Implementation of an Image Processing Library for the TMS320C8x (MVP)

Literature Number: BPRA059
Texas Instruments Europe
July 1997

IMPORTANT NOTICE

Texas Instruments (TI) reserves the right to make changes to its products or to discontinue any semiconductor product or service without notice, and advises its customers to obtain the latest version of relevant information to verify, before placing orders, that the information being relied on is current.

TI warrants performance of its semiconductor products and related software to the specifications applicable at the time of sale in accordance with TI's standard warranty. Testing and other quality control techniques are utilized to the extent TI deems necessary to support this warranty. Specific testing of all parameters of each device is not necessarily performed, except those mandated by government requirements.

Certain applications using semiconductor products may involve potential risks of death, personal injury, or severe property or environmental damage ("Critical Applications").

TI SEMICONDUCTOR PRODUCTS ARE NOT DESIGNED, INTENDED, AUTHORIZED, OR WARRANTED TO BE SUITABLE FOR USE IN LIFE-SUPPORT APPLICATIONS, DEVICES OR SYSTEMS OR OTHER CRITICAL APPLICATIONS.

Inclusion of TI products in such applications is understood to be fully at the risk of the customer. Use of TI products in such applications requires the written approval of an appropriate TI officer. Questions concerning potential risk applications should be directed to TI through a local SC sales office.

In order to minimize risks associated with the customer's applications, adequate design and operating safeguards should be provided by the customer to minimize inherent or procedural hazards.

TI assumes no liability for applications assistance, customer product design, software performance, or infringement of patents or services described herein. Nor does TI warrant or represent that any license, either express or implied, is granted under any patent right, copyright, mask work right, or other intellectual property right of TI covering or relating to any combination, machine, or process in which such semiconductor products or services might be or are used.

Contents

1. Introduction 1
2. Basics about Digital Image Processing 2
3. Basics about the TMS320C80 3
4. Software Implementation 7
   - 4.1 Frame Design for the Library 7
   - 4.2 Principles of ADSP Implementations 9
   - 4.3 Thresholding 11
   - 4.4 Histogram Equalization 13
   - 4.5 Convolution 15
   - 4.6 Median 20
   - 4.7 Morphology 25
5. Results 30
6. Conclusion 30

List of Figures

Figure 1: Phases in Digital Image Processing 2
Figure 2: Architecture of TMS320C80 4
Figure 3: Architecture of the Data-Unit of the Parallel Processors 5
Figure 4: Comparison of Algrebraic Syntax and the Traditional Assembly Syntax 6
Figure 5: Scheme of Multitasking Frame for Image Processing Algorithms 7
Figure 6: Double BufferingTtechnique 9
Figure 7: Triple Buffering Technique 10
Figure 8: Example for Thresholding of an Image 12
Figure 9: Histogram Equalization Scheme 13
Figure 10: Edge Enhancement with Convolution 15
Figure 11: 5x5 Dilation and Erosion 26

List of Tables

Table 1: Number of Cycles Required in Core Loop for some Image Processing Algorithms 30

Implementation of an Image Processing Library for the TMS320C80 (MVP)

ABSTRACT

Digital Image Processing is today a continuously evolving market. There are a lot of applications especially in the industrial area, in medical imaging, document imaging, and in the security area. Texas Instruments digital signal multiprocessor TMS320C80 offers the computing performance to support many of these new possible applications. This paper describes the software implementation of some of the most important and common basic image processing algorithms on this device.

1. Introduction

Because of its complexity digital image processing generally requires a large bandwidth in terms of MIPS and system throughput. Hardware solutions have proved to be costly and inflexible. Thus, it seems quite intuitive that innovative computer architectures are often applied to image processing to allow software realizations. One of the most interesting trends in computer architecture of today is parallelism, that is multiprocessors, multi-scalar processors with long instruction words and the well-known pipelined processors. The TMS320C80 comprises all of these features.

A barrier to these techniques is certainly the problem of efficient software development. One can exploit all these parallelism features hardly without adapting the algorithms to the processor architecture. This redesign of algorithms and the permanent consideration of the architectural properties during software development seems often to be very difficult.

The TMS320C80 has reduced these problems through a shared-memory structure with a crossbar and through an easy-to-learn and easy-to-use algrebraic assembler language for the very long instruction word. This paper discusses the implementation of some of the most common image processing algorithms on the TMS320C80. The algorithms are designed in a way that they might exploit the features of this processor as far as possible.

2. Basics about Digital Image Processing

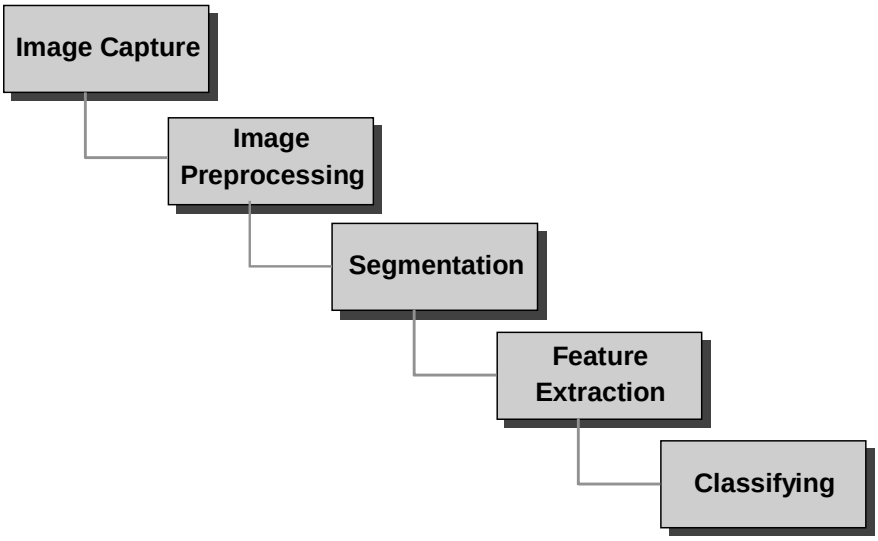


Figure 1: Phases in Digital Image Processing

Most of the digital image processing methods can be divided into five phases. First, the image has to be captured. To be suitable for computer processing the image function has to be digitized both spatially and in amplitude. The digitization of the spatial coordinates is called image sampling -- the amplitude digitization is called gray-level quantization. As a result of these operations we obtain an array with discrete elements -- typically numbers among 0 and 255 as gray-levels for 8 bit values.

In the second phase the image is preprocessed. That means, the image is enhanced, restored or compressed. Enhancement of the image could be achieved through intensification of the contrast or through noise reduction.

A further important step is to segment the image. The main purpose of this phase is to subdivide an image into its constituent parts. The segmentation should isolate the objects of interest in the image.

After the segmentation of images into regions the resulting aggregates of segmented pixels have to be represented somehow. Therefore, many features of regions can be used as descriptors for regions. There are boundary descriptors like the length of the contour or the diameter of a boundary, and regional descriptors like the area of a region or its compactness. More complex features are the topology, the texture or the morphology of a region.

In the last step the region, and hence its descriptor, has to be classified in order to recognize and interpret the image finally. Possible techniques for this phase are decision-theoretic methods like matching and optimum statistical classifiers, structural methods

like string matching and syntactic methods and interpretation methods like logical systems, semantic networks and expert systems.

The TMS320C80 is specially qualified for most of the algorithms in the last four phases. The algorithms of the fifth phase, however, are rather application specific. Therefore, we discuss only algorithms for the second, the third and the fourth phase in this paper. For more details about image processing and also about the theory of the algorithms described in chapter 4 please refer to [2] and [4].

3. Basics about the TMS320C80

The TMS320C80 is a multiprocessor with one Master Processor, four Advanced DSPs, which are also called Parallel Processors, a Transfer Controller and a Video Controller (see Figure 2). The different processors are tightly coupled through a shared memory system, which is maintained by a crossbar. For more details about the architecture of the TMS320C80 please refer to [5].

Imaging applications require processing techniques that range from multiplication-intensive filtering and frequency-domain transform to massive numbers of simple arithmetic operations on pixel data, bit-field extraction, and look-up table (LUT) operations. Because of their architecture, the ADSPs are dedicated to this kind of operations and thus most parts of the algorithms run on these. Special features of the data unit (see Figure 3) within the ADSPs are:

- independent multiplier and ALU;
- multiplier and ALU are splittable for parallel computations with bytes and halfwords;
- the ALU has three inputs -- this is especially useful for masking operations;
- the data unit has special hardware for pixel manipulations like a barrel rotator, a mask generator, an expander and a bit detection logic.

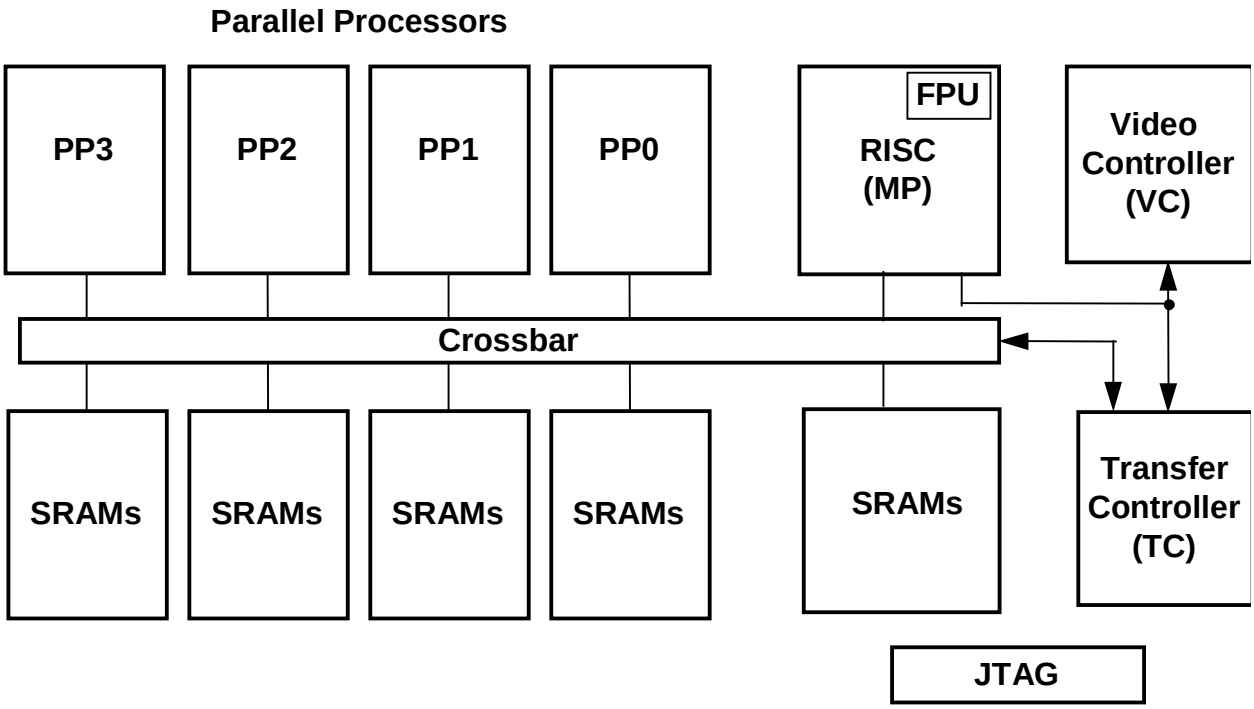


Figure 2: Architecture of TMS320C80

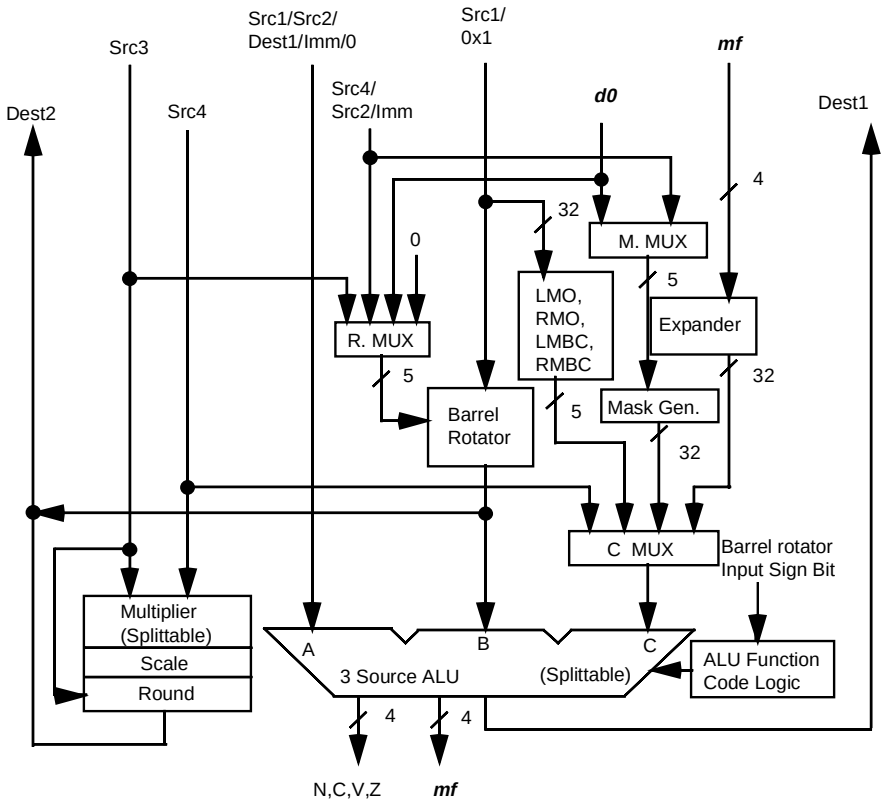


Figure 3: Architecture of the Data-Unit of the Parallel Processors

The three input ALU can perform 256 different Boolean and 256 different mixed Boolean and arithmetic functions. The combination of the three input ALU and a variety of data paths consisting of a barrel rotator, a mask generator and a multiple flag register, feeding the inputs to the ALU, allow over 4000 different operations to be performed. This flexibility generally results in fewer instructions being required to perform a given algorithm. An efficient utilization of all these operations can be easily obtained through the usage of the rather intuitive algrebraic assembler language (see Figure 4).

PP Algebraic Syntax	***Traditional Assembly Syntax***
dst = ~(src1 & src2)	**nand src1,src2,dst**
dst = (src1 | src2) ^ dst	**or src1,src2,dst1** **xor dst1,dst,dst**
dst = src1 >>s src2	**sra src1,src2,dst**
br = ipe + offset	**br offset**
dst = src	**mv src,dst**
src1 = src2 **||src2 = src1**	**swap src1,src2**
dst = src1 & 1 << src2	**tstb src1,src2**
mdst = msrc1 * msrc2 **||adst = asrc1 + asrc2** **|*Gaddr = st_src** **||ld_dst = *Laddr**	**mpy msrc1,msrc2,mdst** **||add asrc1,asrc2,adst** **||st st_src,*Gaddr** **||ld *Laddr,ld_dst**

Figure 4: Comparison of Algrebraic Syntax and the Traditional Assembly Syntax

The TMS320C80 has also two address units, which perform address computations, stores, loads, register-to-register moves and address-unit-arithmetic, which is the usage of the adress units ALU for simple arithmetic operations.

The transfer controller operates in parallel with the memory crossbar and acts as an intelligent DMA controller prioritizing and performing all accesses with off-chip memory without processor interrupt. The transfer controller enables linear and two-dimensional addressing for images. Assuming 8 byte wide memory the TC has a bandwidth of 400 Mbytes per second (50 MHz).

4. Software Implementation

The Master Processor (MP) is a pure RISC processor with a set of around 50 instructions that can be programmed very efficiently in C. The C compiler was developed in parallel with the design of the MP architecture and from there is specially adapted to it. For the Advanced DSPs (ADSPs) a C compiler is also available. For the core loop of an algorithm, however, these processors have to be programmed with the algrebraic assembler language in order to utilize all the capabilities of the long instruction word. Currently, compilers are not able to generate highly optimized code for this kind of computer architecture. In the image processing library, the whole code for the ADSPs is written in the assembler language.

4.1 Frame Design for the Library

Texas Instruments delivers together with the development tools a multitasking executive (ME). This contains a kernel which is running on the MP and a command interpreter running on the ADSPs. The kernel provides methods for communication between tasks like messages and signals and functions for issuing commands to an ADSP. In the library the ME is used according to Figure 5 to define a common calling structure for all the different image processing functions.

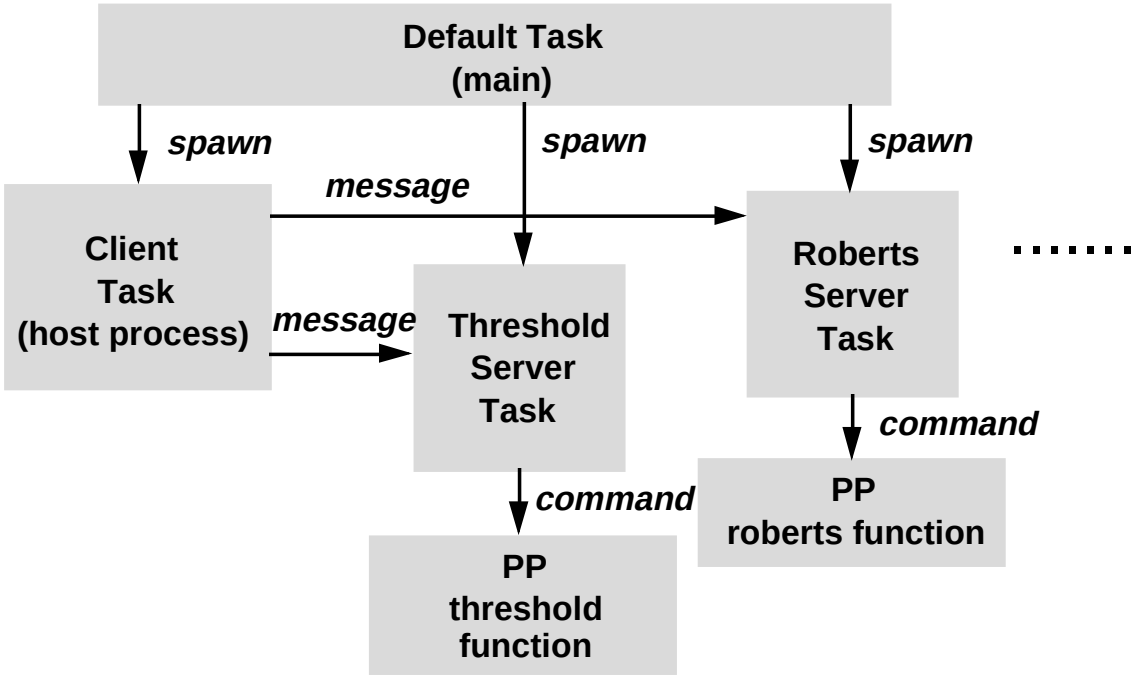


Figure 5: Scheme of Multitasking Frame for Image Processing Algorithms

On the MP a client task is running under the ME in order to simulate a task running on a host processor. The default task, which is the C main program, spawns this host client task and the server tasks, that is, they are created and resumed. The client task could

actually be replaced by a task running on the host processor, which requests special calculations (like thresholding or graylevel-scaling) as services from the different server tasks -- generally there is more than one client possible. The requests are submitted via messages with the necessary arguments.

The server tasks are waiting for the requests, perform some preprocessing and try to distribute the main computing work among the powerful ADSPs. The distribution is achieved via a simple vertical partitioning of the image into four parts. Thus, we achieve the parallelism through data partitioning.

If the width of the image is larger than what the ADSPs can store in one local data RAM block then the image has to be partitioned horizontally too. That means, the image is divided in halves or quarters. Therefore, the number of columns has to be a power of two. The ADSPs handle these parts as if they are processing a complete image.

The server tasks can share one ADSP between each other. This means if different server tasks have been assigned to the same ADSP, then they get this processor according to the first-come-first-serve principle. The mutual exclusion is realized by a semaphore. The host client can assign any number of ADSPs to the different server tasks and so the number of ADSPs is scalable for each service depending on the level of demand for this service or its computing power requirements. Here we have an example for parallelism through functional partitioning.

On each ADSP runs a command interpreter waiting for the commands issued by the server tasks. Depending on the command an appropriate program will be started on the ADSPs. After the ADSPs have executed their computations on the image they send an interrupt to the Master Processor. The interrupt service routine sends through the kernel a signal to the semaphore related to the ADSP notifying that this has completed its work.

The server task waits for these notifications from all assigned ADSPs and performs some postprocessing functions. After that it sends a reply message to the client task with a status code indicating whether the computation had been successful or whether errors had been occurred.

The hostclnt.c program in the library provides an example for a client task requesting several services from the morphological server task. Thus it is computing a morphological smoothing operation on a noisy image. This method consists of an opening followed by a closing.

The parameters of the first message are defined as follows:

```
  morph_msg->ppnr = 4;          /* number of parallel processors        */
  morph_msg->pp[0] = PP0;       /* number of first parallel processor   */
  morph_msg->pp[1] = PP1;       /* number of second parallel processor  */
  morph_msg->pp[2] = PP2;
  morph_msg->pp[3] = PP3;
  morph_msg->width = 128;             /* image width                    */
  morph_msg->height = 128;            /* image height                   */
  morph_msg->inputAdrs = Input;       /* address of input image         */
  morph_msg->outputAdrs = Output1;    /* address of output image        */
  morph_msg->morphop = EROSION;       /* operator type                  */
  morph_msg->filtersize = 3;          /* filter size                    */
```

4.2 Principles of ADSP Implementations

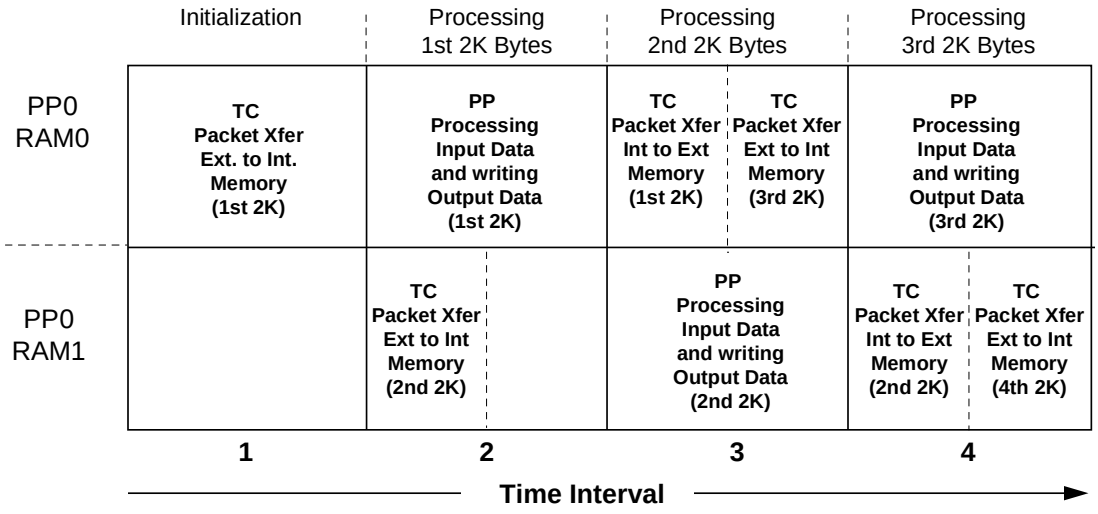


Figure 6: Double BufferingTtechnique

After an ADSP has received a request from a server task it loads the parameters from the argument buffer. In this implementation the ADSPs load and store the required image partitions independently and asynchronously into their local RAM blocks. For that the ADSPs use mostly a double buffering technique. The loads and stores are realized through packet transfers by the Transfer Controller (TC). Since the TC can work in parallel to the ADSPs it is reasonable to reserve one RAM block as buffer for the TC to perform a store of an already processed image partition and subsequently a load of an image partition which should be processed in the next run. At the same time the ADSP can process the data in one of the two other local RAM blocks (see Figure 6).

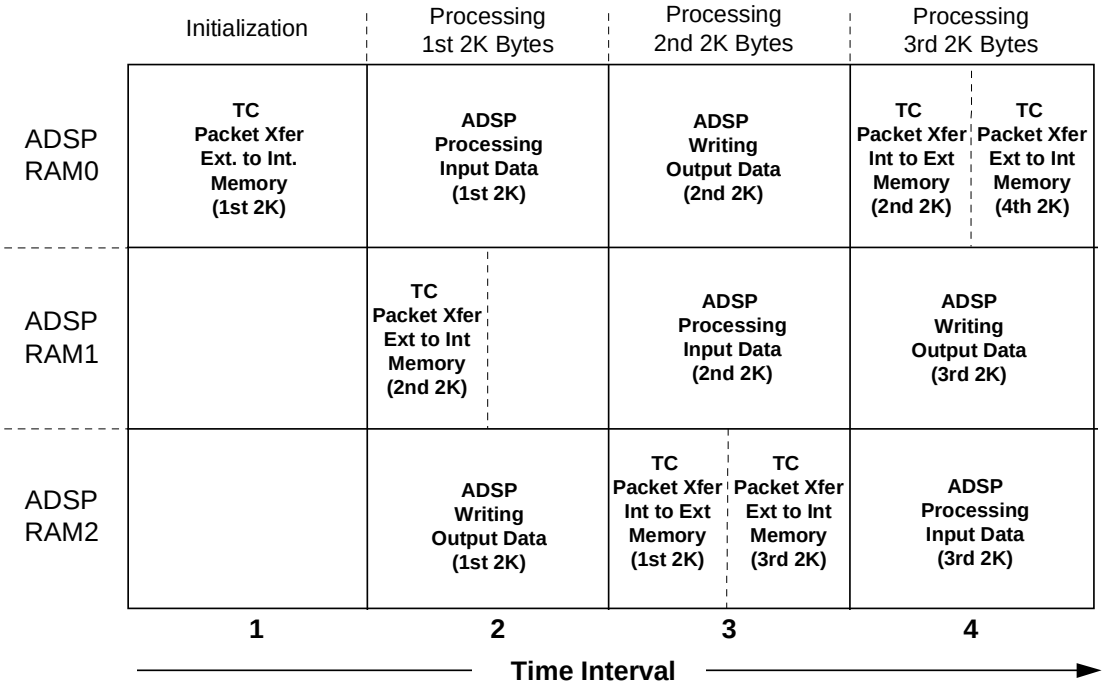


Figure 7: Triple Buffering Technique

If the input image should not be destroyed during the pixel processing we need a third buffer for the output image (see Figure 7). This is for example necessary for the convolution algorithm where always the old pixel values in the neighborhood of a pixel are needed for computation of the new value. The program flow of most of the algorithms in the library is therefore:

```
Flow of typical code:

1) loading of parameters from argument buffer

2) copy filter coefficients from external memory into parameter RAM

3) setup of packet transfer parameters

4) load first partition of the image

5) load second partition of the image and branch to start of
   pixel manipulations 7)

6) packet transfers for double buffer technique
  a) initiate next store of already processed image partition
  b) check whether more computations are necessary --
     if no then branch to 9)
  c) check whether a new load is necessary (the whole image has not
     been loaded yet) -- if no then branch to 7)
  d) initiate next load of a image partition

7) process one partition of the image

8) check whether last packet transfer has finished and branch to 6)

9) check whether last packet transfer has finished and stop

End of Program
```

The data bandwidth should be generally sufficient for image processing applications. If we assume 512x512 pixel images with 1 byte pixels we would have to load around 262 Kbytes and store them back again. A typical frame rate would be 60 frames per second. Thus we would need 512x512x60x2 = 13.5 Mbytes per second. That should not be a problem for the TC with its 400 Mbytes per second. An important precondition for the achievement of this bandwidth and for the efficient execution of the code is that the TC does not have to execute cache operations.

For that, the core loops of the assembler code should always fit into the 2 Kbytes instruction cache in order to avoid cache misses. This means that these core loops should not exceed 256 instructions since each instruction consists of 64 bit. This is typically no problem since the instructions are very powerful.

The other question would be whether the MIPS bandwidth of the ADSP is large enough. We have generally 50 MIPS per ADSP (50MHz). With the image size and frame rate described before we would need 4 x 50 = 200 MIPS divided by 512x512x60 = 15.7 Mpixels per second, which results in 12.7 instructions per pixel. In the library below this number is only exceeded by the 5x5 median and the 5x5 convolution algorithm, which means that either the frame rate or the image size would have to be reduced for these algorithms.

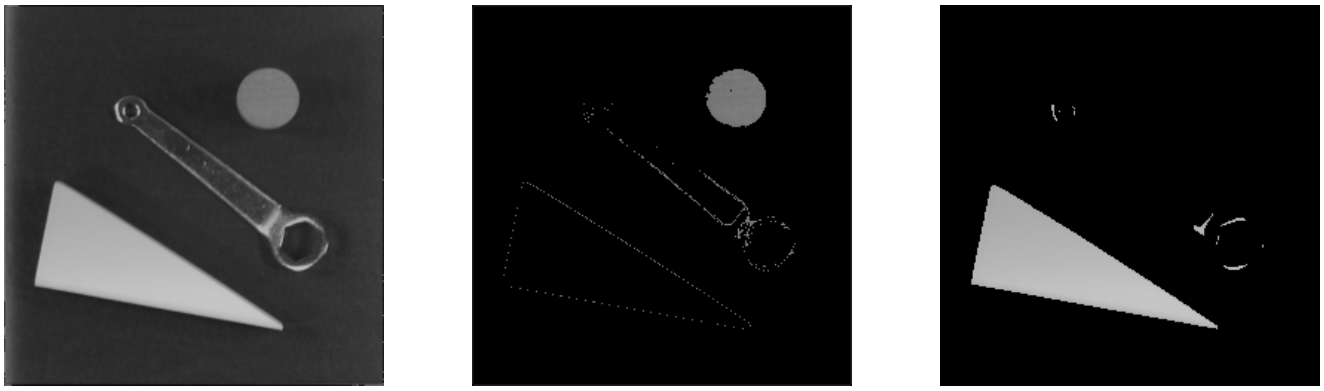
For most of the image processing algorithms the TMS320C80 is neither computation bound nor I/O bound. The estimations above can easily be adapted to the TMS320C82 with two ADSPs, which has 100 MIPS (50MHz) and further TMS320C80 versions with 60MHz.

4.3 Thresholding

Thresholding is a method to segment the image. The thresholding algorithm computes:

$$s'(x,y) = \begin{cases} s(x,y), & \text{if } l \le s(x,y) \le h \\ 0, & \text{else} \end{cases}$$

where "l" is the lower threshold and "h" is the upper threshold. Thus, all pixels with gray levels outside this range are set to zero. An example for the application of this algorithm can be seen in Figure 8. Because of the homogeneous gray levels of the objects in the original picture, it can easily be segmented by this thresholding algorithm. The remaining noisy pixels can be eliminated through a median filter or through morphological operations.



Original **Thresholded images**

Figure 8: Example for Thresholding of an Image

```
      ;
      ; Core loop for thresholding
      ;
Loop:
      temp =mc pixel - low_thresh         ; Compare pixels with lower threshold
            || *Ga_pt++ =w temp           ; Store thresholded pixels
      pixel = (pixel & @mf) | (0x0 & ~@mf); Determine maximum values
      temp =mc high_thresh - pixel        ; Compare pixels with upper threshold

Loopend:
      temp = (pixel & @mf) | (0x0 & ~@mf) ; Determine minimum values
            || pixel =w *La_pt++          ; Load next four pixels
```

The algorithm can be realized very efficiently by splitting the ALU for byte arithmetic. In the first instruction of the core loop the pixel register is compared with the lower threshold register. It is important to note that the registers actually contain four pixels each -- the threshold registers contain the replicated threshold values respectively. The only interesting aspects of the subtraction are the carry-out bits in the multiple flag register (mf). If the resulting byte is positive then the carry bit is set to one, because a borrow did not occur. The expander (@) creates a mask of the status flags in the multiple flags register by replicating each of the four LSBs eight times to fill out a 32 bit word. Then the expander output determines whether the original pixel value or the zero value is taken. This means if the original pixel values were larger than the lower threshold value then this pixel value remains the same otherwise it is set to zero. The same mechanism is used for the compare with the upper threshold values.

4.4 Histogram Equalization

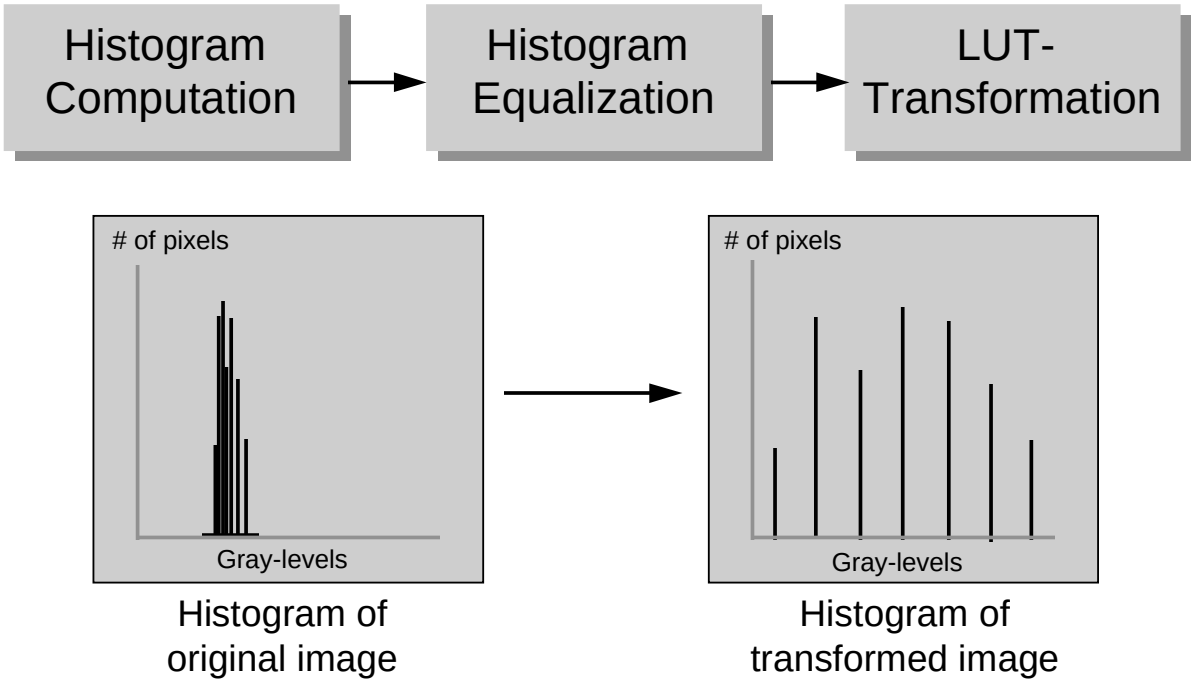


Figure 9: Histogram Equalization Scheme

The histogram equalization algorithm transforms the pixels as follows:

$$s' = T(s), \text{ with } T(s) = \sum_{t=0}^{s} \frac{n(t)}{n}$$

where s is the gray-level, $n(t)$ is the number of pixels with gray level t, that is the histogram of this gray level, and n is the total number of pixels.

The equalization method consists of a histogram computation, the creation of a lookup table, and the transformation in accordance with this table. The histogram-equalization algorithm increases according to Figure 9 the dynamic range of gray levels and, consequently, produces an increase in image contrast. However, you have to obtain that in images with narrow histograms and relatively few gray levels, the increase in dynamic range normally has the adverse effect of increasing visual graininess.

The histogram computation and the lookup table transformation are realized as services. The histogram equalization server task requests these by sending messages to the appropriate server tasks. The creation of the lookup table is executed by the equalization task itself on the MP.

The histogram algorithm simply computes a table h(s), where h(s) is the number of pixels in an image with the gray-level s.

```
Loopstart:
      hist_offset2 = hist_offset1               ; Move offset
            || number =w *(hist_start_adrs + [hist_offset1])
                                          ; Fetch next histogram value
      number = number + 1                       ; Increment histogram value
            || hist_offset1 =ub *Ga_pt++        ; Fetch next value as an offset
Loopend:
      *(hist_start_adrs + [hist_offset2]) =w number   ; Store histogram value
```

The algorithm exploits the long instruction word again by some sort of software pipelining. The first instruction of the core loop loads the actual histogram and also performs a register-to-register move in parallel. hist_offset1 has been loaded in the second loop instruction of the last loop execution. The next instruction increments the actual histogram value. At the same cycle a new pixel value is fetched into the offset register. The third instruction stores the new histogram value. For that, it needs the hist_offset2 register, which has the value of the old hist_offset1 register. With that you achieve a histogram table update for one pixel in three cycles.

After the histogram computation the lookup table has to be created according to the formula above:

```
      /*
       * Histogram-Equalization computations:
       */
      nr_of_pixels = width * height;
      factor = (255.0 / (float)nr_of_pixels);
      sum = 0;
      for (pixel = 0; pixel < 256; pixel++) {
            sum += hist_pt[pixel];
            lut_pt[pixel] = (unsigned char)(sum * factor);
      }
```

During its computation the lookup table is normally loaded into the MPs data cache. In the lookup table transformation phase the ADSPs need this lookup table. Therefore, this part of the data cache has to be flushed. Additionally you have to observe that the lookup table and the histogram table have to be aligned to a 256 byte boundary so that these tables fit in one data-cache block respectively and the overhead for cache fills is reduced.

At the end the LUT algorithm transforms an image according to the lookup table. The lookup table is loaded into one of the local memory blocks. According to this table a new gray level is assigned for each old gray level.

The lookup-table transformation is obtained by two nested loops with each loading two pixels, transforming the values and storing the new gray levels. This is achieved with a rate of one-and-a-half cycles per pixel:

```
      ; Main loop with pixel transformations
Loop:
      lut_offset1 =ub *Ga_pt1++                  ; Load the next pixel
            || pixel2 =ub *(lut_start_adrs + lut_offset2)
                              ; Load next transformed pixel from lookup-table
      lut_offset2 =ub *Ga_pt1++                  ; Load the next pixel
            || *(La_pt2++=2) =ub pixel1          ; Store transformed pixel
Loopend:
      *(Ga_pt2++=2) =ub pixel2                   ; Store transformed pixel
            || pixel1 =ub *(lut_start_adrs + lut_offset1)
                              ; Load next transformed pixel from lookup-table
```

For this algorithm you have to observe that if the contents of an address- or an index-register has been changed by the data unit or has just been loaded then the new contents of this register cannot be used by the address unit in the next instruction. The reason for this is that the ADSP uses a three stage pipeline consisting of a fetch cycle, an address computation cycle and an execution cycle. A data unit operation or a load operation would be executed in the execution stage and the address unit operation would be executed in the same cycle in the address computation stage of the pipeline and, therefore, using the old value of the address- or index-register.

4.5 Convolution

Figure 10: Edge Enhancement with Convolution

Almost all kinds of filtering operations need the convolution algorithm like for example edge detection, gradient filters, and image smoothing.

The formula for the convolution is:

$$s'(x,y) = s(x,y) \times \begin{pmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{pmatrix}$$

The library contains two different convolution algorithms: a simple one, which is easily scaleable and maintainable. The other one is highly optimized and more difficult to understand. The first algorithm can be used for 3x3 and 5x5 convolutions depending on the setting of an assembler constant at assembly-time. The optimized version provides only a 3x3 convolution.

The code below is a clipping of the simple version. In the first line the result variable is set to zero, the result which has been calculated during the last loop execution is stored, and the next pixel value is loaded. In the following lines the pixel value is multiplied with the filter coefficient and the new pixel value and filter coefficient for the next instruction are loaded.

```
Loop:
      result = 0 ||                        ; Initialize result with zero
            *Ga_out++ =ub result ||        ; Store last result
            pixel =ub *La_pt++             ; Load next pixel
      mult = pixel * filter ||             ; Multiply pixel and filter coefficient
            filter =b *Ga_filt++ ||        ; Load next filter coefficient
            pixel =ub *La_pt++             ; Load next pixel
      mult = pixel * filter ||             ; Multiply pixel and filter coefficient
            result += mult ||              ; Accumulate last product
            filter =b *Ga_filt++ ||        ; Load next filter coefficient
            pixel =ub *(La_pt++=offset1)   ; Load next pixel and redirect
                                           ; address pointer to the next line
       .
       .
       .
```

This has to be done for all filter coefficients. After that you can perform an additional saturation. For that, you need around 11 cycles per pixel per PP for a normal 3x3 convolution and 15 cycles per pixel per PP for a saturated 3x3 convolution. This version can easily be adapted to 16 bit pixels, and scaled up to convolutions of higher order.

The more efficient convolution algorithm uses a trick for exploiting the splitted multiplier and the splitted ALU if you work with 8 bit pixels. Through splitting of the multiplier, two by two 8 bit values can be multiplilyed respectively with two 16 bit results. The two by two source bytes have to be ordered within the lower halfwords of two source registers and the two resulting halfwords will be written into one destination register. These two 16 bit values, that is the appropriate destination registers, can then be added on with the results of the other multiplications by the splitted ALU in order to obtain the sums of products which are necessary for the convolution.

For an explanation of the optimized 3x3 convolution routine we take a look in one line of the convolution mask -- the other two lines are computed similarly:

In this example, for simplicity, we just have six pixels named p_0, p_1, p_2, p_3, p_4 and p_5 and the three filter coefficients named c_0, c_1 and c_2. Thus, we would have to calculate the results: $c_0p_0 + c_1p_1 + c_2p_2$, $c_0p_1 + c_1p_2 + c_2p_3$, $c_0p_2 + c_1p_3 + c_2p_4$ and $c_0p_3 + c_1p_4 + c_2p_5$ in order to perform a one-line convolution. The algorithm below requires a preordering of the filter coefficients during a copy-loop at the beginning of the program into c_0 c_2 c_2 c_0 c_1 c_1 within the parameter memory.

For understanding this algorithm you have to keep in mind that we always calculate the convolution sums for different pixels in parallel.

The algorithm consists of three parts: one part for the first two columns, one loop for the other columns, and one part for the last two columns. In our example we just have two columns for the second part -- normally we would have many more columns in this part. We get four different intermediate results during the calculation. The products of pixels and filter coefficients have halfwordsize -- therefore, we need two 32 bit registers to store the four sums of intermediate results: result1 and result2.

For the first part of the algorithm we compute the products c_0p_0, c_2p_1, c_2p_0, c_0p_1, c_1p_0, and c_1p_1 by multiplying the first pixel pair with the three coefficient pairs, which had been preordered before. The first product pair (c_0p_0 | c_2p_1) is rotated by 16 bits and then stored into result2. Observe that this product pair is computed within one cycle by the splitted multiplier. The other results are accumulated in result1 and result2 in normal order. Thus we achieve the sums:

$$\text{result1} = (c_2p_0 \mid c_0p_1)$$
$$\text{result2} = (c_2p_1 + c_1p_0 \mid c_0p_0 + c_1p_1)$$

where | separates the two halfwords. All six products are computed within three cycles. After the first part we store just one zero byte because we did not obtain a full convolution sum. This can be changed if the temporary results above should already be used at the border of the result image. At the end of the first part we rotate the lower halfwords of the result registers and skip the upper halfwords. Afterwards, the two result registers are swapped:

$$\text{result1} = (c_0p_0 + c_1p_1 \mid \_)$$
$$\text{result2} = (c_0p_1 \mid \_)$$

In the second part we reuse the results of the first part, or of the previous executions of the second part, and add the appropriate product pairs, which we computed with the pixels p_2 and p_3 as we did it before with the pixels p_0 and p_1:

$$\text{result1} = (c_0p_0 + c_1p_1 \mid \_) + (c_2p_2 \mid c_0p_3) = (c_0p_0 + c_1p_1 + c_2p_2 \mid c_0p_3)$$
$$\text{result2} = (c_0p_1 \mid \_) + (c_0p_2 \mid c_2p_3)\backslash\backslash 16 + (c_1p_2 \mid c_1p_3) =$$
$$= (c_0p_1 + c_1p_2 + c_2p_3 \mid c_0p_2 + c_1p_3)$$

Here we have now the first two convolution results for the pixels p_1 ($c_0p_0 + c_1p_1 + c_2p_2$) and p_2 ($c_0p_1 + c_1p_2 + c_2p_3$). They are stored in the result image. At the end of this part we swap and rotate the lower halfwords of the result registers and skip the upper halfwords:

$$\text{result1} = (c_0p_2 + c_1p_3 \mid \_)$$
$$\text{result2} = (c_0p_3 \mid \_)$$

If we would have more pixels we would repeat this second part until only two pixels of the input image are left disregarded.

The third part is for this two remaining pixels. Again the product pairs are accumulated to the temporary results of the last part:

$$\text{result1} = (c_0p_2 + c_1p_3 \mid \_) + (c_2p_4 \mid c_0p_5) = (c_0p_2 + c_1p_3 + c_2p_4 \mid c_0p_5)$$
$$\text{result2} = (c_0p_3 \mid \_) + (c_0p_4 \mid c_2p_5)\backslash\backslash 16 + (c_1p_4 \mid c_1p_5) =$$
$$= (c_0p_3 + c_1p_4 + c_2p_5 \mid c_0p_4 + c_1p_5)$$

Here we have again two convolution results, this time for the pixels p_3 ($c_0p_2 + c_1p_3 + c_2p_4$) and p_4 ($c_0p_3 + c_1p_4 + c_2p_5$). They are stored in the result image. The other temporary results are discarded again because no other complete convolution sum can be built with them.

For this algorithm we need the explicit form of the extended ALU (EALU) mechanism. The EALU uses the additional 32 bits of the data-register d0 as opcode-bits in order to achieve a higher number of different operations. The operations are defined with the labels EA1, EA2, and EA3 at the execution points in the program code. The registers ealuA1, ealuA2, and ealuA3 are used to load the d0-register during the pixel manipulation phase in a flexible way. In ealuA1 a parallel split multiply and add with rotate is defined, in ealuA2 a parallel split multiply and add, and in ealuA3 a merge of two upper halfwords.

Here we have again a short section of the optimized convolution algorithm:

```
; Loop for Lines
Loop0:
      ; Computations for the first two columns (first part):
      result1 = 0
            || d0 = ealuA1                ; Initialize d0 for ealu-operation
            || result2 = &*(0)
      mult = 0
            || pixel =uh *Ga_pt           ; Load first pixel pair

      mult =m pixel * filter  ; Execute split multiplication of one pair
                        ; of pixels with one pair of coefficients --
                        ; the result is one pair of halfwords
            || result2 =m ealu(EA1:result2 + mult\\16)
                        ; Dummy ealu-operation -- the split multiply is defined
                        ; as an EALU-Operation therefore the addition with
                        ; rotation has to be executed -- result2 is set to zero
            || filter =h *++La_filt ; Load next filter coefficient pair

      mult =m pixel * filter ; Execute split-multiplication of one pair
                        ; of pixels with the new pair of coefficients
            || result2 =m ealu(EA1:result2 + mult\\16)
                        ; Add last multiplication result with mult-accumulator
                        ; result2. The result is a pair of halfwords. They have
                        ; to be swapped by the rotation to get the right
```

```
                        ; orientation for further computations.
            || d0 = ealuA2              ; Move EALU value for next instruction.
                        ; The different ealu-values can also be fetched from a
                        ; table. Here it is better to use registers (ealuA1,
                        ; ealuA2, and ealuA3), because otherwise we would have
                        ; one more load operation from the parameter RAM which
                        ; would cause a contention (one cycle delay).
            || filter =h *++La_filt ; Load next filter coefficient pair

      mult =m pixel * filter
            || result1 =m ealu(EA2:result1 + mult)
            || pixel =uh *(Ga_pt+=offset1)
            || filter =h *++La_filt

        .
        .
        .
```

You have to observe that this method works only correct if the intermediate results of the 3x3-convolution computation do not exceed halfwordsize. This is the normal case because most filter masks have positive and negative coefficients which result in a value around zero.

The Roberts edge detection algorithm is a specific derivative of the convolution algorithm. However, the filter coefficients are fixed and the size of the filter mask is only two. Therefore it is a very fast edge detection algorithm. On the other hand the results are not as good as the results of a 5x5 Laplace filter as you can see on Figure 10. Since the code is rather straight forward it does not have to be explained in more detail:

```
      ;
      ; Definition of roberts edge detection:
      ;     s'(x,y) = |s(x,y) - s(x+1,y+1)| + |s(x,y+1) - s(x+1,y)|
      ;
      ; Through this operation we obtain some kind of discretisized
      ; differentiation in the two diagonal directions.
      ; This can be seen as an approximation of a Laplace-Operator:
      ;     Laplace(s) = sqrt((ds/dx)^2 + (ds/dy)^2)
      ;
      ; The Roberts crosses are defined as:
      ;   1  0    and    0  1
      ;   0 -1          -1  0
      ; They are similar to a convolution mask.
      ;
      ; Appropriate pixels in the algorithm:
      ;  pixel1   0           and         0           pixel4
      ;  0        pixel2                 pixel3      0
      ;
      ; Computations:
      ; result1 = pixel1 - pixel2
      ; pixel2 = |result1|
      ; result2 = pixel4 - pixel3
      ; result1 = pixel2 + |result2| = |result1| + |result2|
      ; after this result1 is saturated.
      ;
      ; La_pt points to the beginning of the left top pixel in the image.
      ; Ga_pt points to the left pixel in the second line.
      ; We exploit here the split ALU in bytesize in order
      ; to compute four pixels in each run.
Loop0:
      ; pixel2 is loaded and merged in a way that it contains
```

```
      ; the pixels (1,2,3,4):
      pixel2 =ub *(Ga_pt + 4)             ; Pixel 4 is loaded
      pixel3 = *Ga_pt++                   ; Pixels (0,1,2,3) are loaded
            || pixel1 = *La_pt
      pixel2 = pixel3 << 8 | pixel2       ; Pixels (1,2,3,4) are merged
Loop1:
      ; Compute the first cross:
      result1 =mc pixel1 - pixel2
            || pixel4 =ub *(La_pt+=4)
      ; pixel2 = |pixel1 - pixel2|
      pixel2 =mc (result1 & @mf | -result1 & ~@mf)
            || *(Ga_out++=[offset]) = result2
      pixel4 = pixel1 << 8 | pixel4
      ; Compute the second cross:
      result2 =mc pixel4 - pixel3
      ; result1 = |pixel1 - pixel2| + |pixel4 - pixel3|
      result1 =mc pixel2 + (result2 & @mf | -result2&~@mf)
            || pixel2 =ub *(Ga_pt+4)
            || pixel1 = *La_pt
      ; result1 is saturated, i.e. if a byte of result2 had an
      ; overflow then it is set to 0xff
      result2 = @mf | result1
            || pixel3 = *Ga_pt++
Loopend1:
      pixel2 = pixel3 << 8 | pixel2
            || offset = &*(0x1)
```

4.6 Median

In the median algorithm the gray-level of each pixel is replaced by the median of the gray levels in a neighborhood, typically 3x3 or 5x5, of that pixel. The median m of a set of values (9 or 25) is such that half of the values in the set are less than m and the other values are greater than m. For example, in a 3x3 neighborhood the median is the 5th largest value, in a 5x5 neighborhood the 13th largest value. If several values in a neighborhood are the same, all equal values have to be grouped. For example: med(1,2,2,5,6) = 2.

With this nonlinear function you can smooth an image and consequently suppress its noise. This effect can be intensified by repeated application of the median filter or a higher filter order. Generally the median filter forces points with distinct intensities to be more like their neighbors, actually eliminating intensity spikes that appear isolated in the area of the filter mask. The median filter has the desirable property that it does not affect step functions or ramp functions. It is noted that the median filter is more suited to reducing the effect of discrete impulse noise than smoothly generated noise. In other words, the median filter performs very well on images containing binary noise but performs poorly when the noise is Gaussian. The performance is also poor when the number of noise pixels in a window is greater than or equal to half the number of pixels in the neighborhood. Therefore, median filters of higher order might be of special interest. Depending on the setting of an assembler constant at assembly-time a 3x3 or a 5x5 median filter is executed. Median filters of higher order can easily be derived from the 5x5 median. However, one has to observe that higher order median filters tend to blur an image.

This implementation of the 3x3 median filter bases on an example median algorithm by Jeremiah Golston. Here the initial block on a given row of the image requires 19 comparisons between the components of the 3x3 window to determine the median. Initially, a three sort is done on each of the three columns in the block. Next, the first two column maximums are compared with the greater being discarded from median contention. Then the lower is compared with the third column maximum with the greater of the two being discarded. The lower is retained for further consideration. Subsequently, the first two column minimums are compared with the lesser being discarded from median contention. Then the greater is compared to the third column minimum with the lesser of the two being discarded. The greater is retained for further consideration. Next a complete sort of the 3 column medians is performed. The maximum of the column medians is greater than at least 5 pixels; the minimum in its column and the median and minimum of the two other columns. Likewise, the minimum of the column medians is known to be less than at least 5 pixels; the maximum in its column and the maximum and median of the other 2 columns. Thus, only the median of the column medians is retained in consideration for the block median. Finally a sort is done between the minimum column maximum, the median column median, and the maximum column minimum with the median being identified as the block median.

For subsequent blocks in a row, a complete sorting of the first two columns in the block has already been performed and thus the number of compares required is reduced to 13 instead of 19.

In ADSP assembly, four 3x3 blocks can be worked on simultaneously using byte multiple arithmetic. The initial set of four blocks requires 49 cycles and all subsequent blocks on a row require 31 cycles. Hence the tight loop 3x3 median filter performance is 31/4 = 7.75 cycles per pixel.

The program flow is:

1) Pack the pixels of the first two lines: the pixels of a line: 0 1 2 3 4 5 6 7 ... will be reordered to: 0 32 64 96 1 33 65 97 ... Thus, the stepwidth is a quarter of the image width. This is done to achieve that consecutive pixels are in each consecutive word at the same byte position.

2) Pack the pixels of the next line as in 1)

3) Compute maximum, minimum and median values for the three packed lines. Thus:

line0:	0	32	64	96	1	...
line1:	0	32	64	96	1	...
line2:	0	32	64	96	1	...

is transformed into:

line0:	max(col 0)	max(col 32)	max(col 64) ...
line1:	med(col 0)	med(col 32)	med(col 64) ...
line2:	min(col 0)	min(col 32)	min(col 64) ...

This can be done with 2.75 cycles per pixel by using the masking mechanisms of the TMS320C80, which had already been discussed in combination with the thresholding algorithm:

```
Loop1:
      ; Load first word
      pixel0 =w *(Ga_inp2++=offset1)
      ; Load second word
      pixel1 =w *(Ga_inp2++=offset1)

      ; Compare two words with 4 pixels each
      dummy =mc pixel0 - pixel1
      ; Load third word
            || pixel2 =w *(Ga_inp2--=offset2)
      ; Compute maximum pixels of first and second word
      tmp_max = @mf&pixel0 | ~@mf&pixel1  ; tmp_max = max(pixel0, pixel1)
      ; Recompute address pointer
            || a15 = &*(Ga_inp2 += 4)
      ; Compute minimum pixels of first and second word
      tmp_min = ~@mf&pixel0 | @mf&pixel1  ; tmp_min = min(pixel0, pixel1)
      ; Recompute address pointer
            || a15 = &*(Ga_inp1 += 4)

      ; Compare maximum pixels of first and second word with the third word
      dummy =mc tmp_max - pixel2
      ; tmp_med = min(max(pixel0, pixel1), pixel2)
      tmp_med = ~@mf&tmp_max | @mf&pixel2
      ; max = max(pixel0, pixel1, pixel2)
      max = @mf&tmp_max | ~@mf&pixel2

      dummy =mc tmp_min - tmp_med
      ; med = max(min(pixel0, pixel1), min(max(pixel0, pixel1), pixel2))
      ;     = med(pixel0, pixel1, pixel2)
      med = @mf&tmp_min | ~@mf&tmp_med
            || *(Ga_inp1++=offset1) =w max
      ; min = min(min(pixel0, pixel1), min(max(pixel0, pixel1), pixel2))
      ;     = min(pixel0, pixel1, pixel2)
      min = ~@mf&tmp_min | @mf&tmp_med
            || *(Ga_inp1++=offset1) =w med

Loopend1:
      *(Ga_inp1--=offset2) =w min
```

4) Compute the min-value of three consecutive max-values (line0), the med-value of three consecutive med-values (line1), and the max-value of three consecutive min-values (line2). Below we call this values minmax, medmed, and maxmin. Because of the ordering according to 1) and 2) this can be obtained by comparing the consecutive words of each line. Look for example at the first two consecutive words of the maximum line: | 0 32 64 96 | 1 33 65 97| ... you can compare always four pixels by comparing the two words.

```
      ;
      ; loop for calculating
      ;                 - the min-value of max-values (line 0)
      ;                 - the med-value of med-values (line 1)
      ;                 - the max-value of min-values (line 2)
      ;

Loop2:
      ;
      ; calculation of the min-value of max-values
      ;
      pixel0 =w *Ga_inp1
      pixel1 =w *++Ga_inp1

      dummy =mc pixel0 - pixel1
            || pixel2 =w *++Ga_inp1
      ; tmp_min = min(pixel0, pixel1)
      tmp_min = ~@mf&pixel0 | @mf&pixel1
            || a15 =w &*(Ga_inp1-=[2])
      dummy =mc pixel2 - tmp_min
            || pixel0 =w *(Ga_inp1+=offset1)
      ; minmax = min(pixel0, pixel1, pixel2)
      minmax = ~@mf&pixel2 | @mf&tmp_min
            || pixel1 =w *++Ga_inp1

      ;
      ; calculation of the med-value of med-values
      ;

      dummy =mc pixel0 - pixel1
            || *(sp+[10]) =w minmax
      ; tmp_max = max(pixel0, pixel1)
      tmp_max = @mf&pixel0 | ~@mf&pixel1
            || pixel2 =w *++Ga_inp1
      ; tmp_min = min(pixel0, pixel1)
      tmp_min = ~@mf&pixel0 | @mf&pixel1
            || a15 =w &*(Ga_inp1-=[2])
      dummy =mc pixel2 - tmp_max
            || pixel0 =w *(Ga_inp1+=offset1)
      ; tmp_med = min(max(pixel0, pixel1), pixel2)
      tmp_med = ~@mf&pixel2 | @mf&tmp_max
            || pixel1 =w *++Ga_inp1
      dummy =mc tmp_med - tmp_min
      ; medmed = max(min(max(pixel0, pixel1), pixel2), min(pixel0, pixel1))
      ;        = med(pixel0, pixel1, pixel2)
      medmed = @mf&tmp_med | ~@mf&tmp_min
```

5) Compute the medians of the minmax-, medmed-, and maxmin-values which are in fact the medians of the 3x3 neighborhoods.

```
      ;
      ; calculation for next-to-the-last column
      ; (pixel2 on stack)
      ; last columns: ... 30 62 94 126 31 63 95 127
      ; so you need first column: 0 32 64 96
      ; rotated by 8 bit:        32 64 96  0

      ;
```

```
      ; calculation of the median of the three remaining pixels
      ;     minmax, medmed and maxmin
      ;

      dummy =mc minmax - medmed
      tmp_max = @mf&minmax | ~@mf&medmed
            || a15 =w &*(Ga_inp1-=offset2)
      tmp_min = ~@mf&minmax | @mf&medmed
      dummy =mc maxmin - tmp_max
      tmp_med = ~@mf&maxmin | @mf&tmp_max
      dummy =mc tmp_min - tmp_med
      block_med = @mf&tmp_min | ~@mf&tmp_med
      *La_out1++ =w block_med
```

6) Unpack the median-results and store them as output image. The packed pixels of a line: 0 32 64 96 1 33 65 97 ... will be reordered to: 0 1 2 3 4 5 6 7 ... .

7) Branch to 2) as long as there are remaining lines.

Unfortunately, the trick of the 3x3-median does not work for the 5x5-median. Therefore we compute via a simple sorting algorithm 25 values until we know the 13th largest value. Again we exploit the split ALU in order to achieve the median-values of four pixels in one run.

1) Pack the pixels that belong to the 5x5-neighborhood of four consecutive pixels in four pixel packs, that are words, in the Parameter RAM.

2) Sort the values in the packed array until one obtains the 13th largest pixels, which are actually the median values. Thanks to the packing, this sorting can be executed with four pixels in parallel. The sorting of the values in the packed array is realized through a bubble-sort method which is ideal for the parallel execution of four pixel compares in one cycle via the split ALU. The bubble sort algorithm compares always the consecutive pixel pairs and swaps them if they do not have the right ordering. In the first run you have, therefore, (n-1) compares. After one run you have the largest value of all pixels at the bottom of the array. Thus, you just have to perform (n-2) compares in the next run. After computation of the 13th largest element in the 13th run the algorithm can stop.

```
Loop1:
      ; This loop is executed 13 times - because only the 13 greatest
      ; values are required.
      ; Thus the nested loop2 is called:
      ; 24 + 23 + ... + 12 = 234 times per 4 pixels !
      ;
      ; Setup of loop-counter for Loop2:
      ; Loop counter lc1 starts with lr1 and decrements everytime
      ; the loopend register is reached. Therefore it starts with
      ; the number of loop runs minus one.
      ; lr1 = 12 => nr_of_loop1 = 13 - lc1 (Number of runs)
      ; nr_of_loop2 = 25 - nr_of_loop1 (Number of compares: starting
      ; with 24 and ending with 12)
      ; lc2 = 24 - nr_of_loop1
      ; => lc2 = 24 - nr_of_loop1 = 11 + lc1
      ;
```

```
      lc2 = 11 + lc1
      tmp_max =w *(Ga_pack = pba + PACK_ADDRESS)
Loop2:      ; this loop is executed (25 - nr_of_loop1) times
      pixel0 = tmp_max                          ; load pixel0
            || pixel1 =w *(Ga_pack + [1]) ; load pixel1
      ; pixel0 and pixel1 are consecutive pixels in the array

      dummy =mc pixel0 - pixel1
      ; Compute minimum of pixel0 and pixel1
      tmp_min = ~@mf&pixel0 | @mf&pixel1
Loopend2:
      ; Compute maximum of pixel0 and pixel1
      tmp_max = @mf&pixel0 | ~@mf&pixel1
      ; Store minimum pixel --
      ; the maximum pixel will be used in the next compare
            || *Ga_pack++ =w tmp_min
Loopend1:
      ; Store last maximum pixel
      *Ga_pack =w tmp_max
```

3) Write the median values into the output RAM.

4) If there are remaining pixels then branch to 1).

From that, you need 58.5 cycles per pixel per ADSP for this 5x5 median algorithm.

4.7 Morphology

The morphology algorithm computes gray-level morphological operations:

$$s'(x,y) = s(x,y)(\mathrm{op})m(x,y)$$

The supplied operations are erosion, dilation, erosion gradient, and dilation gradient. With this function you can extract image components that are useful in the representation and description of region shape, such as boundaries, skeletons, and the convex hull.

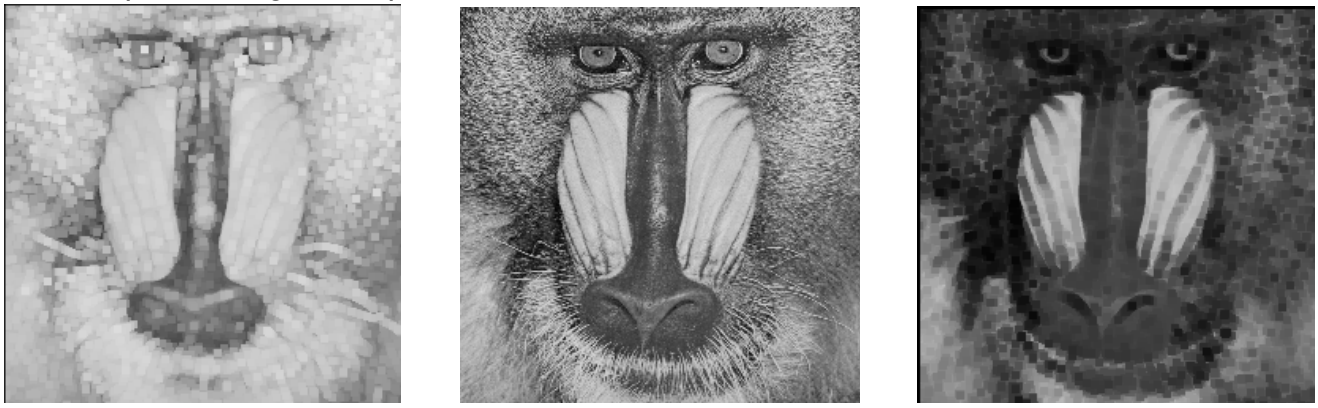


Figure 11: 5x5 Dilation and Erosion

The 3x3-erosion or Minkowski subtraction for gray-levels is defined as:

$$s'(x,y) = s(x,y) \otimes M, \text{ with}$$

$$(s \otimes M)(x,y) := \min_{(a,b) \in M} \{s(x+a, y+b)\},$$

$$\text{where} \quad M = \{(a,b) | -1 \le a,b \le +1\}$$

This formula means that you have to calculate the minimum pixels of a neighborhood of a pixel to compute its erosion value. The flow of computation is therefore:

1) Compare each column of three consecutive lines and determine the minimum value of the three column values. Store the output values in the output row.

2) Compare all triple values (three consecutive values) in the output row respectively in order to achieve the minimum values.

For the realization of the first step the splitted ALU and the masking mechanism with the expanded multiple flag register are exploited again. This mechanism has already been described in the chapter about the thresholding algorithm (chapter 4.3):

```
      ;
      ; This loop computes the minimum pixels of three input
      ; rows and stores it in one output row.
      ;
      ;
eLoop1:
      a15 =mc pixel1 - pixel2 ; Compare 4 pixels of first two rows respectively
      pixel2 = (pixel1 & ~@mf) | (pixel2 & @mf)
                              ; Compute minimum value of the first two rows
            || pixel1 =w *(Ga_pt + offset2)     ; Load pixels of third row
      a15 =mc pixel1 - pixel2 ; Compare minimum pixels with pixels of third row
      pixel2 = (pixel1 & ~@mf) | (pixel2 & @mf) ; Compute new minimum values
            || pixel1 =w *++Ga_pt               ; Load next pixels of first row
eLoopend1:
      *La_out++ =w pixel2                  ; Store minimum pixels in output row
            || pixel2 =w *(Ga_pt + offset1)     ; Load next pixels of second row
```

Evidently, this step requires 5/4 = 1.25 cycles per pixel.

For the second step we use the same masking mechanism again. Additionally the pixels are rotated appropriately so that three consecutive pixels can be compared in four pixel packages. The register `mask` has been set before by "`mask = %8`", where % is the mask generator. From that the eight least significant bits from `mask` are set. We achieve through the extended ALU operation that the most significant byte of the original register (`pixel1`) is shifted out and the most significant byte of register `pixel4` is shifted in. The EALU operation is needed in order to store the result of the rotation explicitly to register `pixel4`. Thus we have two destination registers for one ALU operation.

```
      ;
      ; This loop computes the minimum pixel of three consecutive
      ; pixels in the row with the minimum pixels respectively
      ;
                                          ; Pixels
      pixel1 =w *Ga_in++                  ; 0 1 2 3
      pixel4 =w *Ga_in++                  ; 4 5 6 7

eLoop2:
      pixel2 = pixel1\\8                  ; 1 2 3 0
      pixel2 = ealu(EA:pixel2 & ~mask | pixel4\\8 & mask || pixel4 = pixel4\\8)
                                          ; 1 2 3 4

      pixel3 = pixel2\\8
      pixel3 = ealu(EA:pixel3 & ~mask | pixel4\\8 & mask || pixel4 = pixel4\\8)
                                          ; 2 3 4 5

      ; Compare pixels 0 1 2 3 with 1 2 3 4
      a15 =mc pixel1 - pixel2
      pixel1 = (pixel1 & ~@mf) | (pixel2 & @mf)

      ; Compare pixels min(0 1 2 3, 1 2 3 4) with 2 3 4 5
      a15 =mc pixel1 - pixel3
      pixel1 = (pixel1 & ~@mf) | (pixel3 & @mf)
eLoopend2:
      pixel1 = pixel4\\16                 ; Load next pixels
            || pixel4 =w *Ga_in++
            || *La_out++ =w pixel1        ; Store minimum pixels
```

This step needs 9/4 = 2.25 cycles per pixel. Thus, we have computed the minimum pixels of all 3x3 neighborhoods of all pixels in one row. The whole erosion requires therefore 3.5 cycles per pixel per ADSP. The corresponding 5x5 filter requires 9/4 = 2.25 cycles per pixel for the first step and 16/4 = 4 cycles for the second step. Altogether, the 5x5 erosion needs 6.25 cycles per image.

The 3x3-dilation filter or Minkowski addition for gray-levels is defined similar to the erosion:

$$s'(x,y) = s(x,y) \oplus M, \text{ with}$$

$$(s \oplus M)(x,y) := \max_{(a,b)\in M} \{s(x-a, y-b)\},$$

$$\text{where} \quad M = \{(a,b) | -1 \le a,b \le +1\}$$

The flow of computation is, therefore, rather similar to the erosion algorithm flow with the difference that maximum values are needed instead of minimum values. This means that the source registers for the masking operation with the multiple flag register have to be swapped in order to obtain the maximum values instead of the minimum values. From that, the first line of the first program flow step is:

```
      a15 =mc pixel1 - pixel2 ; Compare 4 pixels of first two rows respectively
      pixel2 = (pixel2 & ~@mf) | (pixel1 & @mf)
                              ; Compute maximum value of the first two rows
```

With that, the rest of the dilation algorithm can easily be derived from the erosion algorithm. The number of cycles per pixel is exactly the same as with the erosion algorithm. For the 5x5 version of the dilation we have similar results.

All morphological algorithms can be generated as combinations of the erosion and the dilation algorithm. The next formulas show some of the possible variations:

Erosion - Gradient:

$$EG(s) := s - (s \otimes m)$$

Dilation - Gradient:

$$DG(s) := (s \oplus m) - s$$

Opening:

$$s \circ m := (s \otimes m) \oplus m$$

Closing:

$$s \bullet m := (s \oplus m) \otimes m$$

Peaks:

$$P(s) := s - (s \circ m)$$

Valleys:

$$V(s) := (s \bullet m) - s$$

The erosion gradient filter can be used for edge enhancement. It is realized through the subtraction of an eroded image from its original. The dilation gradient filter works in an analog way. Here, the original image is subtracted from the dilated image.

The morphological operation code supports basically the erosion, the dilation, the erosion gradient and the dilation gradient, since the gradient algorithms are more efficient if they are implemented explicitely. The other morphological operations can be achieved by appropriate combinations of the erosion and the dilation algorithm.

5. Results

Table 1: Number of Cycles Required in Core Loop for some Image Processing Algorithms

Algorithm	Cycles per Pixel per ADSP
Histogram Computation	3
LUT Transformation	1.5
Gray-level scaling	4
Convolution (3x3)	8.5
Convolution (5x5)	29
Median (3x3)	7.75
Median (5x5)	58.5
Erosion / Dilation (3x3)	3.5
Erosion / Dilation (5x5)	6.25

In Table 1 the required numbers of cycles per pixel in the core loops of the image processing algorithms in one ADSP are shown. Further optimizations seem to be possible -- for example for the 5x5 convolution algorithm. The revision 4.0 of the TMS320C80 will impart a mechanism for saturation through a latched overflow, which will enable further optimizations.

6. Conclusion

A basic library of image processing algorithms for the TMS320C80 has been developed. The main purpose of this library is to show some strategies how to program the MVP effectively. Apparently, the main idea is always to adapt an algorithm as far as possible to the long instruction word architecture.

The results confirm that the TMS320C80 is a dedicated processor for image processing applications. The most important features of the ADSPs for this are the splittable ALU because of the 8-bit pixels, the three inputs of the ALU because of its usefulness for masking operations, the two parallel address units, the conditional operation possibilities and the EALU with its ability to perform 256 different logical and 256 different arithmetical operations and, therefore, reducing the number of required instructions for an algorithm.

References

1. R Chellappa (Ed.); Digital Image Processing; Los Alamitos 1992
2. R. Gonzalez, R. Woods; Digital Image Processing; Reading 1993
3. P. Maragos, R. Schafer; Morphological Systems for Multidimensional Signal Processing; in [1]
4. A. Rosenfeld, A. Kak; Digital Image Processing; New York 1976
5. TMS320C80 (MVP) User's Guide; Texas Instruments 1995