Int. J. Appl. Math. Comput. Sci., 2008, Vol. 18, No. 1, 105–110
DOI: 10.2478/v10006-008-0010-2

amcs

REMARKS ON HARDWARE IMPLEMENTATION OF IMAGE PROCESSING ALGORITHMS

MAREK WNUK

Institute of Computer Engineering, Automation and Robotics
Wrocław University of Technology
ul. Janiszewskiego 11/17, 50–372 Wrocław, Poland
e-mail: marek.wnuk@pwr.wroc.pl

Image processing in industrial vision systems requires both real-time speed and robustness. Modern computers, which fulfill the first demand, are sensitive to hard industrial environment conditions and require considerable amounts of energy. Programmable logic chips are available, which can realize many simple, still time-consuming operations in a parallel or a pipelined manner. The paper discusses particular features of the pipelined architecture and presents selected techniques of implementing early image processing procedures in hardware.

Keywords: On-line image processing, real-time, hardware, pipeline, programmable logic.

1. Introduction

Industrial vision systems are, in most cases, designed to work on-line, as real-time systems. Software implementation of early processing procedures requires computers (microprocessors, DSPs) of great computational power, which work with high clock frequency and hence are very sensitive to hard industrial environment conditions (temperature, electromagnetic noise, etc.). Moreover, the higher the clock speed, the higher the power demand and dissipation. On the other hand, most of the required procedures can be implemented in hardware, using programmable logic chips.

Image acquisition in most cases (CCD, CMOS sensors) is based on line-by-line scanning of the image plane with a constant pixel rate. This results in an input data stream of limited speed. For example, standard VGA-size sampling of a typical CCTV image (640×480, 25 frames per second) requires the sampling rate of approximately 11MHz.

In order to preserve real-time system constraints, a vision system has to guarantee an explicitly defined worst-case latency during on-line image processing.

Many image processing algorithms are based on local image features, which requires simultaneous access to many input image pixels, forming the neighborhood, in order to calculate the result for a single pixel of the output image. For example, in the case of the 5×5 (radius $r = 2$) neighborhood, meeting the 11MHz pixel rate requires at least 275 MHz memory access rate (assuming sequential readouts).

Moreover, the complexity of many procedures increases the requirements for computational power. Contemporary DSPs offer enough speed and architectural features for image processing (DMA, multiple cores, vector processing, etc.), but still at the cost of high clock rates, power consumption and unit price of the devices.

The solution based upon hardware implementation of image processing algorithms is free of the above drawbacks. Using programmable logic devices (FPGA) is a cheap and easy way to build dedicated processors for many widely used image transformations. The solutions are flexible, in contrast to the early implementations (Drzazga *et al.*, 1983), as FPGA based implementation is fully programmable. Moreover, many powerful design tools are available, which makes the development process fast and effective. For example, Spartan-3A DSP FPGA (Xilinx, 2007) offers 53712 Equivalent Logic Cells and 126 DSP48A slices (enhanced MACs) at 250 MHz with a very good price/performance ratio.

2. Idea of pipelined image processing

Consider a memory based image processing system which implements local operators, defined for a given neighbor-

hood of the currently processed image pixel (Fig. 1). Assuming the region of interest $W \times H$ inside the framebuffer of line width L, a standard way of accessing pixel $f(j, j)$ is to calculate its address as shown in the figure. Many processors (especially DSPs) provide memory address generator blocks, facilitating this task. Nevertheless, a more effective way is to set up a pointer to the memory and provide consecutive accesses with auto-postincrementation (available in most advanced processors). This resembles the situation when we receive the image as a sequential data stream (e.g. from CCD or CMOS image sensors or USB/Firewire/Ethernet devices).

In a general case, the local operator calculates the resulting value of the pixel $g(i, j)$ on the basis of the values of all the pixels from the given window, accessible with constant offsets from the current (central pixel) pointer. Actually, there is no need for multiple pixel access. As-

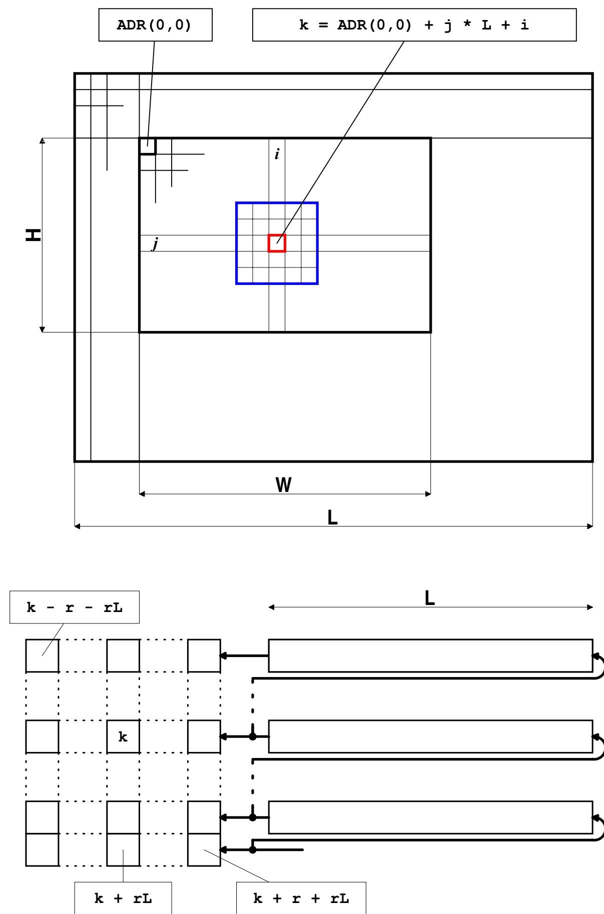


Fig. 1. Pipeline structure.

suming an 8-connected neighborhood of radius r (a square window of the size $(2r + 1) \times (2r + 1)$), one can create a pipeline consisting of $2r$ delay lines (SISO registers) of the image line length L (Fig. 1). The currently accessed (received, in the case of a serial input data stream) pixel, together with the outputs of all $2r$ delay lines, forms one column of the requested window (the data are accessible in parallel). In a general case, a $(2r+1)\times(2r+1)$ array of additional pixel-size registers (forming $2r + 1$ SIPO row buffers) provides simultaneous access to the surrounding pixels. The delay T_D introduced by such a pipeline depends on pixel sampling period T_S, image line period T_L and the neighborhood radius r:

$$T_D = rT_L + (r + 1)T_S.$$

In the case of a line containing L pixels with no blanking period, we have

$$T_D = T_S(r(L + 1) + 1).$$

Any local operator can thus be implemented as a static function Φ of multiple inputs and one output:

$$g(i, j) = \Phi\left(f(i + m, j + n) \middle| -r \le m \le r\ ,\ -r \le n \le r\right).$$

The output $g(i, j)$, delayed from the original data stream by T_D, can be used as input data for the next processing stage of the same form. The delays of the cascaded procedures accumulate, but the overall latency remains strictly defined and constant.

Operators that use only one pixel value to perform the transformation can be considered as a special case of the local ones, with the neighborhood radius $r = 0$. The implementation is much simpler, as the delay lines are not needed and we use only one input. Typically, such transformations are realized via programmable LUTs (*LookUp Tables*), memory arrays addressed by the input value and containing the output values for all possible input values.

The presented implementation concept is suitable for a great variety of early image processing (linear and non-linear): filtering (hi- and low-pass, gradients, edge enhancement, background subtraction, etc.), segmentation (thresholding, clipping, double thresholding, template matching, etc.), morphology (hit-or-miss, dilation, erosion, opening, closing, etc.), parameterization (labeling, moments, moment invariants, etc.). The implemented procedures can be cascaded and combined parallely, forming fast image preprocessing systems, well suited to a given task.

Note that a brute force implementation of Φ is not always efficient, or even possible. Even in the case of the smallest non-trivial $3 \times 3 (r = 1)$ neighborhood and 8-bit gray-scale image, Φ requires a 72-bit input word. Good results can be obtained via the decomposition of the operator, which will be shown next.

3. Separable operators

A special class of local operators (both linear and nonlinear) are separable ones. The problem size decreases significantly if the operator Φ can be decomposed in such a

manner that every column is processed independently and the partial results for the columns are composed to form the result.

Consider the so-called Gaussian filter defined by the convolution kernel:

$$\begin{bmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{bmatrix}.$$

It is widely used for image smoothing (low-pass filtering), as it is easily normalized by 16 (bit shift instead of division). Moreover, to obtain a Gaussian filter of a greater radius, we can compose (cascade) two Gaussian filters:

$$G_{r1} \star G_{r2} = G_{r1+r2},$$

which implies that it is sufficient to implement G_1.

Introducing a three-input operator Γ:

$$\Gamma(a, b, c) = a + 2b + c$$

we can calculate partial results $\gamma(i, j)$ for the consecutive columns of the neighborhood, storing them in a single SIPO buffer (Fig. 2):

$$\gamma(i, j) = \Gamma(f(i, j-1), f(i, j), f(i, j+1)).$$

The final result $g(i, j)$ is calculated in another Γ block, using $\gamma(\cdot, j)$ as inputs:

$$g(i, j) = \Gamma(\gamma(i-1, j), \gamma(i, j), \gamma(i+1, j)).$$

Instead of a function with a 72-bit input, we need two copies of a function with a 24-bit input.

Local minimum and local maximum operators on large windows are used for finding lower/upper image envelopes:

$$l_{\min}(l_{\max}) \ , \ l_{\max}(l_{\min}),$$

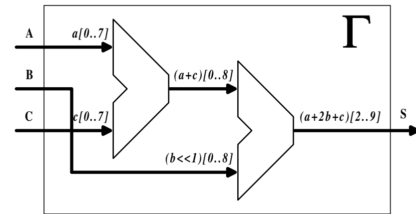
which are very useful in background subtraction methods. Both the operators are separable. Every neighborhood column can be minimized in the pipeline:

$$\mu(i, j) = MIN_{2r+1}\Big\{f(i, j-r), \ldots, f(i, j), \ldots, f(i, j+r)\Big\},$$

and the final result $g(i, j)$ is the minimum of $2r+1$ partial results (Fig. 3):

$$g(i, j) = MIN_{2r+1}\Big\{\mu(i, j-r), \ldots, \mu(i, j), \ldots, \mu(i, j+r)\Big\}.$$

Thus the decomposition results in reducing the problem size from $(2r + 1)^2$ to $2(2r + 1)$.



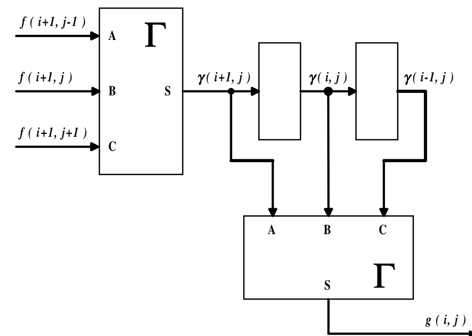


Fig. 2. 3×3 Gaussian filter.

4. Median filter: A special case

The median operator replaces the original pixel value by the median of the surrounding pixel values. A typical neighborhood size varies from 3×3 ($r = 1$) to 7×7 ($r = 3$). It is particularly useful for suppressing impulse noise, as it rejects extremal values from the sampled window. Preserving step and ramp functions minimizes image blurring, but results in poor efficiency in the case of additive (e.g., Gaussian) noise.

The described hardware implementation is based on an algorithm by Jeremiah Golston, included in a software library for a TMS320C8x DSP family (Texas Instruments Europe, 1997). The main block is a 3-input sorter ORD_3 (Fig. 4). It returns the minimal input value on output N, maximal on X, and median on D. The algorithm works for the 3×3 neighborhood only. In the case of nine values, the median cannot be greater (nor less) than five or more values in the window. Finding the median value reduces to rejecting pixel values which do not meet the above conditions.

In the described pipeline scheme, we sort consecutive columns $f(i + 1, \cdot)$ with the ORD_3 block (Fig. 4). The results (N, D, X) are pushed into three SIPO registers and thus are available simultaneously for three consecutive columns. From the minimal values (row N) we can reject the minimum value $N(N)$, which is the global

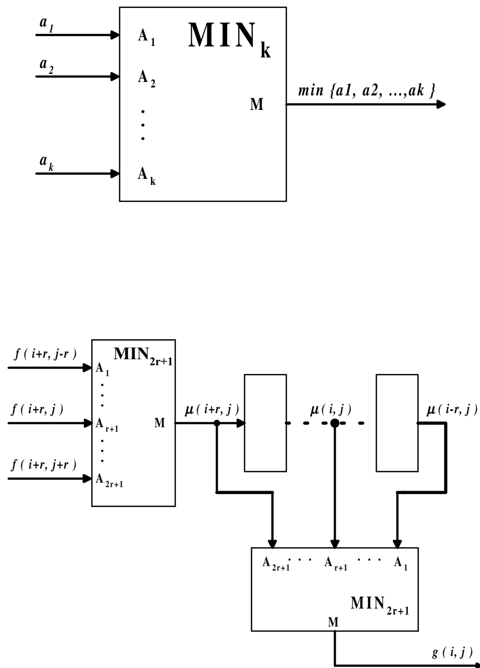


Fig. 3. Local minimum filter.

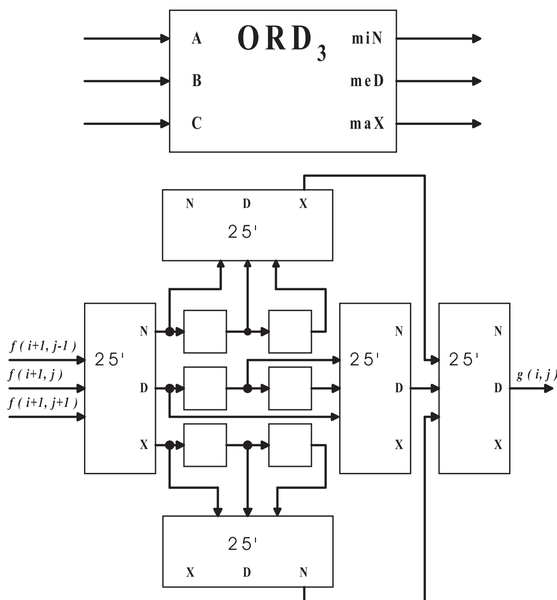


Fig. 4. 3×3 median calculation.

minimum in the neighborhood. Moreover, for the median value of this row ($D(N)$) there exist at least five pixels with greater or equal values (two in its own column and three in the maximal column). The only pixel left for further consideration is the X output of this block ($X(N)$). The calculations for the row X are dual and result in leaving only the pixel $N(X)$. For the median row D, the maximum value $X(D)$ is greater than five pixel values (row N in its column and rows D and N in the other two). Dually, the minimum value of $N(D)$ is rejected, leaving only the median value ($D(D)$) for sorting. The last ORD_3 block determines the median value of the candidates, which gives the final result $g(i,j)$. The actual implementation requires fewer comparisons than five full ORD_3 blocks, as three of them use only one output. Note that the median is a non-linear operator and hence the cascading of medians gives different results than using a higher order median.

5. General convolution filters

Linear local operators are in general performed by convolution with a given kernel of radius r. Consider the one-dimensional example

$$g(i) = \sum_{k=-r}^{r} a_k f(i+k),$$

where a_k are kernel elements (weights).

In Fig. 5 two realizations of the convolution are shown. The first one reflects directly the definition. It requires an SIPO register of the length $2r$, for consecutive $f(k)$ values, $2r+1$ multipliers with predefined (programmable) weights a_k, and an adder with $2r+1$ inputs.

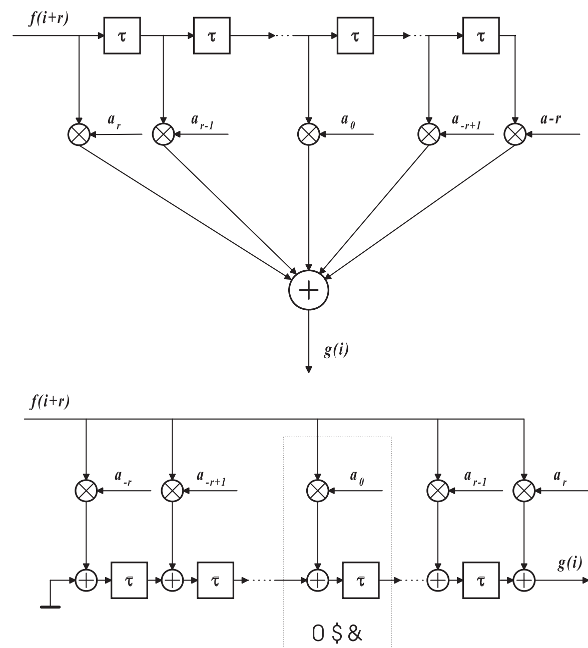


Fig. 5. Convolution and MAC blocks.

Alternatively, we can define the convolution as fol-

lows (Fig. 5):

$$g(i) = a_r f(i+r) + (a_{r-1} f(i+r-1) + (\cdots + (a_{-r} f(i-r) + 0))\dots),$$

which can be realized by MAC (*Multiply and ACcumulate*) blocks:

$$M_k = a_k f(i+k) + M_{k-1}$$

for $k \in [-r, r]$ and $M_{-r-1} = 0$.

In this case we need $2r + 1$ MACs, each consisting of the multiplier and a two-input adder. The shift register stores accumulated partial results M_k, rather than $f(\cdot)$ values, and is distributed among MACs.

The second implementation is easily scalable and particularly good for both software and hardware implementation, as MAC processing units are available in all DSPs and many FPGAs.

A good example of a MAC-based hardware pipeline convolver is an IMSA110 integrated circuit (SGS-THOMSON, 1994) (Fig. 6). It contains three delay lines

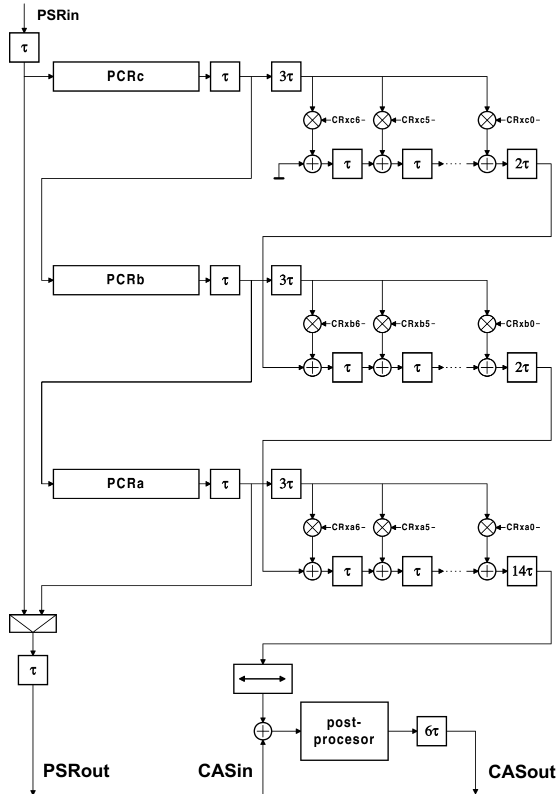


Fig. 6. IMSA110 convolution processor.

PRC_y of programmable length (0–1220), and a 3×7 array of MAC blocks with programmable weights $CRx_{y,i}$ (where $y \in [a, c]$, $i \in [0, 6]$). The input is *PSRin*, and the output signal is available on *CASout* after a barrel shifter and a post-processor. The additional multiplexer, the *PSRout* output, and the *CASin* input provide a possibility of cascading the convolvers in order to increase the MAC processing array.

Programming the delay lengths, the weights, and the barrel shifter is available via a parallel microprocessor interface. The coefficients ($CRx_{y,i}$) are 8-bit signed values and the barrel shifter provides division/multiplication by a power of 2 in the bit range $(-2, +14)$. The implementation of the convolution kernels for many linear operators requires taking the above constraints into account. For example, a simple 3×3 averaging kernel:

$$\frac{1}{9}\begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix}$$

can be approximated by

$$\frac{1}{2^9}\begin{bmatrix} 57 & 57 & 57 \\ 57 & 57 & 57 \\ 57 & 57 & 57 \end{bmatrix}.$$

The normalization of the operator (division by 9) was replaced by shifting the result to the right by 9 bits (division by 512). Appropriate weights ($-128 \le 57 \le 127$) were applied.

Another example can be a rotation-invariant 3×3 Laplacian:

$$\frac{1}{6}\begin{bmatrix} 1 & 4 & 1 \\ 4 & -20 & 4 \\ 1 & 4 & 1 \end{bmatrix} \approx \frac{1}{2^5}\begin{bmatrix} 5 & 22 & 5 \\ 22 & -108 & 22 \\ 5 & 22 & 5 \end{bmatrix}.$$

The post-processor contains a programmable LUT, which offers a possibility to implement linear and non-linear point-based operations (e.g., negation, gamma correction).

6. Cumulative image parameters

Calculating global image characteristics (histograms, moments of inertia, etc.) requires pixel-by-pixel image reading and the accumulation of the calculated parameter. With no special effort this can be done during the input data stream reception. For example, histogramming requires a one-dimensional array of counters, addressable with the input pixel value. A two-dimensional histogram (neighborhood matrix) will require a pipelined arrangement and a two-dimensional addressable counter array. In some cases (mean value, standard deviation, etc.) a kind of post-processing may be required, which is performed once, at the end of the image frame.

A good example of cumulative parameters can be moments of inertia. In a standard application, these parameters provide a description of both the shape and the location/orientation of a silhouette represented by a given value (e.g., 1) on the segmented and labeled image. On the basis of moments up to the second order, it is easy to find the location (centroid) and orientation (principal axis direction) as well as several parameters, which are position-, scale- and orientation-invariant (Dudani *et al.*, 1977).

Standard moments of order $p + q$ are defined as follows:

$$m_{p,q} = \sum f(x, y)x^p y^q.$$

Substituting $x^p y^q$ with $r^{p,q}_{x,y}$, we obtain

$$m_{p,q} = \sum f(x, y)r^{p,q}_{x,y}.$$

Such a representation leads to the calculation of pipelined moments. For the moments of the second order, we get

$$r^{2,0}_{x+1,y} = (x + 1)^2 = x^2 + 2x + 1 = r^{2,0}_{x,y} + 2x + 1,$$

$$r^{1,1}_{x+1,y} = (x + 1)y = xy + y = r^{1,1}_{x,y} + y,$$

$$r^{0,2}_{x,y+1} = (y + 1)^2 = y^2 + 2y + 1 = r^{0,2}_{x,y} + 2y + 1,$$

and we can iteratively calculate consecutive values of $r^{p,q}_{x,y}$ with no multipliers.

The structure of the second-order moment calculator is shown in Fig. 7. Auxiliary elements $r^{2,0}, r^{1,1}, r^{0,2}$ are accumulated in three iterative adders using x and y coordinates available in *modM* and *modN* counters. S_P is the pixel synchronization clock, S_H denotes the line (horizontal) synchronization, and S_V means the frame (vertical) synchronization. The second row of iterative adders is gated by the (current pixel value) input $f(x, y)$ and thus accumulates the second-order moments $(m_{2,0}, m_{1,1}, m_{0,2})$. In the case of a labeled image, several copies of the second row of adders are used, each gated by a selected label value. The results are valid at the end of a frame and should be stored before the edge S_V (not shown for clarity).

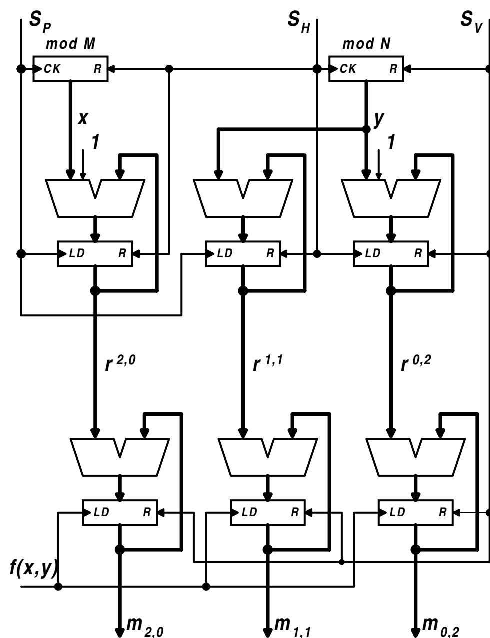


Fig. 7. Calculation of second-order moments.

Moments of the zeroth (histogram of the labeled image) and first (gated accumulation of the x and y coordinates for desired label values) orders are trivial.

7. Conclusions

The pipelined architecture implemented in hardware, especially in programmable logic devices, provides a constant, strictly defined latency of the image processing path, which fulfills the main condition of real-time systems.

The cost of the image processing hardware is relatively low, and will decrease with FPGA chips enhancement. Implementing the procedures is well supported by widespread design tools (VHDL compilers, libraries, etc.).

Low power consumption and small size of the devices encourage constructors to put the preprocessor into the image sensing unit. In the case of remote vision systems, this can lead to reducing the bandwidth between the vision-based sensor and the host (e.g., a robot controller).

Moreover, the possibility to implement selected microprocessor and DSP cores in FPGA provides means of implementing the required low level post-processing and additional, high level procedures (image analysis, pattern recognition, etc.).

References

Drzazga A., Hajdul J., Malec J. and Wnuk M. (1983). Hardware image preprocessor, *Technical Report*, Wrocław University of Technology (in Polish).

Dudani S., Breeding K. and McGhee R. (1977). Aircraft identification by moment invariants, *IEEE Transactions on Computers*, **26**(1): 39–46.

SGS-THOMSON Microelectronics (1994). *IMSA110 Image and Signal Processing Sub-system*, http://www.datasheetcatalog.com.

Texas Instruments Europe (1997). *Implementation of an Image Processing Library for the TMS320C8x*, BPRA059, http://www.datasheetcatalog.com.

Xilinx, Inc. (2007). *Spartan-3A DSP FPGA Family: Complete Data Sheet*, DS610, http://www.datasheetcatalog.com.