

Graphics and imaging are two-dimensional applications. Just as in the one-dimensional case, it is often desirable to manipulate the data through filtering. This chapter describes the implementation of two-dimensional filtering using 3×3 kernels, as well as median filtering. Histogram equalization, a technique for enhancing images, is also described.

9.1 TWO-DIMENSIONAL CONVOLUTION

According to digital signal processing theory, a linear time-invariant system (LTI system) is completely characterized by the impulse response function, $h(n)$, which is the system's response to the unit sample sequence $\delta(n)$. From this principle, the response $y(n)$ of an LTI system is the *convolution sum* of an input to the system $x(n)$ with the impulse response, $h(n)$. The following equation defines convolution in one dimension:

$$y(n) = \sum_{k=0}^n x(k) h(n - k)$$

A two dimensional LTI system can again be described by it's impulse response $h(n_1, n_2)$. The corresponding two-dimensional convolution sum can be describe by the following equation:

$$y(n_1, n_2) = h(n_1, n_2) ** x(n_1, n_2)$$

where

$x(n_1, n_2)$ describes the two-dimensional input
 $y(n_1, n_2)$ describes the two-dimensional output

Note: The “**” symbol signifies a two-dimensional convolution.

This equation expands to the following equation:

$$y(n_1, n_2) = \sum_{k_1=-\infty}^{+\infty} \sum_{k_2=-\infty}^{+\infty} x(k_1, k_2) h(n_1 - k_1, n_2 - k_2)$$

9 Image Processing

A matrix is a natural data structure for storing two-dimensional data. For the two dimensional convolution, the data input values (x-values) are stored in the data matrix (in data memory), and the impulse response values, h-values, are stored in the transfer function matrix (in program memory).

9.1.1 Implementation

FIR filters are used in the two-dimensional signal space just as they are in the one-dimension signal space. For two-dimensional signals kernal convolution performs FIR filtering on an image matrix. The kernal convolution implements the following equation:

$$y(n_1, n_2) = \sum_{k_1=-\infty}^{+\infty} \sum_{k_2=-\infty}^{+\infty} x(k_1, k_2) h(n_1 - k, n_2 - k_2)$$

where

$y(n_1, n_2)$ is the output filtered image
 $h(k_1, k_2)$ is the filter kernal
 $x(n_1, n_2)$ is the input image

The kernel convolves a 3×3 coefficient matrix by an $M \times N$ image matrix. This code segment can use any size data buffer; the `#DEFINE` statements at the start of the program determine the size of the input data matrix. The input data (in data memory) is assumed to be in row major format. The kernel (convolution coefficients) are loaded from the file `coeff.dat` into program memory.

Image Processing 9

Figure 9.1 shows A graphical depiction of the 3×3 kernel convolution.

Data Array:

0,0	0,1	0,2	0,3	0,4	. . .	0,M-1
1,0	1,1	1,2	1,3	1,4
2,0	2,1	2,2	2,3	2,4
~		~
~		~
		
						N-2,M-1
N-1,0		. . .			N-1,M-2	N-1,M-1

Figure 9.1 3x3 Convolution Matrix

The data array elements in the upper left are referred to as $\{d_{00}, d_{01} \dots\}$.
 The convolution kernel elements are referred to as $\{c_{00}, c_{01}, \dots, c_{32}, c_{33}\}$.
 The first two convolutions are formed from these products:

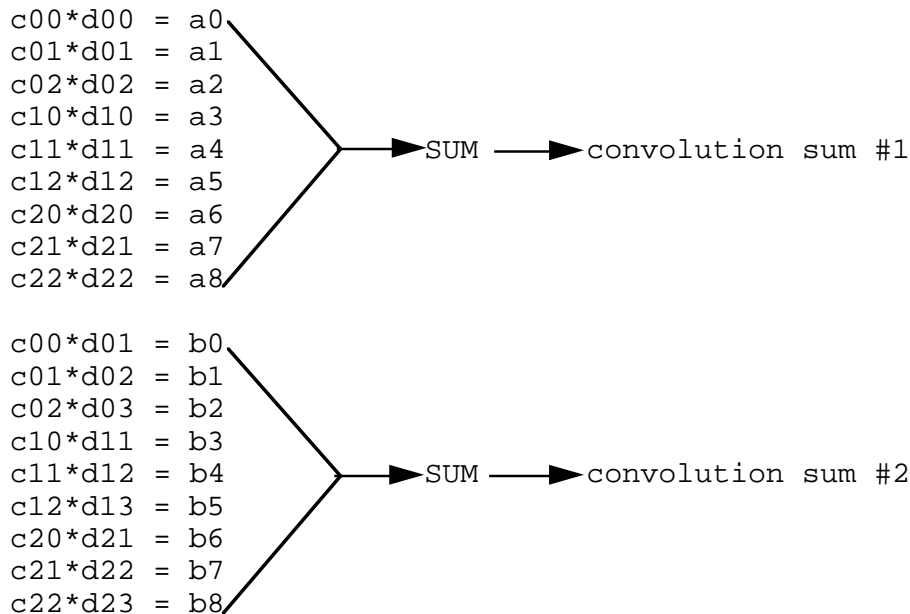


Figure 9.2 3x3 Convolution Operation

9 Image Processing

Figure 9.2 shows that for every convolution sum there are

- nine reads from data memory (input data)
- nine reads from program memory (kernel coefficients)
- eight additions
- one write to program memory for the convolution sum

The inner loop of the algorithm performs these 27 operations.

The program executes the outer loop $m-2$ times because the size the data matrix is 3×3 the data starts at the upper left hand corner of the matrix. At the $m-2$ th iteration through the outer loop, the 3×3 kernel matrix covers the data values through m ; further loop iterations beyond $m-2$ times are redundant. Therefore, the outer loop of the program determines the starting point of the convolution and manages the update of the array indices from row to row. The advantages of the automatic post modify DAGs are obvious; one modify statement and one read statement can manage the indices to process the data array.

The inner loop of the algorithm performs all of the calculations. While the outer loop determines the starting point of the convolution and manages the update of the array indices from row to row, the inner loop of the algorithm performs the calculations. The third kernel coefficient is stored in F5 in the `setup` segment of the code so as to avoid an extra cycle for program memory read within the loop. (The ADSP-210xx multifunction operations allow a data memory read/write and a program memory read/write in one cycle, and the algorithm requires 18 reads and one write.)

The calculations start at the upper left hand corner of the image matrix. The first iteration of the kernel matrix by the image matrix performs the operations over the first, second, and third rows. The second iteration covers the second, third, and fourth rows, etc. Therefore, at the $M-2$ iteration, the $M-1$ and M rows have already been dealt with, and, in the interest of time and space, we set the outer loop equal to $M-2$.

The write operation to the PM location that contains the convolution sum is at the second instruction in the inner loop, which may seem like an unusual place for it. To minimize the cycle count, this write of the partial sum cannot occur at the bottom of the loop. F12 is used to hold the partial products, and F8 is used to hold the running total of the partial sums.

Image Processing 9

The first time through the loop, F8 (which only contains one product) is written to program memory at the relative bottom of the circular buffer output (where the base register is currently pointing). When the circular buffer is written to again, with the valid sum of products, the memory is correctly written at the relative top of the circular buffer.

9.1.2 Code Listing

```
/
*****

File Name
    comv3x3.asm
Version

Purpose
    This code performs the Kernel Convolution.

Equations Implemented
    y(n1, n2) = SUM (SUM ( x(k1, k2) h(n1 - k, n2 - k2))

Calling Parameters
    none
Return Values

Registers Affected

Cycle Count
    Setup: This code initializes register constants and address pointers needed.
           cycles=10+1
           time=11*50ns=550ns

    Benchmark: This code performs the Kernel Convolution.
               cycles=(9*(n-2)+3)*(m-2)+5+11      (note: 11 cache misses)
               time (at m=780, n=1024, cycletime=50ns) =7158394*50ns=.3579197s
               MFLOPS sustained in core loop=20MIPS(9mpy+8add)/9=37.77

# PM Locations
    27 words of instructions + 9 words of PM data
# DM Locations
    m * n + (m -2) * (n -2) where m = rows and n = collumns

*****/
```

(listing continues on next page)

9 Image Processing

/*The third coefficient is stored in the register file to free up a cycle to output a result. The first write to the output buffer is unused, the pointer then wraps around to the proper location at the start of output.*/

```
#DEFINE    m 780          /*m by n image*/
#DEFINE    n 1024

.SEGMENT/DMDATA dmsegment;
.VAR
.VAR
.ENDSEG;

.SEGMENT/PMDATA pmdataseg;
.VAR
.ENDSEG;

.SEGMENT/PMCODE pmcodeseg;
setup:      M0=1;
            M1=n-2;
            M2=-(2*n+1);
            M3=2;
            M8=1;
            M9=2;
            B0=input;          L0=0;
            B8=kernal;        L8=@kernal;
            B9=output+(m-2)*(n-2);  L9=@output;
            F5=PM(kernal+2);    /*store in register file to save cycle*/

kern_conv:  F0=DM(I0,M0), F4=PM(I8,M8);
            LCNTR=m-2, DO in_row UNTIL LCE;
            LCNTR=n-2, DO in_col UNTIL LCE;
            F8=F0*F4, F8=F8+F12, F0=DM(I0,M0), F4=PM(I8,M9);
            F12=F0*F4,          F0=DM(I0,M1), PM(I9,M8)=F8;
            F12=F0*F5, F8=F8+F12, F0=DM(I0,M0), F4=PM(I8,M8);
            F12=F0*F4, F8=F8+F12, F0=DM(I0,M0), F4=PM(I8,M8);
            F12=F0*F4, F8=F8+F12, F0=DM(I0,M1), F4=PM(I8,M8);
            F12=F0*F4, F8=F8+F12, F0=DM(I0,M0), F4=PM(I8,M8);
            F12=F0*F4, F8=F8+F12, F0=DM(I0,M0), F4=PM(I8,M8);
            F12=F0*F4, F8=F8+F12, F0=DM(I0,M2), F4=PM(I8,M8);
in_col:    F12=F0*F4, F8=F8+F12, F0=DM(I0,M0), F4=PM(I8,M8);
            MODIFY(I0,M0);
in_row:    F0=DM(I0,M0);
            RTS (D), F8=F8+F12;
            PM(I9,M8)=F8;
            NOP;
.ENDSEG;
```

Listing 9.1 CONV3x3.ASM

9.2 MEDIAN FILTERING (3X3)

Like any other data transmitted through a channel, images are subject to noise corruption. One type of noise is *shot* or *impulse noise*—a strong spike-like noise scattered evenly through the image. *Median filtering* can remove shot noise and is particularly useful when the image sharpness must be preserved. It replaces a given data value with the median value of its neighboring elements. The median is the value m such that half the values in an array are less than m and half the values are greater than m .

The program presented here implements a 3×3 median filter (a nine-element array) which is applied to every pixel in the image and replaces that pixel with the median value.

In any six element subset of nine data values, at least one of the values will be larger than the median (i.e., at least one value will have a rank of sixth largest). This algorithm of median filtering considers the first six values in the order that they appear in memory—no presorting is required. The highest and lowest values are discarded and a new data value is read. Performing this compare iteratively on successively smaller groups yields a median of three values ranked in the middle of the array—the median is the “middle” value of these three.

For further information see [GLASSNER90], p.171, 711; [GONZALEZ87], p. 162-163; [JAIN89], p. 246-247.

9.2.1 Implementation

The median filter algorithm has a simple implementation in ADSP-21000 family assembly language. Most of the program consists of a few instructions repeated over and over again—the use of macros leads to code that is easy to read and visualize.

The `comp` (compare) operator is central to the macro invocation: it sets the appropriate flag in the arithmetic status (ASTAT) register depending on the relative value of the operands. This conditional operation is always valid because the ADSP-210xx updates the ASTAT register after every operation. The AZ flag is set (ALU result is 0) if the operands of `comp` are equal, and the AN flag is set (ALU result is negative) if the first operator is smaller than the second. The state of the ASTAT, if neither AZ or AN is set, is positive; operands a and b will be swapped if and only if a is greater than b .

9 Image Processing

The ADSP-210xx has a sufficient number of registers to perform the memory read and write operations and the comparisons and swaps that follow them. If registers were not available, it would not be possible to read and compare using the same register; an intermediate storage area in memory and overhead cycles used for memory management would be needed.

9.2.2 Code Listing

```
/
*****

File Name
    med3x3.asm

Version
    1.0

Purpose
    Median filtering: replace a given data value with the median value of its
    neighboring elements.

Equations Implemented
    Y=SIN(X) or
    Y=COS(X)

Calling Parameters
    i0          index to data values
    m0          column offset (usually 1)
    m1          row offset (usually [#pixels/row - 2])
    m2          offset to next 3x3 block (usually -[2*m1 + 1])

Return Values
    r0          median value

Registers Affected
    r0-r6      tmp values

Cycle Count

    56 cycles
    2.24µs per 3x3
    587ms for median filtering a 512x512 imag

# PM Locations
    16 words of instruction
# DM Locations
    9 words
```


Image Processing 9

```
*****/

#define s2(a,b)      comp(a,b); if gt b = pass a, a = b;
                        /*one macrdefinition*/
                        /*that gets repeated*/

#define mnmx3(a,b,c)  s2(b,c); s2(a,c); s2(a,b)

#define mnmx4(a,b,c,d) s2(a,b); s2(c,d); s2(a,c); s2(b,d)

#define mnmx5(a,b,c,d,e) s2(a,b); s2(c,d); s2(a,c); s2(a,e); \
                        s2(d,e); s2(b,e)

#define mnmx6(a,b,c,d,e,f) s2(a,d); s2(b,e); s2(c,f); \
                        s2(a,b); s2(a,c); s2(e,f); s2(d,f)

.SEGMENT /pm pm_code;
.GLOBAL med3x3;

med3x3:  r1=dm(i0,m0);
        r2=dm(i0,m0);
        r3=dm(i0,m1);
        r4=dm(i0,m0);
        r5=dm(i0,m0);
        r6=dm(i0,m1);

strt:
        mnmx6(r1, r2, r3, r4, r5, r6);

mm6:
        r1 = dm(i0,m0);

        mnmx5(r1, r2, r3, r4, r5);

        /*
        /*smallest and greatest
        /*values have dropped out*/

mm5:
        r1 = dm(i0,m0);

        mnmx4(r1, r2, r3, r4);

mm4:
        r1 = dm(i0,m2);

        mnmx3(r1, r2, r3);

mm3:
        rts (db);

        r0 = r2;
        nop;

.ENDSEG;
```

Listing 9.2 med3x3.asm

9 Image Processing

9.3 HISTOGRAM EQUALIZATION

A histogram tallies the intensities of all 8-bit pixels of an image into the 256 bins of a histogram array. If the pixel resolution is increased to 10-bits or 12-bits, a larger histogram array is required (1024 or 4096 bins, respectively). An intensity of zero (0x00) is the darkest pixel and is tallied in bin 0, the first bin. An intensity of 255 (0xFF) is the brightest pixel and is stored in bin 255, the last bin. The ADSP-210xx supports a 32-bit integer data type, and thus can tally large numbers of pixels without overflowing the histogram array.

After all pixel intensities have been tallied, the histogram array can be analyzed to determine the darkness or brightness of the image. If the image is too dark, most of the pixels will record in the first 128 bins (Figure 9.3). If the image is too bright, most of the pixels will record in the second 128 bins (Figure 9.4).

The process of *histogram equalization* enhances the contrast of the image by applying a gain and offset to each pixel, which produces an image with pixels that cover all intensities.

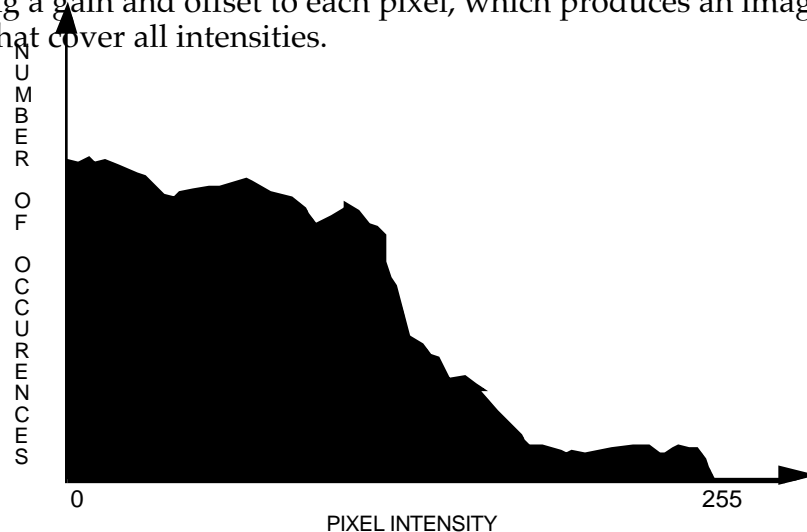


Figure 9.3 Histogram Of Dark Picture

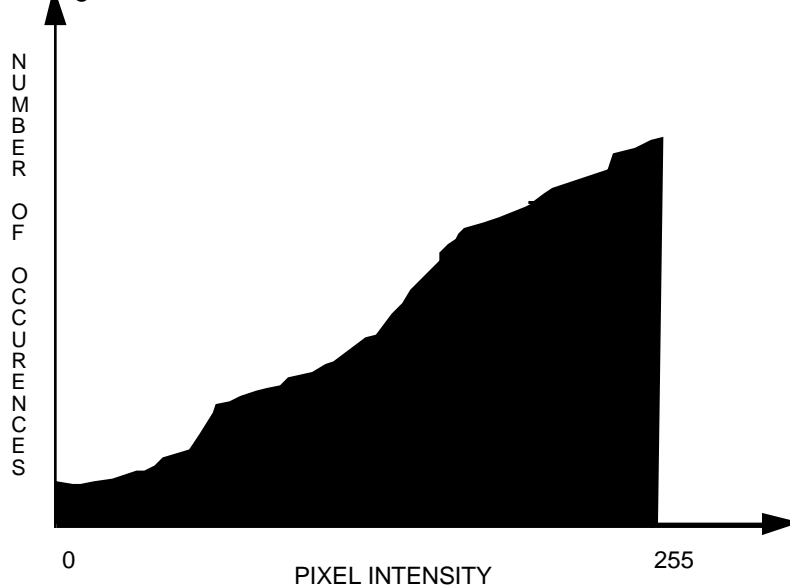


Figure 9.4 Histogram Of Bright Picture

9.3.1 Implementation

The following sequence of tasks generalize the calculation of the histogram array:

1. Read next pixel from memory.
2. Add the pixel value to the base address of the histogram array.
3. Use this address value as a pointer to read the current number stored in the corresponding bin in the temporary histogram array.
4. Increment that bin value.
5. Write the bin value back to the temporary histogram.

The `histo.asm` program (Listing 9.3) is an optimized implementation of the pseudo-code listed above. The core histogram loop, `hloop`, processes two pixels per iteration. This technique yields an average calculation rate of 3.5 instruction cycles per pixel. Due to register pipelining, the two pixels processed in the loop must be tallied in two different histogram arrays. After all pixels are processed, these two temporary histograms in program memory are added together to produce a composite histogram in data memory. The subroutine declares the two

9 Image Processing

temporary and the final composite histogram arrays.

9.3.2 Code Listing

```
/
*****

File Name
    histo.asm

Version

Purpose
    An image histogram is an array summarizing the number of occurrences of each
    grey level in an image. If a pixel in a typical monochrome image can take on
    any of 256 distinct values, the histogram would need 256 bins. In each bin
    the number of occurrences of this grey level is stored.

    The algorithm used here assumes the monochrome image is stored in a buffer in
    Data Memory, and the histogram is formed in Data Memory.

Equations Implemented
    None

Calling Parameters
    b0,i0    = data buffer start
    l0       = data buffer length
    m0       = 1
    m15      = 0

    Note: l1 = l8 = l9 = N = # of bins. N must be even, positive, and >= 4

Return Values
    histogram output bins pointed to by il

Registers Affected
    r0        input data 1
    r1        bin value 1
    r2        input data 2
    r3        bin value 2, tmp register
    r14       temp bin buffer #1 start
    r15       temp bin buffer #2 start

Cycle Count
    3.5N + 2B + 30 cycles
        where N = number of data values
        B = number of bins

# PM Locations
    32 words instructions + 512 words data
# DM Locations
    N * 256, where N = number of data values
```

Image Processing 9

```
*****/

* declare two temporary histogram buffers in program memory */

SEGMENT /pm pm_data;
VAR  temp1[256];
VAR  temp2[256];
ENDSEG;

* declare histogram result buffer in data memory */

SEGMENT /dm dm_data;
VAR  histogram[256];
GLOBAL histogram;
ENDSEG;

SEGMENT /pm pm_code;
GLOBAL histo;

* Initialize data addresses, loop count */

histo:    b8 = temp1; l8=@temp1; r14=i8;
          b9 = temp2; l9=@temp2;
          r3 = l1;          /* r3 = N = # of bins */
          r3 = lshift r3 by -1; /* r3 = N/2 */
          r3 = r3 - 1, r15=i9; /* r3 = N/2-1, initialize r15 */

* Do the histogram into 2 temporary bins in PM */

          r0=dm(i0,m0);
          r0=r0+r14, r2=dm(i0,m0);
          lcntr = r3, do hloop until lce;
              r2=r2+r15, i8=r0;
              i9=r2;
              r1=pm(i8,m15); /* 2 cycles due to I register load latency */
              r1=r1+1, r3=pm(i9,m15);
              r3=r3+1, r0=dm(i0,m0), pm(i8,m15)=r1;
hloop:    r0=r0+r14, r2=dm(i0,m0), pm(i9,m15)=r3;

          r2=r2+r15, i8=r0;
          i9=r2;
          r1=pm(i8,m15);
          r1=r1+1, r3=pm(i9,m15);
          r3=r3+1, pm(i8,m15)=r1;
          pm(i9,m15)=r3;

* Now combine the bins back into DM */

          b1=histogram; l1=@histogram;
          i8=b8;
          i9=b9;
          r2=l1;
          r2=r2-1, r0=pm(i8,m8);
```

(listing continues on next page)

9 Image Processing

```

                                r1=pm(i9,m8);
                                lcctr=r2, do combine until lce;
                                    r2=r0+r1,      r0=pm(i8,m8);
ombine:      dm(i1,m0)=r2,  r1=pm(i9,m8);

                                rts (db);
                                r2=r0+r1;
                                dm(i1,m0)=r2;

                                ENDSEG;
```

Listing 9.3 histo.asm

9.4 ONE-DIMENSIONAL MEDIAN FILTERING

A median filter is designed to sort samples in an array by magnitude, lowest to highest. The middle sorted sample is the median value. Median filters require an odd number of samples to guarantee a middle position.

Median filters are used in image processing to average the image without blurring edges, like low pass and mean average filters do. Median filters are non-linear functions and are not used in speech or audio signal processing.

Some median filters are calculated on samples that cover a two-dimensional area. The median filter discussed in this section is one-dimensional; it finds the median of a horizontal line of samples. One-dimensional median filters may be used if the image scanning device is line array or if it necessary to reduce the processing power required of the DSP.

9.4.1 Implementation

Median filters have a delay line similar to the FIR filter that works on the last N samples. After the median filter processes a sample, it outputs the results and waits for the next input sample. When the next sample is received, it replaces the oldest sample in the delay line.

Figure 9.5 demonstrates the first pass through the median filter to resolve the lowest magnitude sample. The median filter result is four in this example.

Listing 9.4 is a fixed point implementation of the median filter for an ADSP-210xx family DSP. The first task is to transfer the samples from the delay line to the median filter buffer, because they are shuffled. Next, the

Image Processing 9

median filter is performed, as demonstrated in figure 9.5. Each pass of the outer loop resolves the next highest magnitude sample in the median filter buffer. Each pass through the inner loop compares the current lowest sample in the outer loop pass to the next sample in the median filter buffer. If the next sample is less than the current lowest, they are swapped. The last outer loop pass resolves the middle or median sample in the median filter buffer. After the median sample has been resolved, it is not necessary to resolve the remaining higher magnitude samples.

Listing 9.5 is the floating-point version of listing 9.4.

Outer Loop Pass 1: R4 = 4

```
cmp R4 = 4, r2 = 6; 6 < 4, no, R4 = 4
cmp R4 = 4, r2 = 7; 7 < 4, no, R4 = 4
cmp R4 = 4, r2 = 5; 5 < 4, no, R4 = 4
cmp R4 = 4, r2 = 1; 1 < 4, yes, R4 = 1, base_adr + 4 = 4
cmp R4 = 1, r2 = 2; 2 < 1, no, R4 = 1
cmp R4 = 1, r2 = 3; 3 < 1, no, R4 = 1
```

First pass resolves lowest in buffer, since lowest is not median in is not restored into the median filter buffer. Next pass starts with r4 = base_adr + 1 =5. The next pass will resolve the next lowest. The third pass of the outer loop will resolve the next lowest. The fourth pass will resolve the median.

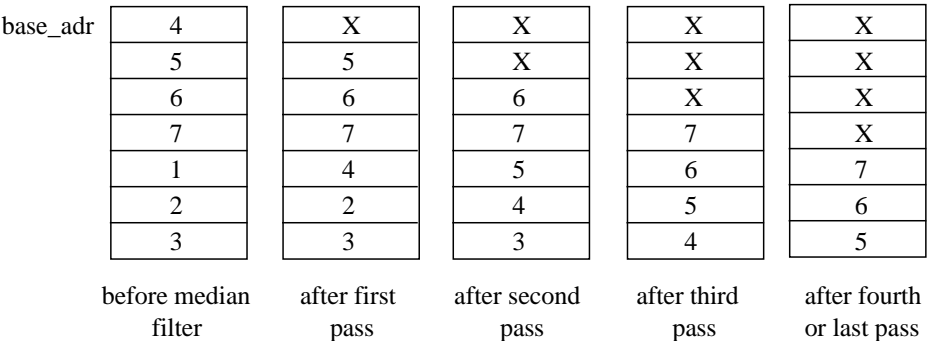


Figure 9.5 Median Filter Algorithm

9 Image Processing

9.4.2 Code Listings

```
/
*****

File Name
    med_fix.asm

Version

Purpose
    N tap one-dimensional median filter subroutine for fixed point data
    Equations Implemented

Calling Parameters
    b1, i1 = start address of input delay line in data memory
    l1      = length of delay line buffer
    m1      =1 - to modify index registers
    b8, i8 = start address of median filter buffer in program mem
    l8      =length of delay line buffer
    m9      =1 - to modify index registers

Return Values
    r4 = median of values in delay line

Registers Affected
    r0, r2, r4, r5, r15

Cycle Count
    FILTER_LEN + 6*[(FILTER_LEN+1)/2] + 14 + 3*sum[N]
    where N=(FILTER_LEN-1)/2 to FILTER_LEN-1
    99 cycles for FILTER_LEN=7

# PM Locations
    16 Instruction + N Words of PM Data, where N is the order of the median
filter
# DM Locations
    N Words, where N is the order of the median filter
```


Image Processing 9

```
*****/

#define FILTER_LEN 7                /* must be an odd number */

.segment/pm pmcode;

.GLOBAL start_median;

start_median:

/* xfer loop - transfer delay line data in DM to median filter buffer in PM */

    r0=dm(i1,m1);
    lcntr=FILTER_LEN-1, do xfer until lce;
xfer:    r0=dm(i1,m1), pm(i8,m9)=r0;    /* transfer to filter buffer */
                                         /* read from input buffer */
    pm(i8,m9)=r0;                      /* transfer to filter buffer */

/*
    median filter loop - find median value in delay line using this technique:

        for N=0 to (FILTER_LEN+1)/2    - outer loop

            for M=N to FILTER_LEN      - inner loop

                if (median[N] < median[M])

                    median[M]=median[N], median[M]=median[N]

                inc M

            inc N
*/

    b9=b8;                            /* i8, i9 point to median filter data */
    r15=FILTER_LEN;                    /* r15 is loop count for inner loop */

/* each pass through the outer loop resolves the next greatest magnitude */
/* in the median filter buffer - median[N] */

    lcntr=(FILTER_LEN+1)/2, do outer_loop until lce;

    r4=pm(i8,m9);                      /* read median[N] */
    modify(i9,1);                      /* i9 points to median[M] */
                                         /* where M=N+1 first */
    r15=r15-1, r5=pm(i9,m9);           /* decrement inner loop count */
                                         /* read median[M] */

/* inner loop finds minimum of the remaining values in median filter buffer */

    lcntr=r15, do inner_loop until lce;
    r2=pass r5;                        /* f2=median[M] */
    comp(r2,r4), r5=pm(i9,m9);         /* cmp median[M], median[N] */
```

(listing continues on next page)

9 Image Processing

```
inner_loop:    if lt r4=pass r2, pm(-2,i9)=r4;    /* if median[M] < median[N] */
                                                    /* median[N]=median[M] */
                                                    /* median[M]=median[N] */

outer_loop:

                i9=i8;                            /* init i9 to median[N+1] */

                rts;                                /* return is non-delayed */
                                                    /* 3 cycles */

.endseg;
```

Listing 9.4 Fixed-Point 1-D Median Filter

```
/
*****

File Name
    medflt.asm

Version
    1.0

Purpose
    N tap one-dimensional median filter subroutine for floating point data

Calling Parameters
    b1, i1 = start address of input delay line in data memory
    l1      = length of delay line buffer
    m1      =1 - to modify index registers
    b8, i8  = start address of median filter buffer in program mem
    l8      =length of delay line buffer
    m9      =1 - to modify index registers

Return Values
    f4 = median of values in delay line

Registers Affected
    f0, f2, f4, f5, r15

Cycle Count
    FILTER_LEN + 6*[(FILTER_LEN+1)/2] + 14 + 3*sum[N]
    where N=(FILTER_LEN-1)/2 to FILTER_LEN-1
    99 cycles for FILTER_LEN=7

# PM Locations
    16 instructions + N Words of PM Data,
    where N is the order of the median filter

# DM Locations
```

Image Processing 9

```

        N Words, where N is the order of the median filter

*****/

        Execution Time:

*/

#define  FILTER_LEN 7                /* must be an odd number */

.segment/pm pmcode;

.GLOBAL start_median;

start_median:

    /* xfer loop - transfer delay line data in DM to median filter buffer in PM */

    f0=dm(i1,m1);
    lcntr=FILTER_LEN-1, do xfer until lce;
xfer:    f0=dm(i1,m1), pm(i8,m9)=f0;    /* transfer to filter buffer */
        /* read from input buffer */
        pm(i8,m9)=f0;                /* transfer to filter buffer */

/*
    median filter loop - find median value in delay line using this technique:

        for N=0 to (FILTER_LEN+1)/2    - outer loop
            for M=N to FILTER_LEN      - inner loop
                if (median[N] < median[M])
                    median[M]=median[N], median[M]=median[N]
                inc M
            inc N
*/

    b9=b8;                            /* i8, i9 point to median filter data */
    r15=FILTER_LEN;                    /* r15 is loop count for inner loop */

    /* each pass through the outer loop resolves the next greatest magnitude */
    /* in the median filter buffer - median[N] */

    lcntr=(FILTER_LEN+1)/2, do outer_loop until lce;

        f4=pm(i8,m9);                /* read median[N] */
        modify(i9,1);                /* i9 points to median[M] */

```

(listing continues on next page)

9 Image Processing

```

                                /* where M=N+1 first */
                                /* decrement inner loop count */
                                /* read median[M] */
                                /* inner loop finds minimum of the remaining values in median filter buffer */

                                lcntr=r15, do inner_loop until lce;
                                f2=pass f5;                                /* f2=median[M] */
                                comp(f2,f4), f5=pm(i9,m9);                /* cmp median[M], median[N] */
inner_loop:  if lt f4=pass f2, pm(-2,i9)=f4;                            /* if median[M] < median[N] */
                                /* median[N]=median[M] */
                                /* median[M]=median[N] */
                                /* median[M]=median[N] */

outer_loop:

                                i9=i8;                                    /* init i9 to median[N+1] */

                                rts;                                    /* return is non-delayed */
                                /* 3 cycles */

                                .endseg;
```

Listing 9.5 Floating-Point 1-D Median Filter

9.5 REFERENCES

- [GLASSNER90] Glassner, Andrew S., ed. 1990. *Graphics Gems*. San Diego, CA: Academic Press, Inc.
- [GONZALEZ87] Gonzalez, R.C. and P. Wintz. 1987. *Digital Image Processing*. Reading, MA: Addison Wesley.
- [JAIN89] Jain, Anil K. 1989. *Fundamentals of Digital Image Processing*. Englewood Cliffs, NJ: Prentice Hall.