LIPP - A SIMD MULTIPROCESSOR ARCHITECTURE FOR IMAGE PROCESSING*

by T. Ericsson and P.E Danielsson

Department of Electrical Engineering, Linköping
University, Linköping, Sweden

Abstract

LIPP (Linköping Image Parallell Processor) is a multiprocessor system intended mainly for image analysis and image processing but even other computing tasks where large amount of data should be manipulated in forms of matrices, such as weather forecasts or other related problems namely systems of differential equations. The processors within the processor array are of bit-serial type with the capability of directly processing data with wordlengths in the range of 1 bit to 32 bits in one bit increments without time penalty. Bit-serial operation gives the possibility of designing suprisingly fast algorithms. To each processor is a fairly large memory (64 Kbit) associated. A processor can instantly reach 8 neighboring memories through an interconnecting network. The processor array whose size is thought to be 16 by 16 it running in SIMD mode. In this way memory access collisions can be minimized. Image and matrix data are mapped in the memory space so that each memory holds a subimage. We call this mapping distributed processor topology. Because of the memory mapping and interconnection network neighborhood operations such as two dimensional convolution are easily performed.

Introduction

Image processing is a constant challenge to conventional computers for reasons of speed. The natural answer is parallelism. As have been shown in [1,2], for neighborhood operations, four different forms of parallelism are possible: Operator, Image, Neighborhood and Pixel-bit parallelism. Consider the fact that the size of a task varies from one operation up to maybe 100, that the neighborhood kernel size goes from one to maybe 100 points and that the number of bits per pixel varies from one (for binary images) to maybe 100 (for complex multispectral images). We conclude that we shall put our money on what we call image parallelism only.

* This work is supported by the Swedish National Board for Technical Development.

This implies a SIMD-system where several bit-serial processors execute the same operation on the same image or images but on different neighborhoods. No pipe-lining (only one level of operation going on at the same time), no parallel fetch of neighborhoods, not even one pixel but one bit is to be fetched at a time to each processor.

Like in other contemporary designs of this kind, e.g. CLIP 4 [3] and MPP [4] we assume a central control unit that generates and broadcasts a global address and control word. This information is picked up by each unit in the two-dimensional array O, that is by the memory modules (M:s) and the processors (P:s) respectively.

An extremly important difference between the above-mentioned machines and the LIPP-proposal of ours concerns the image-to-memory mapping for the case (overwhelmingly common) that the number of pixels in the image is larger than the number of array points (M:s and P:s). In most machines this is done in the fashion of Figure 1a) which shows a case of an 8 x 8 image mapped on a 4 x 4 array.

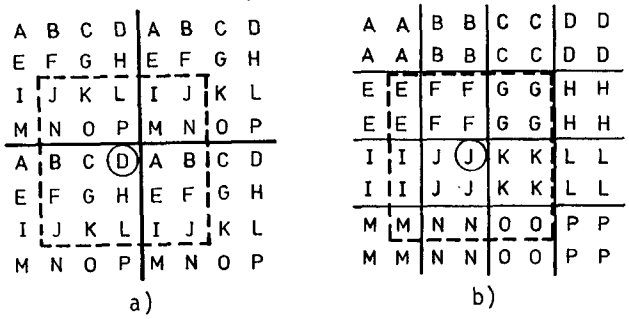


Figure 1 The two mappings of image array

Here the 16 M:s (A,B,C,...,P) are all distributed over each 4 x 4 subimage. A 5 x 5 neighborhood is covering all 16 memory modules as seen for the D-pixel at the encircled position. So, the simultaneous and direct access for all processors to a certain pixel in their respective neighborhoods requires that all processors are connected to all memory modules, or at least up to a radius corresponding to the neighborhood of maximum size. For an array size of 16 x 16 or larger and a neighborhood size of, say, 9 x 9, there would be required a form of 81-connectedness which is impossible to implement as a wiring scheme.

The memory mapping showed in figure 1b) the neighborhood access problem is much more relaxed. Each of the 16 memory modules holds a subimage (in this simplistic example only 2 x 2 pixels). We call this mapping distributed processor topology. It was first suggested for DAP [5] but its processors are not able to access data from all nine neighboring processor memories, which is essential when doing neighborhood operations without need of copying image data.

The image is mapped on the memory array in such a way that each memory contains a subimage of the whole picture as shown in Figure 2. A subimage is marked by M in the figure and the processor array can be seen as being moved over the image plane by the global control unit. The size of subimages varies with memory and processor array size and size of image as shown in the following table.

Image size	Array size	Subimage size	Max-neighborhood size
512x512	16x16	32x32	65x65
512x512	32x32	16x16	33x33
512x512	64x64	8x8	17x17

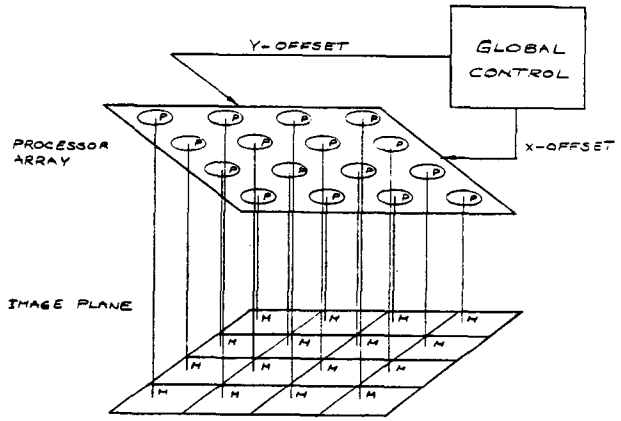


Figure 2 Processor - Image interconnection

The problem of large neighborhood access is not the only contest where distributed processor topology as in 1b) comes out as a winner over the mapping of 1a). Even for the limited size of a 3 x 3 neighborhood, an access problem arises in Figure 1a). In this example, the 16 processors are first processing, say the top leftmost quadrant. The border processors D, H, L, P, O, N, M need data from neighboring quadrants of the image. In the physical array structure we therefore need to wrap around the plane in both directions creating a torus-like shape. Unfortunately, the global common address is not common anymore when the border processors are trying to fetch data across the quadrants while other processors fetch data inside the present quadrant.

In contrast, the scheme of Figure 1b) allow us to store and process images of different sizes quite freely. Each module stores a N/m x N/m subimage of each image. In each module the subimages from different images are mapped into one linear address space. The global address is a pointer in this space of, say, 64 Kbit.

The necessary interconnecting data paths can be implemented in different ways, as was shown in [2]. One of these is shown in Figure 3. Note, that the main purpose is to make it possible for the global control unit to steer data from a memory module to its processor (the normal case) and in the same moment feed data from say South-East to the processor in North-West. This case opens up the dashed data-paths in Figure 3.

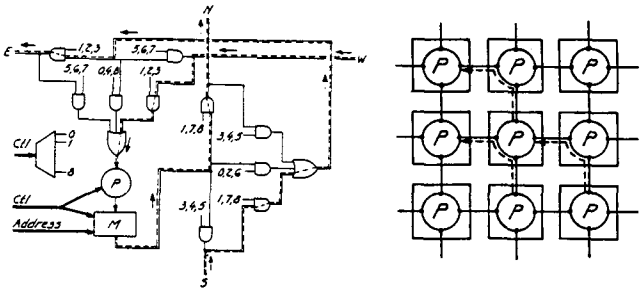


Figure 3 Two-level multiplexing of datapaths and the resulting global interconnection scheme

Input/output methods

Problem statement

All designs of processor arrays requires special attention when it comes to input and output of image data.

The problem we have to solve in LIPP arises from the fact that image data outside the array appears in raster scanned serial format. Assume that an image arrives in TV-fashion, right-left, top-down. Then, with our image to array mapping of Figure 1b) all the leftmost pixels of the first lines shall be stored in module A. Without precautions the I/O datarate would be limited by the small bandwidth of a single bit/serial memory module. Input/output to the parallel array would be a strictly bit-serial procedure.

For LIPP we will see at least three alternative solutions:

- horizontal/vertical unscrambling
- orthogonal memory with flip network
- orthogonal memory with word- and bit access.

In this paper we only discuss the first alternative.

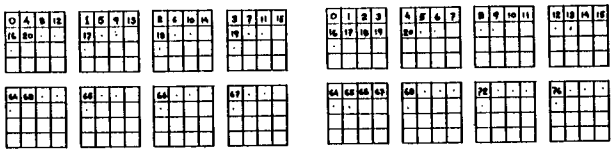
Horizontal/vertical unscrambling

Let us assume that our array has vertical and horizontal one-bit wide buses (H- and V-buses respectively) (highways in DAP [5]) and a control mechanism that allow us to transport data from the edge to a selected row of memory modules (using the V-buses) or a selected column (using the H-buses). A traditional serial-to-parallel converting buffer register is connected to the horizontal as well as the vertical edge.

Assume a binary 16 x 16 image, a 4 x 4 array and that the horizontal buffer register of four pixels has been loaded with input data. Then, we

could transport the set of four contiguous pixels (= four bits in this case) over the V-buses and let the whole image be stored in the way of Figure 4a). Each line of data, e.g. the first line 0, 1, 2, ..., 15 is stored in the correct row of modules. However, compared to the final destination, shown in Figure 4b) the bits are heavily scrambled within each line.

Fortunately, the scrambling pattern is identical in each line. Therefore, by utilizing the H-buses we could do one read column-wise of four pixels and store them in the vertical edge buffer. Then we store them back in their final destination. To avoid over-writing we have to make an initial move of each line to a buffer area in the memory modules. However, these moves can be made with full m x m parallelism and consequently with negligible time consumption.



a) b)

Figure 4 Vertical input, horizontal unscrambling

The great advantage of this method is that no extra hardware has to be introduced. For a 16 x 16 array an 512 x 512 x 8 bit image would require 40 ms which is exactly the frame time in European TV-standard.

An extension of the idea is to use 8 bit wide V- and H-buses. To save lines and IC-pins we also intend to use the buses for global address distribution. In this case the time for transferring the picture mentioned above is 8.3 ms.

The over-all architecture of LIPP

With the adoption of the first Input/Output alternative of previous section a overview of LIPP takes the form of Figure 5.

In the center is the m x m array with memory-processor arrangements in principle organized as in Figure 3. The edge registers are to right and bottom of the array. Some but not all of the global control is shown. Global addresses and constants are issued over the V- and H-buses. One of these lines is used for gating. When the whole array is activated both decoders enable all their m control lines. However, when one row or one column of M/P-units are to be activated only one of the corresponding decoder outputs is enabled.

Each processor can deliver a data-dependent interrupt signal as will be shown in the next section. These are OR-ed together in each row and independently in each column. An m-bit flag register for both coordinates can be set and delivered to the hostcomputer. If only one processor has delivered an interrupt it can be identified immediately, addressed by proper column and row select and data delivered at the feature output. If more than one processor signals an interrupt a more elaborate search process has to take place.

The horizontal bus lines combines to the right in an adder tree. Hereby fast accumulation of counts can be obtained, most typically histograms. Histogramming is extremely common in most image processing applications. It consists of collecting the distribution of gray levels throughout the image. In LIPP each M/P-unit is first creating a local histogram for the subimage of its own, utilizing the table look-up function to be explained in the next section. Second, corresponding table entries from the local histograms are accumulated over the adder tree. The speed is illustrated by the following example.

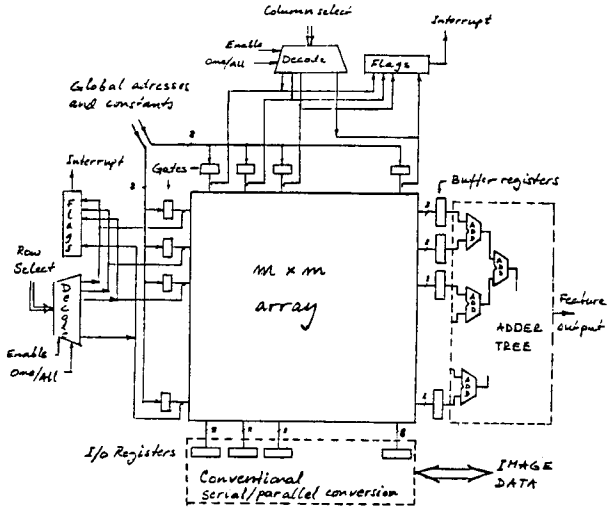


Figure 5 The over-all architecture of LIPP

An image of 512 x 512 x 8 bit processed with a 16 x 16 array requires a 256 x 11 bit table for each local histogram. These are accumulated in parallel over each subimage in (32 x 32) x (8 + 11 + 11) cycles (subimage size is 32 x 32, 8 cycles for serial read out of a pixel, 11 + 11 cycles for serial read-out and storing back of a table entry). The local histogram entries are then byte-wise transported to the merging tree in 256 x 4 x 16 cycles. The total amounts to 30.000 + 16.000 cycles or 4.6 ms assuming a 10 MHz cycle rate.

The control unit is invisible in Figure 5. It generates the common addresses to processor memory array and issues the current ALU function and other control signals essential to all processors. Needless to say, the proper design of this unit is crucial to the success of LIPP. One pit-fall would be to design fast operations in the array that depend on a massive microprogram support. The compilation of the microprogram the fetching of it and finally its interpretation (address and control vector generation) must be fairly swift and simple. Otherwise we might have an unbalanced system that is mostly waiting for the control unit to get ready. For the sake of brevity we obstain from further discussions on this point.

Processor design

From the previous discussions we conclude that the processor should be designed for bit-serial arithmetic and logic. Several surprisingly fast algorithms can be designed that exploits bit-serial

access and processing as will be presented in the next part.

The following list of desirable features are incorporated in Figure 6 of such a processor.

* SIMD processing mode implying that the processor need no control unit of its own.
* Bitserial accumulator/subtractor and logical unit (ALU).
* Internal shiftregisters with variable length.
* Dynamical variation of length of the shiftregisters without loss of stored data.
* Index register for table lookup.
* Shiftable (in/out) up/down counter with status information.
* Status registers for interrupt signalling.
* Activity register for data dependent processing.
* Direct data paths to four fairly large memories (16-64 Kbit).
* Neighborhood data access transparent to processor activity.

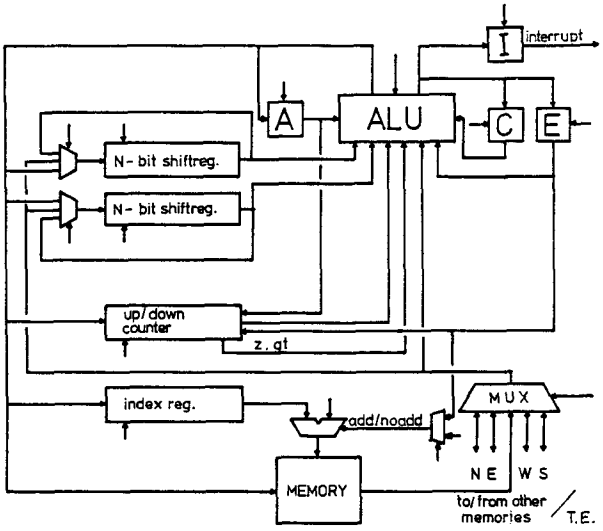


Figure 6 The processor

Arrows throughout Figure 6 indicate control information data paths from a global control unit. It is hard to get all these control signals transferred in each clock cycle due to their large quantity. The amount of these signals can be reduced by observing that several of them are mutual exclusive. All such signals can be grouped and coded.

Another way to reduce the amount of control signals is to distribute some functions among the processors. Each processor is then equipped with a small micro program controller and is thus able to perform suboperations individually. If the global addresses are given by the master controller, common to the whole array, it is possible to partly run the array in a MIMD mode without memory accessing collisions. Even slight local modifications of addresses within a given space can be tolerated.

Interrupt signalling through register flip flop I, see Figure 6, indicates completion of a suboperation. Register I can also be used for arithmetic overflow indication and for object finding purposes in segmentation algorithms.

Activity register, A, is employed in data dependent computations. An example of which is the logarithm algorithm implementation in part 5.2 below.

Register C is used for storing the carry bit produced by the ALU in for the instance the add/subtract operation.

Register E is also employed in the data dependent computations of the logarithm algorithm, refer to part 5.2 of this paper. In this example the counter is only used as a register. The special features of this counter are exploited in median or percentil calculations. This counter can also perform convolution. The signal from the enable register E is used as count enable and the data path from the ALU is used for presetting the counter as well as certain possibilities to alter the operation of the ALU and modifying the address to the memory. Data can be loaded into the counter directly from memory via the ALU. The global control information determines the status of count up or down.

The processor is equipped with two shiftregisters of changable length in order to store intermediate data. These registers should be able to dynamically vary their lengths without loss of stored data. By having this facility it is for example possible to reduce the amount of clock cycles needed for square root extraction by almost fifty per cent.

Figure 6 indicates only one single memory unit. This is the only memory that is allowed for write operations. It contains different subimages and look-up tables. This means that each processor in the array could have different tables, but in most cases the information stored in the different tables is the same. Memories should be large enough to store several subimages and tables. A large number of operations can thus be made without extensive input/output operations.

When often the same large table is needed one may think of attachment of a ROM to the processor, in which several tables can be found.

The inter-connecting network between neighboring memories is represented by the MUX symbol in Figure 6. An implementation is shown in Figure 3 of this paper. Neighboring memories can be reached without time penalty through these bidirectional multiplexers.

The multiplexer can be thought of as being a part of either the memory or the processor. Now,

production of memory chips requires specially tuned processes and layout techniques, which means that it is hard to add extra features to a ready-made design. Therefore it is better to incorporate the multiplexer function into the processor design.

Input and output data paths have not been included in Figure 6.

Local indexing in a look-up table is a very versatile operation as seen by the following important applications.

* Arbitrary grayscale transformation.
* General (logical) neighborhood mapping.
* Distributed arithmetic (for convolution).
* Histogramming and feature counting.

The index register is loaded serially from one of the connected memories via the ALU. However, the register is read in parallel and fed to the adder which is activated by the add/noadd control signal if indexing shall take place. In this way we calculate the address within a single cycle. However, we then have to (optimistically?) assume that index add, memory cycle and processor cycle all can take place in one machine cycle of 100 ns.

Algorithms

In order to design a well balanced system we have to tune it with respect to different tasks and algorithms which we will run on the system. Our main objective is to use it in different image processing applications. However, but other type of applications which could be formulated in terms of algorithms acting upon matrices of data may also be applicable. In the following we give an example of a number representation which is handy for some image processing applications. Besides algorithms for basic algorithm this representation calls for algorithms to directly perform basic arithmetic (addition, multiplication) and conversion algorithms between different representations. The performance of this algorithms are stated below.

More algorithms are neccessary for a complete system and have also been investigated [9] [10]. However, we feel that the bitserial algorithms presented below may be representative for the very often unconventional approach called upon by this architecture.

Signed logarithm number representation

In most applications of image processing the precision need is not very high because of relatively low signal to noise ratio $\sim 10^3$ of commonly used image sensors (CCD arrays and vidicons). In fact 8 bit or in some cases 16 bit integers are in use extensively. When using integers algorithms have to be designed carefully with intevening scaling operations to overcome the possibility of overflow. A signed logarithm number system, suitable for low precision applications, have been proposed by Swartzlander and Alexopoulos [6] that need no scaling.

The representation uses logarithm of base 2 and a multiplicative factor to always keep the logarithm positive. Because of this factor we have to do some adjustments of the result after a multiplication or division. We therefore suggest a twos complement representation of the logarithm which do not need the scaling factor. A logarithm of base 2 gives a poor accuracy. A base of $256\sqrt{2}$ has been suggested by Tood [7] which extends the accuracy at the sacrifice of dynamic range. For a 16 bit representation including sign bit we get an accuracy of $\pm$ 0.14 % and a dynamic range of $3.4 * 10^{38}$. We suggest the following representation of a floating point number ξ with sign S_ξ and logarithm L_ξ.

$$\xi = m \cdot 2^e \tag{1}$$

$$S_\xi = 1 \quad \text{if} \quad \xi \leq 0 \tag{2}$$

$$S_\xi = 0 \quad \text{if} \quad \xi > 0 \tag{3}$$

$$L_\xi = \log_\beta (|m|) + e \cdot \beta \quad \text{if} \quad |\xi| \geq \rho \tag{4}$$

$$L_\xi = \log_\beta (\rho) \quad \text{if} \quad |\xi| < \rho \tag{5}$$

ρ is the smallest number that can be represented in this form.

Multiplication and division in this representation are both easily calculated. Addition and subtraction are however more difficult to compute. A method for this is found in [6], which gives as follows in case of addition.

$$L_{a+b} = L_a + f_\beta (L_b - L_a) \quad \text{if} \quad |a| \geq |\beta| \tag{6}$$

$$L_{a+b} = L_b + f_\beta (L_a - L_b) \quad \text{if} \quad |b| > |a| \tag{7}$$

$$f_s(x) = \log_s (1 + s^x) \tag{8}$$

Subtraction is computed in the same manner, but the function $f_s(x)$ in equation (8) must be altered to (9).

$$f_s(x) = \log_s (1 - s^x) \tag{9}$$

This function can be computed in a table lookup procedure. How large table size Z, do we have to allow for? It naturally depends on the base β.

$$Z > \frac{\ln (2/\ln \beta)}{\ln \beta} \tag{10}$$

If the table size is an even power of two it is easy to check whether we are addressing inside the table or not.

A reasonable table size is 4096.

That gives β equal to $\sqrt[402]{2}$. This base is rather odd. Instead of $\sqrt[402]{2}$ choose β to be $\sqrt[384]{2}$, because 384 happens to be the sum of two even powers of 2. This base gives an accuracy of $\pm$ 0.09 % and 95 % utilization of the look up table. The dynamic range of this base is $4.9 \cdot 10^{25}$.

If we use the two shiftregisters and the counter to store intermediate results we can do the computation a + b or a - b in 93 cycles. Multiplication and division require 48 cycles each.

Applied to standard pictures (512 x 512 pixels), and computed on an array of 16 by 16 processors, we can do pixelwise multiplication in 4.9 ms. The speed for addition (subtraction) is 9.5 ms.

Logarithm computation

If we would like to use the signed logarithm number representation we have to convert the incoming pixel data. The pixel data usually comes in as an integer of fixpoint format.

Many algorithms have been formulated for computing $\log_\beta$ [8]. The unique properties of the logarithm can be utilized in a fast computing scheme. Such a method can be formulated in the following manner.

$$\log_\beta (\xi) = \log_\beta (\xi \pi a_k) - \Sigma \log_\beta (a_k) \quad (11)$$

Choose a_k in such a way so that $\xi \pi a_k$ approaches 1. Then $\log_\beta (\xi)$ becomes $- \Sigma \log_\beta (a_k)$.

Let a_k be of the form

$$a_k = 1 + q\, 2^{-k} \quad , \quad q = -1 , 0 , 1 \quad (12)$$

The algorithm can then be formulated as follows

$$x_0 = \xi \quad (13)$$

$$y_0 = 0 \quad (14)$$

$$x_{k+1} = x_k \cdot a_{k+1} \quad (15)$$

$$y_{k+1} = y_k - \log_\beta (a_{k+1}) \quad (16)$$

$$q = 1 \quad \text{if} \quad 0 < x_k < 1 \quad (17)$$

$$q = -1 \quad \text{if} \quad x_k \geq 1 + 2^{-(k+1)} \quad (18)$$

$$q = 0 \quad \text{if} \quad 1 \leq x_k < 1 + 2^{-(k+1)} \quad (19)$$

$$y_k = \log_\beta \xi + \varepsilon(n) \quad (20)$$

$\varepsilon(n)$ in (20) is an errorfunction with an upper bound of

$$|\varepsilon(n)| < \frac{2^{-n}}{\ell n \beta} \quad (21)$$

To convert a floating point number (10 bit mantissa, 6 bit exponent) to a signed logarithm number representation described in the previous section we need 264 cycles.

The conversion of a full size picture (512 x 512) in a 16 bit floating point format to a picture in a signed logarithm number representation is accomplished in 27 ms on a 16 x 16 processor array.

Conclusions

We believe that the most important step forward is to leave the densely packed processor system and use the distributed processor topology. The former image array mapping is a curse inherited from the old and false idea that the ultimate image processor should look like a two-dimensional iterative automation that tesselates the image plane with one finite-state machine in each gridpoint.

The algorithm examples shows that bit serial processors could be quite effective especially when dealing with various formats and precision of number representation.

References

[1] P.E. Danielsson and S. Levialdi, "Computer Architecture for Pictorial Information Systems", IEEE Computer, pp 53-67, November 1981.

[2] P.E. Danielsson and T. Ericsson, "Suggestions for a Image Processor Array", Internal Report LiTH-ISY-I-0507, Linköping University, S-581 83 Linköping, Sweden.

[3] M. Duff, "CLIP 4. A large Scale Integrated Circuit Array Parallel Processor", Proc. of Third International Joint Conference on Pattern Recognition, pp 728-733, 1976.

[4] K. Batcher, "Design of a Massively Parallel Processor", IEEE Trans. on Computers, Vol C-29, No 9, pp 836-840, September 1980.

[5] S. Reddaway, "The DAP approach", Infotech State of the Art Report on Super Computers, Vol 2, 1979.

[6] E.E. Swartzlander and A.G. Aleaopoulos, "The sign/logarithm Number System", IEEE Trans. on Computer, December 1975.

[7] S. Todd, "Low Precision floating point for Signal Processing", IBM Technical Disclosure Bulletin, Vol 23, no 12, pp 5563-5564, May 1981.

[8] T.C. Chen, "Automatic Computations of Exponentials, Logarithms, Ratios and Square roots", IBM Journal of Research and Development, pp 380-388, July 1972.

[9] P-E Danielsson and T. Ericsson, "LIPP - proposals for the design of an image processor array", Computing structures for image processing, M. Duff (Ed), Academic Press, London 1983.

[10] P-E Danielsson, "Vices and Virtues of Image Parallel Malchines", Digital Image Analysis and Processing, S. Levialdi (Ed), Pitman Books, London 1983.