data to be zeroed with vertical wrap around

Y Scroll and Zoom Register



BIT	LABEL	FUNCTION
0-8	L0-L8	Specify starting line in vertical directi to 0 octal is top to bottom)
9	SL	Line sign bit 0 Data zeroed after vertical wrap if BL is 1(top part of video dis black) 1 Data zeroed before vertical wrap if BL is 1 (bottom part of video d is black)
10-11	Z0-Z1	Zoom control bits Z1 Z0 0 0 1 to 1 (No zoom) 0 1 2 to 1 Zoom 1 0 4 to 1 Zoom 1 1 8 to 1 Zoom
12	DD	Disable DVP writing to image memory channel 0- Enable writing to memory channel by DVP 1- Disable

DESTINATION REGISTERS

Mnemonic	Virtual Address	Function
IDESTX	117420	Destination X - element
IDESTY	117422	Destination Y - line

Destination X Register

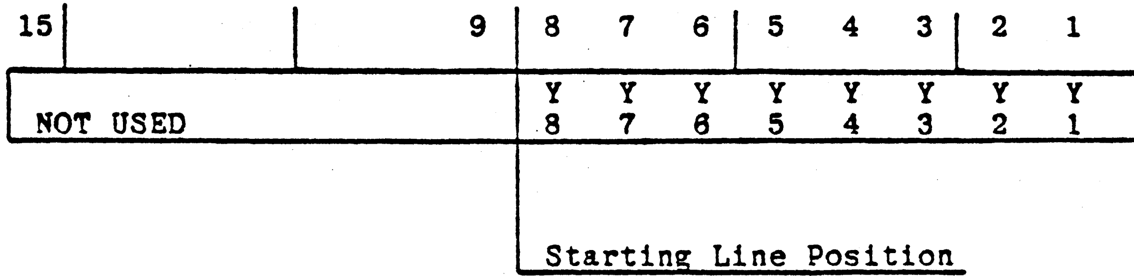
15–12	11	10	9	8	7	6	5	4	3	2	1	0
NOT USED	S 1	S 0		X 8	X 7	X 6	X 5	X 4	NOT USED			

Starting Element Position (bits 8–4)

Shift Code for output from digitizer A/D converter (bits 11–10)

Bit	Label	Function	Bits zero on input the array process 6 bit Digitizer	8 bit Digitize
4-8	S4-S8	Starting Element Position in image memory for data from the DVP. Must be multiples of 16.		
10,11	S1 S0	Direction and number of bits to shift the output of the digitizers A/D converter before input to the array processor.		
	0 0	Normal - no shift. Accept output from the A/D converter.	2 MSB's	None
	0 1	Shift right 1 bit - divide A/D converter output by 2	3 MSB's	MSB
	1 0	Shift left 1 bit - multiply A/D Converter output by 2	MSB, LSB	LSB
	1 1	Shift left 2 bits - multiply A/D converter output by 4	2 LSBs	2 LSB

Destination Y Register



Bit	Label	Function
0-8	Y0-Y8	Starting Line Position in image memory for d; from DVP.

DVP Bit Plane Mask Registers

Mnemonic	Virtual Address	Function
IBPMA	117440	Bit plane mask A
IBPMB	117442	Bit plane mask B

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
M 15	M 14	M 13	M 12	M 11	M 10	M 9	M 8	M 7	M 6	M 5	M 4	M 3	M 2	M 1	M 0
Bit plane mask bits								Bit plane mask bits							

0 - Do not write to bit plane (Disable)

1 - Write to bit plane (Enable)

Bit Plane Mask

Bits	Label	Register	Memory channel
0-7	M0-M7	A	0
8-15	M8-M15	A	1
0-7	M0-M7	B	2
8-15	M8-M15	B	3

Constants Register

MNEMONIC - ICONST

VIRTUAL ADDRESS 117444

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
K7	K6	K5	K4	K3	K2	K1	K0	C7	C6	C5	C4	C3	C2	C1	C0
K1 Constant - Operation ALU								K2 Constant - Test ALU							

Bit	Label	Function
0-7	C0-C7	K2 Constant - alternate input B of Test ALU
8-15	K0-K7	K1 Constant - alternate input D of Operational

Constant value range is 0 to 377 octal.

Arithmetic is performed as 1's complement.

All values are positive and considered as intensities.

DVP Input Data Path

MNEMONIC - INPDP

VIRTUAL ADDRESS 117446 OCTAL

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
T0	AX	A1	A0	DI	K1	B1	B0	Z0	CI	C1	C0	TS	K2	D1	D0

A1, A0: A input memory channel select

AX: Auxiliary input

T0: Test ALU output

A Input Control

Operational ALU

B1, B0: B input memory channel select

K1: MS Constant

DI: Select D input

B Input Control

C1, C0: C input memory channel select

CI: Camera A/D

Z0: Force LS to 0

C Input Control

Test ALU

D1, D0: D input memory channel select

K2: LS Constant

D input control

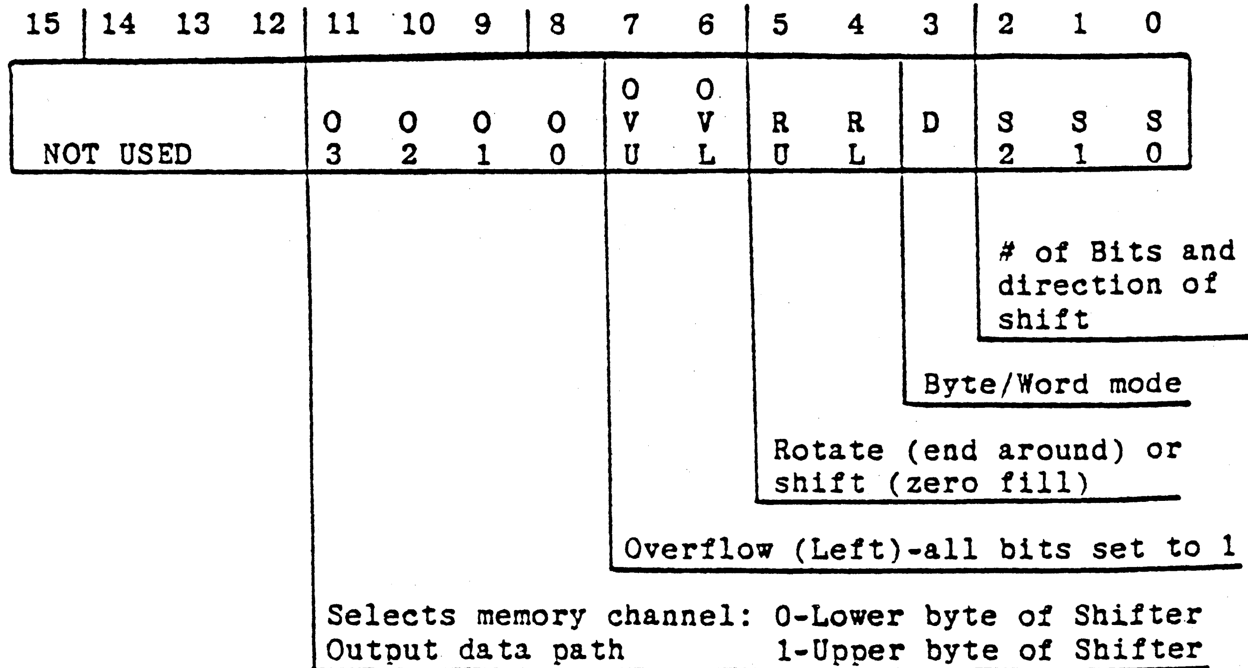
TS: 0-Test ALU to shift
1-B input to shifte

INPUT Name	ALU	OUTPUT Name
A	Operational ALU	MS Byte of SHIFTER
B	Operational ALU	MS Byte of SHIFTER
C	Test ALU	LS Byte of SHIFTER
D	Test ALU	LS Byte of SHIFTER

Output Data Paths and Shift Control Register

MNEMONIC - IODPSH

VIRTUAL ADDRESS 117454 OCTAL



Shift control bits

Upper Byte - MS byte, output of Operational ALU
Lower Byte - LS byte, output of Tet ALU (or B1 selector)

Bits	Label	Function
0-2	S0-S2	Shift Code - specifies the direction and number of shifts

S2	S1	S0	Action
0	0	0	None
0	0	1	Left 1
0	1	0	Left 2
0	1	1	Left 3
1	0	0	Right 4
1	0	1	Right 3
1	1	0	Right 2

Test ALU Operational Code Register

MNEMONIC - ITSTOP

VIRTUAL ADDRESS 117450 OCTAL

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
C3	C2	C1	C0	FI	F1	WE1	WE0	RCC	REQ	M	CC	S3	S2	S1	S0

Condition code for counter to be incremented

Force S0-S3 to 0 if LSB of C input is 0

If CC is set (=1) Force Bit 0 of data in shifter to 0

Write enable region: anywhere, cursor defined, or overlay defined

Select one of 16 operations

Complement of Carry 0-Carry, 1-No Carry

Select Arithmetic or Logical Operation

Determine one of 48 operations to perform

Inputs A = B condition code

Complement carry condition code

2 bit condition code field for OP ALU code.

Bit	Label	Function
0-3	S0-S3	Test ALU operational codes. See Table 1.2.11.F.
4	CC	Complement of Carry 0 - Carry enabled 1 - Carry disabled

Test Operational Code Table

ARITHMETIC FUNCTIONS

INPUTS	M	CC	S3	S2	S1	S0	Output Based on Complement Carr CC = 0	CC =
C, D	0	X	0	0	0	0	C + 1	C
C, D	0	X	0	0	0	1	(C or D) + 1	(C or D)
C, D	0	X	0	0	1	0	(C or $\overline{D}$) + 1	(C or $\overline{D}$)
C, D	0	X	0	0	1	1	0	377 octal
C, D	0	X	0	1	0	0	C + (C and $\overline{D}$) + 1	C + (C and $\overline{D}$)
C, D	0	X	0	1	0	1	(C or D)+(C and $\overline{D}$) + 1	(C or D)+(C an
C, D	0	X	0	1	1	0	C - D	C - D - 1
C, D	0	X	0	1	1	1	C and $\overline{D}$	C and $\overline{D}$ -1
C, D	0	X	1	0	0	0	C + (C and D) + 1	C + (C and D)
C, D	0	X	1	0	0	1	C + D + 1	C + D
C, D	0	X	1	0	1	0	(C or $\overline{D}$)+(C and D)+ 1	(C or $\overline{D}$)+(C anc
C, D	0	X	1	0	1	1	(C and D)	(C and D) -1
C, D	0	X	1	1	0	0	C + C + 1	C + C
C, D	0	X	1	1	0	1	(C or D) + C + 1	(C or D) + c
C, D	0	X	1	1	1	0	(C or $\overline{D}$) + C + 1	(C or $\overline{D}$) + C
C, D	0	X	1	1	1	1	C	C - 1

LOGICAL FUNCTIONS

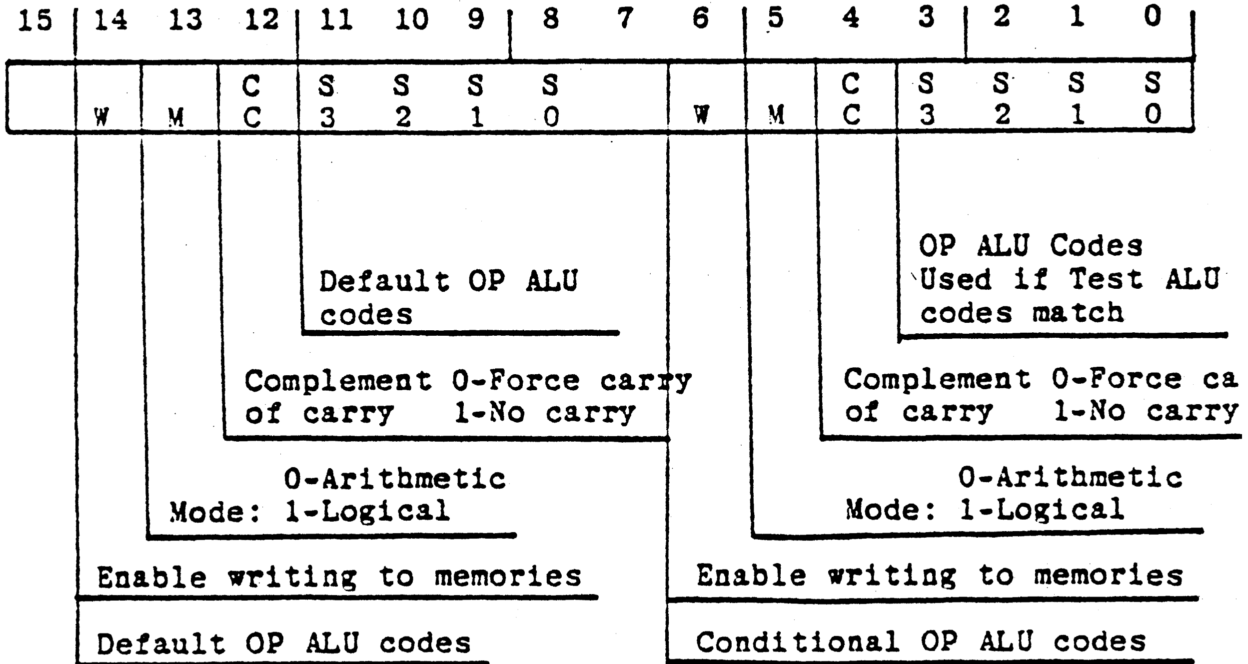
INPUTS	M	CC	S3	S2	S1	S0	Output Based on Complement Carry (CC = DON'T CARE)
C, D	1	X	0	0	0	0	$\overline{C}$
C, D	1	X	0	0	0	1	$\overline{C \text{ or } D}$
C, D	1	X	0	0	1	0	$\overline{C}$ and D
C, D	1	X	0	0	1	1	0
C, D	1	X	0	1	0	0	$\overline{C \text{ and } D}$
C, D	1	X	0	1	0	1	$\overline{D}$
C, D	1	X	0	1	1	0	C exclusive or D
C, D	1	X	0	1	1	1	C and $\overline{D}$
C, D	1	X	1	0	0	0	$\overline{C}$ or D
C, D	1	X	1	0	0	1	$\overline{C \text{ exclusive or } D}$
C, D	1	X	1	0	1.	0	D
C, D	1	X	1	0	1	1	C and D
C, D	1	X	1	1	0	0	377 octal, all bits set
C, D	1	X	1	1	0	1	C or $\overline{D}$
C, D	1	X	1	1	1	0	C or D
C, D	1	X	1	1	1	1	C

Test Operational Code Table

Operational Code Register

MNEMONIC - IOPOP

VIRTUAL ADDRESS 117452



Bits	Label	Function
0-3,8-11		S0-S3 Operational ALU operation codes
4,12	CC	Complement of Force Carry 0 - Force Carry 1 - No Carry
5,13	W	Enable writing to memory channels 0 - Disable writing to memories 1 - Enable writing to memories
0-7		Conditional Operational ALU op code
8-15		Default Operational ALU op code

Operational Code Table

ARITHMETIC FUNCTIONS

INPUTS	M	CC	S3	S2	S1	S0	Output Based on Complement Carry: CC = 0	CC = 1
A, B	0	X	0	0	0	0	A + 1	A
A, B	0	X	0	0	0	1	(A or B) + 1	(A or B)
A, B	0	X	0	0	1	0	(A or $\overline{B}$) + 1	(A or $\overline{B}$)
A, B	0	X	0	0	1	1	0	377 octal
A, B	0	X	0	1	0	0	A + (A and $\overline{B}$) + 1	A + (A and $\overline{B}$)
A, B	0	X	0	1	0	1	(A or B)+(A and $\overline{B}$) + 1	(A or B)+(A and $\overline{B}$
A, B	0	X	0	1	1	0	A - B	A - B - 1
A, B	0	X	0	1	1	1	A and $\overline{B}$	A and $\overline{B}$ -1
A, B	0	X	1	0	0	0	A + (A and B) + 1	A + (A and B)
A, B	0	X	1	0	0	1	A + B + 1	A + B
A, B	0	X	1	0	1	0	(A or $\overline{B}$)+(A and B)+ 1	(A or $\overline{B}$)+(A and B
A, B	0	X	1	0	1	1	(A and B)	(A and B) -1
A, B	0	X	1	1	0	0	A + A + 1	A + A
A, B	0	X	1	1	0	1	(A or B) + A + 1	(A or B) + A
A, B	0	X	1	1	1	0	(A or $\overline{B}$) + A + 1	(A or $\overline{B}$) + A
A, B	0	X	1	1	1	1	A	A - 1

LOGICAL FUNCTIONS

INPUTS	M	CC	S3	S2	S1	S0	Output Based on Complement Carry (CC = DON'T CARE)
A, B	1	X	0	0	0	0	$\overline{A}$
A, B	1	X	0	0	0	1	$\overline{\text{A or B}}$
A, B	1	X	0	0	1	0	$\overline{A}$ and B
A, B	1	X	0	0	1	1	0
A, B	1	X	0	1	0	0	$\overline{\text{A and B}}$
A, B	1	X	0	1	0	1	$\overline{B}$
A, B	1	X	0	1	1	0	A exclusive or B
A, B	1	X	0	1	1	1	A and $\overline{B}$
A, B	1	X	1	0	0	0	$\overline{A}$ or B
A, B	1	X	1	0	0	1	$\overline{\text{A exclusive or B}}$
A, B	1	X	1	0	1	0	B
A, B	1	X	1	0	1	1	A and B
A, B	1	X	1	1	0	0	377 octal, all bits set
A, B	1	X	1	1	0	1	A or $\overline{B}$
A, B	1	X	1	1	1	0	A or B
A, B	1	X	1	1	1	1	A

Appendix B - 3×3 linear window operation

NAME

dwin3_3 - 3×3 window operation

SYNOPSIS

(1) dwin3_3 [w1] [w2] [w3] [w4] [w5] [w6] [w7] [w8] [w9] [s] [a] [c]
(2) dwin3_3 [f] datafile

DESCRIPTION

Three memory planes are used for this operation. The image to be processed should be in memory plane 0. The result is located in memory plane 1. Memory plane 2 is used for the overflow indicators.

The parameters include the nine window coefficients, a scaling factor, underflow complement option, and memory 1 & 2 preblank option described as follows:

w1 to w9 — window coefficients arranged in the order of w1 to w3 as first row of the window, w4 to w6 as second row of the window, w7 to w9 as third row of the window. All coefficients must be numerical values.

s — scaling factor. It is a numerical value, usually selected to prevent overflow of the lookup table (ITT) of DeAnza. As a rule of thumb, when absolute values of some coefficients are larger than one, the absolute value of the largest coefficient should be the scaling factor. This scaling factor scales down the coefficients for arithmetic manipulation in DVP of DeAnza, and then scales up the result. Hence, the scaling factor does not affect the coefficients on the whole window operation. Yet, it is necessary to prevent overflow of the ITT of DeAnza for arithmetic manipulation.

a — This option complements, instead of zeroes underflow. Default would zero underflow.

c — This option preblanks memory 1 and memory 2. Default would not preblank the memory planes.

Parameters can be input from a datafile by typing f before the datafile.

AUTHOR

N. Ansari (7/7/83)

FILES

/dev/deanza
/usr/include/sys/iz.h

SEE ALSO

IP 6400 Programmer's Manual, DeAnza Syste

Appendix C - n$\times$m linear window operation

NAME

dvwin - n×n window operation

SYNOPSIS

(1) dvwin [n] [m] [w[1]] ... [w[nm]] [s] [a] [c]
(2) dvwin [f] datafile

DESCRIPTION

Three memory planes are used for this operation. The image to be processed should be in memory plane 0. The result is located in memory plane 1. Memory plane 2 is used for the overflow indicators.

The parameters include the size of the window, the window coefficients, a scaling factor, underflow complement option, and memory 1 & 2 preblank option described as follows:

n — width, number of rows of the window.

m — length, number of columns of the window.

w[1] to w[nm] — window coefficients arranged in the order of row by row basis. All coefficients must be numerical values.

s — scaling factor. It is a numerical value, usually selected to prevent overflow of the lookup table (ITT) of DeAnza. As a rule of thumb, when absolute values of some coefficients are larger than one, the absolute value of the largest coefficient should be the scaling factor. This scaling factor scales down the coefficients for arithmetic manipulation in DVP of DeAnza, and then scales up the result. Hence, the scaling factor does not affect the coefficients on the whole window operation. Yet, it is necessary to prevent overflow of the ITT of DeAnza for arithmetic manipulation.

a — This option complements, instead of zeroes underflow. Default would zero underflow.

c — This option preblanks memory 1 and memory 2. Default would not preblank the memory planes.

Parameters can be input from a datafile by typing f before the datafile.

AUTHOR

N. Ansari (7/14/83)

FILES

/dev/deanza
/usr/include/sys/iz.h

SEE ALSO

IP 6400 Programmer's Manual, DeAnza Systems, Inc.

Appendix D - $n \times m$ median filtering

NAME

dvmedian - median filtering

SYNOPSIS

dvmedian [n] [m]

DESCRIPTION

dvmedian performs two-dimensional median filtering using the DeAnza IP6400. The default window size is 3×3. The maximum window size is 25×25. Four memory planes are used for this operation. The image to be processed should be in memory plane 0. The result is located in memory plane 2. Memory planes 1 and 3 are used for intermediate processes.

The only parameters are to specify window size.

n width, number of rows of the window.

m length, number of columns of the window.

AUTHOR

N. Ansari (7/23/83)

FILES

/dev/deanza
/usr/include/sys/iz.h

SEE ALSO

IP 6400 Programmer's Manual, DeAnza Systems, Inc.

Appendix E - denhance

NAME

denhance - deblur a picture

SYNOPSIS

denhance

DESCRIPTION

denhance is a deanza version unsharp masking. It uses a Laplacian operator to deblur a picture. It is a 3x3 window operation with a weight of 5 at the center, -1 at the nearest neighbors, and a 0 at other neighbors. The picture to be processed should be placed in memory plane 0, and the result is placed in memory plane 1.

DIAGNOSTICS

None

AUTHOR

N. Ansari (7/7/83)

FILES

/dev/deanza /usr/include/sys/iz.h

SEE ALSO

IP 6400 Programmer's Manual, DeAnza Systems, Inc.

Appendix F - dadd

NAME

dadd - adds two images from Deanza.

SYNOPSIS

dadd chan1 chan2 chan3

DESCRIPTION

dadd will add the images stored in chan1 and chan2 together,and stores the result in chan3.

chan1, chan2, chan3 can be any of the four channels: 0,1,2 or 3.

DIAGNOSTICS

Bad flags are ignored.

AUTHOR

Nancy Cam (3/08/83)

FILES

/dev/deanza

BUGS

No known bugs.

Appendix G - dmove

NAME

dmove

SYNOPSIS

dmove [chanX] [chanY]

DESCRIPTION

This routine allows the user to move an image from chanX to chanY, where chanX and chanY can be from 0 to 3.

DIAGNOSTICS

Synopsis will be given upon detecting illegal parameters.

AUTHOR

N. Ansari (7/7/83)

FILES

/dev/deanza /usr/include/sys/iz.h

SEE ALSO

IP 6400 Programmer's Manual, DeAnza Systems, Inc.

UNIVERSITY OF MICHIGAN
3 9015 02519 6257