Linköping Studies in Science and Technology. Dissertations.
No. 319

Smart Image Sensors

Anders Åström

Department of Electrical Engineering
Linköping University, S-581 83 Linköping, Sweden

Linköping 1993

Linköping Studies in Science and Technology. Dissertations.
No. 319

Smart Image Sensors

Anders Åström

Department of Electrical Engineering
Linköping University, S-581 83 Linköping, Sweden

Linköping 1993

Abstract

Today, modern VLSI technology makes it possible to integrate image input and low-level operations into one single chip. Such a chip can take in an image and apply fairly complex algorithms on the image and also extract features to be sent to higher levels of processing. These chips are known as *Smart Image Sensors*. All devices described in this thesis employ CMOS technology for the sensors as well as for other circuitry.

The first part of this thesis describes a smart image sensor called PASIC (Processor, A/D-converter, and Sensor Integrated Circuit). It is a two-dimensional sensor with a linear array of processing elements working in SIMD mode. The aim is to verify the performance by cycle counting a number of low-level algorithms such as shading correction, histogram based thresholding, edge detection, and segmentation. The studies show that the PASIC architecture is surprisingly effective.

The second part of the thesis describes MAPP2200 which is a commercial available smart image sensor. It has the same basic structure as PASIC (2D sensor array and 1D processor array). The exposure control is performed in a very different manner compared to normal CCD-cameras. In MAPP2200 the sensor array is accessed much like a memory, row-wise in any sequence. The simple ALU-functions in each of the 256 processing elements is enhanced with a global logic for feature extraction. One group of algorithms which seams to suit MAPP2200 very well is moment calculations and we give three application examples with different demands on speed and segmentation. Finally, we show that range data from sheet-of-light can be obtained at very high speed by using a combination of analog and digital computation.

The third part introduces the concept of Near-Sensor Image Processing. It is shown that the analogue-temporal behavior of photodiodes as detected by thresholding amplifiers can be integrated with low-level algorithms such as median filtering, grayscale morphology, convolution, and adaption to different light levels. The quantity to be measured in NSIP is the time from the pre-charge of the photodiode until the voltage passes the reference voltage. This time is inversely proportional to the incoming light intensity. Various mapping strategies to modify this relation are discussed, and it is shown that histogram equalization comes naturally in this concept. For the sake of simplicity, most examples of operations and algorithms are given for the 1D-array LAPP1100. Suggestions for a 2D NSIP architecture are also provided. Finally, an application is described where NSIP is used in an architecture for sheet-of-light range imaging.

ISBN 91-7871-171-1
ISSN 0345-7524
Printed in Sweden by LJ Foto & Montage/Samhall Klintland, Linköping 1993

To my parents

Acknowledgements

There are a number of people without whom this thesis would never have existed. This is my opportunity to thank at least some of them.

First of all I would like to thank my supervisor Professor Per-Erik Danielsson for many valuable suggestions to this thesis, and for making it more readable and understandable.

Invaluable inspiration to this thesis, especially for the parts on MAPP and NSIP has been given by Dr. Robert Forchheimer.

Further thanks goes to

All people involved in the PASIC-, VIP-, MAPP-, and NSIP projects, especially, to Per Ingelhag and Mattias Johannesson.

People at IVP for the opportunity to work with MAPP2200.

All members of the Image Processing Group.

Finally, I would like to thank the Swedish National Board for Technical Development (NUTEK) and Center for Industrial Information Technology (CENIIT) for financial support.

CONTENTS

Part II MAPP2200

Part III NSIP

Smart Image Sensors

Thesis overview

The expression "Smart image sensor" has no official definition and different authors tend to have different meanings of the concept. However, a common characteristic seams to be that the sensor should contain some processing power and/or programmable access to the image data. Hence, ordinary CCD-cameras are not regarded as smart, not even if they have some flexibility in terms of programmable integration time.

We feel that all Smart image sensors (proposed or implemented) can be categorized as follows.

1. Random access of sensor array, no processing capability
2. Integrated sensor and one single processor
3. Integrated sensor and linear processor array
4. Integrated sensor and 2D array of processors (one processor per pixel)

The smart image sensors to be presented in this thesis belong to categories 3 and 4.

Research within the field of smart image sensors has been an ongoing activity at the University of Linköping for a long time. At the beginning of the 1980s, Dr. Robert Forchheimer came forward with an idea to combine sensors and processors on the same chip. He was partly inspired by the optical mouse [19]. In cooperation with Anders Ödmark at the Physics Department, the sensor/processor LAPP [13], Linear Array Picture Processor, was born. A series of prototype chips were designed and manufactured in 1983–84. The project was then moved from the university to the spin-off company IVP Integrated Vision Products, which was founded to commercialize this product. The name of the chip was now changed to LAPP1100 where the first "1" stands for the dimension and the "100" stands for the magnitude of the number of sensor elements, which was 128. The marketing of the LAPP1100 chip began in 1987.

At the same time as the LAPP1100 chip was developed at IVP, several new ideas were investigated at the university as a joint venture between the LSI design group, Dept. of Physics, and the Image processing group, Dept. of Electrical Engineering. One of these was the PASIC architecture, first proposed by Dr. Keping Chen in 1988. At that time, PASIC was assumed to consist of a 128x128 array of photo-sensors and a 128 array of processors, [6][7]. Given the architecture of PASIC several algorithms were simulated and cycle counted. Sometimes the simulations showed that the design should be modified. After a number of architecture and algorithm iterations the architecture converged to a rather complete and powerful smart image sensor. Two different test chips were made and tested in 1991. One contained a 256x256 sensor array and 256 A/D-converters and the other contained a processing array of size 256. This work was published as a licentiate thesis [1] in december 1990 and forms **Part I** of this thesis.

During the work with PASIC, its processor part was discovered to be very powerful on its own. This resulted in a new chip design called VIP, Video Image Processor [4][5][8], which is

not covered in this thesis. This architecture has proved to be highly effective for radar signal processing [14][18]. A modified VIP chip, now called RVIP (Radar VIP), is going to be commercialized by Ericsson Radar Electronics, Mölndal, Sweden.

In 1990, the demand for a two-dimensional sensor persuaded IVP to develop MAPP2200, Matrix Array Picture Processor [12]. MAPP2200 is mainly an outgrowth of LAPP1100 although some of the ideas behind PASIC were also incorporated in the design. Especially, the software environment of MAPP2200 [3] has been influenced by PASIC. MAPP2200 has also been used for research at some universities. One of the applications is to serve as the key component in a sheet-of-light range camera [17]. The MAPP2200 architecture, algorithm implementations, and some applications form **Part II** of this thesis.

The idea of Near-Sensor Image Processing, NSIP, originates from LAPP1100 [13]. In the early 80's, VLSI constraints prevented the LAPP-designers from incorporating A/D-converters for the 128 sensor array. To obtain gray level data the threshold voltage had to be changed step by step and a number of binary image had to be taken care of. It was then discovered that many image processing problems could indeed be solved not by storing a sequence of binary images but by intelligent processing each time the sensor array delivers binary data. The final result is influenced by the full sequence of input data and the processing utilizes certain characteristics in these sequences, e.g. the fact if a sensor input has passed the threshold it will continue to do so. In effect, the A/D-conversion and the digital processing is interwoven in the innermost program loop where the sensor is repeatedly interrogated. This is the NSIP idea, "Near-Sensor Image Processing". **Part III** of this thesis, discusses NSIP both on the algorithmic level [10][11] and on the architectural/VLSI level [2].

Similar projects

The litterature on smart image sensors is not large which is somewhat surprising. For anyone who is familiar with image processing and its applications, the smart image sensor would seam to be an obvious and natural development. Possible reasons why so little has been reported might be a certain conservatsim, feeling that the art of VLSI is still not ready for it, and the fact that the dominating sensor type, the CCD-chip does not lend itself so easily to add-on digital circuitry. In contrast, all smart image sensor reported in this thesis are CMOS for sensors as well as the digital parts.

We will will mention four other smart image sensor projects below; Two of them very breifly.

Carver Mead at Caltech [20] set as his goal to construct a silicon retina to emulate certain functions of the human eye. His strategy is to design a number of functional units in VLSI where each unit has a special neural-like function (axons, dendrites, etc.). He then configures these units according to demand. A motion detector which detects any kind of motion in the scene in the same way as a human eye has been demonstrated as well as a sensor with the same impulse response as the human retina.

Another approach has been reported from **Laval University**, Canada. This is the Multi-port Access photo-Receptor, MAR [22]. The main smartness of this sensor is that pixels in a hexagonal grid can be randomly accessed and for each pixel position, the whole 16 pixels neighborhood will be available at the analog output from the chip. In of the applications men-

tioned in [22] the 16 pixel output is transfered to several different lowpass/gradient filters concurrently.

Smart image sensor projects at the **University of Edinburgh** [21] has been going on for quite some time. The results have been successfully commercialized by the company VVL, VLSI Vision Limited. The idea behind the project and the company is to use a CMOS sensor array and on the same silicon chip implement an image processing system as a full-custom VLSI design dedicated to a certain application [9]. For example, if the application is just a compact video camera a special unit is implemented on the chip which takes the output from the photo-sensors and delivers a standard video signal output. If the application is a system for fingerprint capture and verification [21], the image processing part is totally different. In the latter case the chip consists of

258x258 photo-sensor array
Unit for image preprocessing and quantization
64-cell 2000 Mops/s correlator array
16k bits RAM cache
16k bits ROM look-up table

Together with a 64k bit off-chip RAM and a 8051 micro-controller, this device can capture and compare a fingerprint against a stored reference print within one second.

The photo-sensor array which is used in most of the applications is shown in Figure 1. The size of the array depends on the application. For the fingerprint application it is 258x258 and for the video output chip it is 312x287.

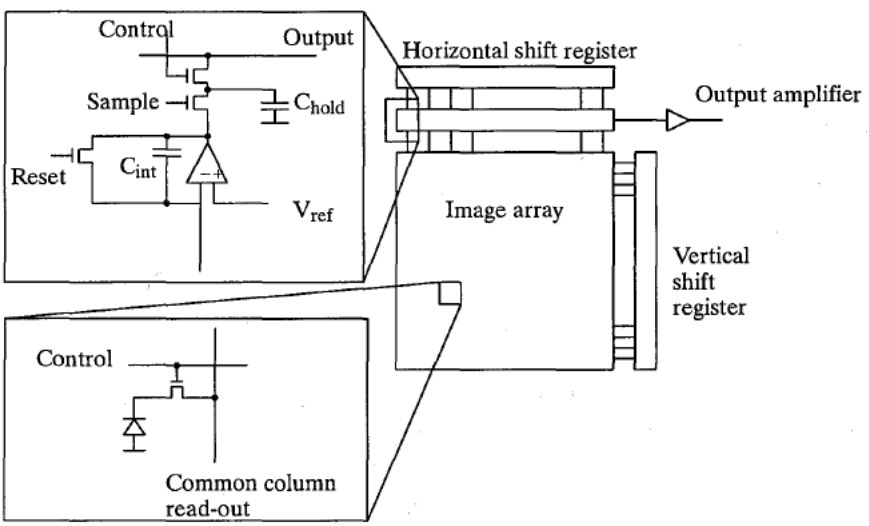


Figure 1. The sensor array in the VVL smart sensor

Each photo-sensing element consists of a photodiode and a CMOS transistor. It is pre-charged and after a certain time it is read-out via the column bus. The pre-charge and read-out is done row-wise in parallel. The row address is controlled by the vertical shift register.

For each column, there is a sense amplifier, which is responsible for both the reset and the read-out of the photodiodes. The charge is moved from the pixel element to the hold position by activating the Sample switch and deactivating the Reset switch. The row can now be read-out, sequentially over the output bus, by addressing one column at a time. This addressing is controlled by the horizontal shift register.

The **VLSI Retina** [23], has been developed at **ETCA**, France. This design has one processor per pixel. Each processor is based on a three-bit semi-static shift register, shown in Figure 2. Each of the memory cell F, G, and H is controlled by the two clock signals M1 and M2. Using the clock signal X the data can be shifted to the right and with Y, data can be moved to the north (with Y and M2) and to the south-east (Y and M1). These three shift possibilities, West, North, SouthEast, are sufficient to perform arbitrary 2D shifting in the array.

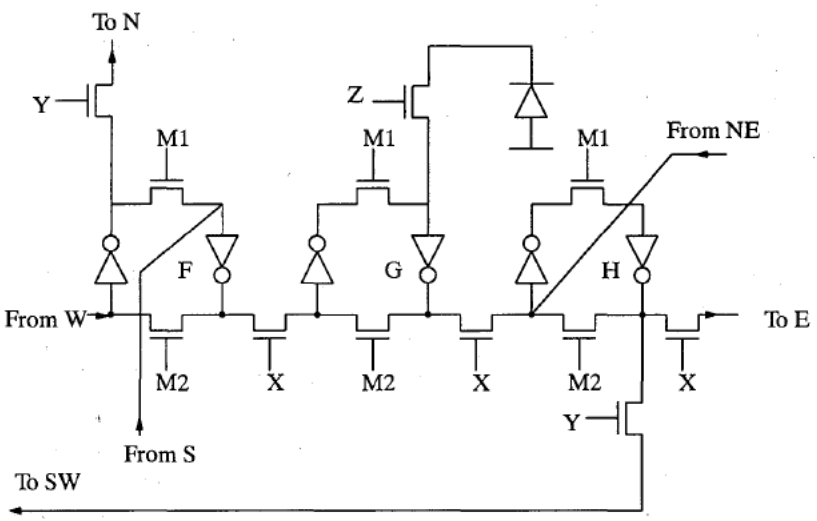


Figure 2. Semi-static shift register

With a control signal T and some extra transistors, not shown in Figure 2, it is possible to perform Boolean operations. The bit F or its complemented version in the first cell, may be copied or and'ed with the G bit in the second cell. It is also possible to take the value from the photo-sensor and latch that binary value into the second cell with the control signal Z. Table 1 shows the content in the three cells F, G, and H, after different operations.

The idea behind the design is to use what is called Neighborhood Combinatorial Processing [15], NCP. It is a concept to define edge detection, dithered halftoning, and other binary operations, in terms of iterated Boolean operations. It is well known that conjunction and complementation are sufficient to perform any computation. Since these are included in Table 1, the claim is that the VLSI Retina sensor/processing element can perform any operation. An edge detection is performed by using classical mathematical morphology technique, where the edge image is the difference between the initial image and the eroded version. This operation takes 30 instruction cycles.

Operation	F	G	H
Shift up	$F_{0,-1}$	G	H
Shift down & left	F	G	$H_{1,1}$
Shift right	$H_{-1,0}$	F	G
Circular permutation and inversion	$H_{0,1}$	$\overline{F}$	G
Copy	F	F	H
Inverting copy	F	$\overline{F}$	$H_{1,1}$
Conjunction	F	F&G	H
Conjunction on inversion	F	$\overline{F}$&G	$H_{1,1}$
Read from photodiode	F	G+(φ>φ_T)	H

Table 1. Some operations on the VLSI Retina

There are at least two major differences between the VLSI Retina and the Near-Sensor Image Processing concept, described in part III in this thesis. Gray-level information in the VLSI retina is obtained by imaging the scene several times with decreasing exposure time, while in the Near-Sensor Image Processing concept, the photodiode is interrogated repeatedly during one single discharge event. The other difference is that feature extraction on a global level is not dealt with at all in the VLSI Retina.

References

[1] Åström A., *A Smart Image Sensor. Evaluation and Description of PASIC.*, Lic Thesis, Linköpings University 1990.

[2] Åström A., Forchheimer R., Ingelhag P., *An Integrated Sensor/Processor Architecture Based on Near-Sensor Image Processing*, To appear in Proc of CAMP93, New Orleans, USA.

[3] Åström A., Forchheimer R., *MAPP2200 Smart Vision Sensor: Programmability and Adaptivity*, Proc of IAPR Workshop of Machine Vision Applications, Japan, 1992.

[4] Åström A., Danielsson P.E., Chen K., Ingelhag P., Svensson C., *Videorate signal processing with PASIC and VIP*, Proc 2nd Int Specialist Seminar on the Design and Application of Parallel Digital Processors, Lissabon, Portugal 1991.

[5] Chen K., Svensson C., *A 512-processor array chip for video/image processing*, Proc. of "From Pixels to Features II," Bonas, France, Aug. 27 – Sept. 1, 1990, pp 349–361.

[6] Chen K., Åström A., Danielsson P.E., *PASIC a smart Sensor for Computer Vision*, Proc of Pattern Recognition, Atlantic City, June 1990, pp. 286–291.

[7] Chen K., Danielsson P.E., Åström A., *PASIC a Sensor/Processor Array for Computer Vision*, Proc of Application Specific Array Processors, Princeton, September 1990, pp. 352–366.

[8] Danielsson P.E., Emanuelsson P., Chen K., Ingelhag P., Svensson C.. *Single-Chip High-Speed Computation of Optical Flow*, Proc IAPR International Workshop on Machine Vision Applications, November 1990, Tokyo, Japan

[9] Findlay K. Personal Communication 1992. Kevin Findlay, <kevin@ee.edinburgh.ac.uk>.

[10] Forchheimer R., Åström A., *Near-Sensor Image Processing. A new approach to low level image processing*, Proc of ICIP, Singapore 1992

[11] Forchheimer R., Åström A., *Near-Sensor Image Processing. A New Paradigm*, accepted for publication in IEEE trans on Image Processing.

[12] Forchheimer R., Ingelhag P., Jansson C., *MAPP2200, a second generation smart optical sensor*, SPIE Vol 1659, 1992.

[13] Forchheimer R., Ödmark A., *Single chip linear array processor,* in *Applications of digital image processing,* SPIE, Vol. 397, (1983).

[14] Ingelhag P., Åström A., Chen K., Svensson C., Ehlersson T., *RADAR Signal Processing Using A 512-Processor Array Chip.* ICDP91, Florence, Italy 1991.

[15] Garda P., Reichart A., Rodriguez H., Devos F., Zavidovique B, *Yet Another Mesh Array Smart Sensor*, Proc of ICPR88, Rome, Italy.

[16] Johanneson M., Åström A, Danielsson P-.E. , *A Range Image Architecture Using Sheet of Light*, Proc of IAPR Workshop of Machine Vision Applications, Japan, 1992.

[17] Johanneson M., Åström A, *Sheet-of-light Range Imaging with MAPP2200*, Proc of SCIA8-93, Tromsø, Norway.

[18] Johanneson M., Åström A, Ingelhag P., *The RVIP Image Processor Array*, To appear in Proc of CAMP93, New Orleans, USA.

[19] Lyon R.F., *The optical Mouse, and an Architectural Methodology for Smart Digital Sensors.* Palo Alto Research Center.

[20] Mead C., *Analog VLSI and Neural Systems*, Addison-Wesley, 1989

[21] Renshaw D., Denyer P. B., Lu M., Wang G., *On-chip CMOS Sensors for VLSI Imaging Systems,* Proc VLSI-91, 1991.

[22] Tremblay M., Larendeau D., Poussart D., *Multi Module Focal Plane Processing Sensor with Parallel Analog Support for Computer Vision*, Proc of Symposium on Integrated Systems, Seatle, 1993.

[23] Bernard T., Zavidovique B, Devos F., *A Programmable Artificial Retina*, IEEE Journal of Solid-State Circuits, vol 28, no 7, July 1993.

Part I PASIC

1 Introduction

1.1 Background

Over the last 15 years image processing has established itself as a respectable scientific discipline. However, the great breakthrough in industrial applications has not yet been seen. The reason seems to be that the industry wants small, well defined and inexpensive systems that are easy to maintain, while the hardware developers often come up with big, general purpose and, due to that, expensive systems that require a continuous professional maintenance. This situation will force the vendors of image processing system to change their policy. They will probably have to make more compact systems that are less all-purpose but has the advantage of being cheap. The enormous development that has taken place within the field of VLSI the last 20 years, seems to be the key to the solution. Today we have efficient tools to produce compact, inexpensive, specialized, but yet powerful systems.

A typical machine vision system is shown in Figure 1.1. First, we have to get an image of the real world into some appropriate digital form. This is accomplished by using a number of photo-sensors followed by an A/D-converter. Secondly, we apply a number of low level operations on our image, such as convolutions, morphological operation and so on. These operation are often performed with a high degree of locality, i.e. they are mostly neighborhood operations. Finally, the output from the system could be either a preprocessed image, in which case we are finished, or used to produce a feedback in form of a control signal to the machine that the system is supervising. In the later case we will have to perform some higher level operations such as classification or decision to produce the control signal. Both type of systems are shown in Figure 1.1.

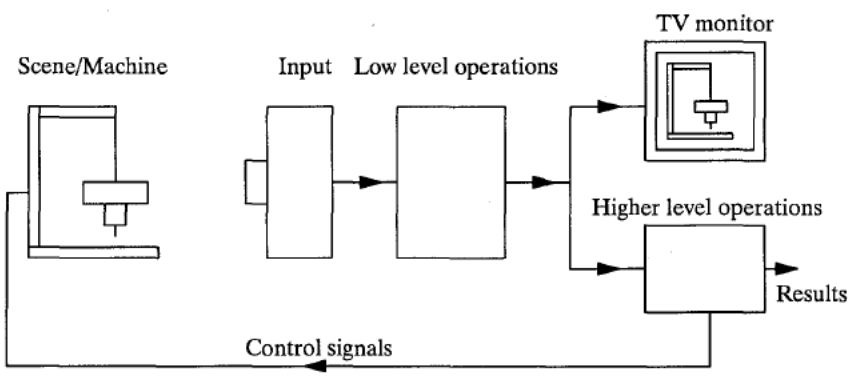


Figure 1.1, A typical image processing system

Today, modern VLSI technology makes it possible to integrate the input and the low level operations into one single chip. Such a chip can take in an image and apply fairly complex algorithms on the image and also extract features to be sent to higher level of processing. This type of chips are known as smart sensors.

This thesis will describe one such smart sensor called PASIC. PASIC stands for Processor, A/D-converter and Sensor Integrated Circuit. The project started in 1988 as a joint venture

between the Department of Electrical Engineering and the LSI Design center at the department of Physics and Measurement Technology, both at the University of Linköping.

The architecture in a typical smart sensor is very VLSI driven. The design strategy is bottom-up meaning that the VLSI implementation sets the constraints for the architecture, which in its turn sets the constraints for the algorithms and the application. This strategy and the common top-down strategy are both illustrated in Figure 1.2. In both strategies the unidirectional influence has to be moderated and interleaved by verification loops. In fact, the main body of this thesis is part of such a procedure which verifies that PASIC efficiently can execute basic image processing algorithms. In several cases, the outcome of this verification has been fed back to a redesign at the VLSI level.

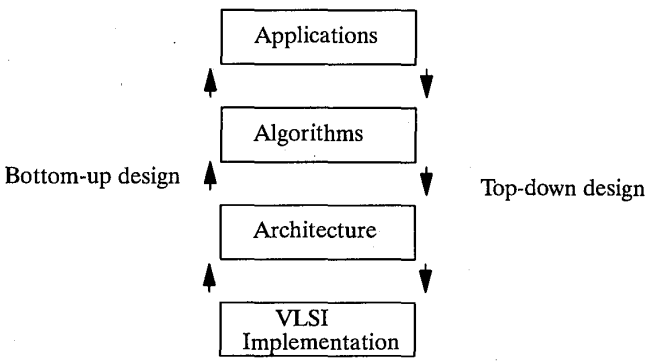


Figure 1.2, Bottom-up and top-down design

Most 2D image sensors are adapted to the standards of broadcast TV, see Figure 1.3(a). In CCD-cameras this means that every pixel first have to be shifted downwards and then, in a high speed, be shifted horizontally. Both shift paths carry data in analog representation. Every pixel then has to pass one common A/D-converter before it can be processed. The demands on the A/D-converter and the analog shift register are very high since we are dealing with pixel frequencies above 10 MHz.

In PASIC we have a 2D sensor, the A/D-conversion and the processing power integrated on a single chip. An immediate consequence is the possibility of a sensor layout as in Figure 1.3(b). To reduce the pixel frequency we insert more A/D-converters, ideally we would like to have one A/D-converter for each vertical column. If we could accomplish this we would avoid the high speed horizontal analog shift as well as the high speed A/D converter.

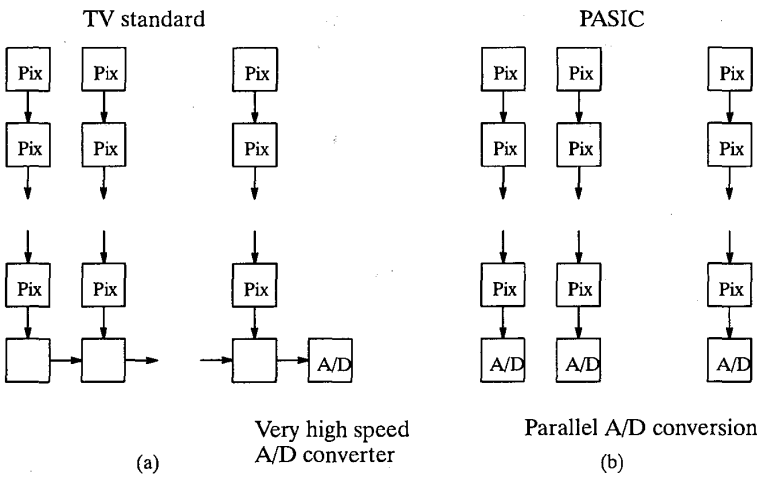


Figure 1.3, TV standard versus PASIC

The parallel A/D-conversion will become worthless unless we could process the data in a parallel fashion. This implies that beneath the A/D-converters there should be some form of linear processor array. The main character of the PASIC architecture took shape in this manner as will be shown in the next chapter of this thesis.

1.2 Bit-serial parallel processors

1.2.1 An overview

Bit-serial processors have been used for quit a number of parallel computers most of them of type SIMD, (Single Instruction stream, Multiple Data stream). Many, but not all have been targeted for image processing. Some have been organized to communicate in a 2D mesh, for instance CLIP4 [10], DAP [14], MPP [1], CM1 and CM2 [17], and GAPP [11]. Others have a linear organization in form of a 1D array, for instance LUCAS [16] and AIS-5000 [18]. In the following sections we will take a brief look at the processor element in three of them, namely LUCAS, GAPP, and AIS-5000. Finally we will make a short presentation of LAPP, see [9], which is both a bit-serial machine, a smart sensor, and in many ways a predecessor of PASIC.

1.2.2 LUCAS

LUCAS was designed at Lund University, Sweden in the late 70's and is also a linear processor array. Each PE has an ALU using 5 inputs and 4 output functions. It also has its own

memory of 4096 bits, a special communication network called Select first network, and a communication multiplexer into the ALU, see Figure 1.4.

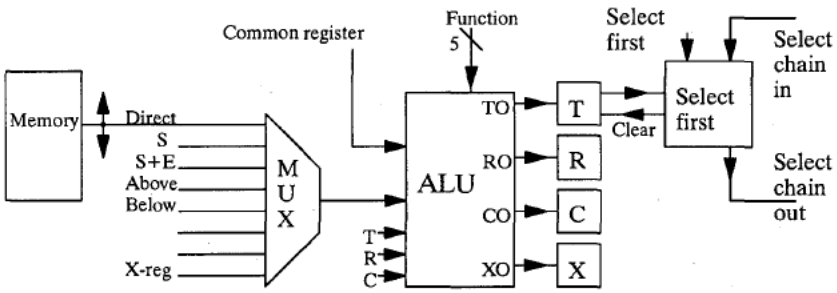


Figure 1.4, The processing element in LUCAS

The ALU has 5 input variables. Three of them are from the output registers, T(ag), R(esult), and C(arry). One is a common bit to all PEs to generate constants and one comes from the multiplexer. The multiplexer can select data from the PE's own memory or the auXilary register, called Direct and X. It can take data from its nearest neighbors, called Above and Below. Finally, it can take data from a perfect shuffle/exchange pattern, called S and S+E. This exchange pattern is used in for instance FFT computation.

The functions from the ALU to the output registers can be chosen from a set of 32 different functions. This means that we can perform both arithmetics and logic operations. An addition of two bit vectors consists of three steps. First load one bit from the first vector into the R register, then add the bit from the second vector with the R and C, and finally store the content of R in the memory. These three phases are repeated b times, where b is the number of bits. Thus, an addition is performed in **3b** cycles.

With the Select first network, all the T-bit except the one in first PE is reset. The select feature is part of LUCAS' ambition to do associative processing.

1.2.3 GAPP

GAPP was developed at NCR in United States and presented as a product in 1984. The processor element, the PE, in GAPP contains four 1-bit registers, 128 bits of static RAM, 5 multiplexers to feed the input registers and the memory, one ALU, and connections to the four nearest neighbors, north, east, west, and south. This is shown in Figure 1.5.

The data input to the PEs are carried out by the CM registers. By opening the CMS-input, data is shifted in without interfering with the logical and arithmetic computation within the PE. The ALU unit is a full adder/subtracter which enables us to perform arithmetic as well as logic. A normal addition is setup by loading the NS and the EW registers with the two bits that we want to add. For each pair of input bits we store the result in the memory and at the same time feed the Carry from the previous result back to the C register. This means that an addition is done in **3b** cycles, where **b** stands for the number of bits.

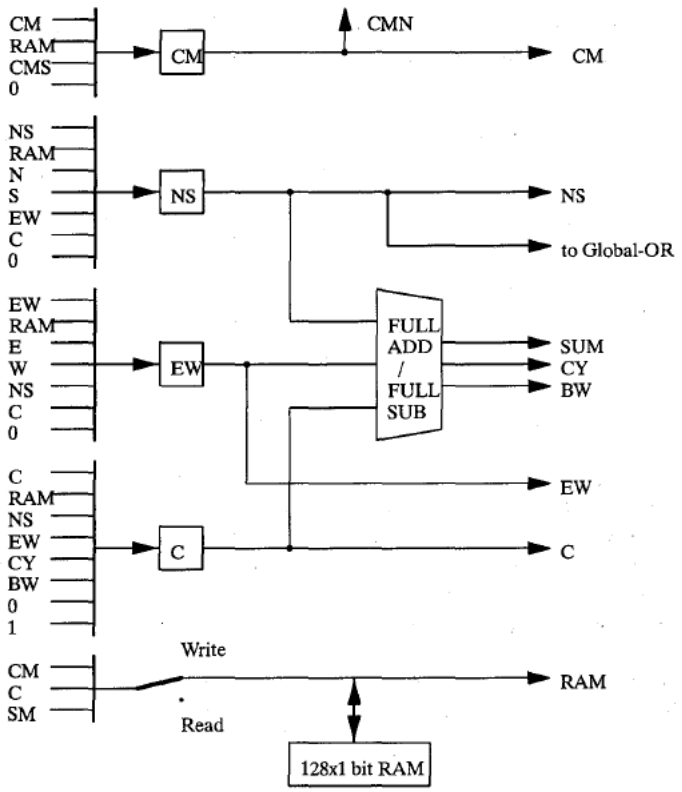


Figure 1.5, The processing element in GAPP

GAPP is equipped with a Global-OR function. This is an output pin on the chip that delivers a logical-OR of all NS registers in the array, which means that if any NS register holds the value 1, the Global-OR will be 1. This function can be used to terminate loops and to check for unwanted conditions.

GAPP has been used in a number of test design and experiments on parallel architecture but was never a great success.

1.2.4 AIS-5000

AIS-5000 is a commercially successful linear array of processors from Applied Intelligence, Ann Arbour, USA. The PE of AIS-5000 has four input registers. These can be arranged in different configurations depending on what task we want to perform. To perform a Boolean operation the registers are connected as in Figure 1.6. The truth table is pre-loaded with a arbitrary 16-bit value, which is common to the whole array. We can now load the S1 to S4 register with values and then read out the result from the truth table back into the memory.

The registers S1 to S4 are connected as a shift register which means that if we only want to change the value of for instance S2, we will have to shift in S1, S3, and S4 again.

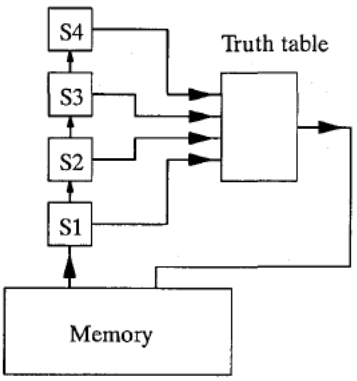


Figure 1.6, AIS-5000 configurations for boolean operations

The arithmetic configuration is shown in Figure 1.7. The two input registers are loaded with the bit vectors from the memory. For each bit pair the result is written back to the memory and the Carry output is stored in the carry register, which means that it takes **3b** cycles to perform an addition.

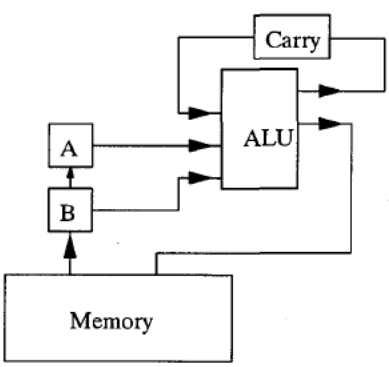


Figure 1.7, AIS-5000 configurations for arithmetic operations

Finally AIS-5000 has the possibility to compute neighborhood binary templets in one cycle. The north, south, and center pixels are fetched from each PE. The east and west pixels are taken directly from the neighboring PEs, see Figure 1.8. The data paths makes it possible to perform erosion, dilation, and any other Boolean function on the neighborhood data in one cycle, by programing the truth table and the logic block in the appropriate way.

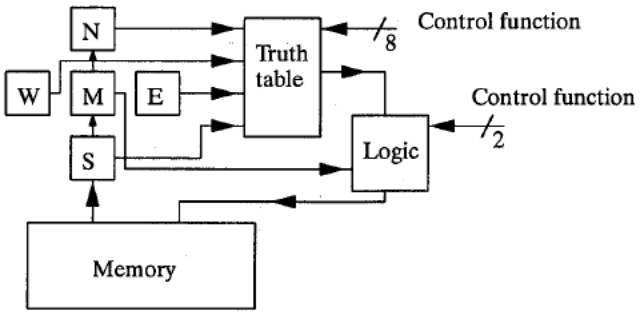


Figure 1.8, AIS-5000 configurations for neighborhood operations

1.2.5 LAPP

LAPP differs from the other processors that we have described, since it is has both input capability and processing power. It even contains these two features on one single chip. In many aspects LAPP has been an inspiration for PASIC, which can be thought of as a second generation of LAPP.

LAPP consists of a linear photo sensor array, 14 registers, one accumulator, and an ALU complex consisting of GLU, NLU, and PLU, see Figure 1.9.

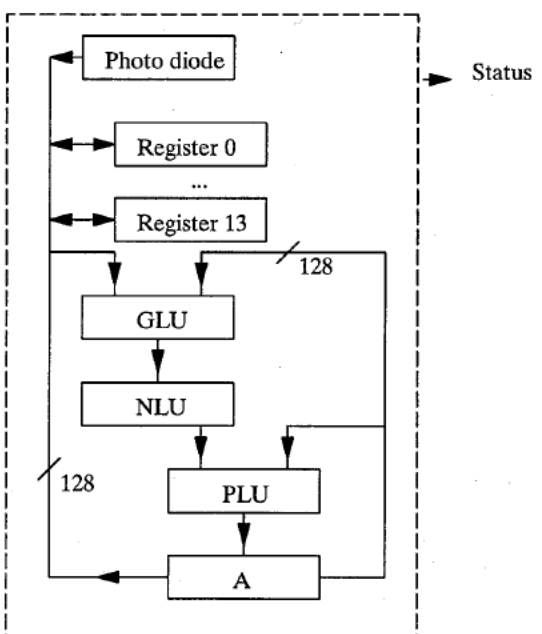


Figure 1.9, The LAPP architecture

The image that we get from the photo sensor array is binary. However, as it is possible to adjust the thresholding level of the input line by an external reference voltage, it is possible

to some extent to work with gray scale images. The size of the photo diode array, the registers, the accumulator, and the logic is all 128 units.

A special hardware logic, the status output, detects and reports the number of 1s in the accumulator and, in the case there is a single 1, the position.

The ALU consists of three parts, the GLU, the NLU, and the PLU. The GLU, Global Logical Unit, performs global operation on the line, such as fill hole and mark object. Figure 1.10 shows the instruction MARK. All following instructions examples is taken from [13]. An object is considered to be a group of connected 1's, represented as black in the following Figures. MARK R0 means that all 1's in R0 which are within the same object as a pixel that has a 1 in the accumulator, is set to 1.

Figure 1.10, Global operation, mark objects

The NLU, Neighborhood Logical Unit, performs operation in a 3 by 1 neighborhood, like Erosion and Dilation. Each PE match its neighborhood with a common pattern containing three symbols, 0,1, and X don't care. To extract the left edges of the objects we use the LEDGE function, see Figure 1.11. This corresponds to the 01X pattern, which means that all 1 pixels remain set if they have a 0 neighbor to the left.

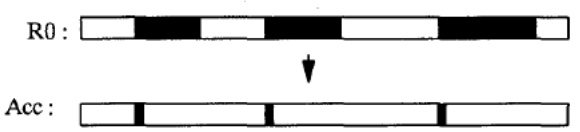


Figure 1.11, Neighborhood operation, find left edges

Finally, we have the PLU, Point Logical Unit, which performs logical and command instructions, such as AND, EXCLUSIVE-OR, and Load accumulator. Figure 1.12 shows an example of an OR operation, where the content of R0 and the accumulator are logically Ored.

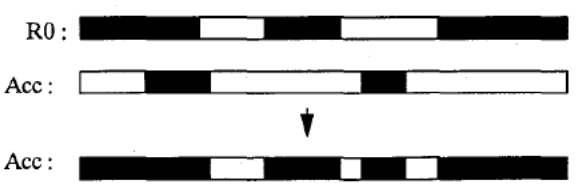


Figure 1.12, Point logic operation, OR between R0 and the accumulator

LAPP has been a commercially successful product and a number of systems has been delivered. One common application for LAPP in industry is to measure length and width of moving objects.

2 The PASIC Architecture

2.1 Overview

PASIC consists of three major parts: Sensors, A/D-converters, and Processors. The sensors form a two-dimensional array of photo sensors. Each column of the sensor area has its own A/D-converter and Processing Element. Thus, if we include the sensor column and the A/D-converter in the PE concept it makes perfectly sense to claim that the whole PASIC chip is organized as a 1D array of PEs. Each PE consists of a column of sensors, an A/D-converter, a register, an ALU, and a RAM. An out-line of the floor-plan of PASIC will then look like Figure 2.1.

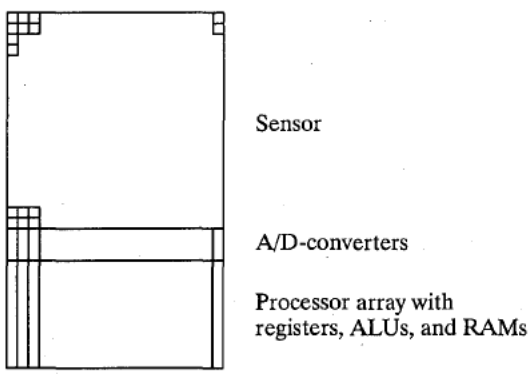


Figure 2.1, PASIC

2.2 The photo sensors

The sensor array consists of a matrix of photo diodes. Each sensor column is connected to its A/D-converter over a single bit bus, see Figure 2.2. The readout from the sensors is done line-by-line controlled by a vertical address decoder. The readout is non-destructive and the sensor array may be addressed randomly as well as scanned in any ordered manner. The reset of the elements is performed over a pre-charge line common to all photo sensors in a row. Each photo diode is guarded against blooming. In the present version of PASIC the size of the photo diodes is $48x48\mu m^2$, with a pitch of $60\mu m$ in both dimensions.

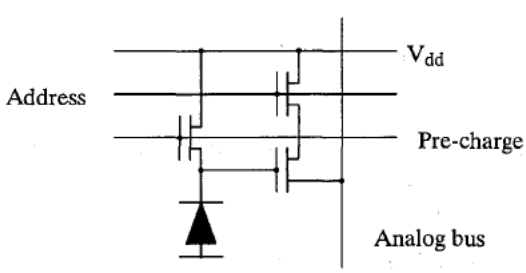


Figure 2.2, An image sensing element

Control and addressing of the sensor array requires 8 bits as shown in Table 2.1. The leading bit selects between pre-charge and read-out.

P/$\overline{R}$	Address	Operation
0	0000000	Connect row 0 to A/D converter
0	0000001	Connect row 1 to A/D converter
0	.....	
0	1111111	Connect row 127 to A/D converter
1	0000000	Pre-charge row 0
1	0000001	Pre-charge row 1
1	.....	
1	1111111	Pre-charge row 127

Table 2.1, Sensor array operations

2.3 The A/D converters

As was indicated in chapter 1, PASIC contains one A/D-converter for each column, which greatly reduces the requirement for high speed transfer of analog data over the chip. The Parallel A/D converter array consists of one counter, one D/A-converter, 128 comparators, and 128 8-bit registers, see Figure 2.3. The Counter is controlled by one control signal Start and one control field which specifies Precision. The Start signal activates the A/D-converters to perform an A/D-conversion with the precision specified by Precision field.

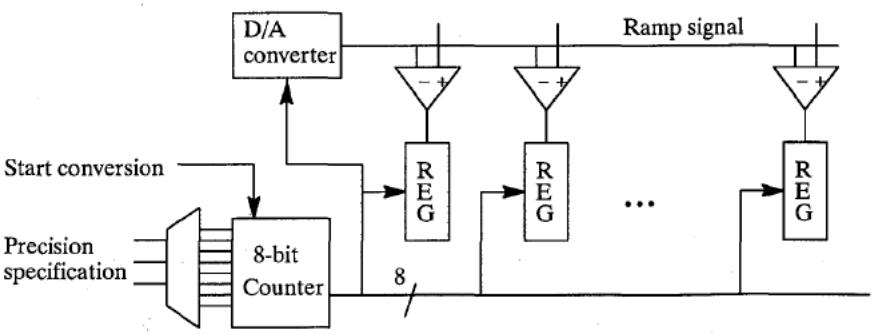


Figure 2.3, The parallel A/D-converter

Each bit in the register is a three transistor RAM cell. The value of the counter is broadcasted and written into the register as long as the ramp signal is less than the analog input. When an input analog signal in one channel is equal to the ramp value, the output of the comparator prohibits further writing. The 128 A/D-conversions are completed after a full ramp generation. The system clock of PASIC is 20 MHz. However, the A/D converter is stepped in 10 MHz rate to allow for the comparators to stabilize. One 8-bit A/D-conversion is therefore done in 256 * 0.1 μs = 25.6 μs. Since the resolution in the conversion is programmable it is possible to trade speed for precision. For instance, if we settle for 7 bits we may double the speed, for 6 bits quadruple it and so on. In many vision tasks such a programmability seems to come in handy. Since we have full control over the read out from the sensor, including the frame rate, it is possible to increase or decrease the signal-to-noise level.

The registers in Figure 2.3 are actually designed as RAM cells in a 8 x 128 bits memory. The read out to the memory bus is controlled by a 3-to-8 decoder, see Figure 2.4. Altogether the A/D-conversion needs 5 bits in the control word shown in Table 2.2. The first bit is to start the A/D-conversion. The second bit changes the effect of the three last bits, CD_0 to CD_2, from the indicating the bit to be read out from the A/D to selecting the precision in the A/D-conversion. If the two first bits are 00 or 11 the A/D-converter is either in data latch (quiescent) mode or in progress with an A/D-conversion.

Start	Control	CD_{210}	Operation
0	0	x x x	Data latch
0	1	0 0 0	Write bit position 0 to bus
0	1	0 0 1	Write bit position 1 to bus
0	1	. . .	. . . . .
0	1	1 1 1	Write bit position 7 to bus
1	0	0 0 0	Start A/D-conversion with 1-bit precision
1	0	0 0 1	Start A/D-conversion with 2-bit precision
1	0	. . .	. . . . .
1	0	1 1 1	Start A/D-conversion with 8-bit precision
1	1	x x x	Data latch

Table 2.2, A/D-conversion operations

2.4 The processing elements, PE

Each processing element consists of three parts, one bi-directional parallel shift register, one ALU-unit, and a memory. Figure 2.4 illustrates that all these units plus the A/D-converter are connected to the same bus. This is a 1-bit bus architecture, meaning that all operations can be seen as transfers over the 1-bit bus. As we will see later on there are a few exceptions from this. This architecture was first proposed by dr Keping Chen, former project leader of the PASIC project.

The parallel shift register contains 8 bits and is a bi-directional shift register, see Figure 2.5. Thus, we can shift a pixel value one step to the left or the right in one clock cycle. 8-bit parallel communication seems a reasonable choice since image processing often deals with 8-bit pixels. This shift register has 7 modes of operation, and it uses 3 control bits, PSC_0, PSC_1, and PSC_2, listed in table 2.3. In 000 and 001 the register communicates with the bus. For 011 and 100 the register content is shifted one step in the desired direction. For a left shift data will be shifted one step to the left leaving the leftmost PE's data to the left I/O-pins, and at the same time input data are accepted from the right I/O-pins. Operations 101 and 110 are identical to 011 and 100 except that end around shift takes place and no I/O operation is performed.

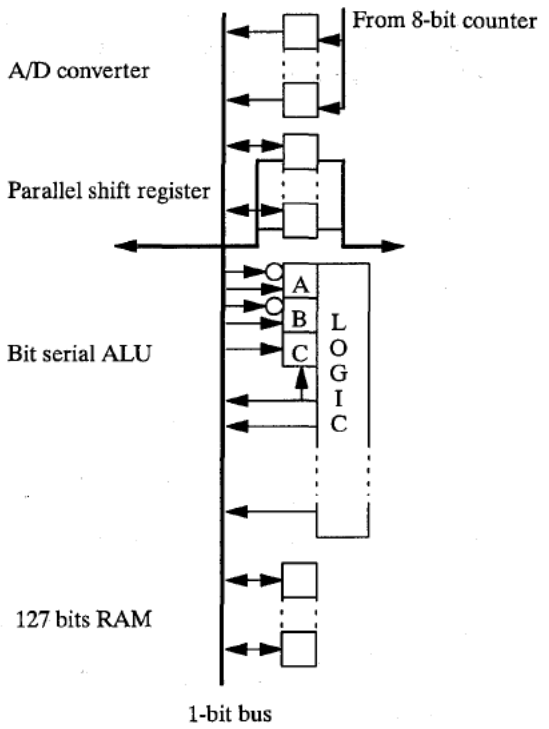


Figure 2.4, The digital architecture of PASIC

PSC_{210}	Operation
000	Write into specified bit position in register from the bus
001	Read from specified bit position in register to the bus
010	Latch
011	Shift left
100	Shift right
101	Rotate left
110	Rotate right

Table 2.3, Parallel shift register operations

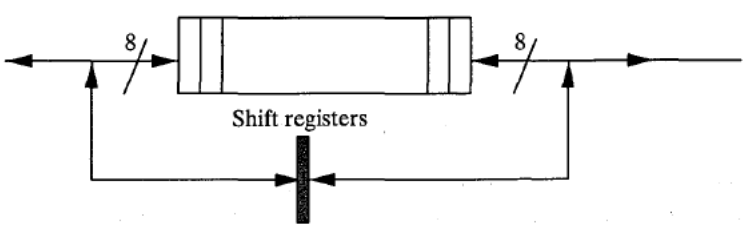


Figure 2.5, Possible data path in the parallel shift register

The ALU is capable of evaluating 1-, 2-, and 3-input Boolean functions. It has 3 input registers and 8 output functions. The three input registers are called A, B, and C. The output functions are listed in Table 2.4. It is possible to load A and B with inverted data.

Name	Boolean function
Carry	$AB+AC+BC$
Sum	$A \oplus B \oplus C$
$\text{Multiplex}_{\overline{c}}$	$C\overline{A}+\overline{C}B$
Multiplex_{c}	$\overline{C}A+CB$
NAND	$\overline{AB}$
NOR	$\overline{A+B}$
XOR	$A\overline{B}+\overline{A}B = A \oplus B$
Not	$\overline{B}$

Table 2.4, Boolean functions in PASIC

The ALU is designed in such a way that it is not possible to take data from one output to the bus and load it into an input register in a single clock cycle. Therefore, the input and output selection of the ALU can use the same decoder. The control signals are called PC_0, PC_1, PC_2, and PC_3, see Table 2.5.

A very common operation in all types of algorithms is addition and when we perform an addition the carry is to be fed back to one of the input registers. To speed up the addition we have provided an extra data path from the carry output to the C register. This extra path is called Carry-feedback and it is controlled by a single bit. It enables us to perform various arithmetics with the same efficiency as other bit-serial processors.

PC_{3210}	Function
0000	A NOR B to bus
0001	A NAND B to bus
0010	Carry to bus
0011	Bus to C
0100	Mux A to bus
0101	Mux $\overline{A}$ to bus
0110	$\overline{\text{Bus}}$ to B
0111	$\overline{B}$ to bus
1000	Bus to B
1001	A XOR B to bus
1010	$\overline{\text{Bus}}$ to A
1011	Bus to A
1100	Sum to bus
1111	Idle

Table 2.5, ALU operations

The memory consists of 127 bits of dynamic RAM and one bit of ROM. The latter is set to zero and located at address zero. With this bit we can produce constants. The memory is controlled with 8 bits, listed in Table 2.6.

$\overline{\text{Read}}$/Write	Address	Operation
0	0000000	Read a constant zero from memory
0	0000001	Read from memory cell 1
0	.....	
0	1111111	Read from memory cell 127
1	0000000	No operation
1	0000001	Write to memory cell 1
1	.....	
1	1111111	Write to memory cell 127

Table 2.6, Memory operations

The 127 bits of RAM are a dynamic non-refreshed memory. This means that the user must make sure that all memory cells are rewritten with a frequency of at least 1 kHz, either as a natural course of events in the program execution or by means of extra rewriting cycles. A refresh cycle takes two clock cycles per bit as we have to read from the memory to the ALU or the shift register, and then write that bit back to the memory. In the worst case the refresh of the memory, where $N_b=127$, $f_{refresh}=1$ kHz, and $f_{system}=20$ MHz, will take 1.3 % of the execution time, see Equation (2.1).

$$\eta = 2N_b \frac{f_{refresh}}{f_{system}} \tag{2.1}$$

In most cases the algorithm uses the memory cells in such a way that we do not need any extra refresh cycles.

2.5 VLSI

The physical VLSI layout of PASIC is shown in Figure 2.6. The boxes to the left of the array is control logic for decoding. Further reading about the VLSI design can be found in [3]. The vertical size of each block is shown in Table 2.7. The horizontal size of PASIC is 7.7 mm.

Region	Size (mm)
Sensor	7.7
A/D-converter	0.3
PE	2.0
PSR	0.5
ALU	0.5
Memory	1.0
	10.0

Table 2.7, The size of the building block of PASIC

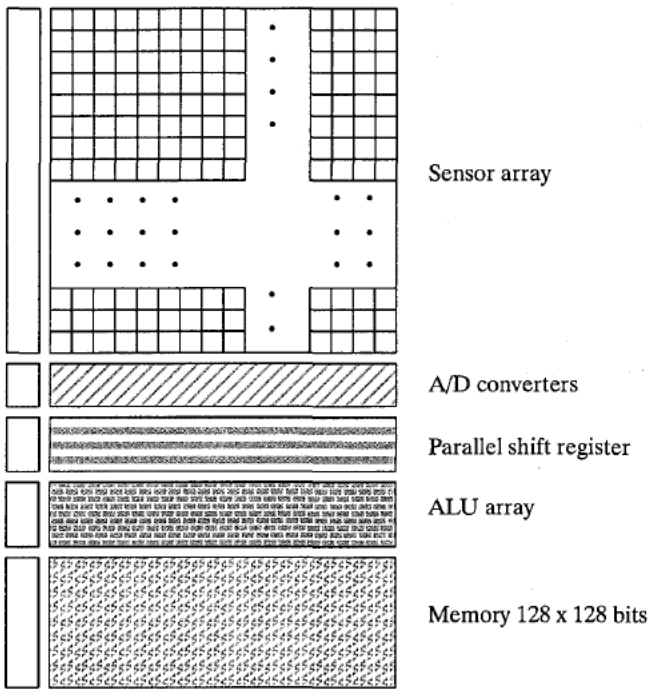


Figure 2.6, The VLSI layout

2.6 The programming model

The control word for PASIC consists of 32 bits, as shown in Figure 2.7. Each unit in Figure 2.6 has its own field in the micro word. Due to the decoding, timing, and delay effects, we are limited in the way that the units can interact with each other. For instance, it is not normally possible to take the content of a boolean function and feed it back to some register, nor to read data from the parallel shift register while it is shifting.

8 bits	5 bits	6 bits	5 bits	8 bits
Sensor address	A/D control	Parallel shift register	ALU	Memory

Figure 2.7, The micro word

The execution flow consists of two parallel processes, one part to control the sensor and the A/D-conversion, and the other to do the digital computation. These two parts are integrated into one single program.

The addressing of the sensor array and the control of the A/D-conversion is typically done once every 256 cycles, while the computation is done both during and between the conversions. A typical example is a low pass filtering of the image, shown in pseudo code in Table 2.8. In this example the result image is obtained by adding the value of the pixel to the left, its own value, and the pixel value to the right.

```
Loop over all rows
  Loop over all bits            ! Read from A/D registers
    Read_A/D
  End-loop
  Reset(R-1)                    ! Pre-charge row R-1
  Sensor_to_AD(R),Start_AD(P)   ! Connect row R to sensor and perform an
                                ! A/D conversion with P bit precision.
  Loop over a neighborhood      ! Get left and right pixel
    Loop over all bits          ! Add them together
      ADD
    End-loop
  End-loop
  Output_pixel                  ! Output the pixels
End-loop
```

Table 2.8, A program example for low pass filtering

The outer loop starts by taking the data from the A/D register, resetting corresponding row in the sensor and starting the conversion of the next row. The time window for light exposure in the sensor is therefore skewed from one line to the next.

Next follows a loop over the neighborhood, in this case the left and right neighbors. We get their values and add them together with the PE's own value. The adding is done in the innermost loop, the bit loop. The row-loop ends with a transfer of bits to the shift register and shift-out of data.

Typically, as in the above example, the computation in PASIC consists of three nested loops. The overall loop is the loop over all rows and is normally done 128 time per frame. However, as we have the possibility to address the row of the sensor randomly, we can reduce the resolution and thereby the number of rows to 64, 32 or less. The size of the neighborhood loop depends on what algorithm we are using, in our case this was 3 x 1. If we use an algorithm that does not use any neighborhood processing, this loop is omitted. Embedded in the other loops are the bit loops, where the number of bits depends on the precision and word length. For the case of simplicity we will hereafter omit all addressing of the sensor and the control of the A/D-converter, and presume that the pixel values are presented in the A/D register when we want them.

As mentioned earlier there are some data moves that seem possible from just looking at Figure 2.4 but which are prohibited in PASIC. The following rules must be obeyed.

(1) Due to self-imposed decoding and addressing limitation, data cannot be taken directly from an ALU output and loaded into an ALU input register in one cycle. Thus, if we want to load an input register with an ALU function we will have to use the memory or the shift register as a temporary storage and execute the procedure in two cycles.

(2) Two input registers cannot receive the same data from the bus in one clock cycle. For instance, we cannot zero two registers in the same cycle. This is also due to the decoding limitation.

(3) We cannot load a bit in its inverted form to the C register, only to the A and B registers.

In some cases the Carry-feedback path helps us to overcome the problem of (1) and (2), as we can retrieve the Carry from the ALU back to the C register without interfering with the bus. An attempt to illustrate the limitations and the data paths is shown in Figure 2.8(a) where each mux-symbol represents a switch, see Figure 2.8(b).

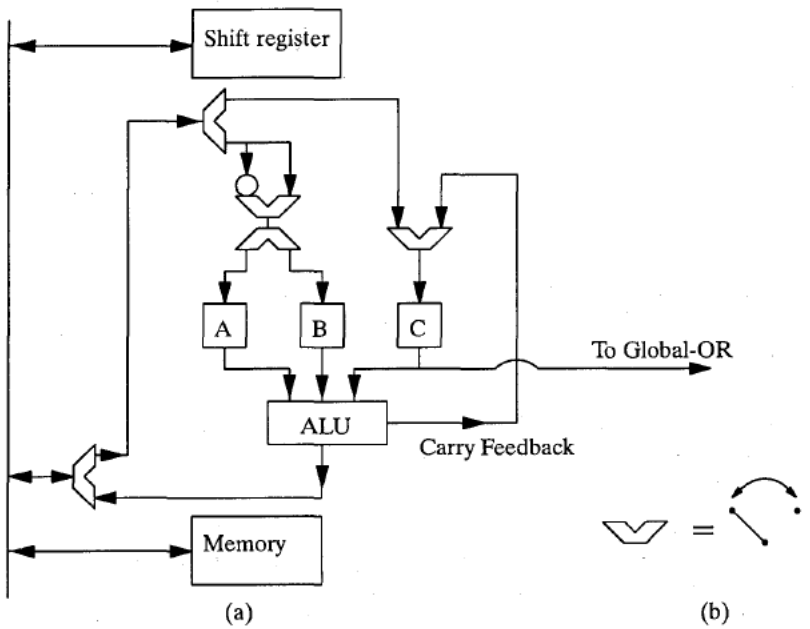


Figure 2.8, A different programming model of PASIC

We can now clearly see that we cannot break the three rules mentioned earlier. For instance, if we use an output function, the only data we can load in the input registers is the Carry to the C register using the CarryFeedback path. Similarly, if we connect the bus to the A register, the output functions and the B input are disabled. However, C can again be loaded from the CarryFeedback.

3 Basic instruction set

3.1 Boolean functions

The ALU in PASIC generates 8 different 3-input Boolean functions. They are all combinatorial functions of the 3 input registers A, B, and C and listed in Table 3.1.

Name	Boolean function
Carry	$AB+AC+BC$
Sum	$A \oplus B \oplus C$
Multiplex_c	$CA+\overline{C}B$
$\text{Multiplex}_{\overline{c}}$	$\overline{C}A+CB$
NAND	$\overline{AB}$
NOR	$\overline{A+B}$
XOR	$A\overline{B}+\overline{A}B = A \oplus B$
Not	$\overline{B}$

Table 3.1, Boolean functions in PASIC

As was shown in chapter 2, the A and B registers can be loaded from the bus with inverted values, ($\overline{X}$ and $\overline{Y}$ respectively). Hereby, all non-trivial 2-input functions can directly be retrieved at the outputs as shown in Table 3.2.

X :	0	0	1	1	Boolean	Input	PASIC
Y :	0	1	0	1	function	operation	function
	0	0	0	0	0		
	0	0	0	1	XY	$A:=\overline{X}, B:=\overline{Y}$	$\overline{A+B}$
	0	0	1	0	$X\overline{Y}$	$A:=\overline{X}, B:=Y$	$\overline{A+B}$
	0	0	1	1	X		
	0	1	0	0	$\overline{X}Y$	$A:=X, B:=\overline{Y}$	$\overline{A+B}$
	0	1	0	1	Y		
	0	1	1	0	$X \oplus Y$	$A:=X, B:=Y$	$A \oplus B$
	0	1	1	1	$X+Y$	$A:=\overline{X}, B:=\overline{Y}$	$\overline{AB}$
	1	0	0	0	$\overline{X+Y}$	$A:=X, B:=Y$	$\overline{A+B}$
	1	0	0	1	$\overline{X \oplus Y}$	$A:=\overline{X}, B:=Y$	$A \oplus B$
	1	0	1	0	$\overline{Y}$	$B:=Y$	$\overline{B}$
	1	0	1	1	$X+\overline{Y}$	$A:=\overline{X}, B:=Y$	$\overline{AB}$
	1	1	0	0	$\overline{X}$	$B:=X$	$\overline{B}$
	1	1	0	1	$\overline{X}+Y$	$A:=X, B:=\overline{Y}$	$\overline{AB}$
	1	1	1	0	$\overline{X \cdot Y}$	$A:=X, B:=Y$	$\overline{AB}$
	1	1	1	1	1		

Table 3.2, All possible two input Boolean functions

The actual execution of a 2-input function is illustrated in Figure 3.1 by an AND operation between the memory cell X and memory cell Y. The result is going to be stored in memory cell Z. First we place an inverted version of the memory content of X in register A. In the next cycle we put an inverted version of Y in register B. Finally we take the content of the output register $\overline{A+B}$ and place that in memory cell Z.

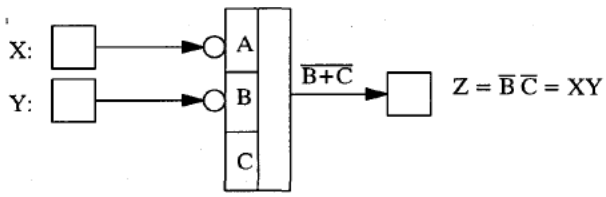


Figure 3.1, Z := X AND Y

3-input functions are performed in the same manner. For instance, a very useful operation is to select one of two **b**-bit vectors, in X and Y respectively, depending on the value of bit T, and store this vector in Z. This is called multiplexing. (We will often work with multi-bit words, for example a byte = 8 bits. Therefore, we will hereafter refer to the number of bits by the constant **b**). First we place T in the C register. Then we move the first bit in X and Y over the bus to registers A and B. We then read out the output CA+$\overline{C}$B and store in Z. This process is than repeated **b** times. The whole operation takes **3b+1** clock cycles and is described in Figure 3.2.

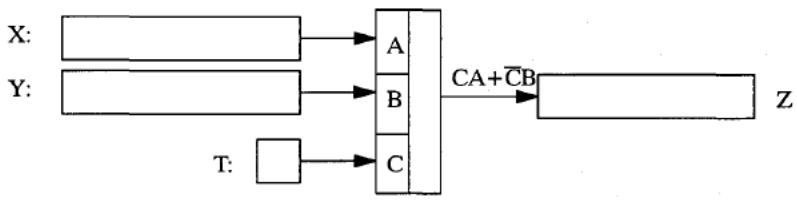


Figure 3.2, Multiplexing

There are 256 possible 3-input functions. To see exhaustively how many of these functions we can realize in PASIC we would have to build a large table in the same manner as table 3.2. However, the 256 entry table can be reduced on the following grounds.

a) If one function f(x,y,z) can be implemented so can f(y,x,z) or any other permutation of the variables, since we are free to permute the inputs to (A,B,C).

b) If a function f(x,y,z) = f(x,y,q) where q is a don't care, this function is really a 2-input function which we proved in Table 3.2 can be implemented directly.

c) If a function f(x,y,z) can be implemented so can f($\overline{x}$,y,z), f($\overline{x}$,$\overline{y}$,z), and so on, since we have the possibility to invert the inputs to the A and B register.

When we use these three rules to find functions which are equivalently or trivially executed in PASIC, the number of different 3-input functions is reduced to 24, see Table 3.3.

	0 1 0 1 0 1 0 1	X	
	0 0 1 1 0 0 1 1	Y	
	0 0 0 0 1 1 1 1	Z	
#		Function	Name/equivalent function
0	0 0 0 1 0 1 1 1	$xy+xz+yz$	Carry
1	0 1 0 1 0 0 1 1	$x\bar{z}+yz$	Mux
2	0 1 1 0 1 0 0 1	$x\oplus y\oplus z$	Sum
3	0 1 1 0 1 1 1 1	$\overline{(x\oplus y)}z$	Comp1
4	0 0 0 0 1 0 0 1	$x\oplus y+z$	Comp2
5	0 0 0 0 0 0 0 1	xyz	3-AND
6	0 0 0 0 0 1 1 1	$xz+yz$	
7	0 0 0 1 0 1 1 0	$xy\bar{z}+x\bar{y}z+\bar{x}yz$	$x(y\oplus z)+\bar{x}yz$
8	0 0 0 1 1 0 0 0	$xy\bar{z}+\bar{x}yz$	
9	0 0 0 1 1 0 0 1	$xy\bar{z}+\bar{x}yz+xyz$	$xy\bar{z}+\overline{(x\oplus y)}z$
10	0 0 0 1 1 1 1 0	$xy\bar{z}+\bar{y}z+\bar{x}z$	$(xy)\oplus z$
11	0 0 0 1 1 1 1 1	$xy+z$	
12	0 0 1 1 1 1 0 1	$xy+x\bar{z}+\bar{x}z$	$xy+x\oplus z$
13	0 1 1 0 1 0 1 1	$x\oplus y\oplus z+x\bar{y}z$	
14	0 1 1 1 1 1 1 0	$x\bar{y}+\bar{x}z+y\bar{z}$	
15	0 1 1 1 1 1 1 1	$x+y+z$	3-OR
16	0 1 1 0 1 0 0 0	$\bar{x}yz+x\bar{y}z+xy\bar{z}$	
17	0 1 1 0 1 0 1 0	$x\bar{y}+\bar{x}yz+\bar{y}z$	$\bar{y}(x\oplus z)$
18	1 0 0 0 0 0 0 0	$\bar{x}\cdot\bar{y}\cdot\bar{z}$	3-NOR
19	1 0 0 1 1 0 0 0	$xy\bar{z}+\bar{x}\bar{y}$	
20	1 0 1 0 1 0 0 0	$\bar{x}\cdot\bar{y}+\bar{y}\cdot\bar{z}$	
21	1 0 1 1 1 1 0 0	$x\bar{z}+\bar{x}z+\bar{y}\bar{z}$	$x\oplus z+\bar{y}\cdot\bar{z}$
22	1 1 1 0 1 0 0 0	$\bar{x}\cdot\bar{y}+\bar{x}\cdot\bar{z}+\bar{y}\cdot\bar{z}$	
23	1 1 1 0 1 0 1 0	$\bar{x}\cdot\bar{z}+\bar{y}$	
24	1 1 1 1 1 1 1 0	$\bar{x}+\bar{y}+\bar{z}$	3-NAND

Table 3.3, All unique three-input boolean functions

The first three functions in table 3.3, functions number 0 to 2, are the only one which are implemented in the original version of PASIC. In fact we have function number 1 in 2 versions, both $CA+\overline{C}B$ and $\overline{C}A+CB$ since we do not have the possibility to invert the C input. Number 3 and 4 are suggested in section 10.1 as an extension to the PASIC functions. Possibility to invert the input to the C register would reduce Table 3.3 to just 16 unique functions.

In some architectures, e.g. the Connection Machine, all 256 functions have been implemented. Consequently, in the Connection Machine any 3-input Boolean function is executed in four clock cycles. The first three are used to initialize the registers and the fourth to read out the output. One way to realize a 256-function machine is shown in Figure 3.3. The function is loaded into the 8-bit memory common to all the processors on the chip. Each processor generates its function output via a decoder followed by an AND/OR-network.

To say that a 3-input function takes 4 cycles to compute is of course not entirely true since the loading of the 8 bits into the memory takes at least one cycle. However, if we use the same function more than once or if the loading could take place in parallel with some other operation, 4 cycles per output bit is close to the true number.

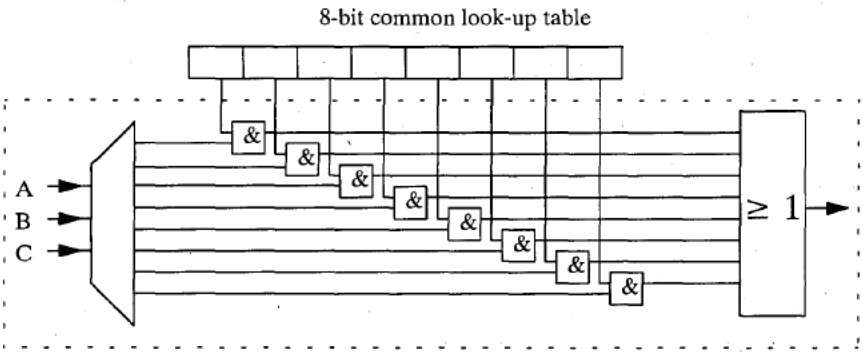


Figure 3.3, A 256-function table look-up implementation

In PASIC the ALU only produce the three first 3-input functions in Table 3.3. The question then is: in how many clock cycles can we realize all 3-input functions. This is shown in Table 3.4, where we have compared PASIC with 256-M, a fictitious machine, that has the capability to realize all 256 function.

Table 3.4 shows that PASIC realizes most of the functions losing only one or two clock cycles compared to the optimum. Also, one should bear in mind that all functions do not have the same usage frequency. In general, Sum, Carry, and multiplexing, which we have implemented in PASIC, are far more common than other operations.

#	Boolean function	256-M	PASIC	Diff
0	$xy+xz+yz$	4	4	0
1	$x\bar{z}+yz$	4	4	0
2	$x\oplus y\oplus z$	4	4	0
3	$\overline{(x\oplus y)}z$	4	6	2
4	$x\oplus y+z$	4	6	2
5	xyz	4	5	1
6	$xz+yz$	4	5	1
7	$xy\bar{z}+x\bar{y}z+\bar{x}yz$	4	8	4
8	$xy\bar{z}+\overline{xy}z$	4	10	6
9	$xy\bar{z}+\overline{xy}z+xyz$	4	8	4
10	$xy\bar{z}+\bar{y}z+\bar{x}z$	4	6	2
11	$xy+z$	4	6	2
12	$xy+x\bar{z}+\bar{x}z$	4	8	4
13	$x\oplus y\oplus z+x\bar{y}z$	4	9	5
14	$x\bar{y}+\bar{x}z+y\bar{z}$	4	8	4
15	$x+y+z$	4	5	1
16	$\overline{xy}z+x\overline{yz}+\overline{xyz}$	4	9	5
17	$x\bar{y}+\overline{xy}z+\bar{y}z$	4	6	1
18	$\bar{x}\cdot\bar{y}\cdot\bar{z}$	4	5	3
19	$xy\bar{z}+\overline{xy}$	4	8	4
20	$\bar{x}\cdot\bar{y}+\bar{y}\cdot\bar{z}$	4	6	2
21	$x\bar{z}+\bar{x}z+\overline{yz}$	4	8	4
22	$\bar{x}\cdot\bar{y}+\bar{x}\cdot\bar{z}+\bar{y}\cdot\bar{z}$	4	6	2
23	$\bar{x}\cdot\bar{z}+\bar{y}$	4	6	2
24	$\bar{x}+\bar{y}+\bar{z}$	4	5	1

Table 3.4, Performance of PASIC in three-input functions

PASIC does not have a specially dedicated accumulator. However, by using the Carry-feedback we can in some cases emulate an accumulator with the C-register. Depending on the value of A, the Carry function (3.1) can be interpreted as (3.2) or (3.3).

$$Cy = AB+AC+BC \tag{3.1}$$

$$Cy_{A=0} = BC \tag{3.2}$$

$$Cy_{A=1} = B+C \tag{3.3}$$

Figure 3.4 illustrates this in a logic-diagram of Equation (3.1).

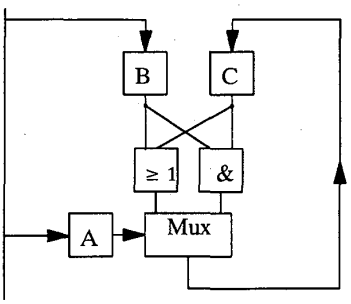


Figure 3.4, Carry feedback used as an accumulator

As another example how to use the Carry-feedback data path, we show in pseudo code in Table 3.5 how to implement XZ+YZ, function 6 in Table 3.4.

```
Mem(X)->A
Mem(Z)->B
Mem("0")->C
Mem(Y)->A, Carry-Feedback          A=Y, B=Z, C=XZ
Carry->M(Result)                   Carry=YZ+XYZ+XZZ=XZ+YZ
```

Table 3.5, Implementation of XZ+YZ

The operation takes 5 clock cycles, which is just one cycle more than the minimum, and that is used to initialize the C register with a 0.

3.2 Addition and subtraction

Basic arithmetic is supported in PASIC by two Boolean functions:

```
Carry = AB + AC + BC
Sum   = A ⊕ B ⊕ C.
```

When performing an addition the object is to add the **b**-bit vectors of X and Y and place the sum in the **b+1** long vector Z, see Figure 3.5. First we place a '0' in the C register, the LSB of X in A and the LSB of Y in B. We are now able to read out the result from the Sum output into LSB of Z. At the same time we use the carry-feedback facility and place the Carry in C without interfering with the bus. We repeat this process until we has processed the MSB of X and Y. Finally we copy the Carry output into the **b+1** location in Z, i.e. the MSB. Hence, we have extended the output word-length from **b** to **b+1**. The whole addition takes **3b+2** clock cycles.

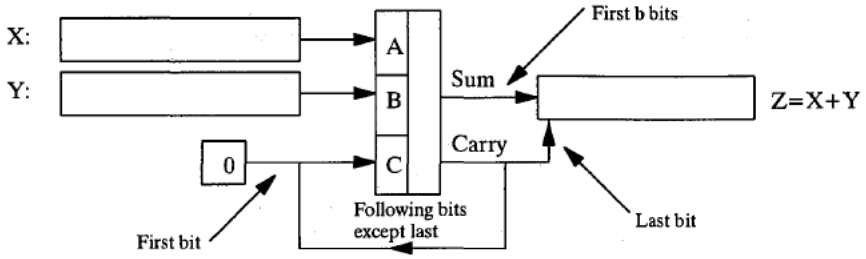


Figure 3.5, Addition of unsigned positive numbers

Addition of numbers in two-complement representation is shown in Table 3.6. The same sub cycles are carried out for the first **b** bit positions. To generate bit **b+1**, i.e. the sign bit S_{b+1}, we let the MSB of X and Y remain in A and B, while we perform a Carry-feedback into C. We now have X_{b-1} in A, Y_{b-1} in B and C_b in C. Finally, we place the sum in the **b+1** position in Z, see Table 3.6. The number of clock cycles is **3b+2** the same as in Figure 3.5.

	C_b	C_{b-1}	C_{b-2}	..		E	A	
	X_b	X_b	X_{b-1}	..	X_1		B	
+	Y_b	Y_b	Y_{b-1}	..	Y_1		C	$D = A \oplus B \oplus C$
	S_{b+1}	S_b	S_{b-1}	..	S_1		D	$E = AB + AC + BC$

Table 3.6, Two-complement addition

Two-complement subtraction is performed almost in the same manner as a two-complement addition. The difference is that the number in the Y-vector is now inverted when loaded into the B register and the C register is set to 1, see Figure 3.6. Hence, we use the identity $X-Y = X+\overline{Y}+1$, where $\overline{Y}$ is the bit vector Y with all bits inverted. The number of clock cycles are **3b+2**, which is the same result as for the addition.

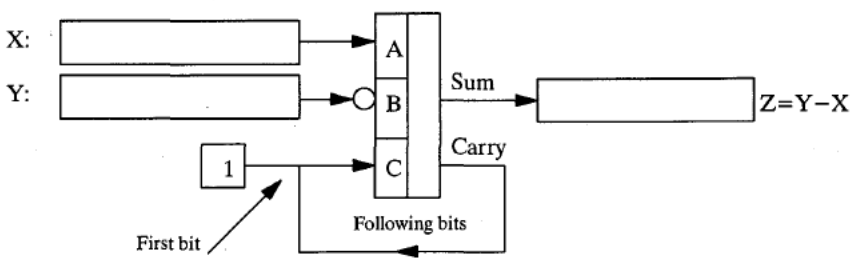


Figure 3.6, Subtraction, in two-complement representation

3.3 Multiplication of two variables

The following description and analysis is adapted from [8]. We assume that we have two **b**-bit vectors in two-complement representation, in memory location X and Y. The result, this time **2b** bits long, will be placed in memory location Z, see Figure 3.7.

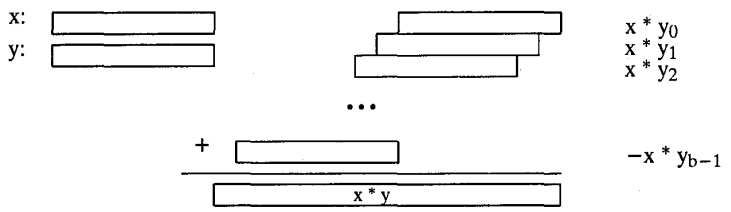


Figure 3.7, Multiplication with bit vectors

For each Y-bit = 1, we add a shifted version of X, and for Y-bit = 0 we add zero. The proper vector (P:=X or 0) to be added is obtained by using the multiplexing function, $CA+\overline{C}B$, see Figure 3.10. Thus we feed the A register with the X vector, B with zero and C, the multiplexer controller, with the appropriate Y bit. This operation, taking **2b+2** clock cycles, is repeated **b** times. Thus, the number of clock cycles so far is $\mathbf{2b^2+2b}$. The first **b−1** P-vectors are added together and the last one is subtracted from the resulting sum. The very first P-vector is just stored in the place of the final product as shown in Figure 3.10. The first addition consists of **3b+1** clock cycles. That is because the LSB in the upper operand in Figure 3.8 is left unchanged, i.e. no operation in a bit serial machine, and that the MSB in this operand is left unchanged and copied to itself in the two last bit additions.

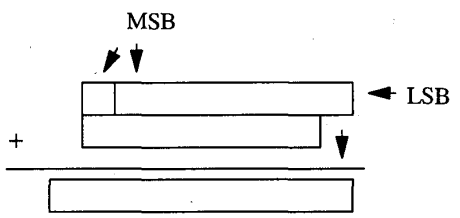


Figure 3.8, First addition step in multiplication

The next **b−3** additions consists of **3b+2** clock cycles. The LSB in the upper operand is left unchanged, which is the same as in the first addition. The final operation is a subtraction, which is carried out in the normal way. We do not have to consider the fact that the last vector has a number of invisible zeroes to the left, see Figure 3.9. The reason is that we will invert this vector which means that all these 0's will become 1's, and the C register will be initialized to 1. This will have the effect that **b−1** bits in the previous result will just be copied. Thus, the subtraction works just like the other addition and requires **3b+2** clock cycles.

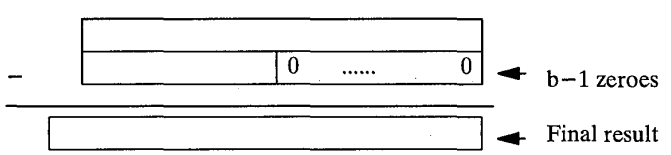


Figure 3.9, Last addition step in multiplication

The whole multiplication procedure is illustrated in Figure 3.10. Thus, the complete process will take $\mathbf{(2b+2)b+(b-2)(3b+2)+3b+1=5b^2+b-3}$ clock cycles.

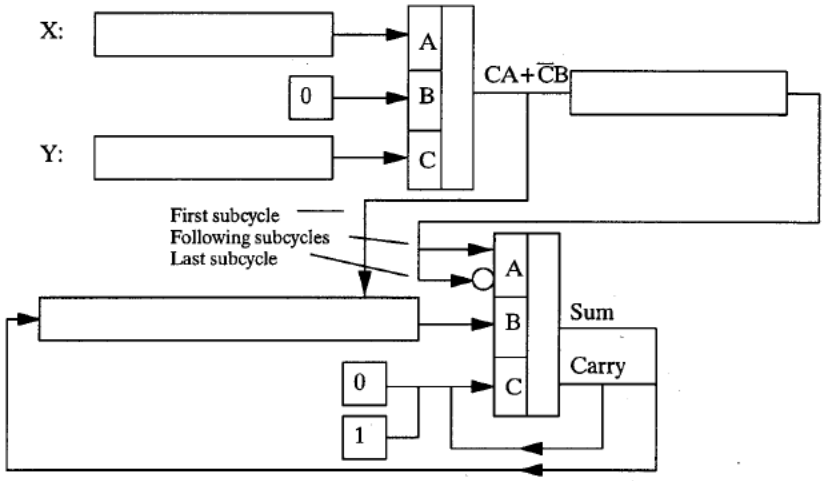


Figure 3.10, Multiplication with two variables

3.4 Multiplication with a constant

In many cases one of the factors in a multiplication is a constant, i.e. common for all the PE's and known at compilation time. For instance, in standard convolution the coefficients are determined before the execution starts. In those cases we can reduce the number of clock cycles by embedding the constant in the micro-program. The reductions of micro code instructions and clock cycles amounts to the following.

(a) The multiplexing phase of the ordinary multiplication procedure can be omitted as the control unit knows whether to add zero or a shifted version of the vector.

(b) All additions with a zero vector can be omitted on the same premisses as in (a).

Figure 3.11 illustrate a multiplication with the constant y=−19.

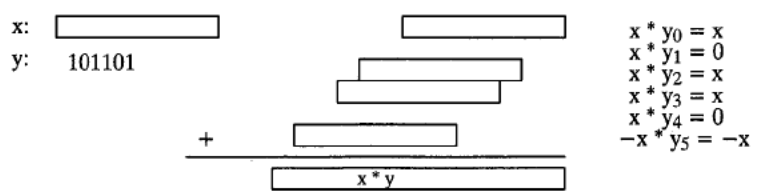


Figure 3.11, Multiplication with a constant

We transform Figure 3.10 into Figure 3.12 where the multiplexing is left out and the addition/subtraction is only carried out for $y_i = 1$.

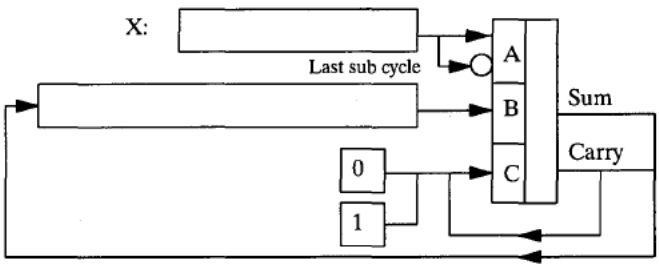


Figure 3.12, Multiplication with a constant

In an arbitrary case, we can expect to find roughly 50% 1's and 50% 0's in the constant vector. Hence, we reduce the number of additions from **b−1** to **b/2−1** and as we do not perform any multiplexing, the multiplication only needs $(3b+2)(b/2-1)=3b^2/2-2b-2$ clock cycles. This is a speed-up with a factor of 3.3 compared to the two-variable case.

Further reduction can be made by using the Signed Digit Code, SDC. In the SDC each bit represents a value of 0, +1, or −1. In the micro-program this corresponds to no operation, add, and subtract. By using the SDC we are able to increase the number of 0 in representation of the constant, and thereby reducing the number of additions, see Table 3.7.

Number	2-comp code	SDC
6	00110	0 0+1+1 0
7	00111	0+1 0 0−1
15	01111	+1 0 0 0−1
−5	11011	0−1 0+1+1

Table 3.7, Conversion to SDC

We see that the execution speed benefits from this code if we have three or more consecutive 1's, otherwise we do not gain anything. On the average we could expect to reduce the number of addition/subtraction by 25%, i.e. from **b/2−1** to **3b/8−1**. The number of clock cycles would then be $(3b+2)(3b/8-1)=9b^2/8-9b/4-2$. Compared to the two variable case we have then reduced the number of cycles by a factor of 4.4.

3.5 Processor communication

In PASIC interprocessor communication takes place over the 8-bit bi-directional parallel shift register. A transfer of data from one processor to another, consists of three phases: Transfer of data into the shift register, Shifting the data N steps in the desired direction, Moving the data from the shift register to the memory or the ALU. These three steps are shown in Figure 3.13(a)−(c).

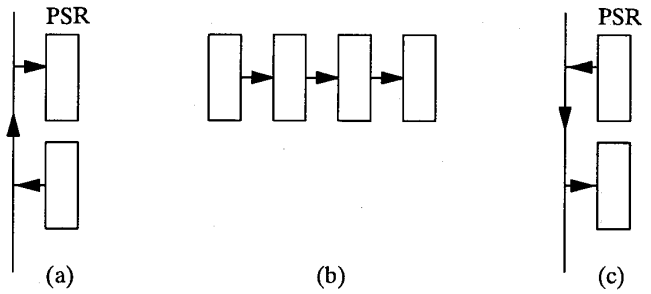


Figure 3.13, The three shift phases

If the data size **b** is less or equal to 8 this operation will take **b+k+b=2b+k** cycles. However, in some cases data size is larger and we will have to do the transfer process more than once. In the general case the number of clock cycles needed is shown in Equation (3.4).

$$t = \begin{cases} 2b + k & b \leq 8 \\ (16 + k) * int(\frac{b}{8}) + 2 * (b \mod 8) + k & b > 8 \end{cases} \quad (3.4)$$

3.6 Interprocessor operations

The communication between the processors is often a part of the computation. We will illustrate this fact by describing the following three binary image operations: Dilation, Erosion, and Connectivity preserving shrinking, followed by some multi-bit communications and operations.

Dilation expands the regions in the image that contain 1's. It can be implemented by an OR-function that delivers a 1 if there are at least one 1 in the predefined region, see Figure 3.14.

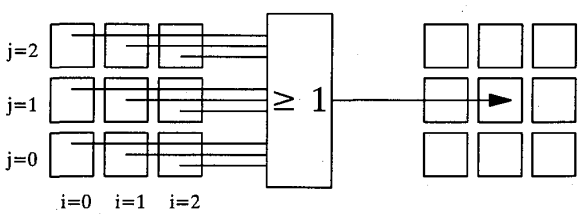


Figure 3.14, Dilation

If we denote each pixel p_{ij}, where i and j are the position in the 3 by 3 neighborhood in Figure 3.14, we get Equation (3.5). This expression can be separated into one horizontal and one vertical OR operation, see Equation (3.6).

$$f_{dilation}(p_{ij}) = p_{00} + p_{01} + p_{02} + p_{10} + p_{11} + p_{12} + p_{20} + p_{21} + p_{22} \quad (3.5)$$

$$f_{dilation}(p_{ij}) = (p_{00} + p_{01} + p_{02}) + (p_{10} + p_{11} + p_{12}) + (p_{20} + p_{21} + p_{22}) \quad (3.6)$$

The pixels within one parenthesis in Equation (3.6), belongs to the one and same PE. Thus, the parenthesis are computed in parallel and taken 5 cycles according to Table 3.4. Each processor then takes the results from its nearest neighbors and compute the final result.

These steps are graphically shown in Figure 3.15. For each line of the image the 3x3 dilation takes **10** cycles.

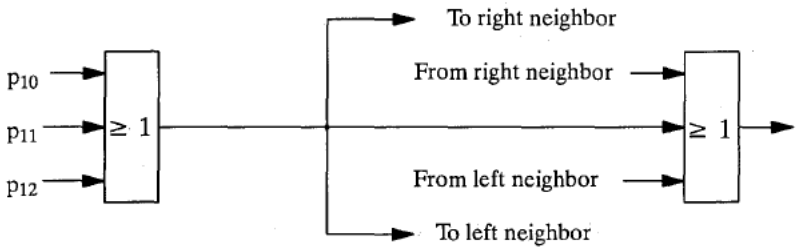


Figure 3.15, Two-step computation of dilation

Erosion is performed in the same way as the dilation. The difference is that we use logic-And instead of logic-OR, see Equation (3.7) and (3.8). An erosion therefore also takes **10** cycles to perform.

$$f_{erosion}(p_{ij}) = p_{00} \cdot p_{01} \cdot p_{02} \cdot p_{10} \cdot p_{11} \cdot p_{12} \cdot p_{20} \cdot p_{21} \cdot p_{22} \tag{3.7}$$

$$f_{erosion}(p_{ij}) = (p_{00} \cdot p_{01} \cdot p_{02}) \cdot (p_{10} \cdot p_{11} \cdot p_{12}) \cdot (p_{20} \cdot p_{21} \cdot p_{22}) \tag{3.8}$$

If we perform an erosion, objects like in Figure 3.16(a) will be destroyed in the sense that they will be separated into two pieces, as in Figure 3.16(b).

Figure 3.16, Erosion effects

In some algorithms these effects should be replaced by Connectivity preserving erosion. One implementation employs an operator from four different directions, north, south, west, and east. For each direction we use a set of three different templates. Figure 3.17(a) shows a set of west direction templates. If any of the templates matches a region in the image, the center pixel in the region will be set to 0. A template is matching a region in the image, if the positions in the template where we have 1's correspond to 1's in that region and 0's in that region and 0's in the templates correspond to 0's. The - means don't care.

$$\begin{matrix} - & - & 1 \\ 0 & 1 & 1 \\ - & 0 & - \end{matrix} \qquad \begin{matrix} - & - & 1 \\ 0 & 1 & 1 \\ - & - & 1 \end{matrix} \qquad \begin{matrix} - & 0 & - \\ 0 & 1 & 1 \\ - & - & 1 \end{matrix}$$

(a)

$$\begin{matrix} - & b & d \\ a & x & e \\ - & c & f \end{matrix}$$

(b)

Figure 3.17, West directions templates

The template definition of Figure 3.17(a) and the notation in Figure 3.17(b), corresponds to the Boolean function (3.9), which can be reduced to (3.10).

$$s = x\ \overline{(\bar{a}xe\bar{c}d + \bar{a}xedf + \bar{a}xe\bar{b}f)} \tag{3.9}$$

$$s = x\ (a + \bar{e} + \bar{d}(b + \bar{f}) + c\bar{f}) \tag{3.10}$$

The computation of Equation (3.10) takes **22** cycles. However, this has to be done for all four directions. Thus, one step of Connectivity preserving shrinking will take **4*22=88** cycles.

In section 3.5 we transferred data from one processor to another. A more typical use of the shift register in neighborhood communication, is shown in Figure 3.18. The procedure implements the convolution with the kernel 1 2 1 which can be decomposed to `1 1 * 1 1`. Convolutions are more fully discussed in section 4.1.

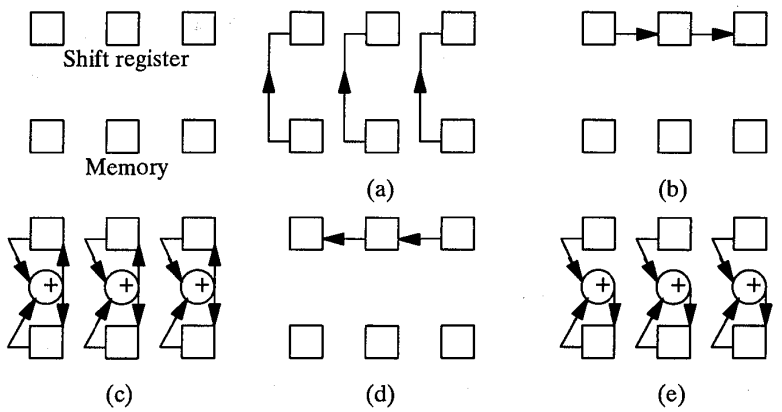


Figure 3.18, Parallel shift register

In phase (a) we transfer the data from the memory to the shift register, which takes **b** cycles. Then we shift the PSR one step to the right, which takes one clock cycle. In (c) we compute the 1 1 convolution by adding the memory with the PSR in **3b+2** clock cycles. Phase (d) means a shift to the right in **1** cycle. Finally, in phase (e), we compute the complete convolution when we add the PSR with the temporary result in the memory. This takes **3(b+1)+2** clock cycles and altogether this convolution takes **7b+9** cycles.

3.7 The Global-OR communication

In many cases the control unit, hitherto invisible, has to have some information on what is going on in the chip. For instance, if we are doing a multiplication we would like to know if we had had overflow, underflow or neither. The means to implement such a link to the control unit in PASIC is Global-OR. Physically, the Global-OR is a wired OR-gate which connects all the C register in the processor array, see Figure 3.19.

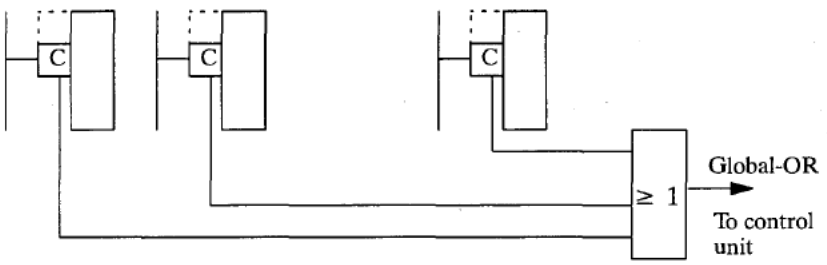


Figure 3.19, Global−OR

One somewhat more sophisticated utilization of the Global-OR function is to find the largest value in the array. We assume that all the values are positive and unsigned. Initially, each PE sets a status bit in the C register which is to stay 1 as long as the PE has a value that might be the largest.

We first perform an AND between the status bit and the MSB of the value, called D_7, and place the result in the C register so that it is connected to the Global-OR, see Equation (3.11).

```
C (Global-OR) <- Status & D7                                    (3.11)
```

If the control unit, CU, finds that Global-OR is 1, all PEs with $D_7=0$ will have there status bit reset to 0, since we know that there is at least one PE with a value $D_7=1$. Thus, we execute Equation (3.12). This means that we keep the status bit set as long as the D_i bit is 1. If D_i is 0 and no other PE with the status bit = 1 has a 1 in its D_i, the status bit also remains set.

```
Status <- Status & Global-OR + Status & D7                      (3.12)
```

This operation takes $4*(^2\log N)+2$ cycles which is **30** cycles for N=128. In Table 3.8 we can see how this is done on the programming level, where the computation and content of the registers can be seen step by step.

```
A:=0;
C:=M(Stat);                          ! A    B      C
FOR i:=7 TO 0 DO                     ! ========================
  B :=M(Di);                         ! 0    Di     Stat
  B :=M(Stat); CarryFeedback;        ! 0    Stat   Stat & Di
  iA:=Gor (From CU via constant)     ! Gor  Stat   Stat & Di
  A :=0; CarryFeedback;              ! 0    Stat   Stat & Di + Stat & Gor
ROF
```

Table 3.8, Pseudo code for Max finding

We here assumed that the Global-OR test only took 1 clock cycle. However, we have to consider that the test consists of two parts, computation of Global-OR and depending on that result, computation of the next address to be executed in the micro-sequence. The operation speed is only 20 MHz which presents no time critical problem for the computation of Global-OR, even though the signals have to travel over the entire chip. The address computation is on the other hand very time critical. A normal micro-sequencer first take the input signal, then compute the next address, which can take a lot of time. However, in this case we,

where the only data dependent operation is to load the A register with a 1 or a 0, we only need to multiplex the next address rather than an arithmetic computation. If we have a multiplexing system with the possibility to pipeline the code, the Global-OR check will only take two cycles. One, to store the multiplexed value in the memory address and one to load the next instruction. In that case the max finding will take **5b+2** cycles, where **$b=^2\log N$**.

The Global-OR can not only be used for maximum finding, but for more general operations. For instance, if we invert the D_i bit to the B register we change the operation from max- to min-finding. As another variation, if only some PEs have their status bit set initially due to some previous intermediate result of processing, and we use the position number of each PE in the array as the D value, we select the rightmost PE which had the status bit set from the beginning. Table 3.9 lists some possibilities of the Global-OR operation.

Input to D_i	Invert D_i	Result	Name
Bin number	No	Rightmost pixel	P_{right}
Bin number	Yes	Leftmost pixel	P_{left}
PE value	No	Maximum value	P_{max}
PE value	Yes	Minimum value	P_{min}

Table 3.9, Possible Global operations

These operations can also be extended to cover a the most of the Global operations in LAPP, see[9]. For instance in the operation RFILL, where we want to set every status bit to 1 to the right of the leftmost status bit that is 1. This is done by storing the position of the leftmost 1, called p_{left}. This number comes for free in the Leftmost algorithm, since we just have to store the Stat & $\overline{D_i}$ result in the memory. Then by taking the difference between the PE's position and p_{left}, and only saving the sign bit, each PE has a bit that indicates if it is to the left or to the right of p_{left}. This will take one extra cycle within the loop plus **$2*(^2\log N)+1$** cycles for the sign detection. This makes a total of **$7*(^2\log N)+3$** cycles.

3.8 Summary of PASIC

Table 3.10 summarizes the basic instructions set of PASIC. In spite of its simplicity PASIC for the most parts executes basic instructions with the same speed as its competitors.

Operation	Number of clock cycles	≈
2-input logic	3	3
3-inputs logic (typical)	4	4
Multiplexing	3b+1	3b
Addition/Subtraction	3b+2	3b
Multiplication Unsigned	$5b^2+b-2$	$5b^2$
Multiplication signed	$5b^2+2b-3$	$5b^2$
Multiplication with constant	$3b^2/2-3b/2-2$	$1.5b^2$
Multiplication with SDC	$9b^2/8-9b/4-2$	$1.1b^2$
Dilation/Erosion	10	10
Connectivity preserving shrink	22	22
Convolution 1 2 1	7b+9	7b
Max finding with Global-OR		5b

Table 3.10, Summary of PASIC execution

4 Basic routines

4.1 Convolutions

One of the most common operations in image processing is convolution. A convolution consists of multiplications with constants followed by accumulations. Figure 4.1 shows some examples of convolution kernels.

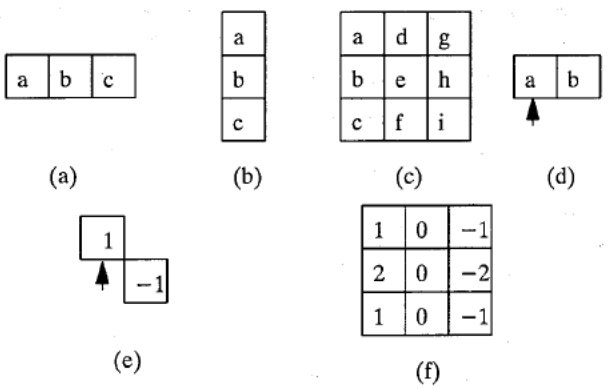


Figure 4.1, Some convolutions kernels

Figures 4.1(a) and (b) are 1D kernels, oriented in horizontal and vertical direction respectively, and (c) is a 2D kernel. In cases where we do not have an odd filter size or when the result is to be placed somewhere else than in the middle, the position where the result will be stored in the output image is indicated with an arrow, as in Figure 4.1(d). Figures (e) and (f) are two edge detection kernels, called Roberts cross and Sobel respectively.

To compute an arbitrary kernel, for instance a 3 by 3, requires 9 multiplications with constants and 8 additions. The transportation time is normally negligible and transparent since we place the pixel in the parallel shift register at the same time as we compute the multiplication. If we use the SDC described earlier we obtain the cycle count in Table 4.1. For **b=8** this will be 856 clock cycles.

Operation		No of cycles
Multiplication	$9 * (9b^2/8 - 9/4b - 2)$	$81b^2/8 - 81b/4 - 18$
Addition	$8 * (3(2 * b) + 2)$	$48b + 16$
Total		$81b^2/8 + 111b/4 - 2$
	$\approx$	$10b^2+27b$

Table 4.1, A general 3 by 3 convolution

In many cases we have the possibility to separate a large kernel into a sequence of smaller ones. Figure 4.2 shows the decomposition of the Sobel operator.

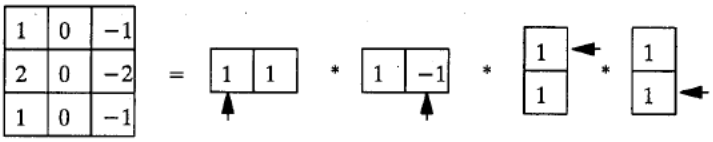


Figure 4.2, Decomposition of the Sobel operator

This decomposition shows that this 3 by 3 convolution can be reduced to 3 additions and 1 subtraction, see [7]. Thereby, we will only need **4*(3b+2)=12b+8** cycles. For **b=8** this means a reduction from 856 to 104 cycles. An other advantage of using separable kernels is that we can use the temporary results in other kernels, see Figure 4.3.

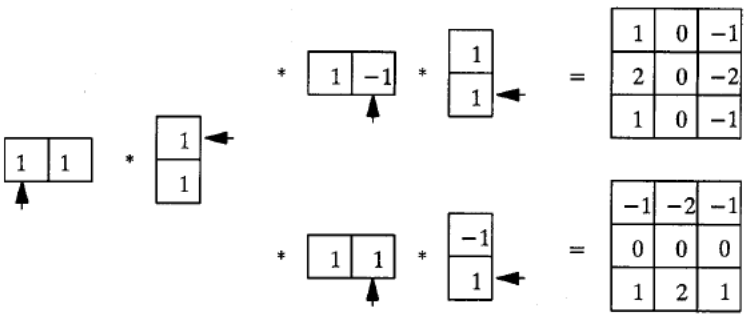


Figure 4.3, Decomposition of the Sobel operators

The computation of the two Sobel kernels in Figure 4.3 now only requires 6 operations compared to 8 operations if we did not use the temporary results.

4.2 Sparse filter kernels. Averaging

Averaging over a neighborhood of N pixels seems to require N−1 additions. However, by using so called sparse filters we are able to reduce the number of additions to $O(^2\log N)$. Averaging could be described as a convolution with decomposable filter kernels. For instance 8*1 convolution with 1's could be decomposed as in Figure 4.4. The *-sign means a convolution and we notice that each convolution on the right side is a simple addition.

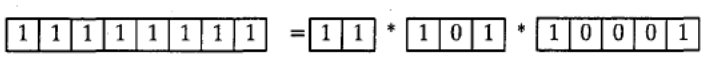


Figure 4.4, Decomposition with sparse filter

Obviously, if N is a power of 2 we can perform averaging with $^2\log N$ additions. When N is not a power of 2 things are less obvious. The general strategy is that when we create a filter with a size of $N=2^n$, we also, as an intermediate result, create the filter of sizes 2^{n-1}, 2^{n-2},.., 2^1, see Figure 4.5. The original value, which by the same notation is called 2^0, is also available.

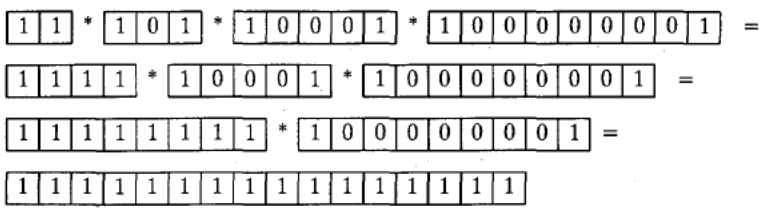


Figure 4.5, Intermediate results in sparse filtering

Now, if the averaging filter size is N where N is not the power of 2 we have to arrange the procedure with some extra operations, see Figure 4.6. Here, a filter of size 11 is computed as a size 8 filter the result of which is added to a shifted version of a size 2 filter and a shifted original pixel. The arrows indicate where the result is placed after the convolution. The `0 0 0 0 1 1` kernel is a normal intermediate result `1 1` shifted an appropriate number of steps. Altogether, the size 11 filter requires **3** additions for the 8 filter, **1** addition to add the intermediate size 2 filter, and **1** addition to add the original pixel. This makes a total of **5** additions.

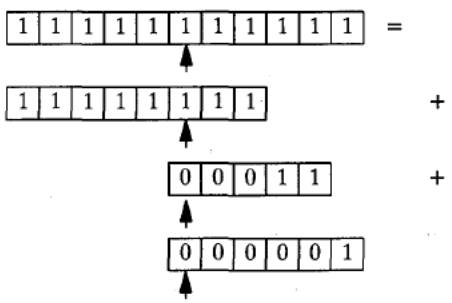


Figure 4.6, Combining intermediate results for the final result

Thus, if we have a size N filter, it can be constructed as a size 2^n filter to which we add some of its intermediate result. Here $n=[^2\log N]$ where [x] stand for the integer part of x.

In the worst case, when N+1 is a power of 2, all the intermediate results have to be added to get the final result. For instance to get a size 15 filter we need to add the original pixel, a size 2, a size 4, and a size 8 filter together. This means that in the worst case a general averaging needs **n+n=2n** additions, where $\mathbf{n}=[^2\log N]$. However, it is possible to use the same strategy as for the Sign Digit Code, which means that we use subtraction instead of addition in some cases. For example in the case where N+1 is a the power of 2, we subtract the original pixel from the N+1 filter, see Figure 4.7. The number of additions in the worst case is now reduced to $\mathbf{n+(n-1)*2/3=(5n-2)/3}$.

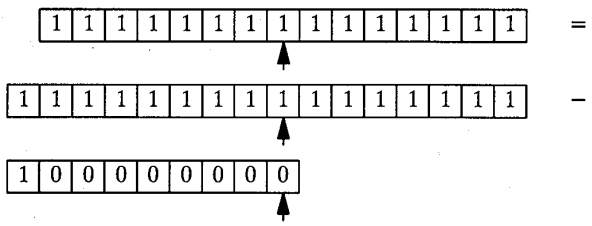


Figure 4.7, Filter size not being a power of 2 using plus and/or minus

When we use even filter sizes, we will have a slight problem. After a convolution the result is shifted half a pixel to the right or to the left depending on where we put the result. However, in most routines this is not a problem because we perform a lot of convolutions, and by placing the first result to the left, the next to the right and so on we minimize this problem. However, we have to ensure that the result is stored in the middle, i.e.half a step away. Thus, all shifts in the first $^2\log N-1$ steps should be shifted in one direction and the last step in the other direction. Figure 4.8 shows how this is done and we can see how the dark pixel in the lowest row, the result pixel, is shifted half a step from the center of the 8*1 kernel at the top.

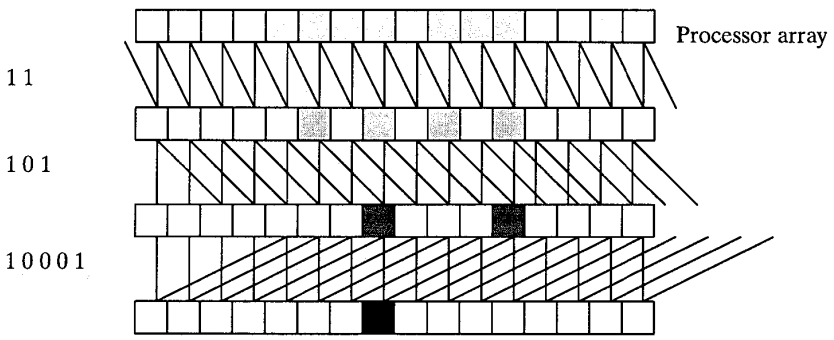


Figure 4.8, Shifting during a 8*1 convolution

With a fixed precision of **b** bits throughout the operation and truncation after each step, an N-points averaging takes $(^2\mathbf{logN})*(\mathbf{3b+2})+\mathbf{N}$ clock cycles. The first term is the number of additions and the second is the total number of shifts. However, the number of bits in the result increases for each addition.To maintain a high accuracy throughout the convolution, we have to increase the number of bits in each step. The first addition takes **3b+2** clock cycles, the next **3(b+1)+2** cycles and so on. The number of extra clock cycles will be $\mathbf{3*(^2logN-1)^2/2}$. Thus, if we avoid all truncation, we will need $\mathbf{(^2logN)*(3b+2)+N+3*(^2logN-1)^2/2}$ cycles altogether.

4.3 Maximum value finding and distribution

In some algorithms, for instance thresholding, an intermediate result in one PE must be distributed to all the other PEs in the array. In the thresholding case we want each PE to have the position number of the PE that has the largest value. Each PE holds two numbers, the position number and a corresponding value. If we use the Global-OR algorithm in section 3.7, to find the maximum value and then distribute the position number that corresponds to

the largest value in the array, we could end up with two or more equally large values which is not acceptable. To break the tie we extend the Max finding operation in section 3.7, to include some other global operations.

First we find the maximum value using Global-OR, which takes **5b+2** clock cycles, in the present case **b=^{2}logN** and **N=128**. To break a potential tie we choose the leftmost PE with a maximum value. The leftmost value is selected using the global operation in section 3.7, which takes **5b+2** cycles. This value is broadcasted all PEs, by loading the status bit into A and a zero into B. Then, bit by bit, we place the value in C, perform a CarryFeedback. Depending on the result of Global-OR we write a 0 or a1 into the shift register, see Figure 4.9, which takes **4b+2** cycles. As it is only one PE with the status bit set and the value from each PE is ANDed with the status bit, the output from Global-OR represents the max-value in the array.

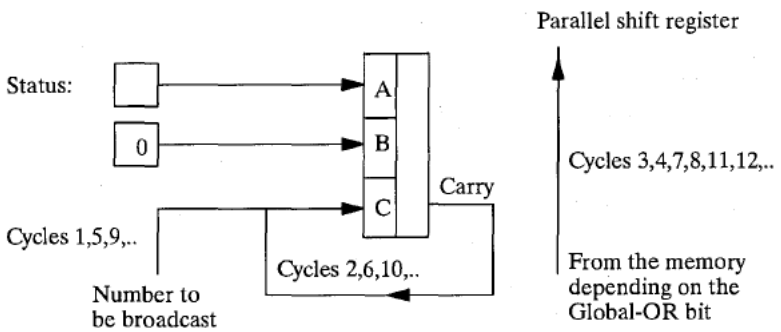


Figure 4.9, Broadcasting from one PE

Thus, the whole procedure will take **14b+6** cycles, see Table 4.2. When **b=^{2}logN=7** this will be **104** cycles.

Procedure	Time
Set all Status = 1	
Find Maximum value	5b+2
Find Leftmost maximum	5b+2
Propagate maximum	4b+2
	14b+6

Table 4.2, Summary of max finding and propagation

4.4 Counting bits in a PE

Counting bits is a common operation in many algorithms. Within a PE, i.e. in the vertical direction of the image, we can proceed as follows. The top vector in Figure 4.10 represents a bit vector in the memory in each PE. This is the vector that we want to compress into a binary count number. We first compress the first three bits into 2 bits, which takes **5** cycles, one for each arrow in Figure 4.10. Having done one pair of such compressions we can compress the first 7 bits into 3 as is described in [8].

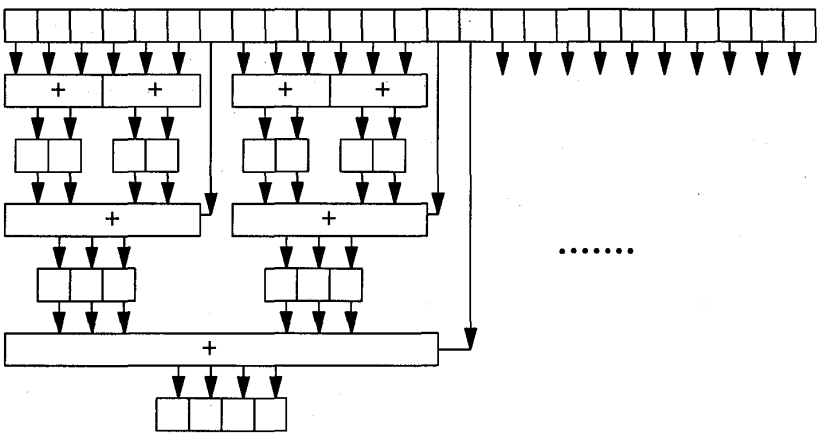


Figure 4.10, Compression of the bit count

The number of clock cycles needed is given by the recursive Equation (4.1) which can be checked against Figure 4.10. It takes C(n) cycles to count/compress n bit:

$$\begin{cases} C(1) & = 0 \\ C(2n+1) & = 2C(n) + 3\log(n+1) + 2 \end{cases} \tag{4.1}$$

To obtain an explicit analytic expression for C(n) we first make the variable substitution in (4.2).

$$n = 2^{p-1} - 1 \tag{4.2}$$

Substituting (4.2) into (4.1) gives us after some manipulations Equation (4.3).

$$\begin{aligned} C(2^p - 1) &= 2C(2^{p-1} - 1) + 3p - 1 = \\ & 2 \cdot (2C(2^{p-2} - 1) + 3(p-1) - 1) + 3p - 1 = \ldots = \\ & \sum_{i=0}^{p-2} 2^i(3(p-i) - 1) = (3p-1)(2^{p-1} - 1) - 3\sum_{i=0}^{p-2} i2^i = \\ & (3p-1)(2^{p-1} - 1) - 3(2^{p-1}(p-3) + 2) = 2^{p+2} - 3p - 5 \end{aligned} \tag{4.3}$$

We rewrite Equation (4.2) into (4.4)

$$p = 1 + \log(n+1) \tag{4.4}$$

Substituting (4.4) into (4.1) gives us (4.5) which we rewrite into (4.6).

$$C(2n+1) = \left[n = 2^{p-1} - 1\right] = 8n - 3\log(n+1) \tag{4.5}$$

$$C(n) = 4n - 3\log(n+1) - 1 \tag{4.6}$$

For larger n, C(n)→4n and when n is small, n<31, n≈3n. Thus, in general, n>32, a bit compression within a PE takes 4n cycles.

4.5 Counting bits on a line

In the horizontal direction, along the array, a bit summation is equal to a convolution with a 128-point kernel `1 1 1..1`, see section 4.2. This would require sparse filter of sizes

2,4,8,16,32, and 64. Depending on the type of result we want, the shift register should be set in different modes, see Figure 4.11. If we want each PE to have the total sum of all bits in the array, the shift register should be set in rotation mode. If we want each PE to have the sum of all bits to the left of the PE, we should disable the rotation mode, shifting in 0s from the left, and store the result after each convolution in the rightmost PE. These two operations are described in Figure 4.11. Figure 4.11(a) shows the original bit vector, (b) shows the result when all PEs obtain the total count, and (c) shows the result when each PE receives the count corresponding to the number of bits to the left of and including its own position.

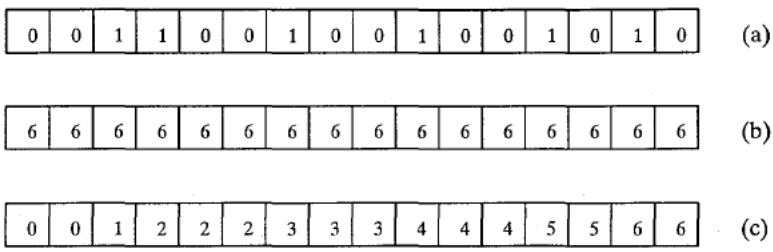


Figure 4.11, Different summation results

The formula for the time consumption in the general case is shown in Equation 4.7. For **N=128** and **b=1** this take **217** cycles.

$$t = \log N(3b + 2) + N + \frac{3}{2}(\log N - 1)^2 = N + \frac{3}{2} + (3b - 1)\log N + \frac{3}{2}(\log N)^2 \quad (4.7)$$

Various conditions could be imposed on the counting procedure. One condition is that during the summation of the bits to the left of each PE, the summation is restarted at various predefined positions along the array. In other words, each PE receives the number of bit to the left since the last restart point, which is illustrated in Figure 4.12. The upper vector in (a) is the original bit vector and the lower one indicates where the summation is restarted.

Figure 4.12(b)–(d) show the temporary results from the input vector (a) and the final result is showed in (e). The idea behind the algorithm is to perform an ordinary horizontal summation and at the same time propagate the restart bits. These bits are propagated and ORed in each PE. The point is now that if a PE receives a pixel tagged with a restart bit the PE will add zero. Thus, we will get the result in Figure 4.12.

This algorithm is described in pseudo code in Table 4.3 and the number of clock cycles it takes is shown in Equation (4.8).

$$t = 2 + \sum_{b=0}^{n-1}\left(2^b + 1 + 2b + 1 + 3b + 4\right) = 2 + \sum_{b=0}^{n-1}\left(2^b + 6 + 5b\right) = 1 + \frac{7}{2}n + \frac{5}{2}n^2 + 2^n \quad (4.8)$$

In our case, when $n={}^2\log N={}^2\log 128=7$, Equation (4.8) will give us **276** cycles.

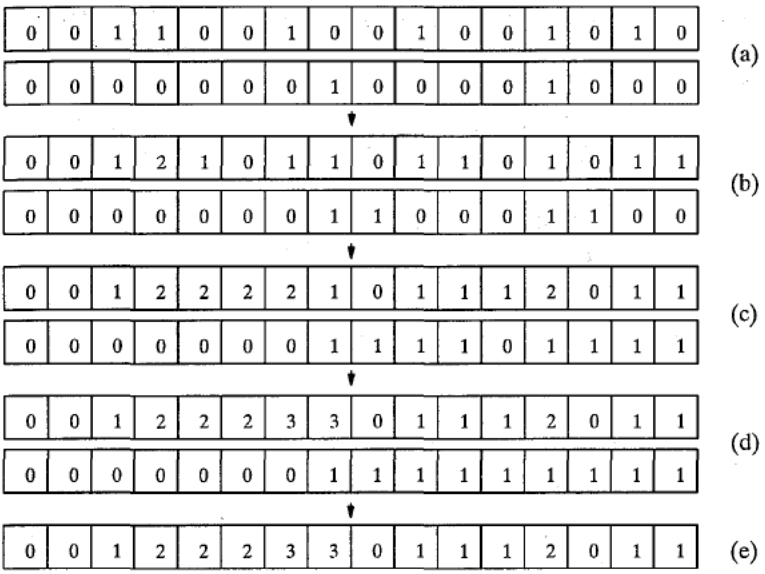


Figure 4.12, Counting objects with restart

```
M(C)->PS(C)
M(0)->PS(0)
for b:=0 to n-1
        Shift_right(2^b steps)
        M(C)->A
        for i:=0 to b
                PS(i)->B
                AB->PS(i)
        rof
        M('0')->C
        for i:=0 to b
                PS(i)->A
                M(i)->B
                Sum->PS(i),M(i)
        rof
        Carry->PS(i),M(i)
        M(C)->A
        PS(C)->B
        A+B->PS(C),M(C)
rof
```

Table 4.3, Pseudo program for bit count with restart

4.6 Labeling objects on a line

This algorithm labels 1D connected components, called objects, on a horizontal line. Thus, an object is defined as a sequence of consecutive 1s. For example, the sequence 010010 contains two objects whilst 011110 contains one object.

To label the object we start by detecting the right edge of all objects. Each PE sets a control bit to 1 if the pixel to the left was an object pixel and its own pixel is a background pixel, otherwise the control bit is set to 0, see Figure 4.13(a) and (b).

By using the routine in 4.5, Figure 4.11(c), we sum the bits to the left of each PE, and obtains labels for each object. The intermediate result in Figure 4.13(c) is masked with the original 4.13(a) to get the final result 4.13(d).

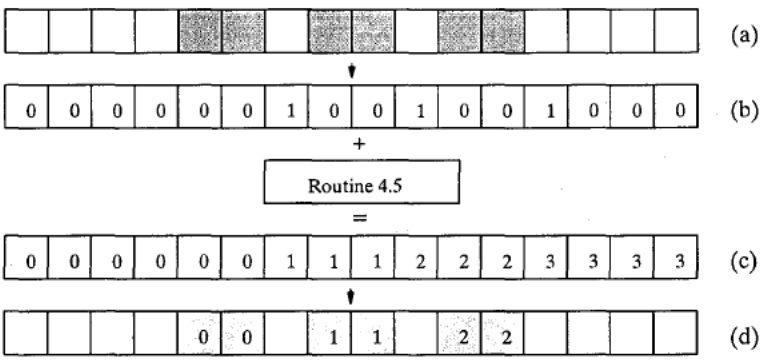


Figure 4.13, Labeling steps

This routine will take **21b+146** cycles which is the same as in 4.5.

The rightmost PE in Figure 4.13(d), now contains the number of objects plus 1. This could be used as a feature when we want to label the next line. In the later case, we just rotate the shift register and shift in this value instead of 0's. Figure 4.14 shows an example where the input is the value of the rightmost pixel's label, in this case a 3. The two object on the line therefore get the labels 3 and 4.

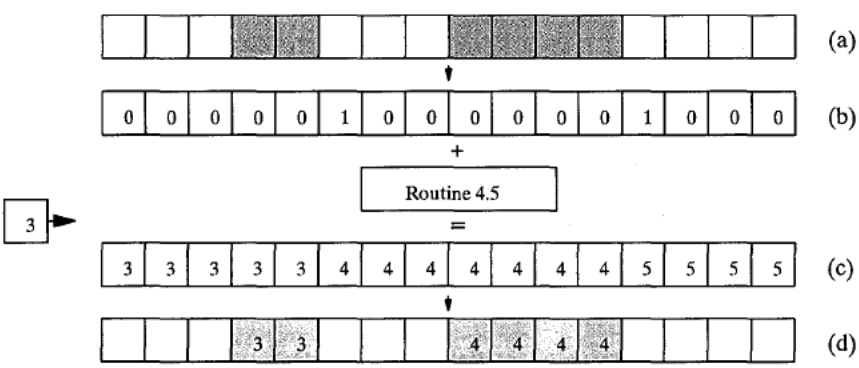


Figure 4.14, Labeling row number two

4.7 Histogram collection

In traditional sequential computers histogram collection is performed as follows, see Figure 4.15. First every entry in the histogram table is set to zero, then for each pixel in the image the table is incremented at the address corresponding to the pixel value.

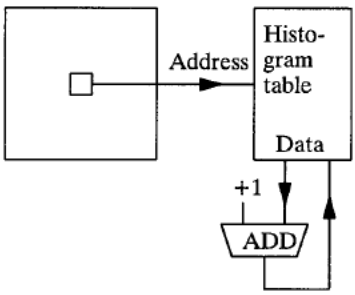


Figure 4.15, Sequential histogram collection

In bit-serial SIMD machines we have to do it in a different way, for instance as suggested in [8]. This method adopted for PASIC is shown in Figure 4.16.

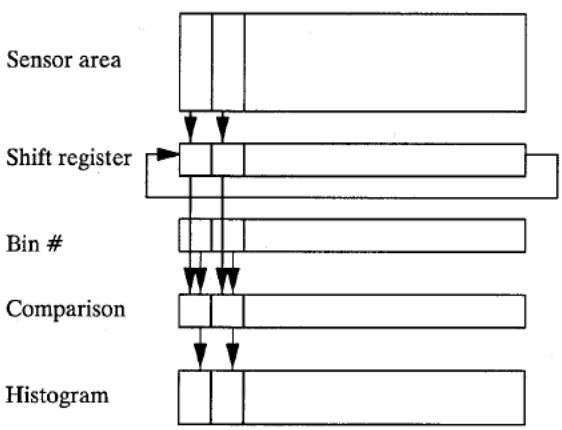


Figure 4.16, Parallel histogram collection

Every PE is assigned a unique bin number between 0 and 127. This number, which is stored in the memory, correspond to the pixel value the instance of which the PE will count. Each line that is read out from the sensor area is stored in the parallel shift register and at the same time compared with the bin number in the PE. The result is 1 if there is a match. We now shift the shift register one step, and make a second comparison, with the next bin number. We repeat this 127 times which results in a 128 bit response vector in each PE. This vector is replaced by the bit count, see section 4.4, and the result is accumulated to the histogram bin in each PE. After 128 lines we have the final histogram. Thus, the histogram collection consists of the following three steps.

```
1. Make comparisons
2. Compute the sum of all 1's in the response vector
3. Accumulate this result to the histogram
```

Each comparison consists of the Boolean expression in Equation (4.9), where $(X_1,X_2,..,X_b)$ and $(Y_1,Y_2,..,Y_b)$ are the two input vectors and C is the result of the comparison.

$$C := \overline{(X_1 \oplus Y_1)+(X_2 \oplus Y_2)+..+(X_b \oplus Y_b)} \quad (4.9)$$

The execution of Equation (4.9) in PASIC is a two-phase procedure described in Table 4.4, and is illustrated in Figure 4.17.

```
For i:= 1 to b
     M(Xi)->A
     M(Yi)->B
     A⊕B->M(Zi)
End
1->B
0->C
M(Z1)->A
For i:=2 to b
     M(Zi)->A, CarryFeedback
End
Carry->M(C)
```

Table 4.4, Pseudo code for two phase comparison

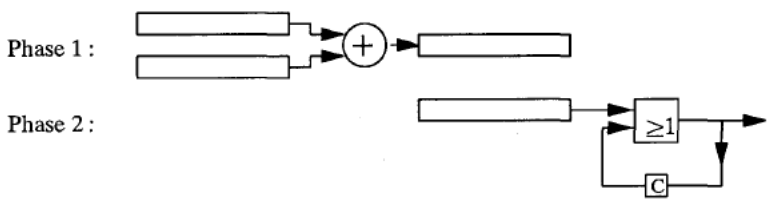


Figure 4.17, An Exclusive-or comparison without special feedback

The algorithm executes in **3b+b+2=4b+3** cycles. The intermediate bit vector Z and the two phases are needed because the PASIC's ALU cannot directly produce the 3-input Boolean function in Equation (4.10), which also must be feed back to the C register.

$$C := C+A\overline{B}+\overline{A}B \quad (4.10)$$

As shown in Figure 4.18, with the function feedback in Equation (4.10), the the task could be accomplished in **2b+2** cycles. This indicate very strongly that the PASIC ALU should be extended with the feedback function in Equation (4.10), as histograming is an extremely common operation in image processing.

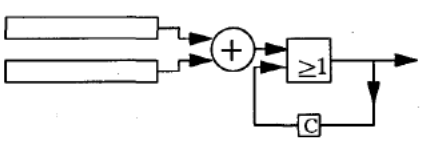


Figure 4.18, An Exclusive-or comparison with feedback

However, we will now show a variation of the previous strategy for histogram collection which makes a comparison in **2b+3** cycles using the existing ALU-functions only. In this version, for each line, we start the histogram collection by subtracting the pixel values p(x) from the bin number h(x) having the line in its original position. Each difference d(x) is now a pointer to the bin where the pixel belongs, see Table 4.5.

Bin number	h(x)	0	1	2	3	4	5	6	7
Incoming pixel	p(x)	2	1	6	6	5	2	7	2
Difference	d(x)	-2	0	-4	-3	-1	3	-1	5
$[d(x)]_{mod\ 8}$	c(x)	6	0	4	5	7	3	7	5

Table 4.5, Creating pointers by subtraction

The algorithm proceeds as follows. The line of differences (pointers) are decremented and shifted one step at a time to the left. When the difference is zero a bit-contribution to the bin is recorded in form of a 1-bit, otherwise a 0-bit is recorded. At each step, the decrementation and the test for zero is performed in PASIC as shown in Figure 4.19.

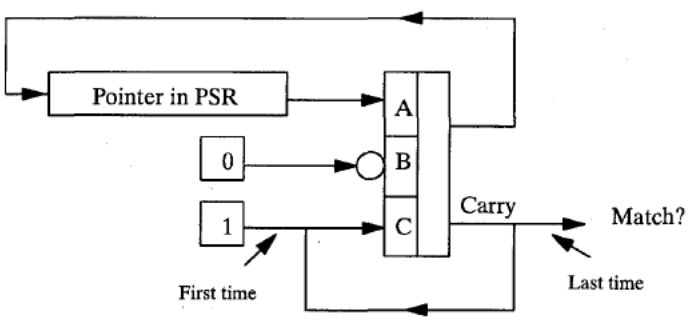


Figure 4.19, Decrementation and test for zero

Since $c(x)=[d(x)]_{mod\ 8}$ values are all positive, the test for zero in this case is equivalent to detect when the 8-bit value of c(x) changes from 00000000 to 11111111. This and only this case gives a remaining Carry = 1 after the complete decrementation. The c(x)-value changes at same time from 0 to $[-1]_{mod\ 8} = 255$. The cycle count for the operation is **2b+3** clock cycles.

The second phase of the histogram collection is the bit vector compression, described in 4.4. For large bit vectors a compression takes **4N** cycles, where **N** is the length of the vector, in this case 128. This value is added to the value that up to now has been collected in the bin. It is an **n** bit addition which takes **3n+2** clock cycles, where **n=14** to cover the worst case, where all pixels in the image have the same value.

Table 4.6 summarizes the different comparison method for histogram collection. In our case **N=128** and **b=7**.

Type of comparison	No cycle per line	Whole image, b=7	≈
Exclusive-or without feedback	4b+3)N+4N+3n+2	$35N^2$	573440
Exclusive-or with feedback	(2b+2)N+4N+3n+2	$21N^2$	344064
Decrementation	(2b+3)N+4N+3n+2	$21N^2$	344064

Table 4.6, Different comparison methods

Thus, a histogram collection the decrementation method takes 344 kcycles per frame which means 58 frames per second in a 20 MHz system.

5 General comments about applications

5.1 Definitions

Figure 5.1 illustrates the dependency between application, algorithm, subroutine, and instruction. For instance, an industrial application of robot vision might consist of the three algorithms, Shading correction, Edge detection, and Segmentation. Each of these three algorithms then has its own set of subroutines and instructions.

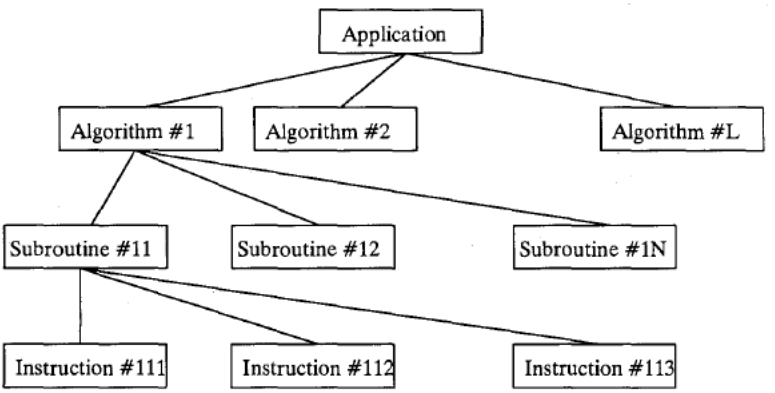


Figure 5.1, Definition of an application

For case of simplicity, we will hereafter omit the application level, as it can be seen as one algorithm with subroutines grouped after each other, see Figure 5.2. This way of looking at the problem makes it easier to implement in a row pipelined form which will be discussed in the next section.

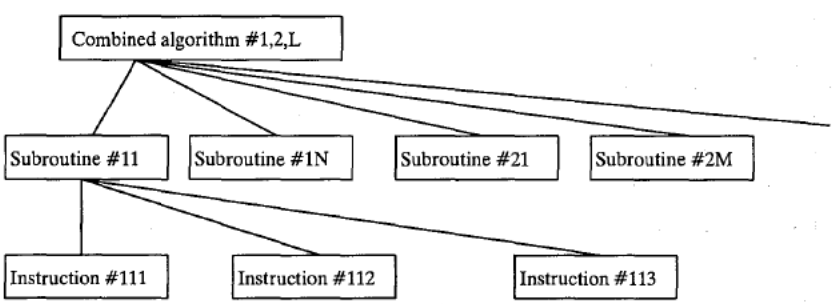


Figure 5.2, Several algorithms combined into one

5.2 Row parallel pipelining

An image processing algorithm can normally be executed in three separate stages: input, computation, and output. However, this way of processing does not apply to PASIC, since we do not have sufficient memory to harbor an entire image. Therefore, the algorithms have to be rearranged in order to have a concurrent input processing and output stream, which

we call row-pipelining. A typical algorithm might consist of a computation that requires data from a number of consecutive lines. For instance, in a 3*3 convolution, each step needs the values of the previous and the next line, see Figure 5.3. This means that we have to have three input pixels at the same time in each PE, one in the A/D-converter and two in the RAM. After the computation the pixel in the A/D register is stored in the memory in the position for the m−2 pixel.

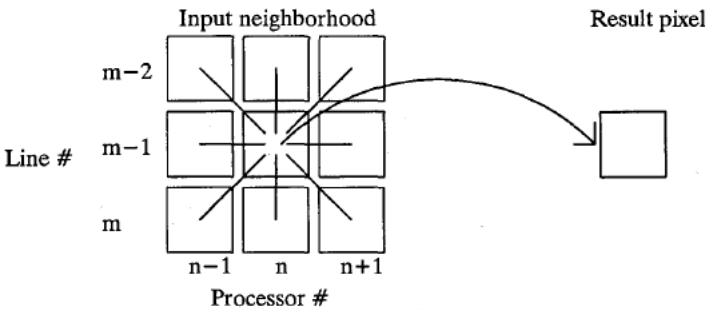


Figure 5.3, Retrieving pixels in a 3*3 neighborhood

Another way to look at this example is shown in Figure 5.4, where we need three line delays. Each line is processed with the following three operations.

(1) Line n+1 is fetched which makes a new 3x3 neighborhood accessible for each PE.

(2) The result at line n is computed.

(3) This result is shifted out and delivered to the output image.

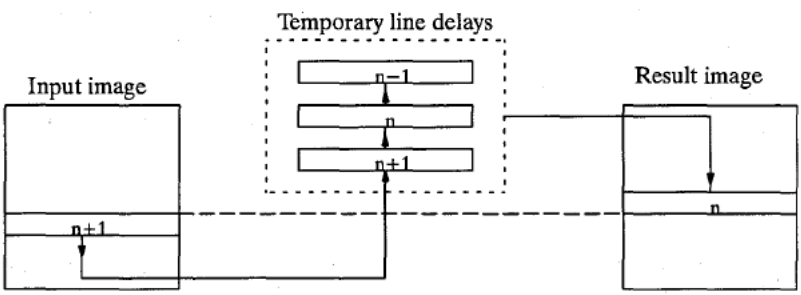


Figure 5.4, Row parallel pipelining

5.3 Combining different algorithms

A typical algorithm consists of many subroutines. To make full use of the PEs and to avoid unnecessary work we have to arrange the flow of computation into on single row-parallel pipeline. For example, assume we have an algorithm that consists of three steps, a 3*3 median, a 5*5 low pass filter, and finally a 3*3 maximum finder operation. Figure 5.5 shows a dependency graph that describes this problem. Each rectangle represents an pixel, an intermediate result pixel, or an output pixel, and the lines between the layers represent data de-

pendent between layers. This picture can be seen as a memory map for each PE in PASIC. The other PEs should be visualized as placed perpendicular to the plane of the Figure 5.5. We only need to store the rectangles that are lightly shaded. Thus, we only need to do three operations to compute a new pixel. These computations corresponds to the pixels (a), (b), and (c) in Figure 5.5.

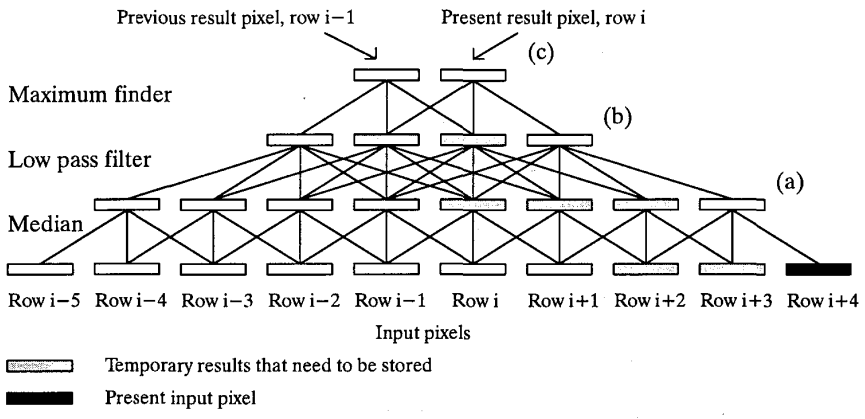


Figure 5.5, Result dependency graph

The most common bottle-neck in a PASIC application is the lack of memory. It very often limits which and how many operations that can be pipelined. We therefore have to be able to compute which operations that can be combined using the available memory. The memory requirement for an operation varies during execution time. The minimum requirement occurs when the operation is non-active, i.e. idle, where only the intermediate results is stored which will be used during the next time the operation is active. These intermediate results corresponds to the shaded rectangles in the two middle layers in Figure 5.5.

The total amount of memory needed in a complete system is the sum of all minimum requirement plus the run-time memory in the application that uses most memory, see Equation (5.1).

$$Memory_{tot} = \sum_i Idle_i + \max_i [Active_i - Idle_i] \quad (5.1)$$

Figure 5.6(a) shows an algorithm that consists of routines with different memory requirement. The total memory needed, according to Equation (5.1), is shown in Figure 5.6(b).

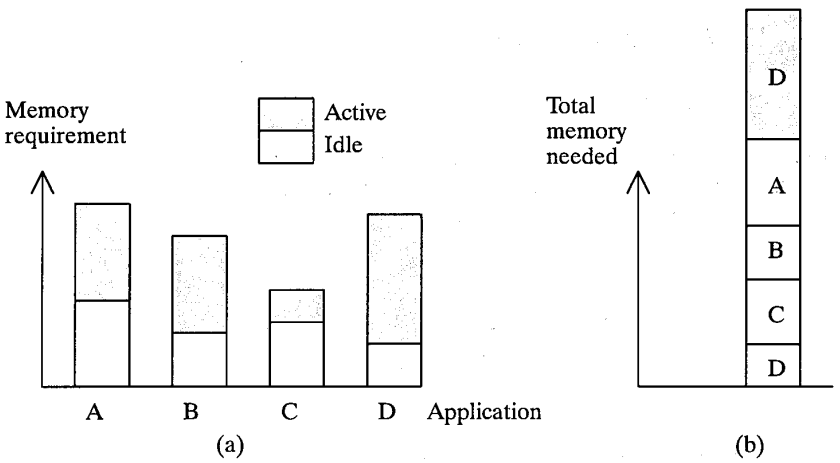


Figure 5.6, Memory requirement

We notice in Figure 5.5, that if we only store the original pixels, corresponding to the rectangles at the bottom level, we will have to store 8 previous pixels in each PE. In the proposed scheme we also have to store 8 pixel values. Thus, we do not gain any memory reduction. However, we do save clock cycles, which of course is very important.

6 Shading correction

6.1 The need for shading correction

The image that we get from the system input, i.e. the photo sensors, is far from ideal. For various reasons and causes a shading function s_{xy}, unique for each (x,y) position, has operated on the original image I_o giving the detected image I_d, see Equation (6.1).

$$I_d(x,y) = s_{xy}(I_o(x,y)) \tag{6.1}$$

This unwanted effect is called shading. Examples of shading causes are

(1) non-uniform illumination of the scene

(2) optical effects ($\cos^4$-law)

(3) non-uniform gain in the photo sensors

(4) non-linearity in the A/D-converter system

As long as these effects do not totally corrupt the signal and, which is very important, do not vary in time, they are quite possible to correct for. With the following notation

r(x,y) = true reflectivity at the point (x,y)
p(x,y) = actual detected value at (x,y)
q(x,y) = corrected pixel value at (x,y)

it has been showed that we in most cases can do a satisfying correction using Equation (6.2).

$$q(x,y) = C_1(x,y)[p(x,y) + C_2(x,y)] \approx r(x,y) \tag{6.2}$$

If we omit the position (x,y) we will get (6.3).

$$q = C_1(p + C_2) \tag{6.3}$$

The two constants C_1 and C_2 have to be computed in a special calibration procedure. Normally, the shading effects vary slowly over the image, at least for the examples (1) and (2) presented above. The case (4) is constant for all pixels in the same column while (3) does not have any of these nice properties. Normally, the effects is not dominating. Presuming that all effects in general are varying slowly, we are able to reduce the number of constants needed by letting neighboring pixels use the same constants. However, without precautions this may cause artefacts in the form of a false edge-pattern. For example, if we use an 8 by 8 neighborhood with the same C values, we will see the contour of a square grid, 8 by 8, superimposed over the picture. Most of these artefacts can be avoided by interpolations, as will be explained in section 6.3.

6.2 Calibration

To estimate the two constants C_1 and C_2, we have to use two reference images, $r_A(x,y)$ and $r_B(x,y)$ for which we know or rather have decided a priori the exact values. For the calibration procedure the two constants $C_1(x,y)$ and $C_2(x,y)$ are all set to 1 and 0 respectively. When

the two reference image are applied we get two uncorrected results, see Equation (6.4) and (6.5).

$$q_A = p_A \tag{6.4}$$

$$q_B = p_B \tag{6.5}$$

With the proper correction coefficients we should rather have the outputs in Equations (6.6) and (6.7).

$$q_A = r_A = C_1(p_A + C_2) \tag{6.6}$$

$$q_B = r_B = C_1(p_B + C_2) \tag{6.7}$$

The C_1 and C_2 can therefore be computed with (6.8) and (6.9).

$$C_1 = \frac{r_B - r_A}{p_B - p_A} \tag{6.8}$$

$$C_2 = \frac{p_B r_A - p_A r_B}{r_B - r_A} \tag{6.9}$$

To reduce the memory requirement we must minimize the number of bits for the constants C_1 and C_2, which means that we should find the lowest number of bits that produces a 'good' picture. Figure 6.1 shows the three worst linear functions s_{xy}, (a), (b), and (c), which are still possible to correct for. Curve (d) is a non-linear function where $p_{xy}=0$ for $r_{xy}<255$.

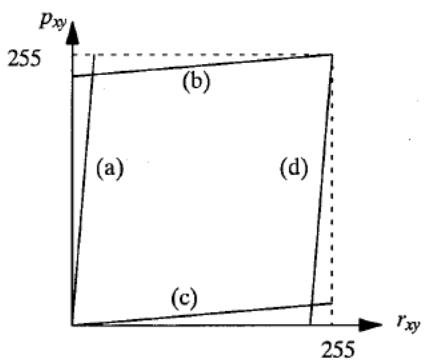


Figure 6.1, Four extreme curves that can be corrected

The four curves in Figure 6.1 corresponds to the following constant pairs.

(a) $C_1 = 1/256$ $C_2 = 0$

(b) $C_1 = 256$ $C_2 = -255$

(c) $C_1 = 256$ $C_2 = 0$

(d) $C_1 = 1/256$ $C_2 = +255$

Thus, the constant C_2 has a range from -255 to $+255$ and C_1 a range from 1/256 to 256. We therefore assign 9 bits to each C_2 value, and 16 bits to each C_1 value, 8 bits for the integer part and 8 bits for the fractional part, see Figure 6.2.

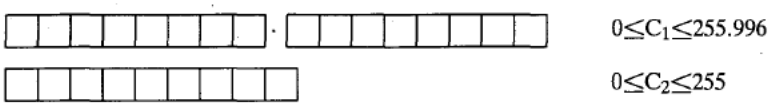


Figure 6.2, Memory size of C_1 and C_2

The spatial resolution of the correction image $C_1(x,y)$ and $C_2(x,y)$ was discussed in the previous section. Ideally, every pixel should have its own C_1 and C_2 values. In the horizontal direction, this is not a problem, since there is one PE for each column. However, in the vertical direction we are limited by the size of the memory. The size that the memory requires is a function of the number of pixels sharing the same pair of C:s, see Equation (6.10).

N_P = *Number of pixels in a column*
N_C = *Number of pixels with the same constants*
N_M = *Number of memory cells required*
b_1 = *Number of bits in* C_1
b_2 = *Number of bits in* C_2

$$N_M = (b_1 + b_2) * \frac{N_P}{N_C} \tag{6.10}$$

This means that in a 128-bit memory we are only able to store 4 C_1 and C_2 constants, which then occupies **4*(9+16)=100** bits. Thus, all pixels in a vertical neighborhood of 128/4=32, will have the same C values, see Figure 6.3.

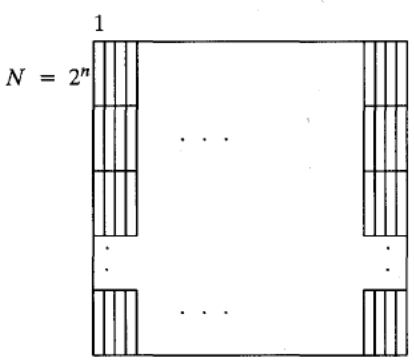


Figure 6.3, Regions with the same C_1 and C_2 values

Thus, we have a rather course resolution for the $C_1(x,y)$ and $C_2(x,y)$ in the vertical direction.If we assume that the artifacts are varying slowly over the image we can still get good results using interpolation as will be shown in section 6.3.

We will here omit all detailed descriptions and cycle counts for the calibration phase as it could very well be computed off the chip and therefore presents no time-critical problem.

6.3 Correction

The correction phase consists of two steps. The first step is interpolation to compute the values of C_1 and C_2, the second is to do the correction by using Equation (6.3). The latter

step consists of one 8-bit add and one 8/16-bit multiplication, which will take **3*8+2+3*8*16+2*8−2*16−2=392** cycles per row.

In the interpolation step we want to compute a C value for a pixel in row i. This pixel is located between two defined values in row A and A+N, called C_A and C_{A+N}, where $N=2^n$ and n is the number of constants in the column. Figure 6.4 shows section through one column of C values.

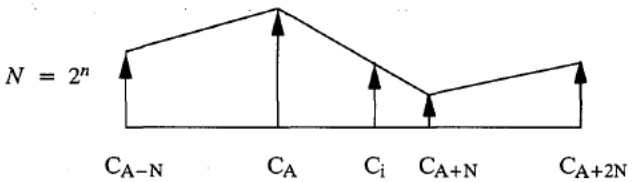


Figure 6.4, Expected value of C between known values

To compute the Ci value we use Equation (6.11), where i=A+k, and the formula for the C value in pixel A+k+1 is in Equation (6.12).

$$C_{A+k} = C_A + k * \frac{C_B - C_A}{N} \tag{6.11}$$

$$C_{A+k+1} = C_A + (k+1) * \frac{C_B - C_A}{N} = C_{A+k} + \frac{C_B - C_A}{N} \tag{6.12}$$

If we multiply (6.12) with $N=2^n$ we get (6.13). This means that the next C value is obtained by adding C_B and subtracting C_A.

$$C_{A+k+1} * 2^n = C_{A+k} * 2^n + C_B - C_A \tag{6.13}$$

The interpolation steps are described in Table 6.1, where the C_i are the corrections values and the C^+ are corresponding temporary values shifted n steps.

$$\begin{aligned} C_0 &= C_A \\ C_0^+ &= C_A \cdot 2^n \\ C_1^+ &= C_0^+ + C_B - C_A \\ C_1 &= C_1^+ \cdot 2^{-n} \\ C_2^+ &= C_1^+ + C_B - C_A \\ C_2 &= C_2^+ \cdot 2^{-n} \\ &\dots \end{aligned}$$

Table 6.1, Computation steps for interpolation

The interpolation requires one addition and one subtraction, whereas shifting takes no time since PASIC is a bit-serial machine. Further reductions can be made since C_B-C_A only need to be computed once for each vertical neighborhood. The interpolation consists of one addition and one Nth of a subtraction, which means that to perform one 8-bit and one 16-bit interpolation takes **3*16+2+(3*16+2)/N+3*8+2+(3*8+2)/N=76+76/N** clock cycles.

Thus, to compute the shoding correction for a system where N=4 takes **95+392=487** cycles per row. A full image needs **62336** cycles and takes 3.12 ms to compute on a 20 MHz system.

7 Histogram based thresholding

7.1 On thresholding and histograms

In order to produce a binary picture from a gray level picture we have to perform a thresholding operation. In the most general case each pixel value p(x,y) is compared with a thresholding function T(x,y,t), which varies in time and space, and generates a binary image b(x,y) as in Equation (7.1).

$$b(x,y) = \begin{cases} 1 & \text{If } p(x,y) \geq T(x,y,t) \\ 0 & \text{Otherwise} \end{cases} \tag{7.1}$$

Such a thresholding is of course both time and memory consuming. We therefore often settle with one common thresholding function for all (x,y), see Equation (7.2).

$$b(x,y) = \begin{cases} 1 & p(x,y) \geq T(t) \\ 0 & \text{Otherwise} \end{cases} \tag{7.2}$$

In a well defined environment, where the objects and the background are well separated in intensity the thresholding value could easily be set to a single fixed value, T, constant in time as in Equation (7.3).

$$b(x,y) = \begin{cases} 1 & p(x,y) \geq T \\ 0 & \text{Otherwise} \end{cases} \tag{7.3}$$

If we use the thresholding (7.3), we must be certain that the light conditions are kept constant. In some cases this is sufficient. However, in other cases we have to be more careful and produce the threshold repeatedly in run-time, as in (7.2). This means that we have to get a new thresholding value T for every frame, depending on the data in this frame. One traditional way to do that is to use a histogram.

A histogram is a vector with N elements, where N represents the number of gray scale values. Each element contains the number of pixels in the picture that has that specific pixel value that correspond to the element position. In section 4.7 the histogram collection was described.

Figure 7.1(a) shows a typical histogram. Ideally, the histogram contains two different gaussian distributions which corresponds to the background and the object, see Figure 7.1(b).

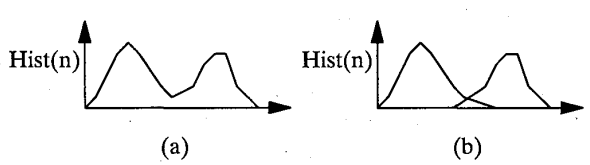


Figure 7.1, Histogram containing two different distributions

The object in thresholding is to find the optimal thresholding point T that best separates the two distribution from each other, see Figure 7.2.

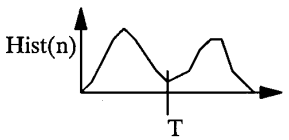


Figure 7.2, Optimal thresholding

There are a number of rather sophisticated algorithms to find an optimal T. One fairly straight forward algorithm is called Mid-point method, see [6]. It starts by guessing an initial value of T which we call T_0. A good estimate of T is the center of mass, called μ, of the histogram, see Equation 7.4. Hist(n) will hereafter be called p(n).

$$\mu = \frac{1}{M}\sum_{n=0}^{127} n * p(n) \quad \text{where} \quad M = \sum_{n=0}^{127} p(n) \tag{7.4}$$

This T_0 divides the histogram into two parts and for each of those parts we compute the center of mass, called μ_0 and $\mu 1$, see Equations (7.5) and (7.6).

$$\mu_0(T_0) = \mu[p(n) | n < T_0] \tag{7.5}$$

$$\mu_1(T_0) = \mu[p(n) | n \geq T_0] \tag{7.6}$$

The new thresholding value is achieved by taking the mean value of μ_0 and $\mu 1$, i.e. the point in the middle between μ_0 and $\mu 1$, see Equation (7.7).

$$T_1 = \frac{1}{2}[\mu_1(T_0) - \mu_0(T_0)] \tag{7.7}$$

We now compute new values of μ_0 and $\mu 1$, which will give us a new T value, and so on. This will iterate until we reach the termination condition in Equation (7.8).

$$|T_n - T_{n-1}| < \varepsilon \tag{7.8}$$

In most cases this method converges. However, in same cases it diverges or converges to a wrong value.

Another approach is to use the zero-crossing of the first derivative. In a well-behaved bimodal histogram we can compute the first derivative of the histogram, Hist'(p), which tells us that the point T is located where the sign of the derivative has a positive zero crossing, i.e. where the sign changes from minus to plus, see Figure 7.3(a) and (b).

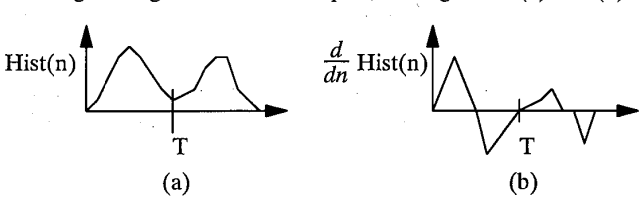


Figure 7.3, Zero-crossing of the first derivatives

The problem with this method is that a normal histogram is not ideal, which means that instead of two smooth well-defined distributions as in Figure 7.4(a), we have a noisy discrete histogram, as in Figure 7.4(b). A noisy histogram will cause a lot of false zero-crossing.

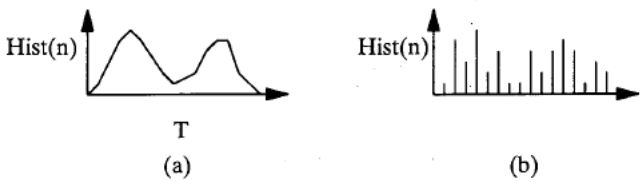


Figure 7.4, An ideal and a normal histogram

We will now describe an iterative algorithm which iteratively smooths the histogram until only the true zero-crossing is left. The input to the algorithm is a histogram, see Figure 7.5.

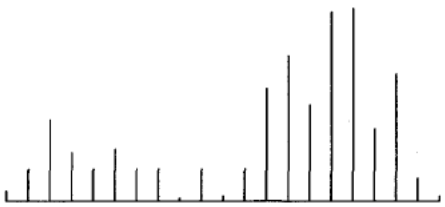


Figure 7.5, Input histogram

We start by computing the first derivative of the histogram. This is approximated with a difference, which is a $-1+1$ kernel, see Figure 7.6.

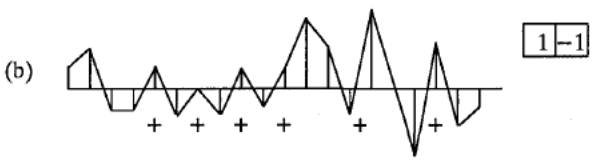


Figure 7.6, Difference kernel applied to input histogram

Then each PE sets a flag if it has a positive value and the neighbor to the left has a negative value, which means that it occupies a positive zero-crossing. This is indicated with a + in Figure 7.6.

The number of flags in the array is counted with the use of Global-OR. If the flag number is zero, there is no possible threshold. If the flag number is one, we have found the threshold, and if the flag number is greater than one we will have to perform a smoothing on the differentiated histogram.

We will continue with the flag counting and smoothing until we find a threshold or until we reach zero flags. This is illustrated in Figure 7.7(a)–(c), where smoothing is performed three times.

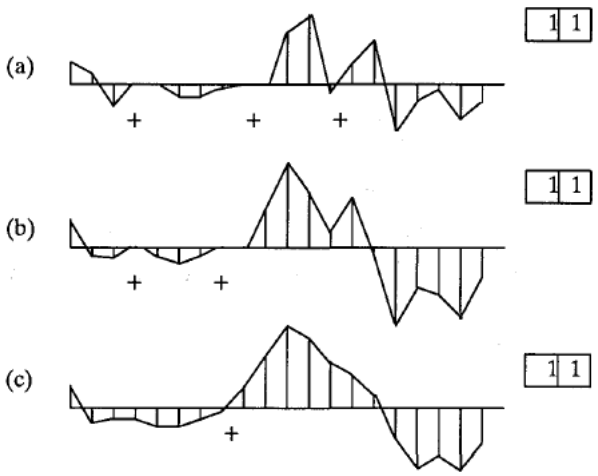


Figure 7.7, Different stage of smoothed histogram

Table 7.1 shows a pseudo program for the algorithm.

```
Compute the difference between two neighboring bins
Loop
        Count the number of zero-crossings
        Case
                Flag_number=0 : No threshold; Exit loop
                Flag_number=1 : Distribute threshold; Exit loop
                Flag_number>1 : Smooth the histogram; Go to Loop
```

Table 7.1, Pseudo code for zero-crossing detection

7.2 Implementation and performance on PASIC

We will begin with the mid-point method. First we compute the center of mass, called μ, using Equation (7.4). The computation of M is very simple since we use sparse filters, described in section 4.2, which will take $(^2\log N)*(3b+2)+N$ cycles. To compute μ we have to multiply each histogram value with its bin number, which takes approximately $5b_0b_1$ cycles, where $b_0=7$ corresponds the bin number and $b_1=14$ to the number of pixels in the bin. The summation of each bin contribution is again done with sparse filter.

Finally, to compute T_0 we have to perform a division. This is preferably done outside PASIC. However, it can be done inside, if every processor does the same division. In that case it will take approximately $8b^2$ cycles, where **b** in this case is 14, which means **1568** cycles.

Assuming that we are able to drop bits to preserve the length of the word, which is the normal case, we will get Table 7.2.

Operation	Time
Compute M	310
Compute n*p(n)	490
Compute sum	310
Division	1568
	2678

Table 7.2, Cycles needed to compute μ

To estimate the two mean values μ_0 and μ_1 we use the two summations (7.9) and (7.10).

$$\mu_0(T_0) = \frac{1}{\sum_{n=0}^{T_0} p(n)} \sum_{n=0}^{T_0} n * p(n) \tag{7.9}$$

$$\mu_1(T_0) = \frac{1}{\sum_{i=T_0+1}^{127} p(n)} \sum_{n=T_0+1}^{127} n * p(n) \tag{7.10}$$

During the first summation every PE with a bin number greater than T_0, contributes with zero and during the other summation the other half of the processor array is idle. Thus, this takes twice the time to compute compared to the computation of μ. However, we do not have to do the multiplication with the bin number since the result was obtained in the first μ computation. Thus, this will take **1240** cycles. The divisions is performed in **1568*2=3136** cycles.

The new T value is computed as a subtraction of the two sums followed by a shift to the right, see Equation (7.11) and the test is on one subtraction, see Equation (7.12). This takes **80** cycles altogether.

$$T_1 = \frac{1}{2}[\mu_1(T_0) - \mu_0(T_0)] \tag{7.11}$$

$$|T_n - T_{n-1}| < \varepsilon \tag{7.12}$$

Table 7.3 summarize the computation.

Operation	Time
Compute μ	2678
Compute μ_0 and μ_1	4476
Subtraction and test	80
	7234

Table 7.3, Total number of cycles needed for the Mid-point method

Thus the Mid-point method takes **7234** cycles per iteration. In the normal case we need between 3 and 10 iteration, providing that it converge. This means between **25000** and **75000** cycles.

The first step in the algorithm for the detection of zero-crossing of the first derivative, is to compute the difference between two neighbors. This takes **3b+2** cycles, where **b=14**. The

next step is to see if there is a zero-crossing within the PE. We define the zero-crossing range for the PE to be between the left PE and itself, see Figure 7.8.

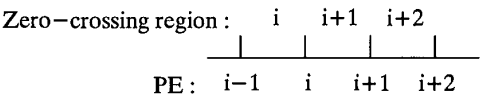


Figure 7.8, Zero-crossing range for each PE

Each PE now checks if it has a positive value and the neighbor to the left has a negative value, which if it is true means that we have a zero-crossing within the pixel. In the test for a negative value of the neighbor we just use the sign bit. To see if we have a positive value in the PE we use Equation (7.13), where b_7 is MSB and b_0 is LSB. It gives us a 1 only if we have a positive number greater than 0.

$$Pos? = \overline{b_7} \cdot (b_6 + .. + b_1 + b_0) \tag{7.13}$$

Altogether, the time to produce the flag for the zero-crossing is **b+2+2=b+4**.

To see how many flags there are in the array, we load the flag bit into the shift register and at the same time into the C register. We now look at Global-OR to see if there is any flag set in the array. If not we quit the routine. If we have at least one flag, we shift one step to the right, and perform an OR between the shift bit and the original bit in the processor. We then perform a logic-And between the two bits and place the result in the C register and perform a Global-OR check. This is shown in pseudo code in Table 7.4. If Global-OR is 1, we have at least two candidates and the test is stopped. Otherwise we repeat the procedure now shifting 2 steps, then 4, then 8, and so on. If we do not get a Global-OR response until we have done a full 128 shift, the test will take **N+7*^{2}logN+3** cycles. In our case when **N=128** this takes **180** cycles.

```
M(Flag)->PS(Flag),C
Global-OR-check
IF Global-OR = 0 Then Quit
FOR b:=0 TO n-1
         Shift_right(2^b steps)
         M(Flag)->iA
         PS(Flag)->iB
         AB->M(Flag)                  ! M(Flag)  = A+B
         A+B->PS(Flag)                ! PS(Flag) = AB
         PS(Flag)->C
         Global-OR-check
ROF
```

Table 7.4, Pseudo code for Two-or-more check with Global-OR

If we during the flag-test phase have a 1-response from the Global-OR, we will perform an averaging, and then go back to flag-set and flag-test again. This averaging is a convolution with a filter that contains 1s, see section 4.2. If we keep the size of the **b** value it will take $\mathbf{Z_1 = (^2logN)*(3b+2)+N}$ clock cycles, and if we let **b** increase it will take $\mathbf{Z_2 = (^2logN)*(3b+2)+N+3*(^2logN-1)^2/2}$ cycles.

If the test shows that we only have one flag we propagate the corresponding bin number to all PE, which is done with Global-OR. Table 7.5 shows pseudo code for the distribution, which takes **4b+2** cycles, where **b=7**.

```
0->A
M(Flag)->B
FOR b:=0 TO b-1
        M(b_i)->C
        CarryFeedback
        Global-OR-test
        CASE Global-OR of
                1 : 1 -> PS(b_i)
                0 : 0 -> PS(b_i)
ROF
```

Table 7.5, Pseudo code for distribution

The cycle count for the whole algorithm are listed in Table 7.6.

Operation	Time
Compute difference	44
Detect zero-crossing	$18N_s$
Count flags	$180N_s$
Smooth the histogram	$Z_x(N_s-1)$
Distribute bin number	30
	$74+188N_s+Z_x(N_s-1)$

Table 7.6, Total number of cycles needed for the Zero-crossing method

7.3 Results from experiments

To estimate the value of N_s and the size and the precision needed in the smoothing, i.e. N and Z_x, we have done some experiments. To test these parameters we used 10 different 128 x128 images, where the object was to find a true thresholding value. A true thresholding point was define as an interval $T_L \leq T \leq T_H$ where all point in this interval according to the human eye, could be considered as correct thresholding values, see Figure 7.9.

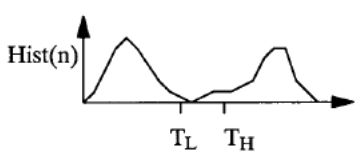


Figure 7.9, Histogram with a thresholding interval

The program tried to threshold all the images with every possible parameter set and registered when the algorithm succeeded in finding a true thresholding point. The program also registered how many time, at the average, we had to loop through the algorithm, i.e. how many passes of smoothing we had to do to produce a correct value. The result is shown in Table 7.7, where the following notation is used.

```
A       Size of the average filter
b+      Number of extra bits used, b=14
S       Number of successfully thresholds
A#      Average number of smoothing for the successful thresholds
```

A	b+=0		b+=1		b+=2		b+=4		b+=8		b+=16	
	H	A#	H	A#	H	A#	H	A#	H	A#	H	A#
2	1	19.00	1	19.00	1	22.00	0	-	4	56.75	10	99.10
4	1	5.00	1	7.00	1	7.00	1	18.00	9	19.78	10	20.20
8	1	4.00	1	4.00	1	5.00	6	5.00	10	6.20	10	6.30
16	0	-	0	-	1	3.00	8	3.12	9	3.11	9	3.11
32	0	-	0	-	2	3.00	6	2.67	7	2.71	7	2.71

Table 7.7, Test result from thresholding

Table 7.7 shows that unless we use 4 extra bits, i.e. b+=4, we will not get any reasonable results. We can also see that with the smallest kernel size, A=2, we can get very good results providing that we use a large number of extra bits. However, A=2 require a very large number of iteration to converge, which takes a lot of time. According to Table 7.7 should the optimal value be A=8 and b+=8, which will result in a N_s value of 6.

Thus, a typical Zero-crossing algorithm takes **74+188*6+285*5=2627** cycles. However, the time to collect a histogram is **344064** cycles which means that the histogram collection is the bottleneck and that efforts should be made to reduce that figure. Comparatively, the thresholding calculation is done very quickly.

We have assumed above in the thresholding analysis that we have a 16-bit parallel shift register. The present version of PASIC do not have that, which means that we will have to increase the cycle count somewhat, at the worst with a factor of 2. In section 10.6 an extension is described which includes a 16 bits parallel shift register and some other features which will speed up this algorithm.

8 Edge detection

8.1 On edge detection algorithms

In the literature on image processing, edge detection is probably covered more intensely than any other subject. The most sophisticated algorithms are often locally data dependent and are virtually impossible to implement in PASIC. Many of those algorithms which may be possible to execute in PASIC can be described by the three-step procedure of Figure 8.1.

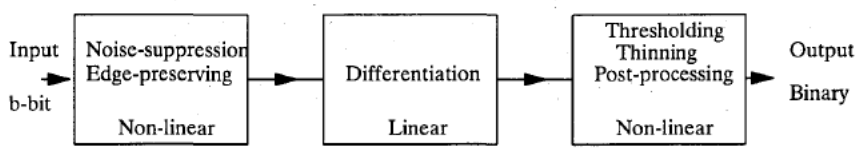


Figure 8.1, Three-phase edge detection

A typical first operator in Figure 8.1 is median filtering. It is especially effective for salt- and pepper noise. As we will see in the next section, median filters of limited size can be implemented efficiently in PASIC. A more sophisticated operator is the Nagao-Ye filter which data-dependently selects a certain subregion of the neighborhood for averaging, see [19]. This operator produces excellent noise suppression and edge preserving result. However, it is almost impossible to implement this operator in PASIC due to the memory requirement and the awkwardness of doing data-dependent processing and addressing in PASIC.

The second step in Figure 8.1 is gradient detection which estimates the gradient in the x- and y-direction. This is mainly a linear type of filtering, implemented as two convolutions. The most common kernels are the well known Sobel pair, described in section 4.1. However, it is worth noticing that these operators can be designed so that they give more or less attention to higher frequencies and that high computation efficiency can be obtained even from larger kernels, see [7]. Figure 8.2 shows an example where we can obtain results from two 5x5 kernels after only 10 additions and subtractions.

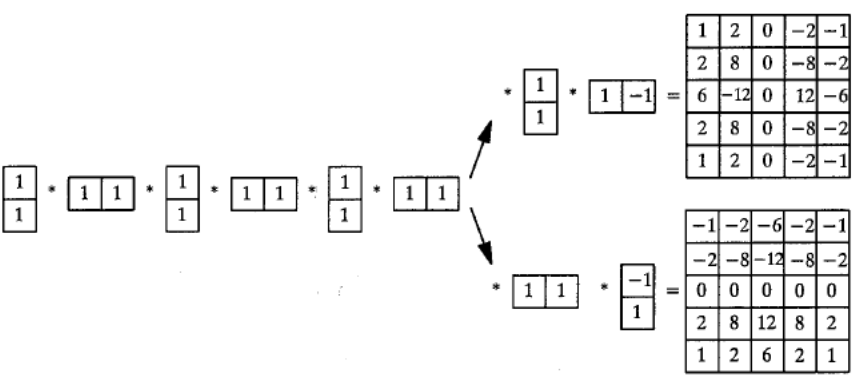


Figure 8.2, 5 by 5 separated Sobel pair

The last step of Figure 8.1 contains thresholding, thinning, and other post-processing. A fix data-independent threshold can sometimes be sufficient providing that the environment of

the system is known. However, in some cases we have to have a more adaptive threshold, for example generated from a histogram as described in chapter 7. To get an edge image where the edge lines are only 1 pixel wide lines we use thinning. Finally, the post-processing tries to enhance the edge image for instance by joining broken edge lines.

In the literature we can find a lot of more or less sophisticated edge detection algorithms. However since we are going to implement these algorithms into PASIC we have a number of constraints. For example,

(a) We do not have floating point capability, at least not anything that runs in normal speed.

(b) We do not have enough storage to harbor an entire image.

(c) We can only use row pipelined algorithms.

This implies that we should use algorithms that use integers and only small regions of the image per pixel. The integer should use as low **b** value as possible and the locality of the pixel makes it possible to implement the algorithms in a row-pipelined fashion, see chapter 5. This constraints will of course limit the possibility to produce a correct edge image. However, since PASIC itself often is used as a preprocessor we will therefore trade speed to precision. The goal will therefore be to have an almost correct edge image in a reasonably amount of time.

8.2 Median - Sobel - Thinning

Figure 8.3 shows a specific example of an edge detection procedure described in [5].

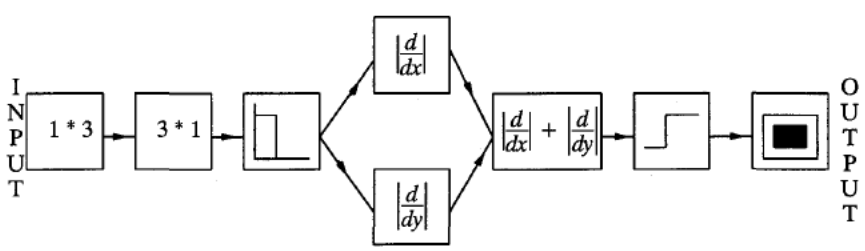


Figure 8.3, Median-Sobel-Thinning algorithm

The first step is a median filtering, which is separated into two steps, one vertical and one horizontal. The vertical filtering is done within each processor and the horizontal filtering is done with each PE's nearest neighbor. Each separated filtering consists of three max/min-tests with multiplexing shown in Figure 8.4(a), where each vertical line stands for a max and min test followed by a multiplexing, see Figure 8.4(b).

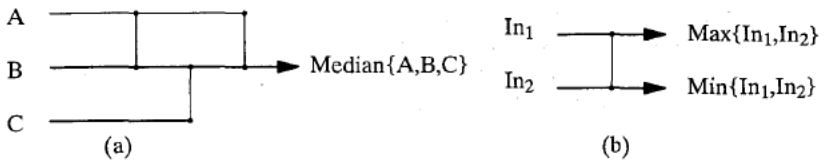


Figure 8.4, 3-input median filter

The operation in Figure 8.5 subtracts In_1 from In_2 without saving the result vector, only propagating the Carry-bit. After the input of the MSBs in In_1 and In_2, the Carry output is 1 if $In_2 > In_1$. It is then fed back to the C register and can be used to perform a one- or a two-way multiplexing using the standard functions as the control bit now is in the C register.

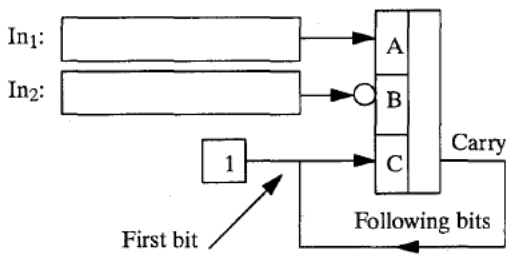


Figure 8.5, Comparison

The time consumed for the different comparison steps are shown in Table 8.1.

Operations			Time
Max-and-Min-test	2b+1+4b	=	6b+1
Max-test	2b+1+3b	=	5b+1
Min-test	2b+1+3b	=	5b+1
			16b+3

Table 8.1, Cycle count for one 3-input median filtering

The reason for doing a separated median filtering rather than a full 3x3 median filtering speed. Figure 8.6 shows that it takes 19 comparisons to do a full 9-input median filtering. However, it is possible to reduce that number to 13 as the three boxes with dotted lines are identical and can be done in parallel in three neighboring PEs.

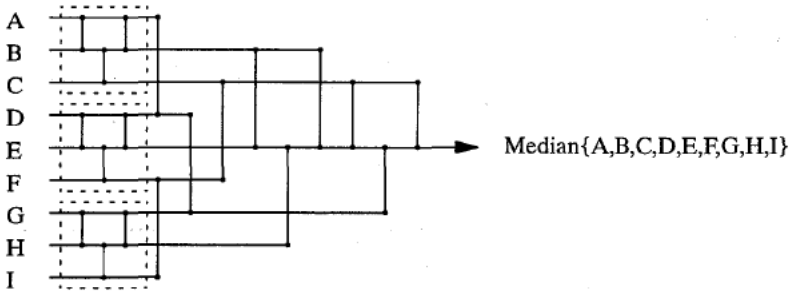


Figure 8.6, 9-input median filter

Table 8.2 shows the cycle count for the separated- and the true median filter. We notice that the shift of data does not add any cycles as it can be done in parallel with other actions.

Operation	# 2-way mux	# 1-way mux	# cycles
True median	5	8	70b+13
Separated median	2	4	32b+6

Table 8.2, Difference in time between true and separated median

With 8-bit precision a true median filtering takes **573** cycles and a separate **262** cycles. Since the results from the two filters are very much the same, the true median filtering is hardly worth the effort.

The low-pass filtering and the gradient computation is combined into one step using the Sobel operator described in section 4.1. This takes **237** cycles and is shown in Figure 8.7.

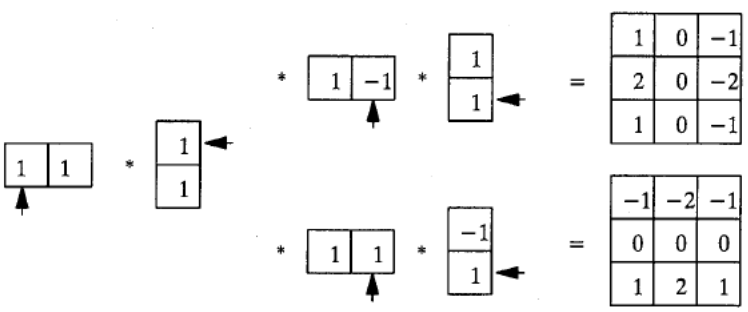


Figure 8.7, Separated Sobel pair

The formula to compute the absolute value of a 2-complement number X is shown in Equation (8.1).

$$|X| = (X \oplus x_{b-1}) + x_{b-1} \tag{8.1}$$

This is done in two steps in PASIC. First, we perform an exclusive-OR between each bit and the MSB, which takes **2b+1** cycles.Then we add the MSB to that result value, which takes **2b+2** cycles. The data flow in the two steps are illustrated in Figure 8.8. We have to compute the absolute value for the gradient in both X- and Y-direction, i.e. two times which means **8b+6** cycles altogether.

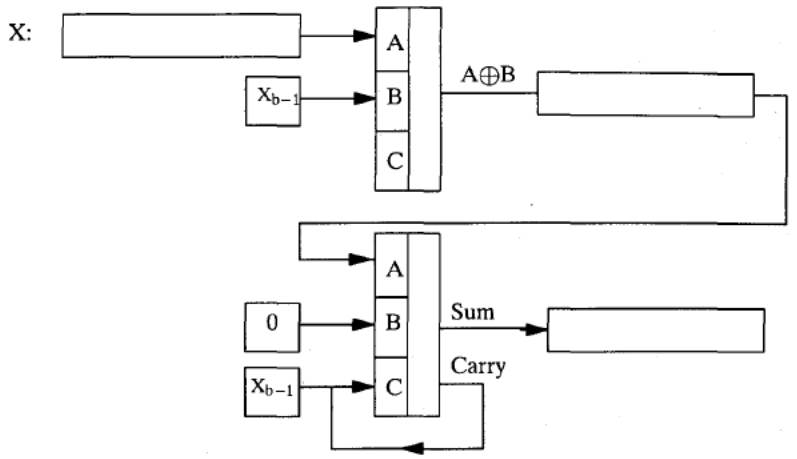


Figure 8.8, Absolute value computation

The magnitude of the gradient is approximated as the sum of the absolute value of the gradients, which is an addition and takes **3b+2** cycles.

The thresholding is performed with a fixed thresholding value and is done in the same way as the max/min-test, with a subtraction and only saving the last Carry-bit. This takes **2b+2** cycles.

The thinning phase is the connectivity preserving shrinking described in section 3.6. The only difference is that we have added an extra template that removes single 1's. The templates are shown in Figure 8.9, and as we will apply a two step thinning this will take **2*88=172** cycles.

```
- - 1    - - 1    - 0 -    - 0 -
0 1 1    0 1 1    0 1 1    0 1 0
- 0 -    - - 1    - - 1    - 0 -
```

Figure 8.9, Special thinning templates

Table 8.3 contains a time consumption summary of the algorithm.

Operation	Time
Separate median filtering	262
Low-pass filtering/ Gradient computation	237
Magnitude	96
Thresholding	18
Thinning	176
	789

Table 8.3, Summary of Median-Sobel-Thinning algorithm

In a 20 MHz system each line will take 39.5 μs to process and a whole image is completed in 5.0 ms.

8.3 Second derivative zero-crossing

As can be seen from Figure 8.10, it seems reasonable to locate an edge at the zero-crossing of the 2nd derivative. This is the main idea behind Canny's edge detection method, see [2].

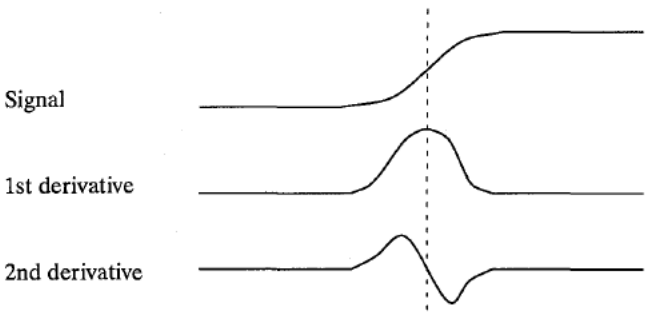


Figure 8.10, Second derivative zero-crossing in 1D

As an image is two-dimensional and discrete, the implementation of Figure 8.10 will include the following three steps, see [8].

(1) Find the maximum |gradient| in at least four directions.

(2) In the direction of the max |gradient| compute the 2nd derivatives.

(3) If the 2nd derivative has a zero-crossing inside the pixel area and if the maximum |gradient| is greater than a specified thresholding value = > the pixel is an edge pixel

Since the algorithm will run on a SIMD machine, we to rearrange these three steps as follows.

(1) Compute the |gradient| in all four directions.

(2) Compute the 2nd derivatives in all four directions to see if there are any zero-crossings.

(3) Select the maximum |gradient| and at the same time the corresponding 2nd derivative response. The result is ANDed with the result from the thresholding comparison and a 1 is obtained if we have an edge pixel.

The kernels for computing the gradients are shown in Figure 8.11. By using the coefficient 5 and 7, the error between the ideal and the approximately gradient is only 1% see Equation (8.2).

$$\frac{7}{5} = 1.4 \approx \sqrt{2} \approx 1.414 = 1.4 * 1.01 \tag{8.2}$$

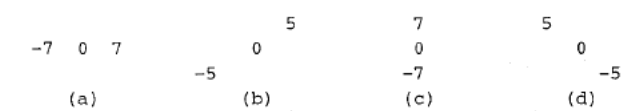


Figure 8.11, Gradient kernels

We begin by using the kernel (c). After shifts to the left and to the right, we compute the results from kernels (b) and (d), and store both these results and the original pixel values in the memory.

The constants 5 and 7, have the binary codes 101 and 111. Thus, a multiplication with 5 means that we take the result and add it with a 2 step shifted version of the result, which takes **2*(3b+2)=6b+4** clock cycles. Similarly, a multiplication with 7 needs 3 additions. However, if we use the SDC-code in section 3.4, 111 could be rewritten into 100−1, which means one addition and one subtraction. This means that the kernels (a) and (c) also take **6b+4** cycles each. In the kernel (a), (b), and (c) we will need **3b** cycles to load the data into the shift register. This makes a total of **27b+16** cycles for all kernels.

The next step is to compute the absolute values of the gradients. As described in section 8.2, this computation takes **4b+3** cycles. For all kernels this makes **16b+12** cycles.

Altogether the gradient computation takes **43b+28** cycles.

To compute the 2nd derivatives of the image we use the kernels in Figure 8.12.

```
                          1           2           1
2  -4  2             -2              -4              -2
                  1                   2                   1
   (e)                (f)            (g)             (h)
```

Figure 8.12, Second derivatives kernels

As the constants 2 and 4 in the kernels are performed as shifts, each kernel only takes one addition and one subtraction, which means **4*2*(3b+2)=24b+16** cycles.

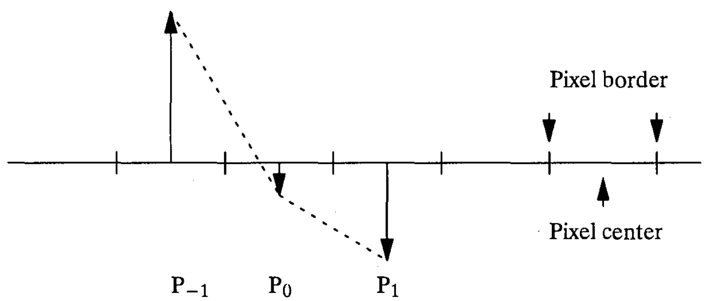


Figure 8.13, Zero-crossing region for each pixel/PE

To determine if we have a zero-crossing within a pixel area we inspect the nearest neighbors. Figure 8.13 shows an example in 1D. The rule is defined as follows, where sign(A) means the sign of the value A.

(a) If $\text{sign}(P_0) \neq \text{sign}(P_{-1} + P_0)$ then there is a zero-crossing.

(b) If $\text{sign}(P_0) \neq \text{sign}(P_0 + P_1)$ then there is a zero-crossing.

This means that we have to compute the sign of the result from two additions in four directions. The sign calculation for each addition takes **2b+2** cycles. After each pair of additions

the results are ORed which takes **3** cycles. This means **4*(2*(2b+2)+3)=16b+28** cycles altogether.

Each pixel now has 4 bits, one for each direction, which are 1 if there is a zero-crossing in that direction. We will now perform step (3) in the computation which means that we will select the maximum gradient and at the same time get the corresponding zero-crossing result followed by a threshold of the gradient. Table 8.4 shows the value that each PE/pixel contains.

G_0	G_1	G_2	G_3	G_i = Absolute gradient in i:th direction
Z_0	Z_1	Z_2	Z_3	Z_i = 1 if zero-crossing in i:th direction

Table 8.4, Contents in each PE

We start by subtracting G_0 from G_1 and only saving the final sign bit. This takes **2b+2** cycles. This bit is used to multiplex the largest G value and its corresponding Z value. This takes **3b+3** cycles and is done three time which makes a total of **15b+15** cycles. We can now compare the outside thresholding value with the greatest G value in **2b+2** cycles, and after that compute AND between the Z value and the sign-bit in the comparison, which takes **3** cycles. We now have a 1 if the pixel is an edge pixel.

The whole algorithm is shown in Table 8.5. In an 8 bits system this means 900 cycles per row and 115200 cycles per frame which gives us one frame each 5.8 ms.

Phase	Operation	Time
(1)	Input pixel	b
	Gradient	27b+16
	|Gradient|	16b+12
(2)	2nd derivatives	24b+16
	Zero-crossing test	16b+28
(3)	Select the max |gradient|	15b+15
	Final edge test	2b+5
		101b+92

Table 8.5, Summary of the second derivative zero-crossing algorithm

8.4 Extended second derivative zero-crossing

One possible extension of the previous algorithm is to try to connect broken lines by conditional expansion, see [15]. This algorithm makes use of two thresholds, T_{low} and T_{high}. The idea is to set a pixel to 1, i.e. an edge pixel, even if they do not have a $|\text{gradient}| > T_{high}$, providing that all the following conditions are met.

(1) The value of |gradient| is greater than a thresholding level called T_{low}.

(2) The direction of the gradient is the same as for an edge pixel in the neighborhood.

(3) The direction to this edge pixel is orthogonal to the gradient.

This means that if a pixel has a N-S gradient greater than T_{low} but smaller than T_{high}, there must be an edge pixel to the east or to the west, with the same gradient direction, for this pixel to become an edge pixel.

Figure 8.14 shows an edge segment where two presumed edge pixels have not been set. In case (a) the pixel becomes an edge pixels as it has the same direction as one neighbor. In case (b) the direction of the gradient is wrong and the pixel remains unset. The pixels (c) are not set as the propagation is only done orthogonal to the gradient direction. This means that the (c) pixels do not get set even though they have the right direction.

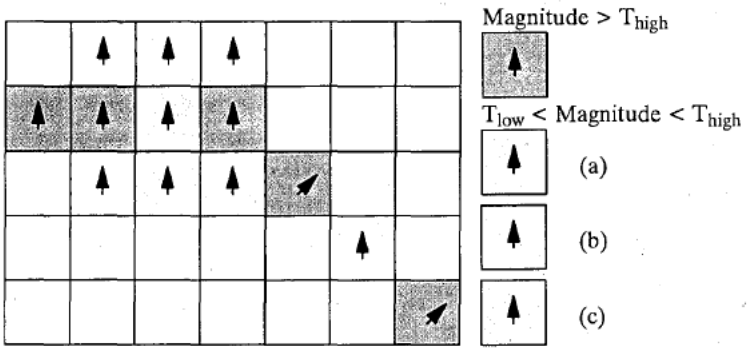


Figure 8.14, Examples of edge propagations

To connect a broken edge line we might need an arbitrary number of propagation steps. However, in this section we will limit this to one step.

This extended algorithm is identical to the first two phases of the zero-crossing algorithm in the previous section. The last phase is almost identical, the only difference is that we save the direction of the gradient for the maximum gradient. This direction is coded with two bits, see Table 8.6. The code is designed so that two directions are orthogonal if and only if we get a 1 from Equation (8.3).

Code	Gradient direction	Corresponding kernel (Figure 8.11)
00	E-W	(a)
01	NE-SW	(b)
11	N-S	(c)
10	NW-SE	(d)

Table 8.6, Direction code

$$Orthogonal? = \overline{Code1_1 \oplus Code2_1} \cdot \overline{Code1_0 \oplus Code2_0} \qquad (8.3)$$

Each pixel now receives direction data and corresponding edge data from each of the possible 8 directions. The pixel is an edge pixel if either of the two following conditions are met.

(a) The pixel is already an edge.

(b) There is a direction match between the pixel and the incoming pixel, and that this pixel is already an edge pixel. (A direction match means that the gradient direction is the same and that the data is fetched from a pixel orthogonal to that direction.)

The object is now to compute equation (8.4).

$$NewEdge = OldEdge + \sum_{all\ dir} \overline{PixDir_i \oplus Dir} \cdot \overline{PixDir_i \oplus \overline{direct(i)}} \cdot OldEdge_i \cdot PossibleEdge_i \quad (8.4)$$

NewEdge : *Edge pixel after one iteration*
OldEdge : *Edge pixel after original algorithm*
PixDir : *Coded direction of neighbor pixel*
Dir : *Coded direction of pixel*
direct(i) : *Coded orthogonal direction i*
$OldEdge_i$: *Neighbor edge pixel after original algorithm*
PossibleEdge : *Edge pixel after original algorithm when* T_{low} *is used*

To compute $PossibleEdge_i$ requires **4*(2b+2)=8b+8** cycles. The expression within the Σ sign in equation (8.4) consists of 4 exclusive-OR and 8 logic-AND. The exclusive-OR takes **10** cycles and the logic-AND takes **12** cycles if we use the Carry-Feedback. To compute the final NewEdge requires 8 logic-OR which takes **10** cycles. The transport between the PE's takes **9+2** cycles as all the directions and corresponding OldEdge values are stored in the shift register and shifted at the same time. This is summarized in Table 8.7.

Operation	# of cycles
Computation of $PossibleEdge_i$	8b+8
Multiplex the direction	12
Transport data	11
Compute equation (8.4)	32
	8b+63

Table 8.7, Summation of conditional expansion

This means that it will take a total of **1178** cycles to compute one line. This is only 10% slower than the original routine. However, a more realistic number of expansion steps will be between 10 and 20, which would result in memory problem. Nevertheless, the architecture in PASIC is powerful enough to compute a few steps of expansion.

8.5 Contrast amplification

One task for an image processing system is to enhance the input image so that it becomes more readable to the human eye. Medical studies have shown that the eye is very sensitive to high spatial frequencies and less sensitive to low frequencies including the DC component, see [12]. The goal is therefore to amplify the high frequencies and to preserve the level of the low frequencies. In a 1D case, the filter response looks like Figure 8.15.

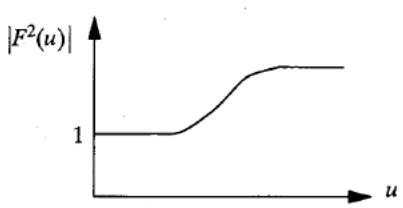


Figure 8.15, Filter response for contrast amplification

One way to obtain this filter response is illustrated in Figure 8.16. We first low-pass filter the image, then we create a high pass filtered image by subtracting the original image from the

low-pass filtered image. The output image is obtained by adding the low-pass filtered image with an amplified version of the high pass filtered image.

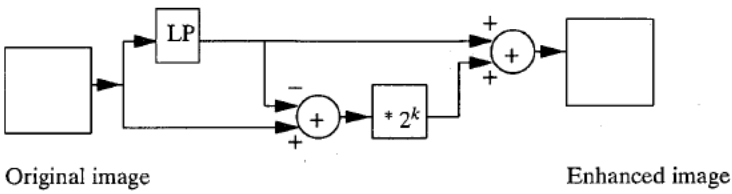


Figure 8.16, Contrast amplification scheme

All these steps are linear and correspond to a convolution shown in Figure 8.17.

$$\left(2^k \left(\begin{bmatrix} 0&0&0 \\ 0&16&0 \\ 0&0&0 \end{bmatrix} - \begin{bmatrix} 1&2&1 \\ 2&4&2 \\ 1&2&1 \end{bmatrix} \right) + \begin{bmatrix} 1&2&1 \\ 2&4&2 \\ 1&2&1 \end{bmatrix} \right) / 16 =$$

$$\left(2^k \begin{bmatrix} -1&-2&-1 \\ -2&12&-2 \\ -1&-2&-1 \end{bmatrix} + \begin{bmatrix} 1&2&1 \\ 2&4&2 \\ 1&2&1 \end{bmatrix} \right) / 16$$

Figure 8.17, Contrast amplification convolution

Depending on the value we assign to k, we get different kernels which corresponds to more or less contrast amplification. Three of the kernels are shown in Figure 8.18.

$$\begin{bmatrix} 0&0&0 \\ 0&16&0 \\ 0&0&0 \end{bmatrix} / 16 \qquad \begin{bmatrix} -1&-2&-1 \\ -2&28&-2 \\ -1&-2&-1 \end{bmatrix} / 16 \qquad \begin{bmatrix} -3&-6&-3 \\ -6&52&-6 \\ -3&-6&-3 \end{bmatrix} / 16 \qquad \begin{bmatrix} -7&-14&-7 \\ -14&100&-14 \\ -7&-14&-7 \end{bmatrix} / 16$$

k = 0 k = 1 k = 2 k = 3

Figure 8.18, Contrast amplification kernels

The low-pass filtering is theoretically accomplished as a separable convolution as in Figure 8.19, which is much the same as the Sobel operator described in section 4.1.

$$\begin{bmatrix} 1&2&1 \\ 2&4&2 \\ 1&2&1 \end{bmatrix} = \begin{bmatrix} 1&1 \end{bmatrix} * \begin{bmatrix} 1&1 \end{bmatrix} * \begin{bmatrix} 1 \\ 1 \end{bmatrix} * \begin{bmatrix} 1 \\ 1 \end{bmatrix}$$

Figure 8.19, Separated low-pass filter

However, practically it is realized as in Figure 8.20, which is the row pipelining method described in chapter 5.

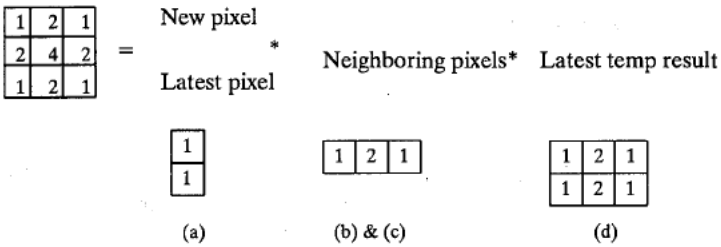


Figure 8.20, Separated row-pipelined low-pass filter

Figure 8.20(a) shows the first operation, which is an 8-bit addition. The next step (b) is also an 8-bit addition as we have truncated the result with a factor of two. Step (c) is once again an 8-bit addition and at the same time a store of the temporary result in the memory. Finally (d) performs a 8-bit addition with previous result, we have a low pass filtered image.

The output image is then computed as in Equation (8.5), where LP( ) stands for Low pass filtering.

$$Out = LP(In) + 2^k * (LP(In) - In) \qquad (8.5)$$

Equation (8.5) is computed with one subtraction and one addition since the multiplication takes no time as it is performed as a shift. Thus, the total number of cycles required is **(3*8+1)*4+(3*8+1)*2=150**.

This type of algorithm is very well suited for PASIC. The reason is it contains neighborhood operations, it can be pipelined in a line wise fashion, and the multiplication with 2^k takes no time since it is just a question where to take the data in a bit serial machine. The amount of memory needed is also very low. In a 20 MHz system it takes 7.5 µs per line and 0.96 ms per frame.

9 Segmentation

9.1 Segmentation on parallel machines

An object in a binary image is defined as a group of connected pixels. Under the 4-connectivity rule each pixel has four connected neighbors, north, west, south, and east. Thus, the binary image in Figure 9.1,is considered to contain two different objects.

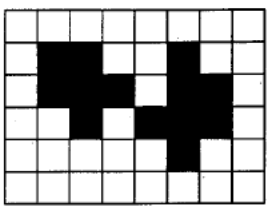


Figure 9.1, Two different objects

A segmentation has been performed when every object in a binary image is marked with an unique label. In Figure 9,2 we have labeled the three objects with the arbitrary labels 1, 2, and 3.

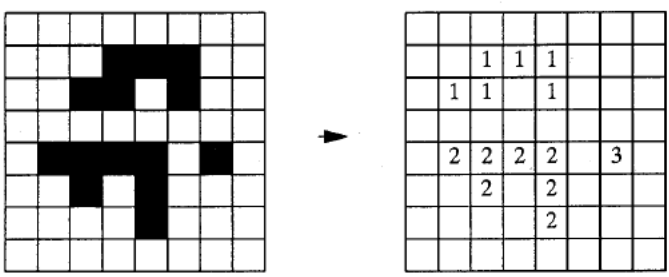


Figure 9.2, Segmentation of a binary image

It is obvious that to label an object, this label has to be distributed to all pixels in the object from some originating points. This means that the time to perform a segmentation is data dependent. To propagate a label through an object requires O(N) steps, where N is the largest distance between two points. In Figure 9.3 the distance between A and B is 8.

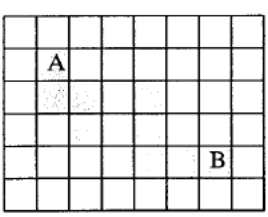


Figure 9.3, The distance between A and B is 8

The worst case of segmentation is a Meander curve, see Figure 9.4. The length of this curve is $N/4*8*N/4=N^2/2$, where N is the size of the image.

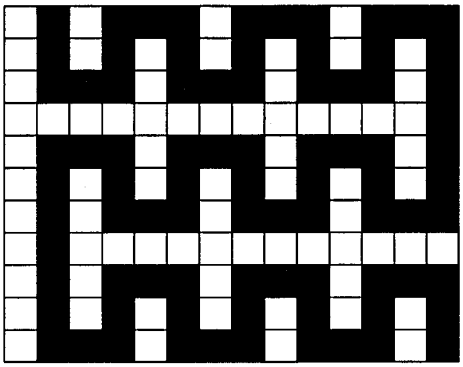


Figure 9.4, The Meander curve

One parallel algorithm that segments an image with objects containing holes is called PLPA, Pure Label Propagation, see [8]. Initially, each pixel is assigned a unique label number, coming from the X and Y coordinates of the pixel. One of the LSB in the X or Y coordinates can be dropped as two adjacent pixels, in either the X- or the Y must be in the same object. This gives us $(n-1)+n=2n-1$ bits per label, where $n=^2\log N$ and N is the size of the image. The basic operation step in the algorithm is the following.

- If a pixel has a label that is larger than the label of its nearest neighbor, and both pixels are object pixels, then the label is replaced by its nearest neighbors label.

The propagation may takes place in all four directions, north, east, west, and south, or in one direction at the time. The propagation then continues until no changes occur in any of the four propagations.

The PLPA algorithm is designed for a 2D processor array. However, it is possible to adopt this type of algorithm to a 1D processor array, i.e. PASIC. In a 1D array the image is processed from the top to the bottom. This means that we have a natural propagation flow from top to bottom, i.e. the south propagation. In the top-down scan we ignore the north propagation, which leads us to the question if we should propagate in both east and west directions or just in one of the directions. Figure 9.5 shows these types of propagation, (a) is a south-east propagation and (b) is a south-east-west propagation. The * indicates the position of the origin PE which means that we first perform a horizontal propagation followed by a vertical.

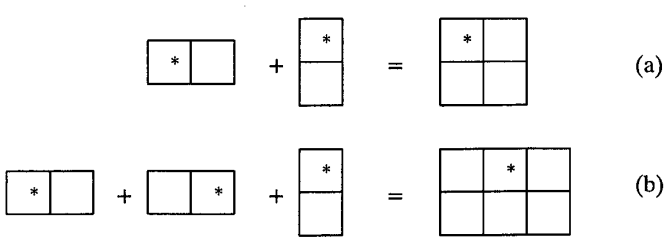
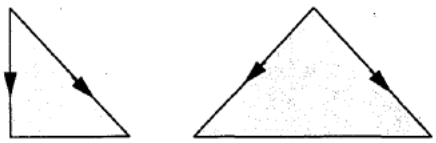


Figure 9.5, South-east- and south-east-west propagation

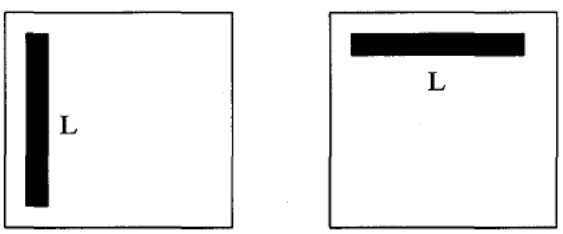
These two operations result in two different propagation patterns, which are showed in Figure 9.6.



(a) (b)

Figure 9.6, Propagation pattern

Both procedures are fast when labeling a long vertical object and slow when the object is horizontal. For instance, the object in Figure 9.7(a) will be correctly labeled after one vertical scan, while the object in Figure 9.7(b) will need L vertical scans in the East-West case and 2L in the East-case.



(a) (b)

Figure 9.7, Different orientation of object

A labeling routine more suited for a 1D array is called Run-Track, i.e. R-T. This algorithm is illustrated in Figure 9.8, and it operates on one line at a time. It start be assigning each left edge a unique label and after that propagate this label to the nearest right edge, see Figure 9.8(a). Next step is to propagate the labels to the next line, see Figure 9.8(b). The labels are propagated in both direction and if we have two labels, we choose the lowest one, see Figure 9.8(c). We finally label the pixels that have not received a label yet, see Figure 9.8(d).

If all the objects in the images were convex our job would be finished. However, since this is not normal the case we have two choice. Either, repeat this procedure until there is no change of labels, or to make a list of which labels that belong to the same object. If we choose to repeat the procedure we will have to work from the bottom to the top of the image, from the top to the bottom, and so on until there is no label change. This will result in Figure 9.8(e). The list procedure stops with the result in Figure 9.8(d) plus the list (1,2) which tells us that label 1 and 2 belongs to the same object.

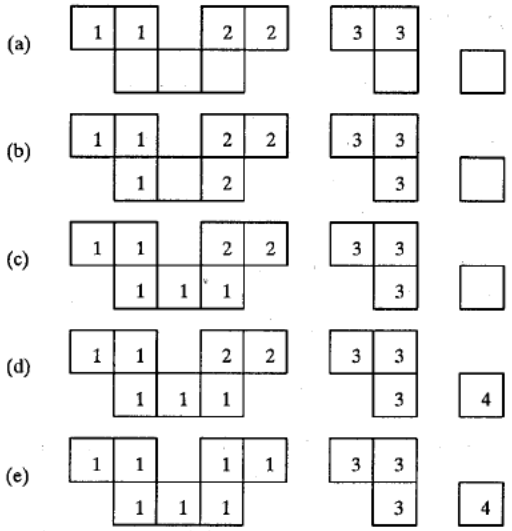


Figure 9.8, Different phase of the R-T algorithm

So far we have mention three algorithms, two types of start labels, and two types of scanning methods. Table 9.1 shows all possible combinations of the methods.

# scans	Start label	PLPA S-E	PLPA S-E-W	R-T
1	Coordinates	(a)	(a)	(b)
	Limited	(c)	(c)	(d)
2 or more	Coordinates	(e)	(e)	(f)
	Limited	(g)	(g)	(h)

Table 9.1, Different segmentation combinations

We will hereafter omit the cases (a) and (e) from further examination since it is possible just to use a limited number of label. This will speed up the process and limit the memory requirement which will be explained in the following sections. Cases (b) and (f) will also be omitted on the same ground. The (c) cases can be omitted since the one scan and list procedure is only meaningful together with the R-T algorithm. The reason is that the other two algorithms we will have unlabeled pixels after one scan. Figure 9.9 shows an image segment after one PLPA S-E-W scan where we have unlabeled pixels.

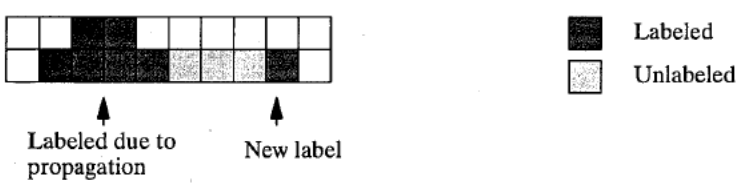


Figure 9.9, Unlabeled pixels after one scan

This leaves us with the (b), (g), and (h) cases which will be investigated in the following sections.

9.2 Limitations for PASIC

The memory needed in each PE to harbor an entire label image is **Nb** where N is the image size and b is the number of bits in each label. The maximum number of different labels occurs when we have a chess-board image. In that case we have $N^2/2$ different labels which means gives us a **b** value of $^2\log(N^2/2)=2*^2\log N-1$. Thus, for a N=128 image each PE needs 1664 bits.

This means that if we want to use a iterating procedure where the image is processed back and forth, we need at least 1664 bits per PE. Even if we could reduce the number of different labels by a factor of $128=2^7$, we still would need 768 bits per PE which seems to be to large even for a next generation of PASIC. The only way to overcome this problem is to use some external memory. One example of that is described in section 10.4.

We will therefore, present the algorithm both as if we had had external memory, i.e. cases (g) and (h) in Table 9.1, and if we did not have external memory, i.e. case (b).

Whether or not we have external memory we want to reduce the maximum number of labels. The reason is to suit the shift register and to reduce the computation time. In the following algorithms the number of bits per label is set **6**, i.e. 64 unique labels. One advantage of using 6 bits is that it fits into our parallel shift register, as the algorithm uses two bits. However, we are forced to household with the labels. The optimal way is be to use one labels per object, which the algorithms do not reach. However, we will show that we can get very close to this optimum with the simple strategy for labeling: wait to use a new labels on a row until we are sure that a pixel is not connected to any labeled pixel on this row or any row above.

9.3 Implementation of the multi-scan procedure

We are now going to describe the implementation and cycle count for the R-T algorithm, which result we will use to compare the two variants of PLPA algorithms. All three algorithms will be adopted to the 1D array of PASIC.

In the R-T algorithm each pixel gets the following set of data.

(1) One bit to indicate object or background called p.

(2) One control bit to indicate if the pixel has been labeled called l.

(3) Six bits for the label called L.

The first step of the algorithm, step (a) in Figure 9.8, is to label the first row. This is accomplished by first setting a 1 in all PEs which has a background pixel and has a PE to the right that has an object pixel. The labels are calculated for all pixels in the first row by simply add all 1 bits to the left of each pixel, see Figure 9.10. This labeling procedure is described in section 4.6, and takes **21b+146** cycles, which is **272** cycles for **b=6**.

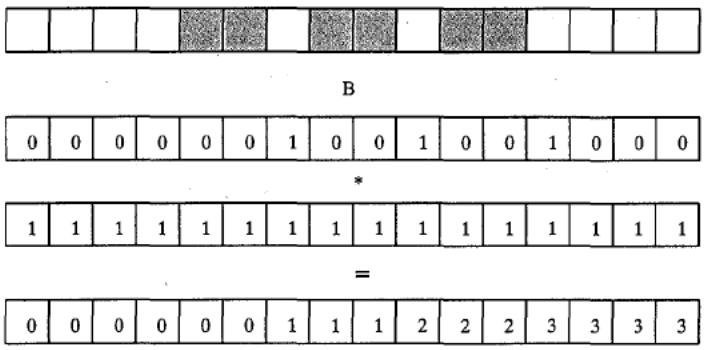


Figure 9.10, Labeling of the first row

We now copy the labels to the next row, see Figure 9.8(b), under the following condition.

- If the above pixel, p_0, and the present, p_1, are object pixels then copy the label number and set the labeled bit, l_1.

In PASIC this is one multiplexing with p_0 as the control bit followed by one assignment of l_1 to p_0, which takes **3b+3** cycles.

The labels are propagated to the right and to the left, using the shift register. This horizontal propagation is done as follows.

- If the neighbor pixel is an object which is labeled, and
- the pixel is labeled with a higher label or not labeled,
- Then propagate the new label and set the l bit.

This corresponds to the following expression

```
IF p0l0(l1(L0<L1)+¬l1) = 1 THEN L0
                           ELSE L1
l1 = l1 + l0
```

During the propagation both the memory and the shift register obtain data of the label and the l bit. This operation takes **2b+2+8+3b+3=5b+13** cycles per shift. We continuously check Global-OR to see if any pixel has an expression in the IF statement which is 1, i.e. we have a change of label. As long as Global-OR is 1 we continue to propagate. When Global-OR becomes 0 we reverse the propagate direction and when Global-OR becomes 0 again we stop the horizontal propagation. This will take $(5b+13)*(N_l+N_r)=43*(N_l+N_r)$ cycles, where N_l and N_r stands for the average number of propagation steps in the left and right direction.

The rest of the line is labeled in the same way as the first line. The difference is that we only create a 1 if the pixel to the right is an object which has not been labeled, i.e. its label bit l is 0. On the first row zeroes were shifted in from the left. This time we will shift in from the left what was shifted out from the right on the last row to get unique labels. This method is described in section 4.6, and takes **272** cycles as before.

We continue the downwards propagation until the whole image is scanned. The process is then reversed to work upwards. However, we do not create any new labels as all the pixels now are labeled. Instead, the next row propagation is performed as the left-and-right propagation, which means **43** cycles per row.

We continue to process the image up-down, down-up, and so on, until one scan is completed with no exchange of labels. The total number of cycles is shown in Table 9.2, where N_s is the number of scans.

Operation	# of cycles	=
New labels to each row	272*128	34816
Propagate new labels to each row	21*127	2667
Left and right propagation	$43*(N_l+N_r)*127*N_s$	$5461*(N_l+N_r)*N_s$
Propagation up and downwards	$43*127*(N_s-1)$	$5461*N_s-5461$
		$32022+5461*N_s*(1+N_l+N_r)$

Table 9.2, Summation of segmentation

To estimate the performance of the algorithm we have to assign some typical values to N_l, N_r, and N_s, which is done in the next section. However, we can compute the cycle count for the worst case, the meander curve. In that case $N_s=2*N^2/16$ and $N_r+N_l=(4+2+4+2)/4=3$, which means **44.8** Mcycles.

9.4 Implementation of the one scan procedure

If we do not have external memory we have to use the one scan and list procedure, which means that every time we have a label change we must report this to the control unit. The means to do this is to check Global-OR to see if we have any change of labels and if that is the case use the select-leftmost-PE routine in section 3.7 and Global-OR to read out the label pairs from one PE at the time.

The change of labels check is done in the same way as for the iterating procedure. However, we now have to read out the label pair(s). We first selects the leftmost PE with the Global-OR function, which takes **5*8+2=42** cycles. Then the two labels are read out from this PE via the Global-OR, which takes **2*6+1=13** cycles. However, in order to reduce the number of duplicates in the connection list we must be sure that each propagation front only contributes with one list entry for each direction on each row. This is done by using an extra bit which is reset after each direction in the horizontal scan. If we did not use this facility the propagation front in Figure 9.11 would result in a number of (1,2) entries in the connection table.

Figure 9.11, Propagation of label 1

Thus, it take **32022+5461*(1+N_l+N_r)+127*N_c*55** cycles to complete one image, where N_c is the average number of label changes on each row. This figure is in most cases very low which means that we can neglect it. This gives us **32022+5461*(1+N_l+N_r)** cycles.

9.5 Results from experiments

To get a better estimation of the N values, the algorithm has been tested on a number of pictures. The images that we used is the same image as in chapter 7, and we used the thresholding values we received from that routine.

The implementation of the two PLPA algorithms within the R-T algorithm is easy since we only limit the propagation in the horizontal direction. This means that in the PLPA S-E, we only propagate one step to the east on the down scan and one step to the west at the up scan. The PLPA S-E-W propagates one step at both directions.

The following data were collected.

No	Image number
Scans	Number of scans of the picture, N_s
Left,Right	Total number of propagation steps to the right and to the left and the average number of propagation steps, N_r and N_l
Labels	Total number of labels that were assigned
Objects	Total number of different objects in the picture

Tables 9.3, 9.4, and 9.5 shows data from one direction PLPA, two directions PLPA, and R-T scan respectively. The bottom row in each Table contains the average value.

Table 9.6 shows the result if we insert the average values from Tables 9.3, 9.4, and 9.5 into the formula in Table 9.2. The R-T algorithm is 2.6 times faster than the PLPA S-E-W and 3.5 times faster than PLPA S-E. Tables 9.3, 9.4, and 9.5 also show that R-T only need half the number of labels compared to the PLPA algorithms.

No	Scans	Left		Right		Labels	Objects
		Total steps	Average steps	Total steps	Average steps		
0	17	1008	0.46	1143	0.53	6	2
1	36	2268	0.49	2286	0.50	26	7
2	26	1638	0.49	1651	0.50	39	15
3	8	504	0.49	508	0.50	17	3
4	13	756	0.45	889	0.53	14	5
5	19	1134	0.47	1270	0.52	25	15
6	26	1638	0.49	1651	0.50	25	2
7	13	756	0.45	889	0.53	7	1
8	15	882	0.46	1016	0.53	16	2
9	43	2646	0.48	2794	0.51	7	1
Mean	21		0.47		0.51	18.2	5.3

Table 9.3, Experiment data from PLPA S-E

No	Scans	Left		Right		Labels	Objects
		Total steps	Average steps	Total steps	Average steps		
0	10	1265	0.99	1265	0.99	6	2
1	18	2277	0.99	2277	0.99	26	7
2	12	1518	0.99	1518	0.99	39	15
3	4	506	0.99	506	0.99	17	3
4	6	759	0.99	759	0.99	14	5
5	10	1265	0.99	1265	0.99	25	15
6	13	1645	0.99	1645	0.99	25	2
7	7	886	0.99	886	0.99	7	1
8	8	1012	0.99	1012	0.99	16	2
9	13	1645	0.99	1645	0.99	7	1
Mean	10		0.99		0.99	18.2	5.3

Table 9.4, Experiment data from PLPA S-E-W

No	Scans	Left		Right		Labels	Objects
		Total steps	Average steps	Total steps	Average steps		
0	3	452	1.18	477	1.24	5	2
1	3	644	1.68	550	1.43	17	7
2	3	546	1.42	488	1.27	21	15
3	3	475	1.24	439	1.14	6	3
4	3	492	1.28	483	1.26	7	5
5	3	507	1.32	474	1.23	19	15
6	3	502	1.31	550	1.43	6	2
7	3	450	1.17	452	1.18	4	1
8	3	513	1.34	477	1.24	9	2
9	3	509	1.33	469	1.22	5	1
Mean	3		1.33		1.27	9.9	5.3

Table 9.5, Experiment data from Run-Track propagation

# scans	Start label	PLPA S-E	PLPA S-E-W	R-T
1	Coordinates	-	-	-
	Limited	-	-	51681
2 or more	Coordinates	-	-	-
	Limited	259090	194760	74618

Table 9.6, Total number of cycles for each algorithm

Thus, a normal picture with $N_s=3$ and $N_r+N_l=2.6$, is processed in **74618** cycles or 3.7 ms on a 20 MHz system using the multi-scan algorithm and the list algorithm takes **51681** cycles on the same system.

10 Improvements of PASIC

10.1 Architecture extension, different approaches

The processing part of PASIC is built around a 1-bit bus, which makes it easy to add and subtract different building blocks. In Figure 10.1 we show two examples of different improvements. In (a) we have added two extra units. From the VLSI point of view, they have been placed below the other units. However, this is of course not necessary as the position on the bus is arbitrary. It is also possible to add units that connect directly to other units, see Figure 10.1(b). In that case the wire connection will be more difficult. A proposed extension of PASIC, where both extra units and direct communication between units are used, can be found in [5].

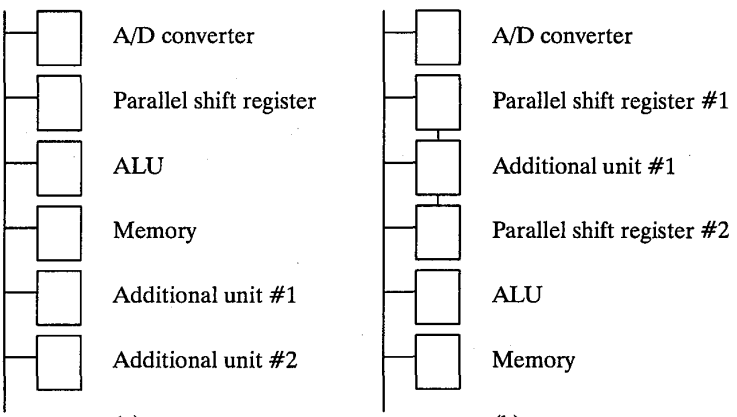


Figure 10.1, Different extension approaches

Another way to enhance the performance is to increase the processing power in the existing units, e.g. more Boolean functions in the ALU, larger memory, and wider shift register. Yet another way is to create more data paths within the units. All these types of enhancements will be discussed in the following sections.

10.2 Extended ALU

The ALU could be extended in the following ways.

(1) More 3-input functions

(2) More feedback functions

(3) Four or more input registers

(4) Two or more feedback register functions

We showed in section 3.1 that we can implement most of the possible 3-input functions in the optimal 4 cycles. We also showed that those 3-input functions that we cannot compute

in 4 cycles can be computed in 5 to 10 cycles. This means that the improvement of the performance is small in case (1) since other 3−input functions are seldom used compared to the add/sub/mux functions.

A more important feature in the ALU is case (2), when we have a data path from an output function, back to an input register without interfering with the bus, similar to the Carry-Feedback case. One such example is feedback for comparisons, see section 4.7, where we had two bit−vectors and compared them bit by bit and if any two bits were not equal the result was 1. Figure 10.2 shows two input vectors which are bit−wise Exclusive−ORed with each other and the result is then ORed with the previous results.

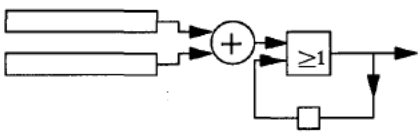


Figure 10.2, Comparison with feedback

If we want to have more feedback signals to the C register than the CarryFeedback, we must have a multiplexing unit for the feedback signals. Figure 10.3 shows one realization of such a feedback control.

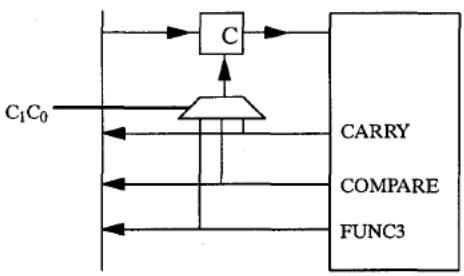


Figure 10.3, Extended feedback

The multiplexer is controlled by two bits, described in Table 10.1. The Compare performs the Boolean function A⊕B+C and the third function Func3 is a feedback function that we can have almost for free as we have a coding position left.

C_1	C_0	Function
0	0	Disable
0	1	Carry feedback
1	0	Compare feedback
1	1	Func3 feedback

Table 10.1, Code to the extended feedback

Another possible way to enhance the ALU would be to add an extra input register, case (3). However, by inspecting described algorithms we find that we almost never used a 4−input functions and in those cases where we did use 4−input functions, the Carry feedback was used to overcome the problem.

In case (4) we use a fourth register to obtain more than two states in the state diagram. Figure 10.4 shows the 2–state Carry function where we go–to/stay–in C=1 if we have a Carry. The output from the function is determined by the input vectors and the current state at the C register.

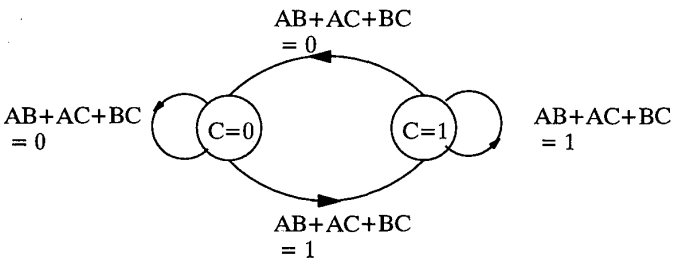


Figure 10.4, State diagram for the Carry feedback

Figure 10.5 shows a 3–state function which requires two registers, C and D. This is used in an operation that in **3b** cycles outputs the largest of to input variables x,y, i.e. z=max(x,y). We initiate the two state registers, CD := 00. Then the two input vectors are loaded, into the A and B registers respectively, one bit at the time starting with the MSB. As long as $X_i=Y_i$ the output will be X_i, i.e. the largest vector. If at any time $X_i \neq Y_i$ the output will be the largest value and we will jump to a state where only the bits from that largest vector is selected. This process takes **3b+2** cycles.

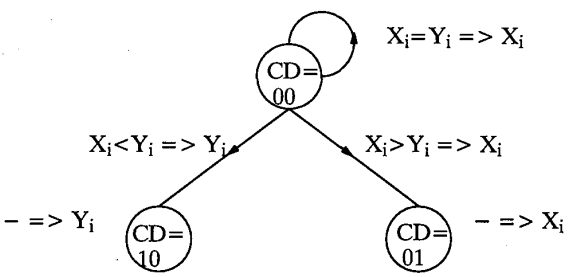


Figure 10.5, Maximum select state diagram

In the present PASIC design this task takes **5b+1** cycles. The two types of median filtering in section 8.2 will with this new feedback take **44b+26** and **20b+12** cycles, true and separated median respectively. The architecture needed for a two–function feedback is shown in Figure 10.6.

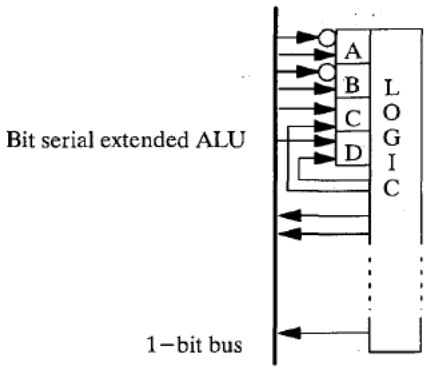


Figure 10.6, Two−function feedback ALU

10.3 Global−OR feedback

In section 3.7 and 4.3 we described the Global−OR function and gave examples of how this function could be used in maximum selection and distribution. However, we also noted that the response time and the complexity of the control unit might become a problem and slow down the execution.

To reduce the burden of the control unit and gain speed, the Global−OR output can be fed back directly to each PE's bus, see Figure 10.7 and [5]. In Figure 10.7 the Global−OR signal is fed back to special memory position which means that one cycle after we put some data into C, each PE can use the result of the Global−OR.

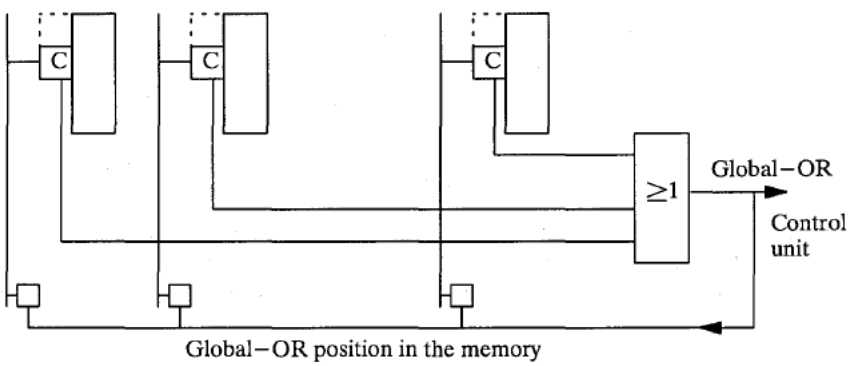


Figure 10.7, Global−OR connected to memory

For the control unit this means that it can run its data−independent micro−program flow uninterruptedly in the algorithm in section 4.3. As another example, Table 10.2 shows the distribution routine in section 7.2 without a Global−OR feedback. This takes **4b+2** cycles if we have a powerful control unit, or maybe **5b+2**, **6b+2**, or worse in a more realistic situation.

```
0->A
M(Flag)->B
FOR b:=0 TO b-1
        M(b_i)->C
        CarryFeedback
        Global-OR-test
        CASE Global-OR of
                1 : 1 -> PS(b_i)
                0 : 0 -> PS(b_i)
ROF
```

Table 10.2, Distribution without internal feedback

If we have a Global−OR feedback the program could most certainly run in **3b+2** cycles, see Table 10.3. Equally important is that we do not need any conditional branching in the micro program for these cases, which simplifies the micro−programming and the design of the control unit.

```
0->A
M(Flag)->B
FOR b:=0 TO b-1
        M(b_i)->C
        CarryFeedback
        Global-OR -> PS(b_i)
ROF
```

Table 10.3, Distribution with internal feedback

10.4 External memory

The memory size is a major problem in many routines. PASIC only contains 127 bits of RAM per PE, which is not enough for many algorithms. By using row−parallel pipelining, the problems can sometimes be overcome. For some other algorithms though, the lack of memory makes them impossible to implement. Those cases can only be handled with external memory.

The ideal solution would be to have each PE's bus connected to one output pin of the chip which in turn is connected to an external memory. However, this solution is not feasible since it requires too many pins. One way to overcome this problem is to let a number of PEs use the same pin, via a multiplexer. The necessary data buffer is a special 1 bit RAM cell in each PE, see Figure 10.8.

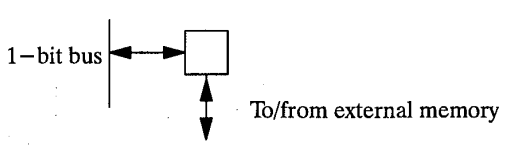


Figure 10.8, The external memory's connection with the PE's bus

The data is transferred to the external memory over the multiplexers, as shown by Figure 10.9. After 8 write cycles, the data from all PEs have been written into the external memory. Read is performed in a similar way.

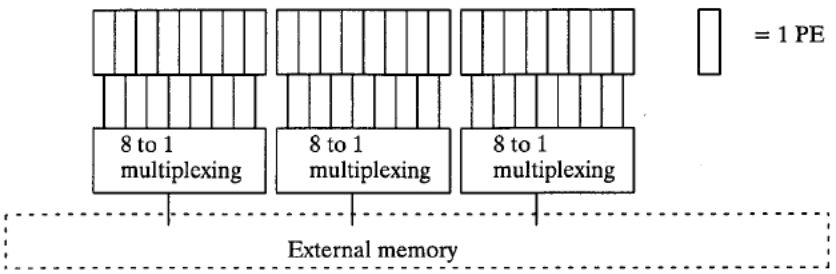


Figure 10.9, The interface to the external memory

One advantage of this solution is that the communication with the external memory can be carried out at the same time as the PE is doing something else. We only loose clock cycles when we store or fetch data into or from the PEs memories. In a general case, the time loss is 1/16 of the total computation time which occurs when data has to be transferred from the external memory cell to the on−chip memory, if we assume a 10 MHz external memory.

External memory is a large bonus for instance in the shading correction algorithm, see chapter 6. For slowly varying shading function we could solve the sparse coefficient problem by interpolation. Fast, random variations can only be compensated for by one set of coefficient for each pixel in the image.

Another obvious application for the external memory is to harbor entire images and thus, make it possible to perform temporal filtering, i.e. a filter that uses two or more time frames.

10.5 Look−up table

In many applications there is a need of a fast table look−up function, LUT, for instance in pseudo−coloring, histogram equalization, or computation of Sin, Cos, Exp etc. In the present version of PASIC this is not supported. Figure 10.10 shows one implementation of a LUT which is built around the original PASIC. The operation code is shown in Table 10.4.

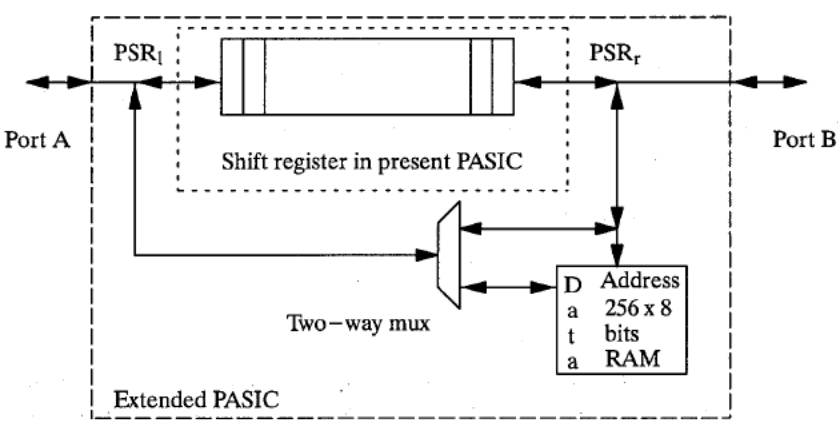


Figure 10.10, Modification for Table look−up

#	Operation
0	Write into specified bit position in register from the bus
1	Read from specified bit position in register to the bus
2	Latch
3	Shift left
4	Shift right
5	Rotate left
6	Rotate right
7	Write Port A into address Port B
8	Read Port A from address Port B
9	Write PSR_l into address Port B, shift left
10	Read PSR_l from address Port B, shift right
11	Write PSR_l into address PSR_r, shift left
12	Read PSR_l from address PSR_r, shift right

Table 10.4, Extended parallel shift register operations

To compute the Sin function for each PE's value, we load the LUT with a Sin function table. Then load the value from each PE to the shift register. We then shift 128 time using function code12 in Table 10.4. After 128 shifts, each PE holds the Sin value in its shift register.

To perform a histogram equalization we do the following steps.

(1) Collect a histogram, h(n).

(2) Compute Inthist, H(n).

(3) Map H(n) to 0..255, H'(n).

(4) Store H'(n) in positions 0 to 255 in the LUT.

(5) Shift out original pixel and receive histogram equalized pixels.

Step (1), the histogram collection is described in section 4.7, and takes **344064** cycles with a result that may look like the one in Figure 10.11.

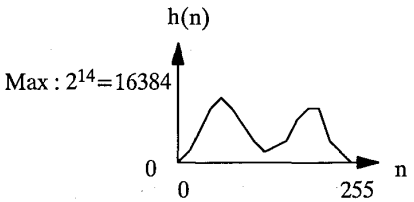


Figure 10.11, After histogram collection

Step (2) is to compute the H(n) which is defined by Equation (10.1).

$$H(n) = \sum_{i=0}^{N-1} h(i) \qquad (10.1)$$

This is done by using sparse filter and storing the result in the rightmost PE, see section 4.2, which takes **436** cycles. The result after this operation on h(n) in Figure 10.11, is shown in Figure 10.12.

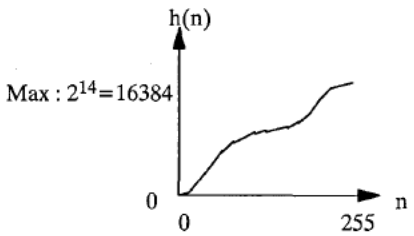


Figure 10.12, After inthist computing

To map H(n) into 0..255, step (3), we use Equation (10.2), which contains only one subtraction.

$$H'(n) = H(n)\frac{255}{2^N} = \left(H(n)2^8 - H(n)\right)\frac{1}{2^N} \qquad (10.2)$$

This takes **3*(N+8)+2=3N+26** cycles, which is **68** cycles for **N=14**, and the result after this is shown in Figure 10.13.

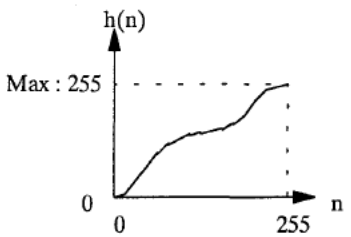


Figure 10.13, After normalization

We can now load the LUT with the values in the shift register, step (4), which takes **128** cycles. We shift in those values to the data input and the address is fed by the values 0 to 255 from the control unit.

Thus, we can now read out a histogram equalized image from either the non–destructive sensor area or from an external memory. We see that it is the histogram collection is by far the most time consuming procedure, which will be the bottle–neck of the system. However, it is often possible to use the same histogram for a number of images taken with small time and spatial differences to each other.

10.6 An application driven extension design for PASIC

One example of an extension package for PASIC is shown in Figure 10.14. This design was proposed in [4] and it consists of 5 extensions compared to the present PASIC. Together, these extensions are used to speed up the thresholding using zero–crossing algorithm from section 7.1. They are:

(1) One extra 8–bit shift register.

(2) A comparison network between the two shift registers which delivers a 1 if both register are equal.

(3) The Global−OR feedback described in section 10.3.

(4) 9 bits of ROM, containing one bit for the generation of constants and 8 bits for the position number of the PE.

(5) External memory, described in section 10.4.

The extra shift register serves two purposes. The first one is to have a sufficient number of bits in the thresholding calculation, as was pointed out in section 7.3. The other is to speed up the pixel comparison in the histogram collection, which is described in section 4.7. The ordinary comparison routine, which implements Equation (10.3) iteratively, takes **2b+2** cycles while parallel comparison of the two shift registers only takes **2** cycles. Thus, we have a speed−up factor of $^{2}\mathbf{logN=b}$ for introducing a parallel comparator.

$$C := \overline{(X_1 \oplus Y_1)+(X_2 \oplus Y_2)+..+(X_b \oplus Y_b)} \qquad (10.3)$$

The advantages of a Global−OR feedback is discussed in section 10.3, and the extra ROM cells are used to load the shift register with the position number for the PE.

Finally, the external memory is capable of storing the whole image so that the thresholding is performed on the same image as the histogram was collected from.

The total gain for these improvements are the following:

(1) A reduction from **344064** to **98304** cycles for the histogram collection.

(2) Higher accuracy in the thresholding computation due to the 8 extra bits.

(3) Simpler micro−code and control unit as we have Global−OR feedback and extra ROM cells.

(4) The external memory gives us the mean to use the threshold on the same image as the threshold was computed for.

Similar substantial gains occur in other applications and algorithms. Above all, the external memory enlarges the set of applicable algorithms immensely.

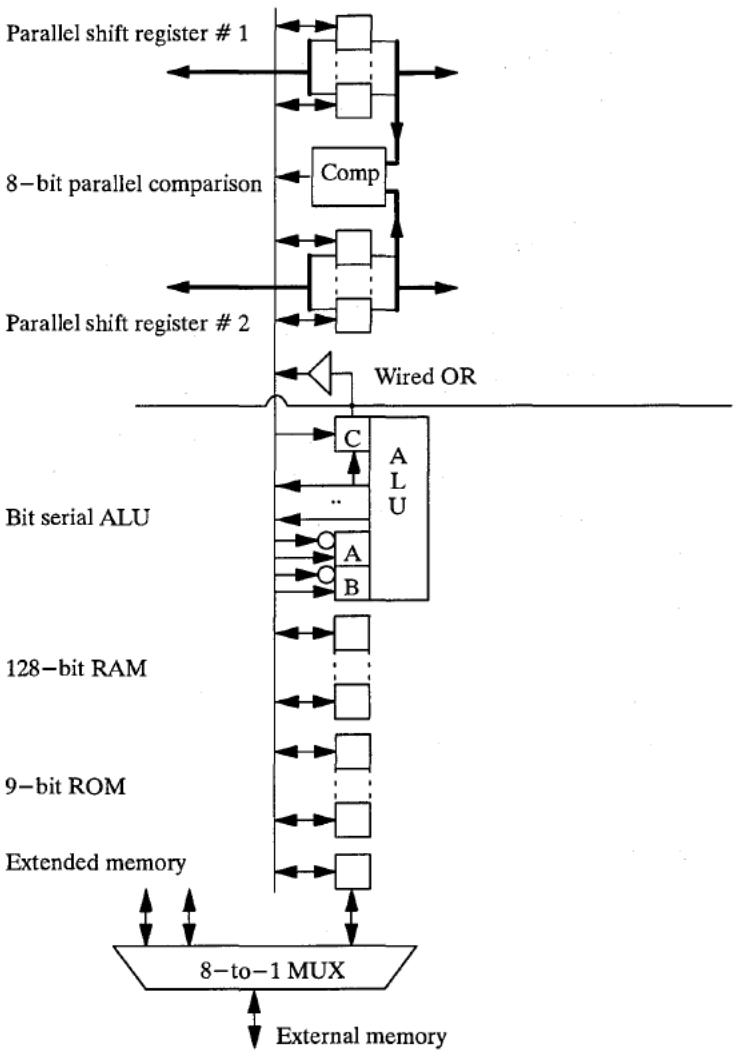


Figure 10.14, Extended PASIC

11 Conclusions

This thesis has described the architecture and the performance of PASIC. It has compared the architecture and the processing capability of PASIC to other processors. It has also described the performance of previously described algorithms on PASIC as well as some new that are special designed for the architecture of PASIC. The thesis has given a few suggestion of improvements for PASIC and described the improved performance each such extension will produce.

A number of unique features in PASIC simplify the programing and reduce the execution time. Some of these are:

- The parallel shift register which is very efficient for neighborhood communications.
- The 1-bit bus architecture which makes the design flexible and VLSI friendly.
- The ALU which is small in size yet indeed competitive with other designs. This means that the performance/mm^2 factor is very high on PASIC.
- The integration of Sensors, A/D-converters, and Processors onto a single chip which reduces the data flow out from the chip.

PASIC should be considered as a prototype for a more full-fledged smart image sensor with higher resolution. By employing sub-micron technology, it seems quite realistic that the next generation of PASIC contains a sensor area of 512x512 and 512 processors. However, since the sensor area grows quadratically while the processing rate only grows linear, the frame rate is subject to a linear decrease for an upscaled version of PASIC. This problem indicates that future large scale smart sensor system must be programmable at the sensor level. The sensor should be able to deliver an image of the scene at a variable resolution and/or deliver just a smaller window of the full scene, so called foveation.

Another goal for the next generation of PASIC is to integrate the control unit on the same chip as the sensor, A/D-converters, and processor array, which is shown in Figure 11.1.

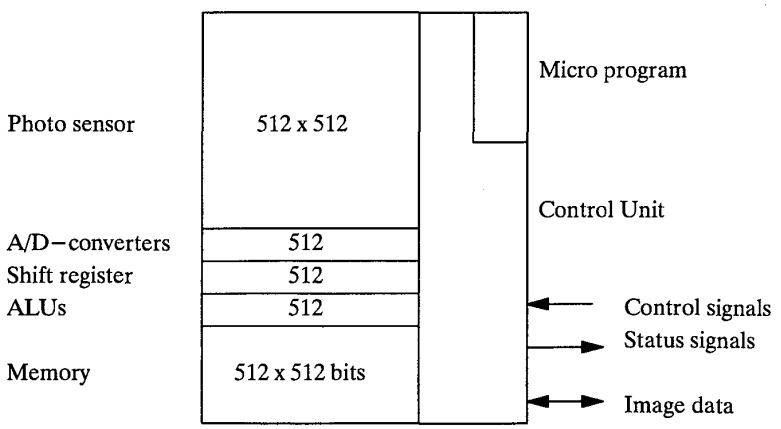


Figure 11.1, Next generation of PASIC

In this thesis, the control unit has been covered only briefly. However, if we want to have a compact system, we are forced to integrate at least some of the control unit functions on the chip. Functions that could be included in this integrated control unit in Figure 11.1, are the following:

(a) Storage of the micro program.

(b) Loop control that can handle nested loops.

(c) Conditional– and unconditional branches.

(d) Output of the image.

(e) Controlling of the external memory.

(f) Reporting status of the program flow via the status output to the external control.

By integrating the the control unit we take the burden off the host computer and reduces the data flow out from the chip. This means that we can use an inexpensive host computer, for instance a PC, remotely positioned relative to the smart camera. We feel that this is what most industrial applications require.

Hopefully, this thesis has convinced the reader that the PASIC or PASIC like designs provides new and promising means for low–level image processing. Very likely, smart sensors of this type is needed to give the field of image processing and computer vision a real breakthrough.

12 References

[1] Batcher K.E., *Design of a Massively Parallel Processor*, IEEE Computer, Vol C29, 1980, pp. 836–840.

[2] Canny J., *A Computational Approch to Edge Detection*, IEEE PAMI, November 1986, pp. 679–698.

[3] Chen K., Afghani M., Danielsson P.E., Svensson C., *PASIC a Processor-A/D-converter-Sensor Integrated Circuit*, Proc of ISCA90, pp. 1705–1708.

[4] Chen K., Danielsson P.E., Åström A., *PASIC a Sensor/Processor Array for Computer Vision*, Proc of Application Specific Array Processors, Princeton, September 1990, pp. 352–366.

[5] Chen K., Åström A., Danielsson P.E., *PASIC a smart Sensor for Computer Vision*, Proc of Pattern Recognition, Atlantic City, June 1990, pp. 286–291.

[6] Danielsson P.E., *Bildbehandling 1990 kapitel 5–7*, Lecture Notes, Dept of EE, Linköping University.

[7] Danielsson P.E., *Generalized and Separable Sobel Operator*, Internal Report LiTH-ISY-I-0975, Linköping, Sweden, 1989.

[8] Danielsson P.E., *SIMD-arrays. A GAPP-exercise*,1986. Lecture notes, Dept of EE, Linköping University.

[9] Forcheimer R., Ödmark A., *Single chip linear array processor*, Proc. of 3rd Scandinavian Conference on Image Analysis, Copenhagen, July 1983, pp 320–325.

[10] Fountain T.J., *Processor Arrays Architecture and Applications*, Academic Press, London, 1987, pp.49–60.

[11] *Geometric Arithmetic Parallel Processor*, Data sheet NCR45CG72, NCR corporation, (1984).

[12] Hubel D.H., *Eye, Brain, and Vision*, Scientific American Library series #22, 1988.

[13] *LAPP 1100 Picture processor, Instruction Set*, Integrated Vision Products AB.

[14] Reddaway S.F., *DAP - a Distributed Processor Array*, First annual symposium on computer architecture, Florida 1973, pp 61–65.

[15] Ruff B.P.D., *A pipelined architecture for a video rate Canny operator used at the initatial stage of a stero image analysis system.* Parallel Architecture and Computer Vision, I. Page, Ed, Clarendon Press, Oxford, pp 171–185.

[16] Svenson B., *LUCAS Processor array - Design and Applications*, PhD thesis, Department of Computer Engineering, University of Lund 1983.

[17] Tucker L.W., Robertson G.G., *Architecture and Application of the Connection Machine*, IEEE Computer, August 1988, pp 26–38.

[18] Wilson S.S., *One dimensional SIMD architectures - the AIS-5000, Multicomputer Vision*, S. Levaldi, Ed, Academic Press, London, pp. 131–149.

[19] Ye Q.Z., *Contributions to the Development of Machine Vision Algorithms*, PhD thesis, Linköping University 1989.

Part II MAPP2200

1 Architecture

1.1 Overview

MAPP2200, Matrix Array Picture Processor, is an image sensor which includes a digital image processor array. It is a commercial product, developed at IVP Integrated Vision Products in Linköping, and samples of this component have been available since August-91. Its predecessor, the 1D sensor/processor LAPP1100 [9] has been available since 1987. MAPP2200 has also a lot in common with PASIC [1], a design which came about in a research project at the University of Linköping in 1989–1992. The MAPP2200 chip is designed in 1.6 µm CMOS and contains approximately 500,000 transistors on an area of 10x15mm^2.

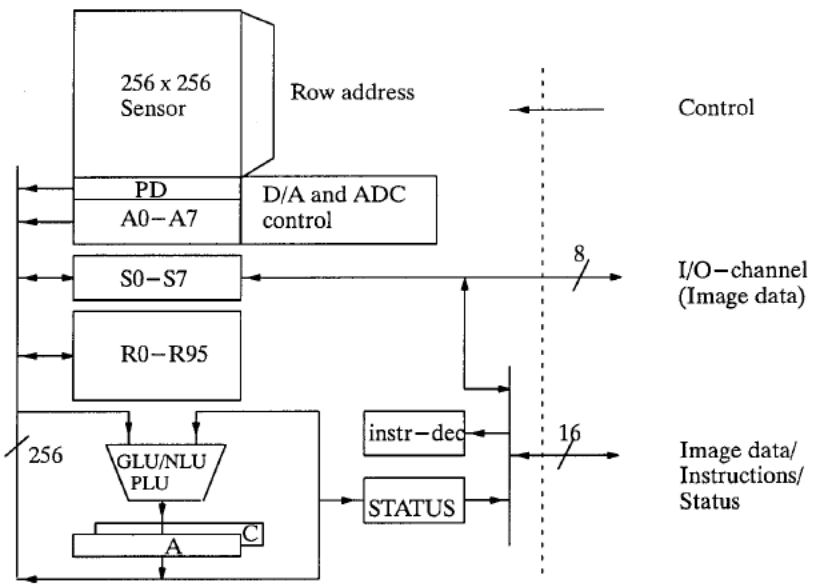


Figure 1.1. Block diagram of the MAPP2200 chip

A block diagram of MAPP is shown in Figure 1.1. At the top is a photodiode matrix which consists of 256x256 sensor elements. The sensor data are read row-wise, in parallel, to an analog register PD and converted to digital values using an internal A/D-converter. The pixel values are digitized, one row at a time, with a precision of 1 to 8 bits. The result is stored in special register positions A0–A7. For each column there is a processing element, PE, consisting of a number of 1-bit registers for storing image data (S0-S7, R0-R95), and an ALU unit (GLU/NLU/PLU) dedicated to low-level image processing. The processors are working in an SIMD mode. The ALU unit operates on the two 1-bit registers A (accumulator) and C (carry), and/or the bus, depending on the instruction. The Point Logic Unit, PLU, performs operations on operands within the PE, the Neighborhood Logic Unit, NLU, performs operations on operand in a 3x1 neighborhood along the PE array, and the Global Logic Unit, GLU, performs global operations involving the whole PE array.

In this section the architecture of MAPP2200 will be described from the programmer's point of view. A more detailed description of the sensor parts of MAPP2200 can be found in [15] and more information about the processor part of MAPP2200 can be found in [14] and [20].

1.2 The sensor

The sensor array occupies about half of the silicon area and consists of 256 rows with 256 photodiodes per row. Figure 1.2 shows one of the 256 photodiode columns and its read-out unit. The image data are read out row-wise in parallel to an analog register (PD). The row number which we want to access is selected by a sensor row address counter which is set with the SETR <row_no> instruction. The transfer of a row analog data is accomplished by manipulating the switches S_1, S_2, and S_3, using the operations listed in Table 1.1.

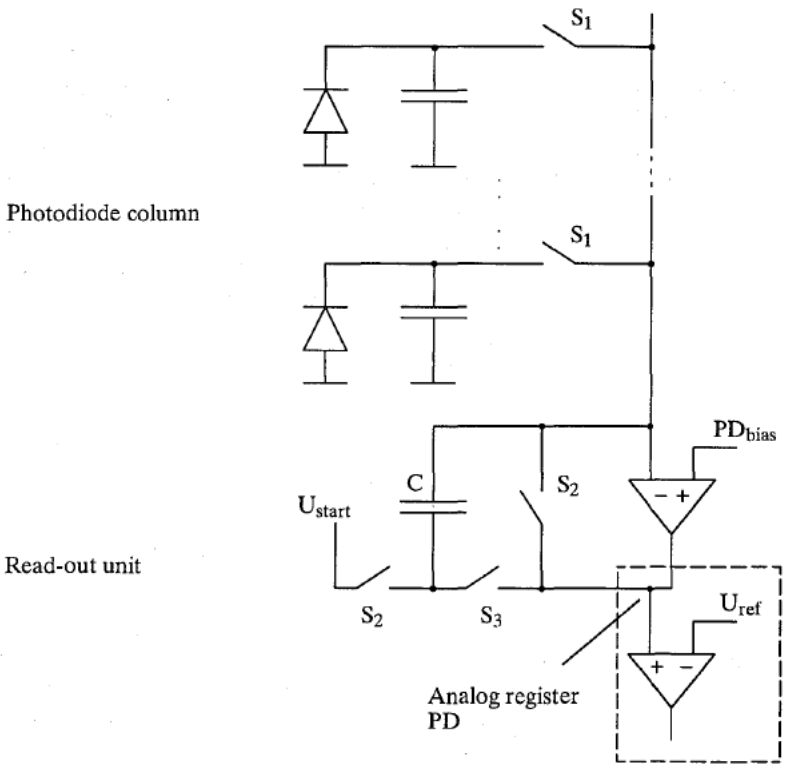


Figure 1.2, Photodiode and read-out circuit

To read data from the photodiode to the PD register, we first reset PD by closing the two switches S_2 and opening S_1 and S_3. Thus, the PD register will obtain the same potential as PD_{bias} while the capacitor C is charged to a pre-defined value. This is the operation RESET in Table 1.1. We can now transfer the data from the photodiode to PD by closing S_1 and S_3 and opening S_2. The charge at the photodiode capacitor will now be transferred to the capacitor C at the same time as the photodiode will be precharged. This is the instruction

INTEG and the read-out is destructive. To prevent further discharge of the photodiode from affecting PD, we close S_1, which correspond to HOLD. If we only want to precharge the photodiode, i.e. not read out its value, we close S_1 and S_2 and open S_3, with RESETP.

S_1	S_2	S_3	Name	Operation
OFF	ON	OFF	RESET	Reset PD
OFF	OFF	ON	HOLD	Hold charge in PD
ON	ON	OFF	RESETP	Reset PD and photodiode
ON	OFF	ON	INTEG	Transfer charge from photodiode to PD

Table 1.1. Read-out circuit function

If an operation in Table 1.1 is accompanied by a ",+", the row address counter is incremented by 1 after the switches are set. For instance, RESET,+ initiates the PD register and then increments the row address counter.

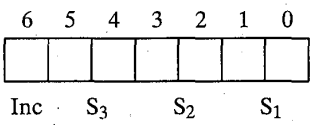
To simplify the programming, four special macros have been defined, shown in Table 1.2. The readout macro is called READPD, which we have described above, and the initialization macro INITPD. The CONNECT and INITCONNECT macros connect the photodiodes to the PD registers so that the current produced by the light on the photodiode discharge the capacitor C at in the read-out circuit instead of the internal capacitor. This is used for non-destructive read-out in Near-Sensor Image Processing (NSIP) where the PD output is interrogated repeatedly, see [7][8] and Part III of this thesis. The CONNECT macro makes it possible to start the interrogation after a certain exposure time, while the INITCONNECT resets the photodiodes before the interrogation.

Macro	Instruction sequence
READPD	RESET : INTEG : HOLD
INITPD	RESETP : RESET
CONNECT	RESET : INTEG
INITCONNECT	RESETP : INTEG

Table 1.2. Sensor macros

If the macros in Table 1.2 are accompanied by a ",+", this will apply to the last operation. For instance, INITPD,+ means RESETP : RESET,+.

The switches S_1, S_2, and S_3 in Figure 1.2 are not limited to the combinations in Table 1.1 but can be arbitrarily controlled by the instruction SETPD <bit pattern>, where the <bit pattern> is to be defined as in Figure 1.3. However, a protection circuit guaranties that the S_2 and S_3 switches are not both switched at the same time. The Inc bit is set when the row address counter should be incremented, i.e. when the instruction in assembler code notation is accompanied with ",+". The operations in Table 1.1 are defined in SETP instructions.



00 : no change
01 : deactivate
10 : activate
11 : illegal

Figure 1.3. Bit pattern for SETPD

The analog value in the PD register can be converted into a digital value in two ways. One way is to produce a binary value using a fixed threshold voltage U_{ref}. This value is hereafter referred to as the (digital) PD register value. Another alternative is to produce a grayscale value using the internal A/D-converter. The A/D-converter is of the ramp-generator/ counter type, and can be programmed for 1 to 8 bits precision, depending on speed and precision requirements. Figure 1.4 shows how the 8-bit digital counter value and its corresponding analog value are broadcasted to all PEs. As long as an analog register PD contains a larger value than the analog ramp, the digital value will be latched into the corresponding ADC-register (A0–A7). Thus, all 256 analog values will be converted in 2^b cycles given a selected precision of **b** bits. Note that the dashed square in Figure 1.4 is identical to the dashed square in Figure 1.2.

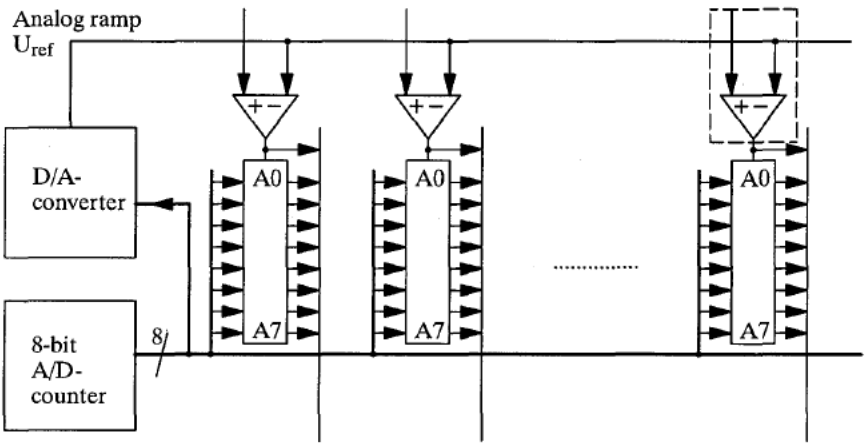


Figure 1.4. The parallel ADC in MAPP2200 from [14]

1.3 Processor architecture and instruction set

Figure 1.5 shows the bus system and all the attached components for one processor slice, i.e. one PE. At the top of each PE, we have the ADC registers, where the A/D-converted values are stored, and the (digital) PD register. They are referred to as A0–A7 and PD. Below them are the registers S0–S7 which constitute an 8-bit bi-directional shift register. This register can be used both for external communication and for communication between processors. The internal memory in each PE consists of 96 bits, R0–R95.

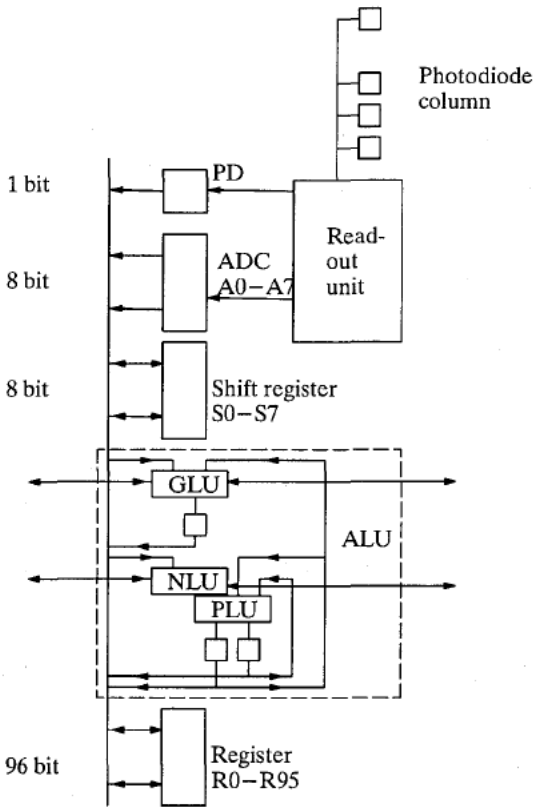


Figure 1.5. One PE

As shown in Figure 1.6, the arithmetic logic unit consists of three major parts: the Global Logical Unit, GLU, the Neighborhood Logic Unit, NLU, and the Point Logic unit, PLU. The GLU performs global operations on the whole PE array, i.e. 256 bits of data, and the execution is split into two cycles. During the first cycle, the bus operand is loaded into the GLU, and at the next cycle, the result is written back into the accumulator via the NLU and the PLU. This means that the assembler instruction

```
LD MARK R0
```

is expanded into

```
MARK R0
LD G
```

This expansion is taken care of by the assembler. The GLU register G is invisible to the programmer.

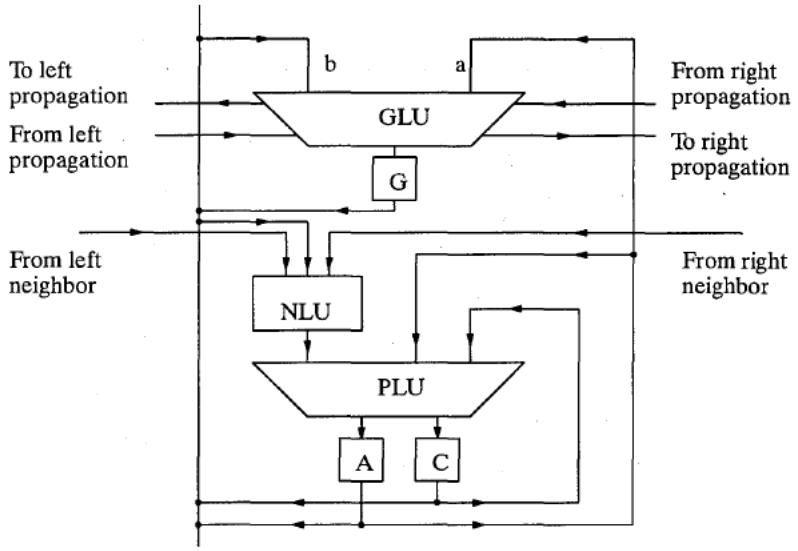


Figure 1.6. The ALU consists of one GLU, one NLU, and one PLU

The fundamental GLU operation in MAPP2200 is the MARK operation. MARK has two operands (a,b). Hence, for our description here we use the notation MARK(a,b) rather than the above assembler notation. Operand a is either the accumulator content A or an instruction-specific constant. Operand b is the content of one of the other registers explicitly designated in the code and brought in via the bus. For the MARK operation as well as the other GLU-operations, it is convenient to visualize the operands as 256-bit binary vectors as shown in Figure 1.7. In this case the operand b is register R0. The effect of the MARK operation is a kind of masking. The result, which is stored in the GLU register (G in Figure 1.6), consists of those connected runs of operand b (R0 in Figure 1.7) which at least in one position overlap a position which is set in operand a (the accumulator). In a final operation cycle the G-values are always loaded to the accumulator.

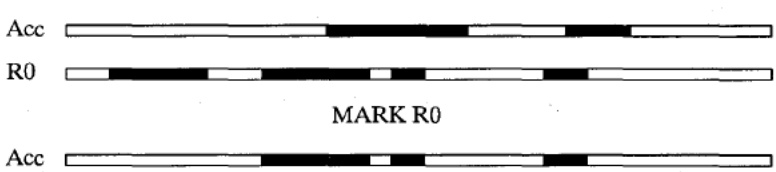


Figure 1.7

The eight different GLU operations in MAPP2200 are shown in Table 1.3. As seen from the Table, the instructions of type fill are complementary to the "pure" mark instruction.

Assembler instruction	Result	MARK(a,b)
MARK	Objects in bus connected to a 1 in A	MARK (A,bus)
LMARK	Objects in bus connected to left border	MARK (LCONST,bus)
RMARK	Objects in bus connected to right border	MARK (RCONST,bus)
LRMARK	Objects in bus connected to left or right border	MARK (LRCONST,bus)
FILL	Holes in b connected to a 1 in A are kept	$\overline{\text{MARK}}$(A,$\overline{\text{bus}}$)
LFILL	Hole in b connected to left border is kept	$\overline{\text{MARK}}$(LCONST,$\overline{\text{bus}}$)
RFILL	Hole in b connected to right is kept	$\overline{\text{MARK}}$(RCONST,$\overline{\text{bus}}$)
FILL	Holes in b connected to left or right border are kept	$\overline{\text{MARK}}$(LRCONST,$\overline{\text{bus}}$)

```
LCONST   = (1,0,0,....,0,0)
RCONST   = (0,0,0,....,0,1)
LRCONST  = (1,0,0,....,0,1)
```

Table 1.3. GLU instructions

The NLU operations employ 3x1 binary masks where the input operands come from the PE's own bus and its two neighbor's buses as shown in Figure 1.6. The mask bits can be either **1**, **0** or **X** (don't care). A **1** means that the corresponding input bit must be 1, a **0** that it must be 0, and a **X** that the input bit can be either 1 or 0. The output from the NLU is 1 if all the input bits matches the mask bits in this sense. For instance, the NLU mask (X1X) means that the output is 1 if the center bit is 1 which reduces the NLU operation to a mere data transfer. The mask (01X) means that the NLU output is 1 if the center bit is 1 and the left neighbor is 0. The NLU accepts all 27 possible masks, although, some of the more frequently used masks have got their own names, shown in Table 1.4.

The Boolean function of the NLU output is

$$NLU_{out} := ((BUS_C \& N_1) + (\overline{BUS_C} \& N_2)) \& ((BUS_L \& N_3) + (\overline{BUS_L} \& N_4)) \& ((BUS_R \& N_5) + (\overline{BUS_R} \& N_6)) \quad (1.1)$$

The input signals BUS_C, BUS_L, and BUS_R are the values of this PE's own bus, its left neighbor's bus, and its right neighbor's bus, respectively. This is shown in Figure 1.6. The signals N_1 to N_6 define the mask. N_1 is set to 1, if the mask is "1" for the center pixel, N_2 is set to 1 if the mask is "0", and both N_1 and N_2 is set to 1, if the mask is "X". The same coding is used for N_3 & N_4 (left neighbor) and N_5 & N_6 (right neighbor).

Name	Mask	Operation
DOT	(010)	Detect isolated 1's
HOLE	(101)	Detect isolated 0's
INV	(X0X)	Invert input data
LEDGE	(01X)	Detect left edges
REDGE	(X10)	Detect right edges
LSHIFT	(XX1)	Shift left
RSHIFT	(1XX)	Shift right

Table 1.4. NLU instructions

The SETB <boundary> instruction controls the NLU input to the two edge PEs. Table 1.5 shows the four possible settings.

Border setting	Argument
Border is always 0	0
Border takes on the inverted value of the edge pixel	1
Border takes on the value of the edge pixel	2
Border is always 1	3

Table 1.5. The set boundary operation

The Point Logic Unit, PLU, has three inputs which are shown in Figure 1.6. They are the output of the NLU, the accumulator (A), and the carry (C). The PLU operates locally within each PE and it performs Boolean operations on its three inputs. Table 1.6 is a list of all PLU instructions. The eight first instructions operate only on the NLU output and the accumulator. Instructions 9 to 11 are used for bit-serial arithmetic and will be described in more detail in section 2.3. Instructions 12 to 14 operate on the NLU and the carry, and instruction 15 computes a two-level logic function which is equivalent to multiplexing the A and NLU content using C as the direction bit. The C-register will be affected only by operations 9, 10, and 11.

No	Name	Operation
1	LD	A := NLU
2	AND	A := A AND NLU
3	OR	A := A OR NLU
4	XOR	A := A XOR NLU
5	LDI	A := $\overline{\text{NLU}}$
6	ANDI	A := A AND $\overline{\text{NLU}}$
7	ORI	A := A OR $\overline{\text{NLU}}$
8	XORI	A := A XOR $\overline{\text{NLU}}$
9	ADD	A := A XOR NLU ; C := A AND NLU
10	ADC	A := C XOR NLU ; C := C AND NLU
11	ADA	A := A XOR NLU ; C := C OR (A AND NLU)
12	ANDC	A := C AND NLU
13	ORC	A := C OR NLU
14	XORC	A := C XOR NLU
15	MUXC	A := (A AND C) OR (NLU AND $\overline{\text{C}}$)

Table 1.6. PLU operations

The operations load (LD), in Table 1.6, and store (ST), in Table 1.7, are used to transfer data between the units within the PE, using the 1-bit bus shown in Figure 1.5. LD transfers data from all units to the accumulator register via the NLU. Unless a mask is appended to the operation the neighbors do not affect the operation. The instruction ST transfers data to the shift registers (Sx) or the memory (Rx) from the other units as well as between Sx and Rx. Table 1.7 shows all legal transfer operations using ST. Note that the shift register and the memory can take data from the accumulator, the carry, the ADC register, and the PD. The shift register can take data from the memory, and both the shift register and the memory can take data from the shift register. This means that we can transfer data within the shift register in one cycle.

```
ST {A,C,Ax,PD},{Sx,Rx}          {Sx,Rx} := {A,C,Ax,PD}
ST Rx,Sx                        Sx      := Rx
ST Sx,{Sx,Rx}                   {Sx,Rx} := Sx
```

Table 1.7. Transfer instructions

The 5 different SET instructions which are shown in Table 1.8. SETR set the sensor row we want to address and SETPD controls the switches shown in Figure 1.3. The SETV instruction sets the U_{ref} voltage in Figure 1.2 and SETAD determines the resolution of the A/D-conversion, which can be set to 1 – 8 bits. SETB sets the border values of the NLU units at the border PEs, shown in Table 1.5.

```
SETR <row_no>               Set row
SETPD <bit pattern>         Set sensor switches
SETV <voltage>              Set voltage
SETAD <precision>           Set ADC resolution
SETB <boundary>             Set boundary bits
```

Table 1.8. Set instructions

Finally, there are miscellaneous instructions, collected in Table 1.9, which control the shifting of the shift register and the initialization of the A/D-conversion. The ROL and ROR perform rotations of the shift register to the left and to the right respectively. The LOAD and SAVE instructions trigger an external unit to take control over the shift register, which then can shift data in and out using the 8-bit port in Figure 1.1. Such data transfer is performed concurrently with the normal MAPP execution. The SRES instruction restores the control over the shift register back to the MAPP2200. This instruction is only used the MAPP2200 is initialized. The shift register data can also be obtained via the 16-bit port shown in Figure 1.1. In this case concurrent MAPP execution is prohibited. The INITAD instruction initiates the A/D–conversion.

```
ROL      Rotate shift register to the left one step
ROR      Rotate shift register to the right one step
SAVE     Store shift register in the external memory
LOAD     Load shift register from the external memory.
SRES     Reset shift register
INITAD   Initialize A/D-conversion
```

Table 1.9. Miscellaneous instructions

To execute a MAPP instruction the host computer or the designated control unit transfers a 16-bit instruction code to the MAPP chip which decodes and executes the instruction

immediately. The status register in Figure 1.1 is accessible from the host via the same 16-bit channel. This 16-bit output is the feed-back from the ALU part of MAPP to its control unit. As such it can be used to influence the subsequent program execution. As shown in Figure 1.8, the 9 least significant bits of the status word contain a COUNT value which corresponds to the number of 1's in the 256-bit accumulator vector. The three most significant bits of the status word are called G_s, A_s, and S_s respectively. The G_s-bit is a Global-OR which is set if the there is at least one 1 in the accumulator. The A_s-bit is a busy-bit which is set while the A/D-conversion is in progress and the S_s-bit is a busy-bit which is set when an external unit has control over the shift register. These three status flags, GOR, ABUSY, and SBUSY are also available at separate pins on the MAPP chip.

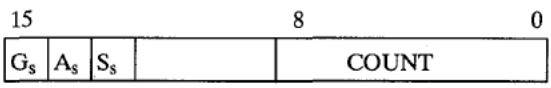


Figure 1.8. The status register

The MAPP-chip occupies two addresses in the address space of the host. One of these, the *low address*, is used for issuing instructions and reading status information. The other, the *high address*, makes it possible to access the shift register from the control unit as shown in Figure 1.1. By reading from this address, we obtain the 8-bit shift register data (S0–S7) from the rightmost PE. A subsequent ROR instruction is automatically performed so that the 8 bits of the next PE is made ready to be accessed. When writing to this address, the 8 least significant bits in the control word is written into the rightmost PE after which a ROL instruction is performed automatically.

1.4 MAPP2200 system configurations

The MAPP chip itself does not contain a control unit. Instead, it is designed to work together with an external controller. In the basic system this controller is a standard microcomputer, where the instruction to the MAPP2200 is furnished via a conventional I/O port. The same port may be used to access the status register in MAPP. This configuration is shown in Figure 1.9. The Ctrl arrow symbolizes address pointer, enable signal, and read/write signals.

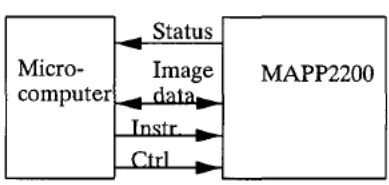


Figure 1.9. Standard configuration

In many applications, e.g. in time sequence analysis, we need to have access to previous images, possibly both in processed and unprocessed versions. For this purpose, the minimal configuration in Figure 1.9 has been augmented in Figure 1.10 with a fast data channel to an external memory. A similar configuration is also needed to produce a video signal simultaneously as we process the image from the MAPP sensor. In this case the external memory could be a dual-ported video ram. To determine which image and what row to access, we use

the same instruction channel as for the MAPP chip and some new instructions codes which are treated as no operation by MAPP.

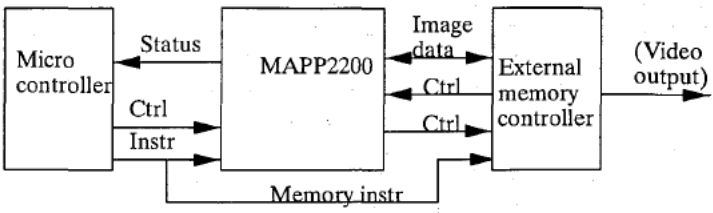


Figure 1.10. External memory configuration

Few, if any microprocessors can deliver instructions as fast as MAPP can execute them (4 MHz). If we want to utilize the MAPP at maximum speed a microprocessor cannot function alone but requires a micro-sequencer between itself and the MAPP-chip as shown in Figure 1.11. This is the configuration used in the sheet-of-light application described in section 6.2 and [17]. In some applications the sequencer can be stand alone. In other applications it has to be supported by a microprocessor. With a microprocessor and a sequencer there are several possibilities to optimize the programming/processing effort. For instance, the normal background mode could be to feed MAPP with a low-speed instruction stream from the microprocessor. However, for highly speed-demanding and time-critical tasks, "the inner loops", the sequencer may take over and feed MAPP with predefined and optimized instruction streams.

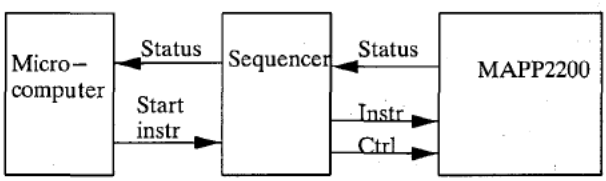


Figure 1.11. A speed-up sequencer configuration

2 Programming MAPP2200

2.1 Environment and syntax

The control unit feeding the MAPP chip with instructions could either be some sort of PC system, a microprocessor, or a dedicated hardware as was explained in the previous section. In this section we will describe a programming environment which assumes a PC system where the MAPP chip receives instructions from and delivers status to a specific port.

Typically, a MAPP application program consists of one outer loop and one inner loop. This program structure is caused by the line-based architecture in MAPP where the same instruction flow (the inner loop) will occur for each line (=row) of the image. The outer loop continues to execute until some termination condition is fulfilled. Within this outer loop we adjust the global parameters, for instance the exposure time, depending on the status acquired in the inner loop. Thus, the program normally has the following structure where the *italic text* are MAPP instructions and the rest belongs to the control structure.

```
REPEAT
        MAPP initialization
        DO 256 TIMES
                MAPP instructions for one line
        END-DO
        Update global parameters
UNTIL Termination condition
```

Accordingly, a MAPP program is divided into two parts. The MAPP2200 instructions are written in a special MAPP2200 assembler language and the control program is written in C. The MAPP code is first run through a macro-expander to be described in section 2.4. The assembler, called MAPPASM, then translates the MAPP instructions into 16-bit words which are stored as integer arrays, one for each procedure. The MAPPASM generates a file which is included in the controlling C program. This C program also includes special functions which at run-time will output the *instructions arrays* to the MAPP chip as fast as possible. Compilation of the C program will generate an executable file. The entire process is illustrated in Figure 2.1.

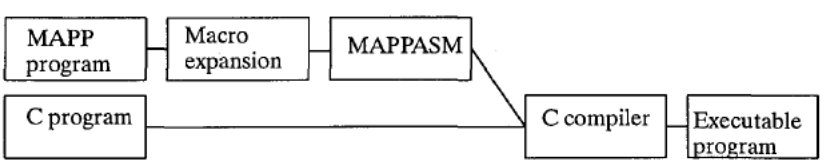


Figure 2.1. The process from source code to an executable program

Apart form the MAPP instructions the MAPPASM accepts three directives: ".FILE", ".PROC", and ".ITER". The ".FILE" directive should be the first statement in a file. It names the output file which is the file containing the arrays of 16-bit instruction words. If there is no name after ".FILE" the file receives the same name as the input file to MAPPASM, except that the file extension is changed from .MAP to .H. At the end of the file there should be a ".ENDF" which terminates MAPPASM. The ".PROC" directive, in combina-

tion with ".ENDP", divides the MAPP instructions into MAPP procedures. The procedure name must begin with "M_". This restriction is to avoid name collisions when included in the control program. The iteration directive ".ITER" will be discussed in the next section. To simplify the programming, the possibility to include macros has been added and will be described in section 2.4. Table 2.1 shows the complete syntax of a MAPP program.

```
MAPP_PROGRAM =
        {INCLUDE_MACRO_FILE}+
        {DEFINE_MACRO}+
        .FILE
          {PROCEDURES}+
        .ENDF

{PROCEDURE} =
        .PROC M_<procedure name>
          {INSTRUCTIONS}+
        .ENDP

{INSTRUCTIONS} = {ITERATION} | MAPPINSTRUCTION

{ITERATION} =
        .ITER ( <no of iterations> <iteration name> )
          {INSTRUCTIONS}
        .ENDI

{INCLUDE_MACRO_FILE} =
        #include <filename>

{DEFINE_MACRO} =
        #define <macro name> {INSTRUCTIONS}+
```

Table 2.1. MAPPASM syntax description

To illustrate MAPPASM, we will use the following program, named TEST.MAP, which outputs a continuous flow of 256x256 images of 4-bit gray levels to an external device, for instance a TV monitor.

```
.FILE test.h
.PROC M_init
   SETAD 5                  ; ADC with 4 bits precision
   SETR 0                   ; Start at the top of the image
.ENDP
.PROC M_run
   ST A4,S4                 ; From A/D-reg to S-reg. Bit 4 to 7
   ST A5,S5
   ST A6,S6
   ST A7,S7
   READPD,+                 ; Read/reset this line, address next line
   INITAD                   ; Initiate the ADC of this line
.ENDP
.ENDF
```

The MAPP program contains one initialization procedure and one procedure which, for every row, will transfer the data from the A/D-register to the shift register, read a new line,

and initialize a new A/D-conversion. When this source code is run through the assembler the instructions are converted into arrays of unsigned integers, i.e. 16 bits codes. The assembler outputs the number of instructions for each procedure.

```
% MAPPASM test
Procedure (number of instructions)
  M_init  (2)
  M_run  (9)
15 input rows read.
```

Each procedure generates its own array and the first element in each array is the length of the array. For example, the output file in the above example would contain the following two arrays of short integers called M_init and M_run.

```
short int M_init[] = {2,0x0385,0x0100 };

short int M_run[]  = {9,0x0a64,0x0ae5,0x0b66,0x0be7,0x0319,
                      0x0326,0x0365,0x031a,0x0319 };
```

This file is then included in the final C program with the *include "test.h"* statement.

```
#include "test.h"
#include "mapp2200.def"
main()
{
  int i;
  while (TRUE) {
    writecode(M_init);
    for (i=0;i<256;i++) {
      writecode(M_run);
      Store(0,i); /* Output device */
    }
  }
}
```

The include file *mapp2200.def* contains the definitions of *writecode*, *Store*, and other MAPP driver routines. The function *writecode* takes the MAPP instruction array and outputs its content to MAPP.

2.2 Iterations

The PEs in MAPP2200 work in a bit-serial fashion. Thus, moving an 8-bit data operand from register positions 0–7 to positions 10–17 has to be done one bit at a time. To reduce trivial and tedious work for the MAPP programmer and, even more importantly, to reduce the possibility of errors in the code, an iteration function has been added to the assembler. The following 16 lines of MAPP code perform the above-mentioned task.

```
LD R0
ST A,R10
LD R1
ST A,R11
LD R2
ST A,R12
LD R3
ST A,R13
LD R4
ST A,R14
LD R5
ST A,R15
LD R6
ST A,R16
LD R7
ST A,R17
```

Using the iteration directive this task can be reduced to

```
.ITER (8 i)
  LD R(0 i)
  ST A,R(10 i)
.ENDI
```

The MAPPASM assembler will unroll this iteration loop and output a final instruction stream identical to the previous one.

The general syntax of the iteration directive is:

```
.ITER({value_1} .. {value_N} {iteration_name})
  {code}
.ENDI
```

This will replicate the instructions in {code}, ({value_1} + .. + {value_N} − 1) times, and for each time the value of {iteration_name} is incremented. To make use of the '.ITER' directive there must be a possibility to modify the statements for each operation. This is done by indexing. By writing the register number and the {iteration_name} within (), for instance R(3 i), the assembler will first use R3, then R4, and so on. Indexing also works for the shift register and the A/D register.For instance

```
.ITER(2 5 -4 i)
  AND A(0 i)
.ENDI
```

will be equivalent to

```
AND A0
AND A1
AND A2
```

Up to 10 iteration loops can be nested which makes it possible to write

```
.ITER(2 i)
  .ITER(2 j)
    OR R(0 i)
    AND R(10 j)
  .ENDI
  XOR R(20 i)
.ENDI
```

which is equivalent to

```
OR R0
AND R10
OR R0
AND R11
XOR R20
OR R1
AND R10
OR R1
AND R11
XOR R21
```

2.3 Bit-serial arithmetic

MAPP performs all arithmetic in a bit-serial fashion. A full-adder operation appears in the instructions ADD, ADC, and ADA in Table 1.6. Here is a MAPP2200 routine for adding two b-bit operands giving a b+1 bits result.

```
LD R0                   ;A := X0
ADD R10                 ;A := A xor Y0 <=> X0 xor Y0
                        ;C := A and Y0 <=> X0 and Y0
ST A,R20                ;Z0 := A <=> sum(X0,Y0)
.ITER (7 i)
   ADC R(1 i)           ;A := Xi xor Ci-1
                        ;Ci := Xi and Ci-1
   ADA R(11 i)          ;A := A xor Yi <=> Yi xor Xi xor Ci-1
                        ;Ci := Ci or A and Yi <=> carry(Xi,Yi,Ci-1)
   ST A,R(21 i)         ;Zi := A <=> sum(Xi,Yi,Ci-1)
.ENDI
ST C,R28                ;Z8 := C8
```

A subtraction in 2-complement representation is performed by setting the carry to 1 at the beginning and inverting the second operand.

```
XORI A                      ; A := 1
ADD R0
ADA (x0x) R10               ; (x0x) = inverted value
ST A,R20
.ITER (7 i)
   ADC R(1 i)
   ADA (x0x) R(11 i)        ; (x0x) = inverted value
   ST A,R(21 i)
.ENDI
ADC R7
ADA (x0x) R17               ; (x0x) = inverted value
ST A,R28                    ;Z8 := sum(X7,Y7,C8)
```

2.4 Macros

In order to help the MAPP2200 programmer, a macro library containing a number of bit-serial arithmetics- and filter functions have been developed. Some of the arithmetic macros are shown in Table 2.2. These macros are expanded by the C pre-processor before the program is run through the assembler. The macros must be defined within the program itself or in a file attached to the MAPP program, see Table 2.1. For example, the move macro, described earlier is defined in the following way:

```
#define MOVE(X,Z,b)        .ITER (b i):\
                             LD R(X i):\
                             ST A,R(Z i):\
                           .ENDI
```

The ":" symbol separates MAPP instructions and the "\" symbol allows the macro to be multi-lined.

Name	Operation			~cycles
ADD_D	Z(b)	:=	(X(b) + Y(b)) / 2	3b
ADD_X	Z(b+1)	:=	X(b) + Y(b)	3b
ADD_DIFF	Z(b1+b2)	:=	X(b1) + Y(b1+b2)	3b
SUB_D	Z2(b)	:=	(X(b) - Y(b)) / 2	3b
SUB_X	Z2(b+1)	:=	X(b) - Y(b)	3b
ABS	Z(b)	:=	X2(b)	4b
INC_A	Z(b)	:=	X(b) + acc	2b
MUX_C	Z(b)	:=	IF C==1 THEN X(b) ELSE Y(b)	3b
MOVE	Z(b)	:=	X(b)	2b
ZERO	Z(b)	:=	0	b
MULT	Z(b1+b2)	:=	X(b1) * Y(b2)	$5b^2$
MULT2	Z2(b1+b2)	:=	X2(b1) * Y2(b2)	$5b^2$

Table 2.2. Some arithmetic macros

2.5 2D filtering

As MAPP is a linear array of processors, it is only possible to process one line of pixels at a time. To implement 2D filters temporary storage for neighboring rows are needed. More specifically, we have to save a number of previous input rows from the sensor in registers to obtain one result row. This type of processing is called Row-parallel pipe-lining, see [1]. Figure 2.2 shows an example where PE number n in the array needs to access a 3x3 neighborhood to compute the result. This is accomplished by storing the two previous rows $m-2$ and $m-1$ along with the incoming row m in registers. After the computation the two last rows are transferred to new locations as shown in Figure 2.3.

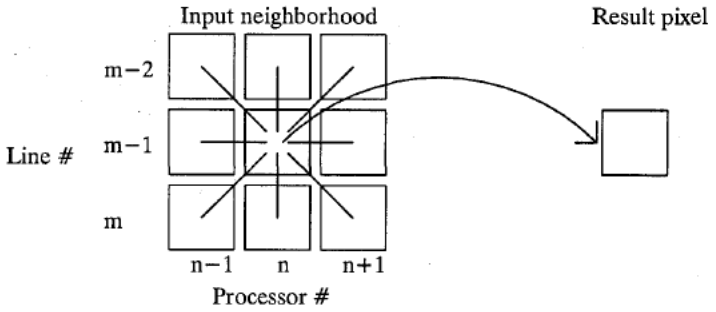


Figure 2.2. Retrieving pixels in a 3*3 neighborhood

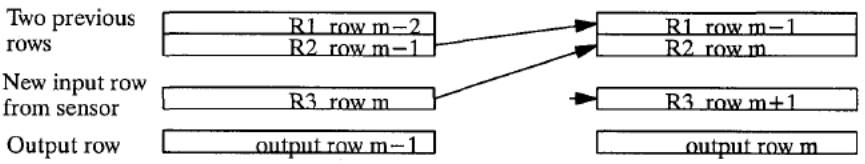


Figure 2.3. Data moves in row parallel pipe-lining

An alternative way to implement (separable) filter functions is to limit the input data to one row but use these data to compute their contributions to three (in the Figure 2.2 case) consecutive output rows. Thus, in this case the two extra registers, besides the one for the input row in Figure 2.3, are used for partial intermediate results which have to be saved for the subsequent input rows.

A number of different filters have been implemented using the row parallel pipe-lining scheme. Some of the filters in the macro library are listed in Table 2.3 and Table 2.4.

Name	Size	Cycles/row	Time/frame (8 bit, 256x256)
Lowpass	3x3	16b	8 ms
Sobel	3x3	22b	11 ms
Laplace	3x3	22b	11 ms
Median	3x3	40b	21 ms

Table 2.3. Gray scale filters

Name	Connectivity	Cycles/row	Time/frame
Expand	4 or 8	9	0.6 ms
Shrink	4 or 8	8	0.5 ms
Thin	8	25	1.6 ms
Open	4 or 8	17	1.1 ms
Close	4 or 8	17	1.1 ms
Median	8	11	0.7 ms

Table 2.4. Binary filters

A typical macro call from the standard filter library in MAPP looks like

```
LOWPASS(IN,PREV,PPREV,OUT,TMP,b)
```

Here, IN is the location of the input register group to the routine. PREV and PPREV are two previous results which must be stored between each run. OUT is the location where the result is put. TMP is the location of a temporary memory. b is the number of bits used in the routine. The macro implements a separable 3x3 lowpass filter which can be executed in 4 additions, see [4].

It is very simple to combine macros which enables the user both to perform different operations and to use larger kernels. For instance in the program

```
LOWPASS(IN,PREV1,PPREV1,TMP1,TMP2)
LAPLACE2(TMP1,PREV2,PPREV2,OUT,TMP2)
```

the output from the macro LOWPASS is passed directly to macro LAPLACE2. Note that we can use the same temporary register in both routines. The effect of this combined filter is a large Laplacian operator as seen from the kernels involved.

```
                          1  4   6  4  1
1 2 1      1   2  1       4  0  -8  0  4
2 4 2  *   2 -12  2   =   6 -8 -28 -8  6
1 2 1      1   2  1       4  0  -8  0  4
                          1  4   6  4  1
```

A compact C program with these specific filter coefficients has the operational structure described in section 2.1. As before, the include file test.h contains two MAPP code sequences, M_init and M_run. The M_init resets the registers used and sets up the ADC etc. The M_run contains the code for the filter and can be written as:

```
.PROC M_run
   .ITER(8 i)          ; Transfer data from ADC to R0-R7
      ST A(0 i),R(0 i)
   .ENDI

   READPD,+            ; Read/reset this line, address next line
   INITAD              ; Initiate the ADC of this line

   LOWPASS(0,8,16,24,32,8)
   LAPLACE2(24,40,48,0,32,8)

   .ITER(8 i)          ; Transfer data from R0-R7 to S0-S7
      ST R(0 i),S(0 i)
   .ENDI
.ENDP
.ENDF
```

3 Exposure control

In conventional image sensors, e.g. CCD-sensors, the exposure time, or rather the integration time of the array, is controlled by a global signal which initiates the exposure. After a certain time, the exposure is interrupted by transferring all the charges in the array in parallel to a set of readout registers. In MAPP we do not initialize the entire image simultaneously but rather one line at a time. Therefore, the exposure time will be the time from the initialization of the line until it is read into the PD register. Suppose that the desired exposure time is T_e, the number of rows is N, and the processing time per row is t_p. We distinguish between two cases $T_e \geq N*t_p$ and $T_e < N*t_p$.

Case I: $T_e \geq N*t_p$

Obviously, an extra delay time T_d,

$$T_d = T_e - Nt_p \tag{3.1}$$

can be inserted in the outer loop of the program. The sequence of events is illustrated in Figure 3.1. The read-out (R) and initialization (I) of each row follow immediately after one another along the diagonals in the diagram.

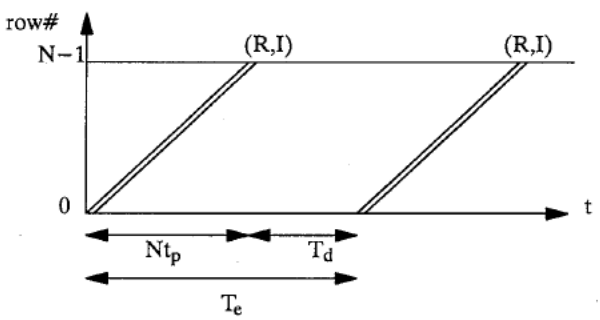


Figure 3.1. Time delays between frame

A pseudo C-program for this operation:

```
while (TRUE) {
  for (row=0;row<=255;row++) {
    SETR row
    READPD
    INITPD
    ..
  }
  WAIT(Td)
}
```

An alternative would be to insert a delay t_d,

$$t_d = \frac{T_d}{N} = \frac{T_e}{N} - t_p \tag{3.2}$$

in the inner loop which would stretch out the exposure-time diagram of Figure 3.1 as shown in Figure 3.2. For simplicity, we have chosen to consider the row number as an (almost) continuous variable in both Figure 3.1 and Figure 3.2. In reality, the discreteness will cause the diagonal lines to be jagged.

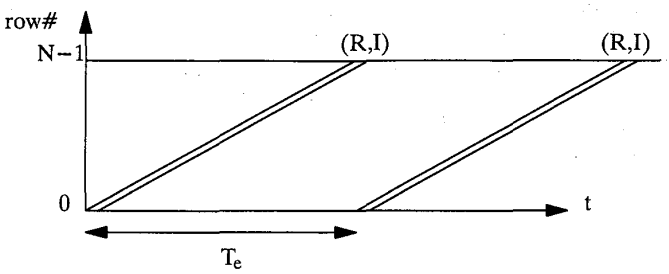


Figure 3.2. Time delays during frames

A pseudo C-program for this operation is

```
while (TRUE) {
  for (row=0;row<=255;row++) {
    SETR row
    READPD
    INITPD
    ..
    WAIT(td)
  }
}
```

Case II: T_e<N*t_p

In this case we have to separate the read-out and the initialization operations in time so that after the read-out of each row, the initialization is postponed for time T_i,

$$T_i = Nt_p - T_e \tag{3.3}$$

The effect is illustrated in Figure 3.3 where we note that the read-out of row n_r should take place at the same time as the initialization of row n_i

$$n_i = n_r - \frac{T_i}{Nt_p}N = n_r - \frac{T_i}{t_p} \tag{3.4}$$

Since T_e is the prime target parameter it is natural to use (3.3) and write n_i in (3.4) as

$$n_i = n_r - N + \frac{T_e}{t_p} = \left(n_r + \frac{T_e}{t_p}\right)_{modulo\ N} \tag{3.5}$$

The parameter t_p might not be known in all cases to the application programmer. In some cases t_p might even be data dependent or vary over time for some reasons. In any case we assume the existence of a real-time clock that facilitates measurements of t_p by running the program with T_d=T_i=0.

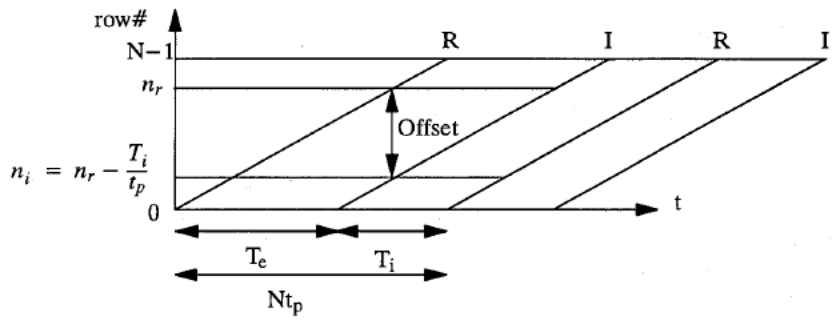


Figure 3.3. Exposure control using a row offset

A pseudo C-program with offset=(T_e/t_p)

```
while (TRUE) {
  for (row=0;row<=255;row++) {
    SETR row
    READPD
    SETR (row + offset) % 256
    INITPD
    ..
  }
}
```

For an arbitrary exposure time T_e, the row offset is set to T_e/t_p or actually, the nearest integer of T_e/t_p. Thus, we experience a maximum error in exposure time ΔT_e when the fraction part of $T_e/t_p = R_{offset}$ is 1/2.

$$\Delta T_e = \frac{R_{offset}\, t_p - [R_{offset}]t_p}{R_{offset}\, t_p} = \frac{R_{offset} - (R_{offset} - \frac{1}{2})}{R_{offset}} = \frac{1}{2R_{offset}} \qquad (3.6)$$

This means that the smaller offset we have, the more difficult it becomes to realize the exact exposure time. For instance, an offset of 100 means that the exposure time can be set with an accuracy of 0.5 %, while an offset of 2 means that the exposure time only can be set with an accuracy of 25%. The remedy to this problem is to insert a delay in the main loop as in Figure 3.2.

4 Adaptive exposure control

4.1 A model for optimal exposure

Adaptive exposure control means varying the exposure with respect to the actual light conditions. One method suitable for MAPP2200 is based on the histogram of the scene. The motivation is that a two-bin histogram comes almost for free during the normal execution and that the histogram approach is robust and insensitive to small regions with high intensities, such as sun reflexes.

If the acquired image is under-exposed it will result in an Inthist diagram (the integrated histogram) which might look like curve (a) in Figure 4.1. On the other hand, if the image is over-exposed, the Inthist might look like curve (b). The idea behind the adaptivity routine is to measure the number of pixels above threshold P_T, and to change the exposure time, iteratively, until a certain number $\hat{n}$ pixels have the pixel value P_T or higher, corresponding to curve (c) in Figure 4.1. Thus, if there are too many pixels which are lower then the P_T value, we will prolong the exposure time. This is the case for curve (a) in Figure 4.1. Correspondingly, we will decrease the exposure time if there are too few pixels below P_T as for curve (b).

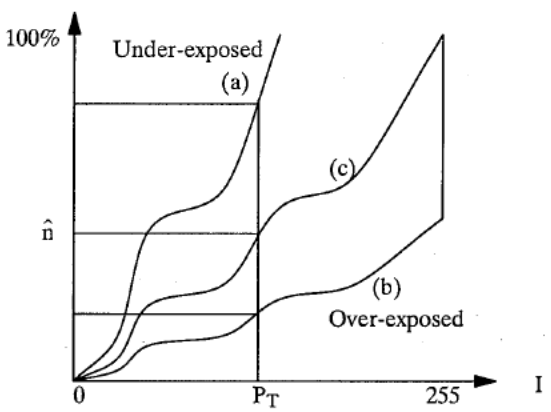


Figure 4.1. Inthists of the same scene but with different exposure times

To create an update rule to change the exposure time T_e when we have discovered that n pixels are below the threshold at level P_T, we need a model of the Inthist diagram. Let us assume that intensity of the pixels are equally distributed between 0 and the highest intensity P_0. This gives us an Inthist as in Figure 4.2 (thick line) and we set the voltage P_T to half of the maximum detectable intensity P_{max} and the ideal number of pixels $\hat{n}$ below the threshold to N/2. One reason to choose this linear Inthist and the point (P_T,N/2) is that in some important respects it matches other typical Inthist curves, for instance the one that is the integrated histogram of a Gaussian histogram and also shown in Figure 4.2.

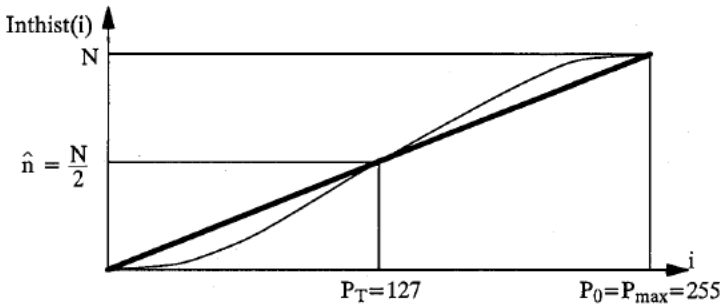


Figure 4.2. The ideal Inthist

The thick line in Figure 4.3 shows the Inthist of a scene exposed during a time T_e which is too short. We notice that the maximum intensity in the scene, P_0, is twice the maximum detectable intensity P_{max} so that half of the pixels in the scene are mapped to P_{max}. The objective is now to change the exposure so that the maximum intensity P_0 is equal to the maximum detectable intensity P_{max}. This is achieved by changing the exposure time T_e according to

$$T_e^{(i+1)} = T_e^{(i)} \frac{P_{max}}{P_0} \tag{4.1}$$

which in this case means that the exposure time will be halved.

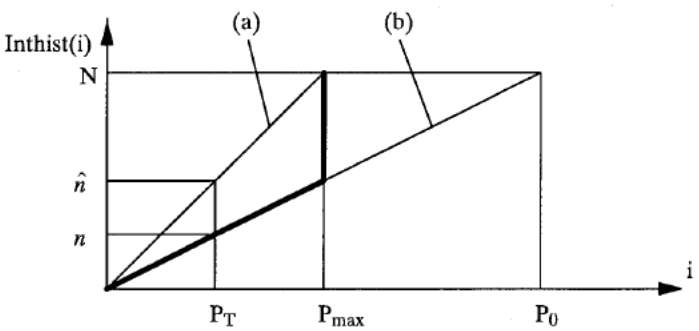


Figure 4.3. An over-exposed Inthist

To compute (4.1) we use the following relation for the desired Inthist curve (a) in Figure 4.3,

$$\frac{\hat{n}}{P_T} = \frac{N}{P_{max}} \tag{4.2}$$

For the measured Inthist curve (b) we have

$$\frac{n}{P_T} = \frac{N}{P_0} \tag{4.3}$$

This gives us the update factor

$$\frac{P_{max}}{P_0} = \frac{n}{\hat{n}} \tag{4.4}$$

We can now rewrite (4.1) in the following way

$$T_e^{(i+1)} = T_e^{(i)}\frac{n}{\hat{n}} = T_e^{(i)}\left(1 + \frac{n-\hat{n}}{\hat{n}}\right) \tag{4.5}$$

and where $\hat{n}$ normally is set to N/2. The formula (4.5) holds for all P_0 values. However, when $P_0<P_T$ the measured entity n will always be equal to N, shown in Figure 4.4. This will limit the update factor in (4.4) and (4.5) to be equal or less than 2. This actually turns out to be very practical as the control loop which uses more than one iteration step will become more stable. Therefore, to obtain the same stability in the previous case when $P_0>P_{max}$ as in Figure 4.3, we change (4.5) to

$$T_e^{(i+1)} = T_e^{(i)}\frac{n}{\hat{n}} = T_e^{(i)}\left(1 + \frac{n-\hat{n}}{2\hat{n}}\right) \tag{4.6}$$

The penalty for the stability of (4.6) compared to the formula (4.5) is a longer convergence time. For instance, given an exposure time error of a factor 100, it will take 7 iteration to converge to the ideal exposure time. However, for a frame rate of 50 Hz the adaptation is still finished in 0.15 s.

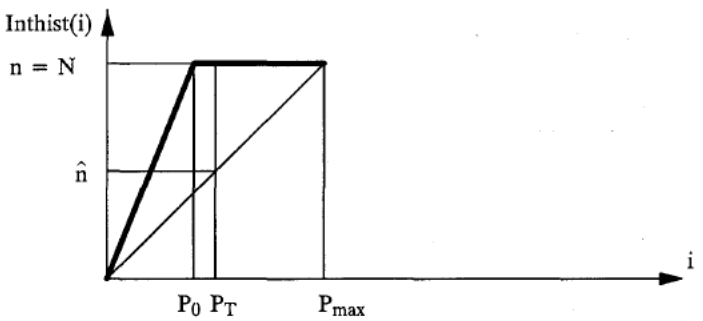


Figure 4.4. An under-exposed Inthist

The update factor as a function of n is shown in Figure 4.5. This curve is y=1+p where p is defined as

$$p = \begin{cases} \dfrac{n-\hat{n}}{\hat{n}} & n \geq \hat{n} \\ \dfrac{n-\hat{n}}{2\hat{n}} & n < \hat{n} \end{cases} \tag{4.7}$$

A more general function, still simple to implement, is

$$T_e^{(i+1)} = (1 + p^{\alpha})T_e^{(i)} \tag{4.8}$$

where α is an integer. Curve (b) in Figure 4.5 shows the update function for an $\alpha>1$. Experiments have shown that a robust update function is achieved for $\alpha=2$.

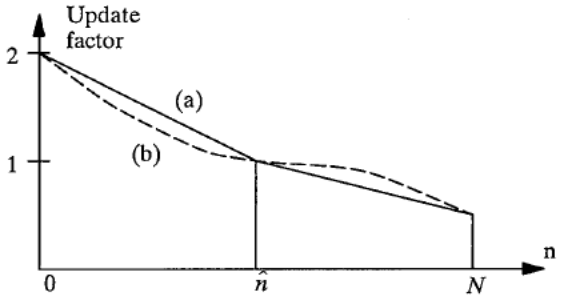


Figure 4.5. The update factor as a function of n

In MAPP2200, the two-bin histogram collection is done by an analog comparison of the pixel values with the global threshold voltage set to Y, followed by reading the COUNT status. This is done row-wise, before the normal A/D-conversion is started. The values for each line are accumulated in the host. After the whole image is read-out, the histogram is complete and we can adjust the exposure time accordingly.

To set a new exposure time, we adjust the row offset within the main loop, described in section 3. The exposure time is the time from the precharge until the photo diode is read out. In the following program, the *update* function performs the calculations according to (4.7) and (4.8).
Program for adaptive exposure control:

```
while (TRUE) {
  sum = 0
  for (row=0;row<=255;row++) {
    SETR row
    READPD
    SETV 128              /* Set i=N/2 */
    LDI PD
    sum += COUNT          /* accumulate Inthist(N/2) */
    SETV 0
    INITAD                /* ADC */
    ..
    SETR (row + offset) % 256
    INITPD
  }
  offset = update(offset,sum);
}
```

Alternatively, we can use one of the true exposure time program, in section 3, by inserting the collection of the Inthist and the update rule into these programs.

Another automatic exposure control method is described in [5]. In this case the pixels are collected into three histogram bins: very white, average, and very black. The exposure is increased or decreased according to whether the image is judged to be too bright or too dark. This is done using a decision diagram shown in Figure 4.6. The update rule is not presented, but it is said to be implemented in hardware using 1000 gates in a CMOS process.

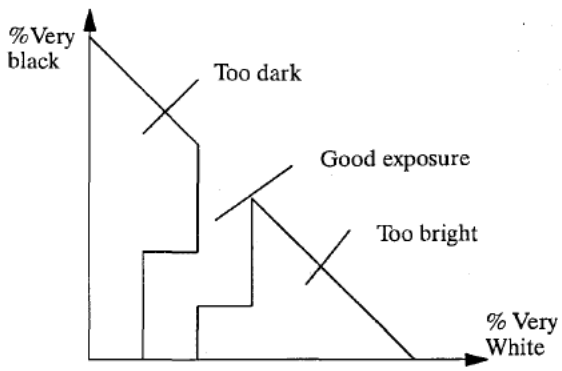


Figure 4.6. Decision diagram, from [5]

The three-bin approach should be quite possible to realize in a MAPP2200 system.

It is also possible to adapt the exposure with respect to the histogram of a limited part of the image as shown in Figure 4.7. Here, the region-of-interest is defined vertically from *first_row* to *last_row*, and horizontally as shown in the mask register R95. By applying the mask before the COUNT read-out, only the pixel within the region-of-interest will contribute to the final summation.
Program for histogramming a predefined area.

```
SETV 128              /* Set i=N/2 */
LDI PD
if (row >= first_row && row <= last_row)
  AND R95             /* AND with mask */
else
  XOR A               /* Set acc to zero */
sum += COUNT          /* collect Inthist */
SETV 0
INITAD
```

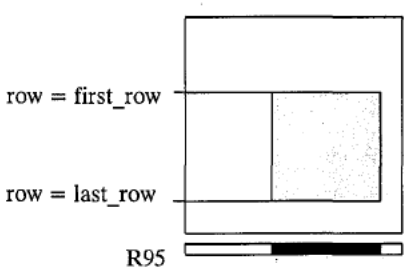


Figure 4.7. Region-of-interest

5 Computation of moments

5.1 2D discrete moments

Moments are features for object recognition and image classification, see for instance [6] and [19]. One of the reasons for their popularity is that they are invariant under translation, scaling, and rotation. In one-dimension the k-th order moment for a function is computed as

$$m_k = \sum_{i=0}^{N-1} i^k f(i) \tag{5.1}$$

For two-dimensions the k-th and l-th order moment is computed as

$$m_{kl} = \sum_{i=0}^{N-1} \sum_{j=0}^{N-1} i^k j^l f(i,j) \tag{5.2}$$

For a binary object such as the one shown in Figure 5.1, the area is the zero order moment m_{00}. The center of gravity $(\bar{x}, \bar{y})$ is given by (5.3) and (5.4). The slope of the principle axis Θ is computed using the zero, the first, and the second order moments (5.5).

$$\bar{x} = \frac{m_{10}}{m_{00}} \tag{5.3}$$

$$\bar{y} = \frac{m_{01}}{m_{00}} \tag{5.4}$$

$$\tan 2\theta = \frac{2(m_{11}m_{00} - m_{10}m_{01})}{m_{00}(m_{20} - m_{02}) + m_{10}^2 - m_{01}^2} \tag{5.5}$$

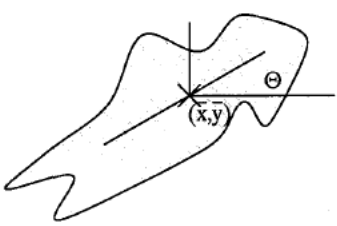


Figure 5.1. Center of gravity and principal axis of an object

The routines for MAPP2200 to be described below, are based on the ideas from Chen [3] and Hatamian [11] which describe moment calculation performed on SIMD arrays.

We observe that the general formula (5.2) is separable so that

$$m_{kl}[f(i,j)] = \sum_{i=0}^{N-1} i^k \sum_{j=0}^{N-1} j^l f(i,j) = \sum_{i=0}^{N-1} i^k m_l(i) \tag{5.6}$$

Here $m_l(i)$ is the one-dimensional l-th order moment of f(i,j) while m_{ke} is the one-dimensional k-th order moment of $m_l(i)$.

Another type of separation consists of expressing higher order moments in terms of lower orders in an iterative formula. Such a formula is translated into a pipe-line (systolic array) in Figure 5.2 which might be helpful to consider together with the derivation below. Note that the input data stream f(i,j) arrives with higher index first. The following expansions are possible from (5.1) and (5.2):

$$m_0(i) = \sum_{j=0}^{N-1} = f(i,0) + \ldots + f(i,m) + \sum_{j=m+1}^{N-1} f(i,j) \tag{5.7}$$

Thus, the first adding operation in Figure 5.2 is

$$\sum_{j=m}^{N-1} f(i,j) = f(i,m) + \sum_{j=m+1}^{N-1} f(i,j) \tag{5.8}$$

For the first order moment we notice that

$$m_1(i) = \sum_{j=0}^{N-1} j\, f(i,j) = \sum_{j=0}^{N-1} f(i,j) + \sum_{j=1}^{N-1} (j-1) f(i,j) \tag{5.9}$$

So that, the second adding operation in Figure 5.2 is performed as

$$\sum_{j=m}^{N-1} (j-m+1) f(i,j) = \sum_{j=m}^{N-1} f(i,j) + \sum_{j=m+1}^{N-1} (j-m) f(i,j) \tag{5.10}$$

To understand the third addition in Figure 5.2, let us first note that

$$j^2 = j + (j-1) + (j-1)^2 \tag{5.11}$$

A first expansion step of the second order moment formula is

$$m_2(i) = \sum_{j=0}^{N-1} j^2 f(i,j) = \sum_{j=0}^{N-1} j\, f(i,j) + \sum_{j=1}^{N-1} (j-1) f(i,j) + \sum_{j=1}^{N-1} (j-1)^2 f(i,j) \tag{5.12}$$

Therefore, the iteration reflected in the third adder node is

$$\sum_{j=m}^{N-1} (j-m+1)^2 f(i,j) = \sum_{j=m}^{N-1} (j-m+1) f(i,j) + \sum_{j=m+1}^{N-1} (j-m) f(i,j) + \sum_{j=m+1} (j-m)^2 f(i,j) \tag{5.13}$$

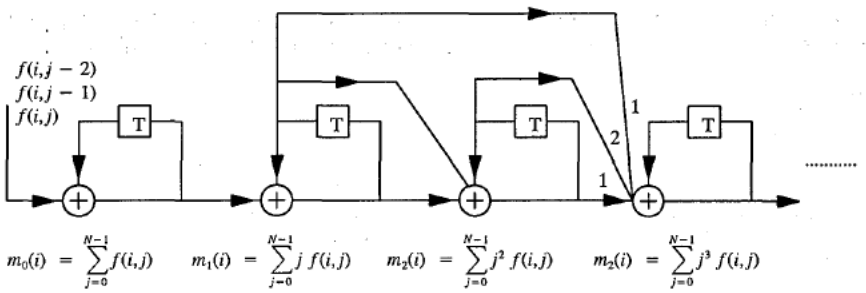


Figure 5.2. Computation flow for the vertical summation for PE number i=x [3]

For higher order moments which correspond to subsequent nodes we notice that (5.11) for arbitrary value of p can be written

$$j^p = j^{p-1} + \sum_{k=0}^{p-1} \binom{p-1}{k} (j-1)^{p-k} \tag{5.14}$$

since

$$j^{p-1} + \sum_{k=0}^{p-1} \binom{p-1}{k} (j-1)^{p-k} = j^{p-1} + (j-1)\{(j-1)+1\}^{p-1} =$$

$$j^{p-1} + (j-1)j^{p-1} = j^p \tag{5.15}$$

Thus, the general recursive formula for the p-th order moment is

$$\sum_{j=m}^{N-1} (j-m+1)^p f(i,j) =$$

$$\sum_{j=m}^{N-1} (j-m+1)^{p-1} f(i,j) + \sum_{k=0}^{p-1} \binom{p-1}{k} \sum_{j=m+1}^{N-1} (j-m)^{p-k} f(i,j) \tag{5.16}$$

5.2 Moment calculation in MAPP2200

Obviously, a MAPP2200 system would be able to perform the computation above for each column. A/D-conversion to eight bits per pixel would then in the worst case accumulate to 16 bits for the 1D 0-th order moments, 23 bits for the 1st order moments, and 31 bits for the 2nd order moments. Together with the temporary variable of the 1st order moments of 23 bits, this totals 93 bits which can be implemented in the 96-register MAPP2200. However, in most applications, the input image is binary which reduces the number of registers needed to 64.

Once the vertical computation in each PE is done, the horizontal computation represents a minor computational load and could possibly be taken care of by the host. However, we will show now that the COUNT feature can be used to great advantage also in this stage. In the MAPP2200 implementation, the vertical computation is performed by a series of addition for each row, which was shown in Figure 5.2.

The MAPP2200 procedure for moment calculation, to be repeated 255 times, using some MAPPLIB function (some of which were described in section 2.4) runs as follows.

```
.PROC M_loop
   READPD,+
   LD PD
   INC_A(0,0,8)                    /* m0(i)  = m0(i) + P      */
   ADD_DIFF(0,8,8,8,8)             /* m1(i)  = m1(i) + m0(i)  */
   ADD(8,24,24,16,8)               /* mtmp   = m1(i) + mtmp   */
   ADD_DIFF(48,24,24,16,8)         /* m2(i)  = m2(i) + mtmp   */
   MOVE(8,48,16)                   /* mtmp   = m1(i)          */
.ENDP
```

Here, $m_0(i)$ is in registers r0–r7, $m_1(i)$ in R8–R23, $m_2(i)$ in R24–R39, and m_{tmp} in R40–R63. This procedure requires 222 cycles per loop, which totals 56832 cycles per frame.

When the vertical scan is completed, each PE contains a contribution to the final moments in form of $m_0(i)$, $m_1(i)$, and $m_2(i)$. We now have to perform a horizontal merge of these data to obtain the moments m_{00}, m_{01}, m_{02}, m_{10}, m_{11}, and m_{20}. The first three of these moments, m_{00}, m_{01}, and m_{02}, can be obtained as a horizontal summation of each PE's values of $m_0(i)$, $m_1(i)$, and $m_2(i)$. We then use the powerful COUNT feature for summing one bit plane at a time weighted with the factor 2^j as showed for $m_0(i)$ in (5.17). Here, $m_0(i,b)$ represents bit number b in PE number i.

$$m_{00} = \sum_{i=0}^{255} m_0(i) = \sum_{i=0}^{255}\sum_{b=0}^{7} m_0(i,b)2^b = \sum_{b=0}^{7} 2^b \sum_{i=0}^{255} m_0(i,b) \tag{5.17}$$

We can interpret this as

$$m_{00} = COUNT_{R0} + COUNT_{R1} * 2 + \ldots + COUNT_{R7} * 128 \tag{5.18}$$

A pseudo-C program to compute m_{00}.

```
m00 = 0;
for (b=0;b<N;b++) {
  LD R(b)
  m00 += COUNT<<b;
}
```

To compute the other three moments, m_{10}, m_{11}, and m_{20}, we have to perform a multiplication with a constant vector before the global summation described above. This vector is a constant ranging from 0 to 255, corresponding to PE 0 to 255. We assume this vector is stored in the memory for each PE. To obtain m_{10}, $m_0(i)$ is multiplied by the constant vector and then globally summed. If we multiply $m_0(i)$ once again by the constant vector we will get m_{20} after summation. Finally, m_{11} is obtained by multiplying $m_1(i)$ by the constant vector. This execution takes approximately

Main loop	57000 cycles
3 multiplications (1000 cycles/mult)	3000 cycles
6 read-out (Host and MAPP operations)	5000 cycles
	65000 cycles

For a 4 MHz system the 65,000 MAPP-instructions corresponds to a maximal frame rate of 60 images/s.

5.3 A MAPP application. Tracking movement in a rat muscle

The instruments shown in Figure 5.3 are used to register the relationships between nerve activation level, muscle contraction velocity and force, and muscle/tendon stress and strain in a wide variety of experiments. The rat is heavily anesthetized, and placed in the box in front of the camera. The skin is removed from one leg and six dots are painted along the muscle. Tests are done in *isometric* mode with a lever arm holding a constant position against the muscle force, and in *isotonic* mode with the lever arm moving to maintain a constant force load to the muscle. By taking images of the six dots at a high frequency (200 frames/s), the relations between the impulse and the reaction of the muscle can be determined. Today, the image are captured by a high speed video camera and the processing is done off-line, after the experiment. In the future, a MAPP2200-camera will do the job in real time on-line.

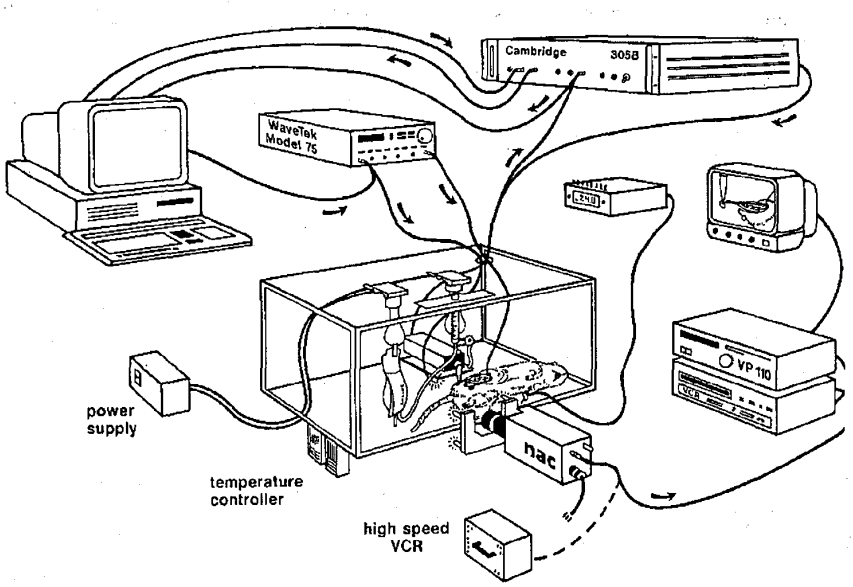


Figure 5.3. The experimental set-up, used today, for the rat muscle registration

Figure 5.4 shows an image from the experiment where we can see the rat, its muscle, and the lever arm which controls the force. The region to be monitored is marked with a small frame in Figure 5.4. In the totally controllable situation it is relatively easy to make the background of the muscle very white and the six dots very black, shown in Figure 5.5. A single predetermined threshold is then sufficient to segment the six dots, see Figure 5.6.

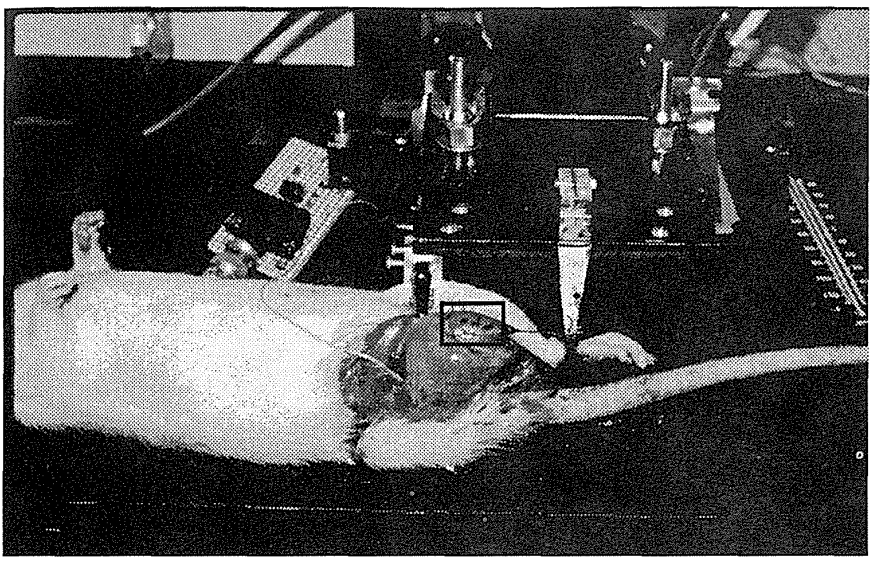


Figure 5.4. The rat and its muscle, and the region-of-interest

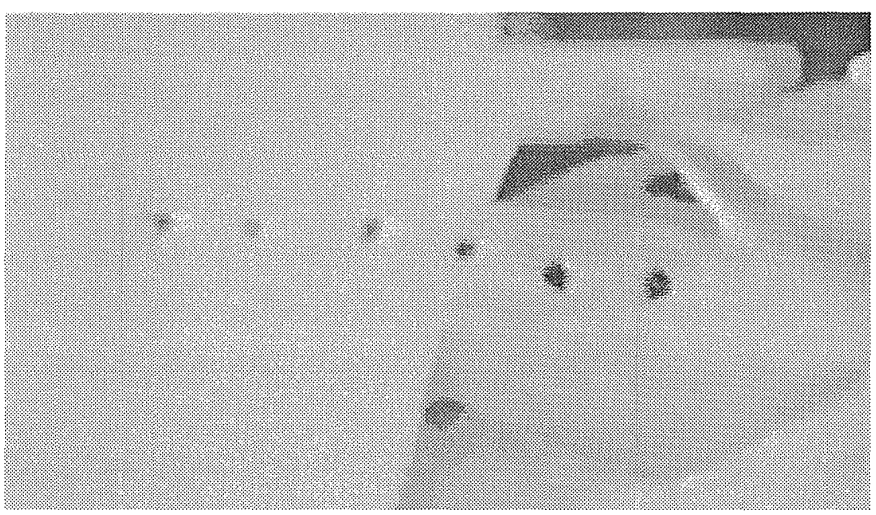


Figure 5.5. Unfiltered image

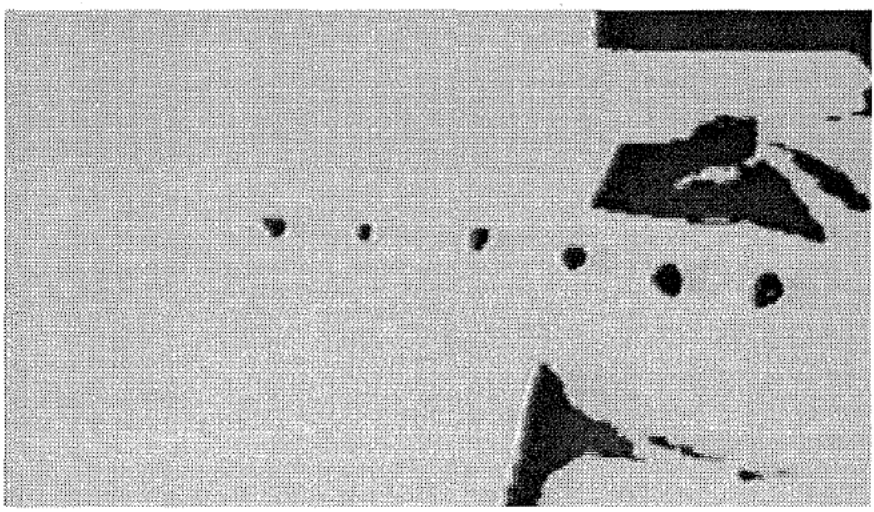


Figure 5.6. Thresholded image

By manipulating the background even more, for instance using a white frame mask, the scene is reduced to contain six 2 mm black markers (dots) of somewhat various size in a 10 cm by 2 cm field. The dots move somewhat irregularly within the scene as illustrated by Figure 5.7.

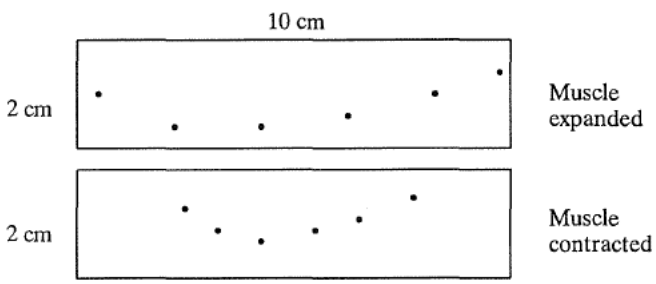


Figure 5.7. The six dots on expanded and contracted muscle

The diameters of the marker objects are in the order of 2 mm. Given the size of the MAPP sensor the resolution will be 0.39 mm/pixel which means that the typical diameter of an object is 5.1 pixels. To cover the area of 10cm x 2 cm, the active part of the sensor is 256 by 52 pixels.

The dot-tracking task was solved as follows. Using the previous moment computation we first retrieve moments of the whole image as if it contained only one object. To compute the moments (in this case only the first order which is the center of gravity) for each of the six objects, we must perform some sort of segmentation. This is possible because the objects are horizontally separated. Two object are considered to be horizontally separated if the vertical projection results in two objects, shown in Figure 5.8 where we have four horizontally separated objects.

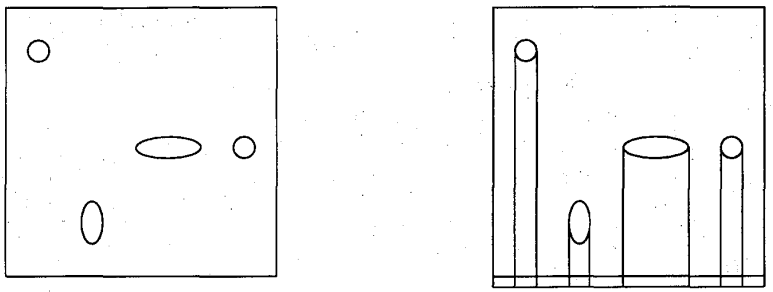


Figure 5.8. Horizontally separated objects

We now recall that the vertical computation of moments in the previous section resulted in three contributions for each column, $m_0(i)$, $m_1(i)$, and $m_2(i)$. To obtain the moments for the whole image we use the same read-out technique as in (5.18). However, the read-out is repeated six times, and by manipulating the read-out bit-vector only those PEs which belong to the same 1D projection, i.e. to one object, is allowed to contribute to each read-out.

In MAPP2200 the vertical summation is carried out in the same way as in the previous section. However, as we only need the position, which corresponds to the 0-th and 1-th order moments given by (5.3) and (5.4), the MAPP2200 program can be reduced to:

```
.PROC M_loop
   READPD,+
   LD PD
   INC_A(0,0,6)                  /* m00x   = m00x + P       */
   ADD_DIFF(8,0,8,8,8)           /* m01x   = m01x + m00x    */
.ENDP
```

The next step is to create the projection of the image to make the segmentation possible. This is done by ORing the bits in $m_0(i)$ for each PE, i.e. to see if $m_0(i)>0$, which gives us the projection vector shown in Figure 5.8(b). We now perform a loop, selecting one object at a time which we place in the C register. Before the horizontal summation, shown in the previous section, we AND $m_{xx}(i,b)$ in (5.17) with the C register. Thus, only PEs belonging to the latest marked object, contributes to this moment.
A program for selecting object in a 1D image:

```
create m00x, m01x etc

for (i=0;i<no_object;i++) {
  LSCAN R64       /* Get leftmost pixel in projection vector */
  MARK R64        /* Mark leftmost object */
  ADD A           /* Place object in Carry reg */

  Compute m00 and m01 in the same way as in the previous section

  XORC R64        /* Remove this object from projection vector */
  ST A,R64
}
```

The number of clock cycles for the vertical computation is in the general case

$$N(5b_1 + 2b_2 + 6) \approx N(7\lfloor^2 \log N\rfloor + 6) \tag{5.19}$$

Here, N denotes the number of lines to process (in our case 52), b_1 denotes the number of bits need in m_{00}, and b_2 denotes the extra number of bits needed in m_{10} and m_{01}. These two numbers can in our case be set to $\lfloor^2 \log N\rfloor = 6$, which gives us 2500 clock cycles using (5.19). We must also add to this number roughly 1000 clock cycles, which is the time it takes to perform the horizontal summation. With an instruction frequency of 4 MHz, this means a frame rate of 1100 frames/s, which is well above the required 200 frames/s.

5.4 Segmentation and moment calculation, the general case

If the objects in the image are located in a way which makes it impossible to separate them with the simple method described in the previous section, we have to perform a more sophisticated segmentation procedure. Such a procedure is needed for the image in Figure 5.9.

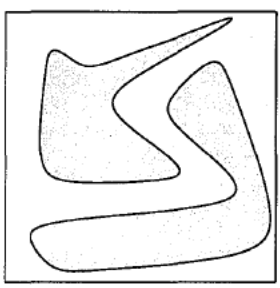


Figure 5.9. Objects which are difficult to separate in a simple way

Compared to the two previous routines the MAPP2200 procedure to be described below is a totally different approach to the segmentation-moment calculation problem. It is based on Run-Length coding where the MAPP2200 is used for fast generation of R-L code while the host computer is responsible for the actual segmentation and moment calculation. Thus, we rely heavily on the host but even so we will show that thanks to the R-L code representation we can achieve frame rates in the order of 20−50 frames/s for typical images.

Given a line segment starting at position P, with a length of L, and at row j=R, its contribution to moments m_{kl} is

$$m_{kl} := m_{kl} + R^l \sum_{i=P}^{P+L-1} i^k \tag{5.20}$$

Knowing the summation formulas for

$$\sum_{i=0}^{n} i = \frac{n^2 + n}{2} \tag{5.21}$$

$$\sum_{i=0}^{n} i^2 = \frac{2n^3 + 3n^2 + n}{6} \tag{5.22}$$

the 6 moment contributions to m_{00}, m_{01}, m_{02}, m_{10}, m_{11}, and m_{20} can be computed as

$$\begin{aligned} m_{00} &:= m_{00} + L \\ m_{01} &:= m_{01} + LR \\ m_{02} &:= m_{02} + LR^2 \\ m_{10} &:= m_{10} + L\frac{2P + L - 1}{2} \\ m_{20} &:= m_{20} + \frac{L^3}{3} + L^2\frac{6P - 3}{6} + L\frac{6P^2 - 6P - 1}{6} \\ m_{11} &:= m_{11} + LR\frac{2P + L - 1}{2} \end{aligned} \tag{5.23}$$

The segmentation is performed by the host and is based on three lists of data, see Table 5.1, 5.2, and 5.3. The first list, Table 5.1, contains all the object, so far encountered. Each object is represented by a label and for each label we have the present values of moments, m_{00}, m_{01}, m_{02}, etc. The other two lists, Tables 5.2 and 5.3, contains the R-L codes for the previous and the new line and a label associated with each segment. In the previous row list, the labels are known for all segments. Thus, the goal for the host computer is to label the objects in the new line.

Label	m_{00}	m_{01}	..	m_{20}
1	xx	xx		xx
2	xx			
3	xx			
..				
23	xx			xx

Table 5.1. Example of label and moment list

RL start	RL length	Label
10	15	2
50	15	7
160	55	11

Table 5.2. Example of previous line list

RL start	RL length	Label
15	40	?
60	15	?
100	24	?
150	22	?
200	23	?

Table 5.3. Example of new line list

The image is read-out line-wise and for each line, we first generate the R-L codes for the new line. We then have the R-L codes for both the previous and the new line, shown in Fig-

ure 5.10 and in more detail in Figure 5.11. We now process one line segment at a time from the new line list and work from left to right. The following rules should be obeyed.

- If we encounter an object which has no support in the previous row, for instance (f) in Figure 5.11, we create a new entry in the label list and compute this segment's contributions to the moments according to (5.23).
- If the object has support from the previous row, for instance (g), the contributions to the moments from this segment are added to the supporting label's moments.
- If the object has support from two or more segments, as in case (d), we choose the label belonging to the leftmost support.
- If the supporting objects have different labels, as in case (d), the labels of the other objects, in this case (b), are changed to the leftmost label, in this case (a), and the moment contribution of (b) are merged into (a).

During the processing of a line, the labels in the previous line might change which is the case for object (b) in Figure 5.11. But this can be achieved through horizontal propagation of the label, which means that we only have to do one vertical scan regardless of the structure of the image. A final unification of the different labels in the same object can be made in the host if necessary. It is not needed if the segmented moments are the only and final result to be extracted.

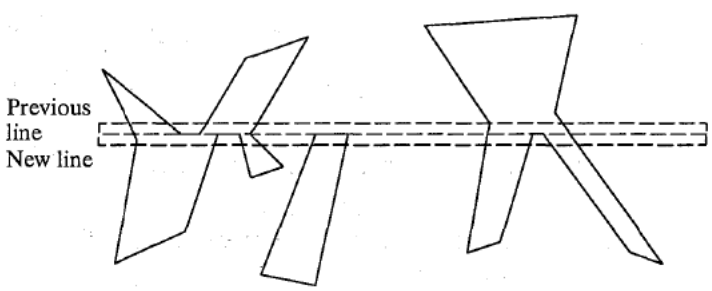


Figure 5.10. Segmentation process at a certain time

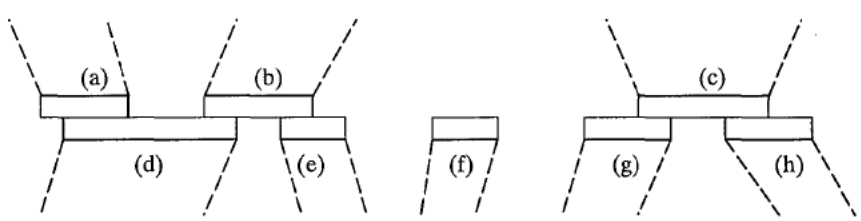


Figure 5.11. Enlargement of the interesting area in Figure 5.10

The execution time for this algorithm is very data dependent. If there are many objects on one line, there will be more tests of the connectivity and more moment calculations. However, tests have shown that for a segmentation program written in C which run on a PC system (which can feed the MAPP2200 chip with instructions at a frequency of 0.5 MHz), we can obtain a frame rate of up to 15 segmented images per second, with six moments for each

object. This hold for a "normal" image, for instance like Figure 5.10. If we have more "difficult" objects, the frame rate may go down to 3–5 images/s. Using a "smart camera" (dedicated controller for the MAPP2200 chip) and an assembler program, we can expect to achieve frame rates 20 to 50 image/s.

The R-L codes from one line are obtained very easily in MAPP using the GLU instructions, shown in Figure 5.12.
Program for generating R-L codes:

```
ledge r0          ; How many objects?
n = COUNT
for (i=0;i<n;i++) {
    lscan r0      ; Select leftmost pixel in leftmost object
    rfill a       ;
    p = COUNT     ; Start position
    mark r0       ; Mark leftmost object
    l = COUNT     ; Length of object
    xor r0        ; Remove this object in line
    st a,r0       ; Store line in r0
}
```

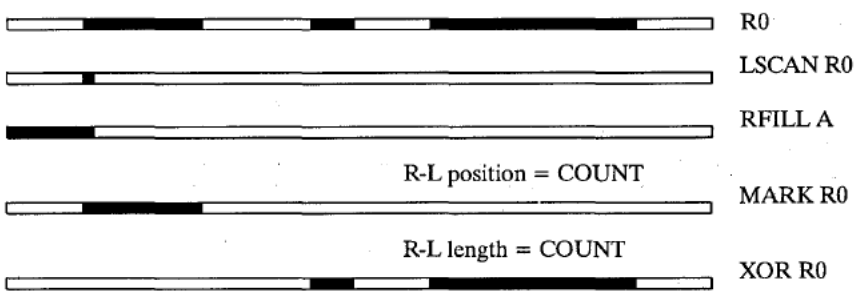


Figure 5.12. R-L coding in MAPP

6 Sheet-of-light range imaging

6.1 General

Range imaging is important in many machine vision applications. If the scene is moving (the conveyer belt case) or if the camera system is moving over the scene (the camera mounted on a robot arm or on a vehicle) a rather attractive range method based on sheet-of-light can be used. In this method, by means of rather simple optics, a laser pencil beam is diverged into a sheet-of-light, a curtain that cuts through the 3D scene as shown in Figure 6.1. It should be noted that the only additional feature needed to image a static scene with a static camera is a rotating mirror that makes the sheet-of-light sweep the scene over time.

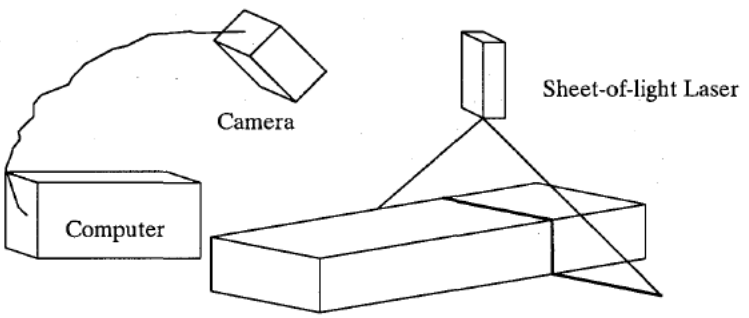


Figure 6.1. A Sheet-of-light camera set-up

The other major optical component in the system is an ordinary camera with an image sensor. the reflected light from the sheet that cuts the scene will be mapped onto the sensor as a jagged line as shown in Figure 6.2. The topography, or rather the depth, will correspond to the vertical position of the points on the line in Figure 6.2(b). This relation is not difficult to express in mathematical terms once the geometrical and optical parameters are known for the laser illuminator and the camera sensor. However, to do this is outside the scope of our treatise.

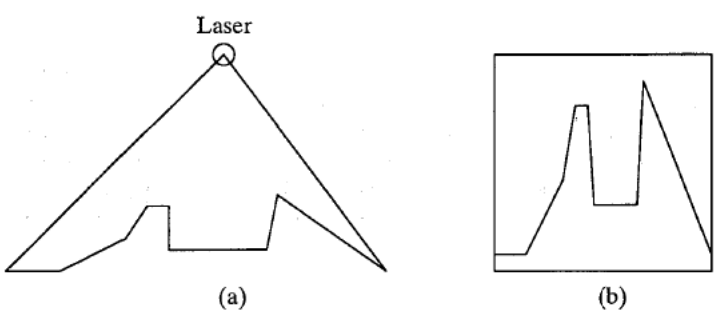


Figure 6.2. A topological view and a corresponding sensor registration

The image processing problem in a sheet-of-light camera system is to establish the true vertical position of the line in Figure 6.2. This should be done in the presence of several non-ideal conditions like low reflectance of the scene, missing parts of the line due to occlusion,

thickness in the sheet-of-light which causes a multi-pixel wide line, noise, false reflections, and ambient light, just to mention a few. Even so, if the position of the points on the line can be established with a reasonable accuracy, the 2D sensor data can be reduced to a 1D vector. In this respect, the image sensor in a sheet-of-light camera is an ideal target for a smart sensor approach which reduces the 2D image input to a refined 1D feature vector.

Sheet-of-light camera design is treated by many authors. For a recent survey, see Johannesson [16].

6.2 Implementation on MAPP2200

MAPP2200 can be used in two ways as a sensor for sheet-of-light. One of these is illustrated in Figure 6.3(a) where the orientation of the jagged line is the same as in Figure 6.2(b). For the input pattern in Figure 6.3(a) the task would be to read out the sensor row by row, and in each PE-position in some way take a note of on which row address the line appears. Another way illustrated in Figure 6.3(b) is to turn the camera 90° so that the jagged line appears vertically on the chip. Here, the task would be to read out row by row and for each row detect the PE position where the line segment appears.

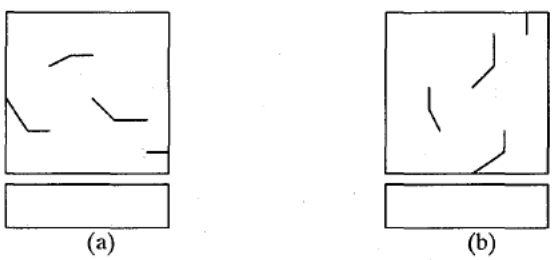


Figure 6.3. Horizontal a vertical placement of the laser line

An range data feature, not mentioned so far, is the *intensity* of the light at the jagged line in the sensor. In some applications, such intensity retrieved in parallel to the range is of great value. In the programs to be presented below we attempt to capture both range (position) and intensity.

For the horizontal case, shown in Figure 6.3(a), we first allocate 8-bits registers to the intensity and the position, both of which are set to zero. Then, for each row, the difference between the maximum intensity so far and the new intensity is computed. If the new intensity is larger than the current maximum value, we adopt the new maximum intensity and record the position of the maximum. This is described in Table 6.1 together with the time consumption in cycles/row. We see that it takes 9b*256 =2304b cycles which equals 18,400 cycles for b=8.

Operation	#cycles
Load P_{in} from ADC	b
Compute P_{in} - P_{max}	2b
Multiplex P_{max} := max(P_{in}, P_{max})	3b
Multiplex Pos_{max} := Current row if P_{in} > P_{max}	3b
	9b

Table 6.1. Horizontal computation (a)

For each row, we have to perform an A/D-conversion which takes $1/2*2^b$ cycles (an ADC cycle is half as long as a normal processor cycle). Full time overlap between A/D-conversion and processing requires

$$2^b \leq 2 * 9b = 18b \tag{6.1}$$

which holds for b≈7. This means that we can use 7-bit precision from the ADC when we apply the operations described in Table 6.1. If we want 8-bit precision the processors will be idle half of the time for each row. However, good use could possible be made of this extra time to achieve a better (sub pixel) resolution of the final maximum.

After the whole frame has been processed, the range and intensity data are shifted out with the shift register.

For the case illustrated in Figure 6.3(b) the sensor array is read one row at the time and the position of the maximum is determined by a successive approximation using all row data simultaneously. Initially, at time t=0, an approximation register in the sequencer or the host, the reference voltage (I1,I2,..,I8), is set to $10000000_2=128_{10}$. If there is at least one position with a voltage above 128, the Global-OR will go high. If the Global-OR is high, the next reference voltage will be set to $11000000_2=196_{10}$, otherwise the voltage will be set to $01000000_2=64_{10}$. Thus, for each step we perform $I_t=G_t$ and $I_{t+1}:=1$. Table 6.2 shows these steps for each t, where Gi represents the value of Global-OR at time i−1. The full intensity value (G1,G2,G3,G4,G5,G6,G7,G8) is found as the I vector after t=8.

	Intensity/Reference voltage bit #							
t	I1	I2	I3	I4	I5	I6	I7	I8
0	1	0	0	0	0	0	0	0
1	G1	1	0	0	0	0	0	0
2	G1	G2	1	0	0	0	0	0
3	G1	G2	G3	1	0	0	0	0
4	G1	G2	G3	G4	1	0	0	0
5	G1	G2	G3	G4	G5	1	0	0
6	G1	G2	G3	G4	G5	G6	1	0
7	G1	G2	G3	G4	G5	G6	G7	1
8	G1	G2	G3	G4	G5	G6	G7	G8

Table 6.2. The value of the threshold voltage at different times

This procedure takes 3b cycles for a b bit digital output if we augment the standard Global-OR as shown below. When we have found the brightest spot, we perform an LFILL

operation on the thresholded line data which gives us the x-position of the brightest pixel. This operation also serves as a tie-breaker in case that two or more pixels have the same maximum intensity so that the leftmost position is chosen. The following program shows the complete procedure.

```
for (i=0;i<255;i++) {
     SETR i
     READPD,+

     volt:=128;
     SETV volt
     LDI PD

     for (j=7;j>=1;j--) {          /* From Figure 6.2 */
          volt =    volt +         /* Old value + */
                    (GOR-1)*2^j +  /* remove if not G-OR */
                    2^(j-1);       /* add next level */
          SETV volt
          LDI PD
     }

     volt = volt + (GOR-1);        /* Remove LSB if not GOR*/
     SETV volt
     LFILL PD
     pos = COUNT;
     intensity = volt;
}
```

Some selected MAPP2200 chips have been found to work well up to 8 MHz, which is twice the normal frequency. Given such a selected chip and a dedicated sequencer implemented as in Figure 6.4, it seems possible to execute up to 700 frames a second. Since each frame produces 256 positions and each position produces one range value (rangel) this MAPP system produces 180,000 rangels/s.

The main point of Figure 6.4 is to show how a fast sequencer and the host can share the responsibility to generate instructions to MAPP. It also emphasizes that fast feed-back is wanted for the Global-OR signal in the above program.

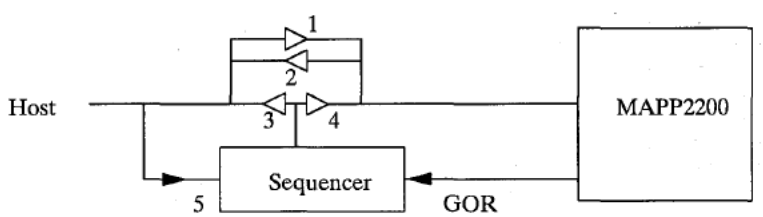


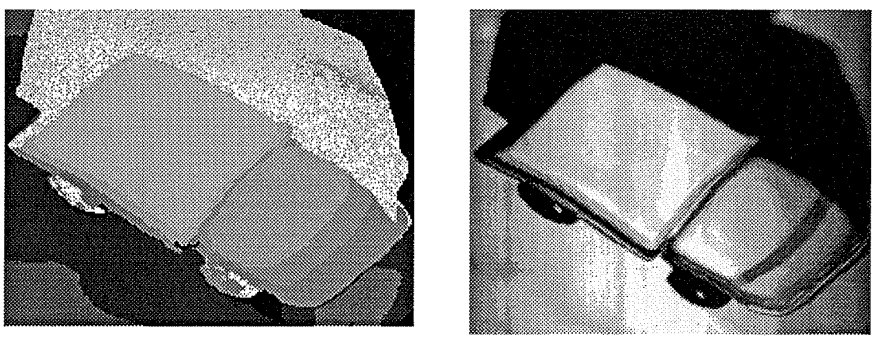
Figure 6.4. A fast transparent control for sheet-of-light system

During normal execution, switches 1 and 2, in Figure 6.4, are used for write to and read from MAPP. When a special trigger instruction from the host is detected by the sequencer at 5, switch 4 is used for issuing instructions from the sequencer. When all bit planes shown in Table 6.2 have been processed, the intensity is sent from the sequencer to the host via switch

3 and the range, i.e. the position of the maximum intensity, is sent to the host computer from the MAPP chip via switch 2.

A number of scenes have been processed using method (b) in Figure 6.3. They are presented in much greater detail in [17] along with suggested post-processing operations to reduce noise in the images.

The scene in the example shown in Figure 6.5 contains a toy car which has been moved through the sheet-of-light. Here, the range mapped onto grayscale is shown in Figure 6.5(a) and the corresponding intensity image in Figure 6.5(b). A combination c of range r and intensity I pixel by pixel is shown in Figure 6.6. The mapping from 6.5 to 6.6 is $c(x,y):=I[x,y-r(x,y)]$.



(a) (b)

Figure 6.5. A range image and an intensity image

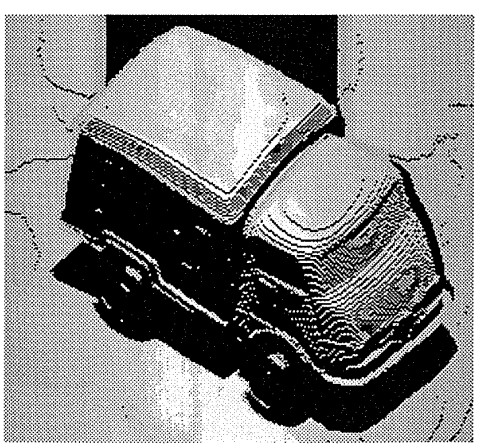


Figure 6.6. A 3D plot using both range and intensity data

6.3 Filtering during computation

Ideally, a cross-section of the image that falls upon the sensor in Figure 6.3(b) contains a single sharp pulse. In practice it may look like Figure 6.7. Even so, in this case the intensity peak where the sheet-of-light hits the object stands out as a well-defined maximum.

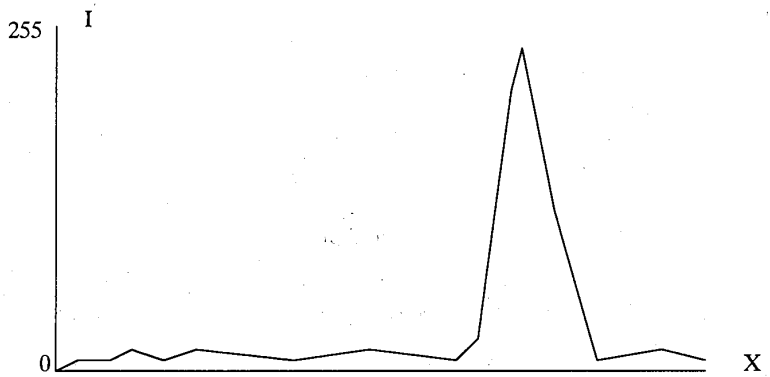


Figure 6.7. The intensity values along the array at a certain time

A more difficult situation is shown in Figure 6.8 which could appear in an outdoor application where uncontrolled reflexes are abundant. The thin high peak in Figure 6.8 is likely to be such a false unwanted response.

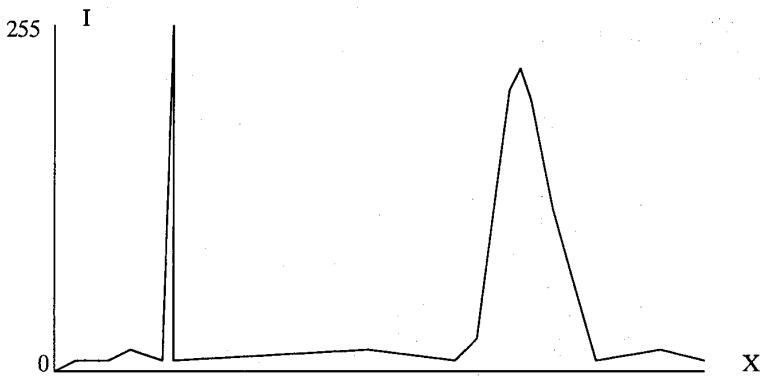


Figure 6.8. A bright spot which might affect the result

Various post-processing techniques to combat this problem have been described, see for instance, Johannesson [17]. Another possibility is to filter the signal (i.e. row data) *before* the successive approximation when the image data is in analog form.

The method we suggest is based on grayscale morphology [22] and it has also a lot in common with Part II of this thesis, Near-Sensor Image processing. We notice that both the thin and the thick peak in Figure 6.8 grows wider, or at least not thinner towards their bases. This is self-evident since the picture of these peaks represents a part of the function p(x,y). Nevertheless, this feature is vital to the success of our method.

The inner loop of the program for the image in Figure 6.3(b) uses successive approximation to find the maximum value. In the case of Figure 6.8 it would pick the wrong one. However, the Global-OR that signals the status in the linear processor array to the sequencer could very well be applied to a processed result of row data instead of the thresholded row sensor data. For instance, using simple morphology operations we should be able to discriminate between various peak widths as they appear in the thresholded row data. We can filter out binary 1D objects in these data which we determine unsuitable to our max-finding goal and we can suppress their influence on the Global-OR result. It can be seen that this filter principle must be consistent for all gray-levels along the row. Hence, the above condition on peaks with widening bases.

The program below is the previous successive approximation augmented with median filtering of the sensor data before Global-OR. A similar augmentation is possible to erode to a single point before the final position read-out. A multi-pixel wide peak will then be registered in position by its mid-point.

```
for (j=7;j>=1;j--) {          /* From Figure 6.2 */
     volt =    volt +         /* Old value + */
               (GOR-1)*2^j +  /* remove if not G-OR */
               2^(j-1);       /* add next level */
     SETV volt
     ld (00x) pd
     and (0x0) pd
     and (x00) pd
}
```

The above way of filtering the 1D image is identical to grayscale morphology filtering. Table 6.3 shows how to implement various other grayscale morphology operations on the image line by adding one or a few instructions in each cycle.

```
Filter name      Instruction sequence
---------------------------------------------------
Normal           ld  (x0x) pd  ( = ldi pd )
Erosion          ld  (000) pd
Dilation         ldi (111) pd
Closing          ldi (111) pd : ld  (111) a
Opening          ld  (000) pd : ldi (000) a
Median           ld  (00x) pd : or (0x0) pd : or (x00) pd
```

Table 6.3. Some grayscale morphology filters and their corresponding instructions

7 Conclusions

The MAPP2200 has been designed to be a general purpose smart image sensor. This means that it should efficient for various types of image processing operations. Thus, these operations must be easy to map onto the architecture, to implement and program in the system, and to verify. We believe that the above six sections have shown that MAPP2200 in these respects is a general purpose system.

As was mentioned in section 1 the MAPP2200 inherited much of its basic operations from LAPP1100 but important novelties were brought in from PASIC, for instance, programmable resolution in the ADC, the shift register, and some of the new ALU-operations. The most obvious difference to LAPP1100 is of course the two-dimensional sensor. As we will show in part III in this thesis this will create a more principle difference between the one-dimensional LAPP1100 and the two-dimensional MAPP2200. LAPP1100 is able to implement what will be defined as Near-Sensor Image Processing (NSIP) where the sensor data are interrogated **and processed** repeatedly during one ongoing exposure. In MAPP2200, this is not possible because of the much higher volume of sensor data which has to be handled by just a linear array of processors. More importantly, the NSIP idea demands that all sensor data are interrogated at the same time. Finally, the present column amplifier in MAPP2200 prohibits non-destructive read-out from the individual sensor. However, it is possible to perform 1D NSIP in MAPP2200.

For an industrial product, like MAPP2200, it is fundamental that the program development environment is simple to use, yet powerful to utilize the computational capacity of the product. For this purpose, a special assembler has been developed which makes it possible for the user to make full use of the the system. To reduce the complexity of the programming, a special function library has been developed. It contains both bit-serial logic and arithmetic operations as well as two-dimensional binary and grayscale filters.

The exposure control in MAPP2200 is different to conventional CCD-cameras. In MAPP2200 we may use both a timer function and/or a row offset control. The present adaptive exposure scheme is based on a two-bin histogram of the image. This approach was chosen because of its robustness and the fact that the collection of the histogram was done with little extra execution time. For instance, the programmability of MAPP2200 makes it possible to use a limited window (region-of-interest) for the exposure control.

For both industrial and robot vision applications, moment calculations have proved to be a powerful tool. We implemented a computation scheme targeted for SIMD processors on MAPP2200 and made some modification to utilize the global features of the chip. For single objects we achieved very high frame rates. With a smaller modification of the computation, we showed that we could use the same scheme in a medical experiment where the speed requirement was 200 frames/s. The goal was to monitor the position of six blobs and the MAPP2200 implementation achieved frame rates of up to 1000 frames/s. In the general case, when we need full segmentation and moment calculation, we can still use MAPP2200 in a powerful way. In this case the MAPP2200 chip delivers run-length codes of the image to simplify the segmentation process and the moment calculation, so that they can be performed by the host computer.

Finally, we described a MAPP implementation of a sheet-of-light range camera. In this application we made use of the global operations to achieve the high frame rates. We also showed how advanced filtering during computation can be included with little extra computational effort.

The computational strength of MAPP2200 comes forward in applications where the output data from the chip are highly reduced in quantity compared to input data. The moment calculations is a good example in case. Here, MAPP2200 delivers moments or contributions to moments which are then converted in to size, position, and slope in the host. Another example is the sheet-of-light range camera application where MAPP2200 together with a sequencer delivers range and intensity but only one pair per row.

In many cases, the performance of MAPP2200 is very competitive compared to conventional image processing systems. However, in certain types of image processing applications the MAPP2200 looses its edge. Among the trouble-spots are large convolution kernels with arbitrary coefficients, histogramming, and floating point operations (FFT etc.). In these cases a MAPP2200 system would be slower than many other image processing systems, and some functions, for instance FFT, would be more or less impossible to realize in MAPP2200.

An important feature of MAPP2200 is compactness and small size which makes it very suitable for air-borne and space-borne applications and in surveillance cameras. An unbeatable feature of MAPP2200.

8 References

[1] Åström A., *A Smart Image Sensor. Evaluation and Description of PASIC.*, Lic Thesis, Linköpings University 1990.

[2] Åström A., Forchheimer R., *MAPP2200 Smart Vision Sensor. Programmability and Adaptivity.*, Proc MVA92, Tokyo, Japan, 1992.

[3] Chen K., *Fast Algorithm for the Calculation of Image Moments in a Linear Processor Array*, Internal Report, Linköping, Sweden, 1988.

[4] Danielsson P.E., *Generalized and Separable Sobel Operator*, Internal Report LiTH–ISY–I–0975, Linköping, Sweden, 1989.

[5] Denyer P. B., Renshaw D., Gouyu W., Lu M., Anderson S., *On–chip CMOS Sensor For VLSI Imaging Systems*, Proc of VLSI91, 1991.

[6] Dudani S. A., Breeding K. J., McGhee R.B., *Aircraft identification by moment invariants*, IEEE trans Computer, vol C–26, pp 39–45, Jan 1977.

[7] Forchheimer R., Åström A., *Near–Sensor Image Processing. A new approach to low level image processing*, In Proc of ICIP, Singapore 1992

[8] Forchheimer R., Åström A., *Near-Sensor Image Processing. A New Paradigm*, accepted for publication in IEEE trans Image Processing.

[9] Forchheimer R., Ödmark A., *Single chip linear array processor,* in *Applications of digital image processing,* SPIE, Vol. 397, (1983).

[10] Forchheimer R., Ingelhag P., Jansson C., *MAPP2200, a second generation smart optical sensor*, SPIE Vol 1659, 1992.

[11] Hatamian M., *A Real–time Two–Dimensional Moment Generating Algorithm and Its Single Chip Implementation*, IEEE trans ASSP, Vol 3, June 1986.

[12] Horii A., *Depth from Defocusing*, Internal report, Stockholm University, Sweden.

[13] Krotkov E. P., *Active Computer Vision by Cooperative Focus and Stereo*, Springer–Verlag, 1989.

[14] Ingelhag P., *A Contribution to High Performance CMOS Circuit Techniques*, Lic Thesis, Linköpings University 1992.

[15] Jansson C., *Image Sensing Using Standard CMOS Techniques*, Lic Thesis, Linköpings University 1993.

[16] Johanneson M., *Sheet-of-light Range Imaging*, Lic Thesis, Linköpings University 1993.

[17] Johanneson M., Åström A, *Sheet–of–light Range Imaging with MAPP2200*, Proc of SCIA8–93, Tromsø, Norway.

[18] Kittler J., Illingworth J, *Minimum Error Thresholding*, Pattern Recognition, Vol 19, No 1 , 1986.

[19] Lucas P., *Moment Techniques in Picture Analysis*, Proc IEEE conf Computer Vision Pattern Recognition, pp 178–187, June 1983.

[20] MAPP2200 Technical Description, IVP Integrated Vision Products AB, Linköping, Sweden, September 1991.

[21] Renshaw D., Denyer P. B., Lu M., Wang G., *A Single–chip Video Camera with On–chip Automatic Exposure Control*, Proc ISIC–91, 1991.

[22] Sternberg S. R., *Grayscale Morphology*, Computer Vision, Graphics, and Image Processing, Vol 35, 1986.

Part III NSIP

1 Introduction

Most image processing applications are burdened with high volumes of data that have to be funnelled though the image processing system. This is especially true in the early part of the system where individual pixels are represented by full gray scale or color. Therefore, all efforts which lead to a reduction of the pictorial data without too much loss of information are of interest. The processing task column of Figure 1.1 shows a typical image processing sequence consisting of image acquisition, noise reduction and filtering, segmentation, feature extraction, and classification. The next two columns show examples of operations for each task, and the amount of input/output data for each operation. Here, N stands for the image size and b for the number of bits in the input image. This type of image processing sequence is very common in robot vision applications as well as in industrial inspection tasks.

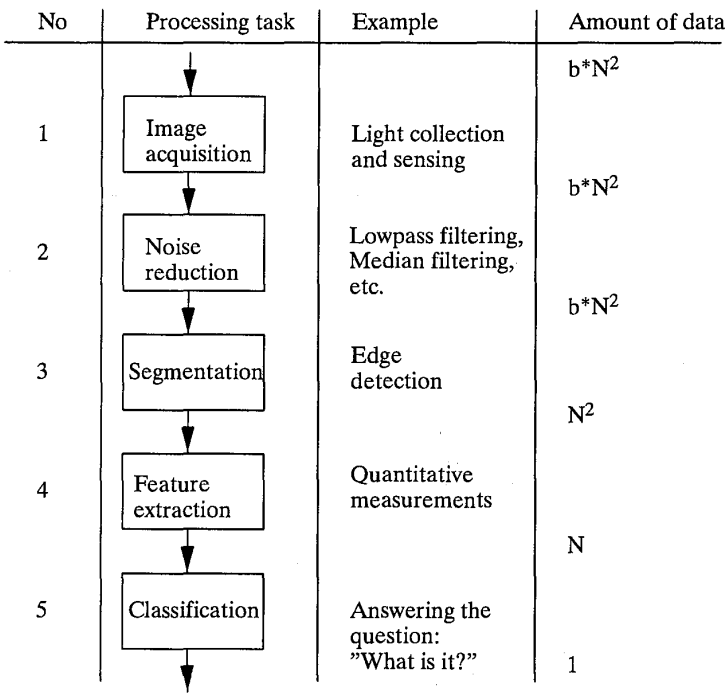


Figure 1.1. Image processing sequence

The amount of input data to the four first processing tasks are of order N^2, as shown in the rightmost column in Figure 1.1. The first three stages consists of local operations while the fourth stage is a global feature measuring operation. As the input data for local neighborhood data operations are of similar type, this suggests that a physically integrated computer architecture is feasible and since it all starts with an optical sensor, the ideal system for the four first stages would be a single solid-state component. The output from such a system would then be a feature vector to be forwarded to a classification system. Thus, what we are

looking for is a system which accepts a time sequence of images, f(x,y,t), and for each image output a feature vector, $\bar{g}(t)$, as shown in Figure 1.2.

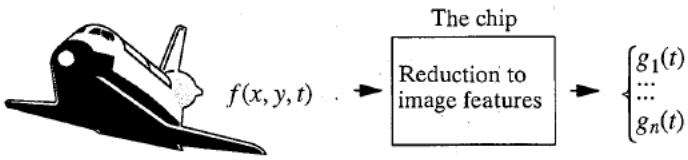


Figure 1.2. The goal for the image information reduction on a chip

The architectures to be presented are derivations of the classical idea of the massively parallel SIMD processor [3][9][16]. However, when implemented on a single chip it will be seen that new constraints as well as new possibilities appear, to the extent that also the most basic image processing algorithms are affected. To decide what type of basic operations one would like to have available within such a component, it is necessary to look at the cost, i.e. the VLSI area occupied by the hardware supporting each operation. For the individual cell in a 2D-array of Sensor-Processing Elements, SPE, a minimal configuration would consist of a photodiode, an Analogue-to Digital-Converter ADC, a bit serial ALU, and some memory for each pixel position. An important benchmark is the transistor count within each such SPE. Table 1.1 shows the transistor count for three different cases (A)–(C) containing sensor and processing units and with a memory of 16 bits per SPE.

Module	Transistors (A)	Transistors (B)	Transistors (C)
Sensor	1	1	1
ADC	110	90	10
ALU	100	60	60
Memory	110	50	30
Total	320	200	100

Table 1.1. Transistor count

In the case (A) we use "standard cell" technology. One possible realization of such a SPE is shown in Figure 1.3. This design has a lot in common with PASIC [1] as it consists of one photo sensor, one ADC, and a 1-bit bus connecting the ADC with three registers and a memory. Attached to the registers is a full adder logic which enables us to perform all types of Boolean operations. Gates for 2D mesh connection are included. A 256x256 SPE array of this type would require 21,000,000 transistors which is far too much for present day technology.

The case (B) in Table 1.1 assumes an optimized VLSI design for this SPE, where dynamic circuitry has been used where appropriate. A 256x256 SPE array still requires 13,000,000 transistors.

The last case (C) corresponds to the new concept of this treatise where the ADC has been removed and the memory per SPE has been reduced to 10 bits which leaves us with only 6,500,000 transistors for the 256x256 array of SPEs. We believe that this is a feasible number for a chip design in a foreseeable future.

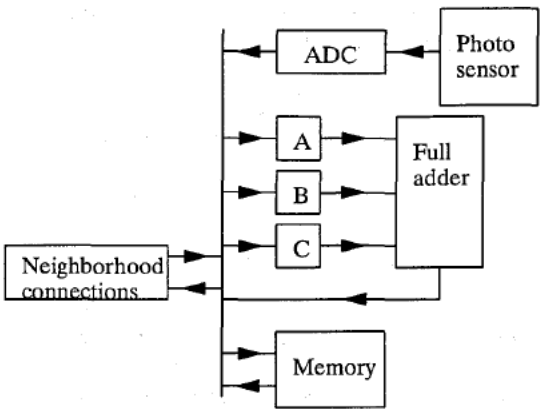


Figure 1.3. Logic scheme of a minimum configuration SPE based on traditional thinking

Actually, in the sequel we will show that conventional A/D conversion is neither necessary nor particularly efficient. Also, by using non-conventional digital representation we will be able to save on memory registers. The concept that makes this possible is hereby coined **Near Sensor Image Processing (NSIP)**.

To fully understand the NSIP concept it is essential to have a working model of the opto-electronic process that takes place within each SPE. For the discussion to be meaningful it is also necessary to define the way in which the analog sensor signal is converted into a measurable entity. These two topics are treated in section 2. For the sake of simplicity we will start to describe NSIP for a one-dimensional SPE-array that has been used in practice, the LAPP1100 [8]. Section 3 describes a number of simple NSIP operations and how they are implemented. It also describes how the special NSIP data representation is used throughout the system. Sections 4, 5, 6, and 7 describes adaptive thresholding, linear- and non-linear grayscale mapping, and error modelling, respectively. Grayscale operations are discussed in sections 8 and 9 for grayscale morphology and linear convolutions, respectively. In section 10 we present a 2D SPE model and describe the consequences for the architecture and programming. Section 11 describes how global operations on the array are performed in 2D and section 12 how features are extracted in 2D using the COUNT net. Section 13 describes a possible VLSI implementation of a 2D NSIP architecture. Section 14 describes an NSIP application which produces range data using sheet-of-light technique.

2 A Near-Sensor Image Processor

2.1 The photodiode sensor

The photo-electric effect is the basic physical principle used in all light sensors. Under certain conditions it is possible for a photon which hits a material to leave its energy to an electron. This causes a charge to build up in the sensor material. In a CCD photo element used in many TV cameras, the charge is collected in a well created by applying suitable voltages to a system of electrodes. During exposure the charge will grow until the well is full (overexposure) or until the charge is forced to move away.

The *photodiode sensor* utilizes the photo-electric effect differently, see Figure 2.1. Here, the photo-current is used to discharge a small capacitor which initially has been charged to a nominal level. Thus, the remaining voltage over the capacitor will be an indicator of the accumulated light over the exposure time. It is essential to sample this voltage level before the capacitor is completely discharged, or over-exposure (saturation) will result. Typically, the capacitor is obtained for free as a byproduct of the inversely biased diode (p/n-junction) in a semiconductor material. For silicon based material the spectral response of a photodiode is rather broad; from infrared (1100nm) to ultra violet (300 nm).

There are several reasons to choose a photodiode element as the sensing device in machine vision applications. This sensor takes up more space than a CCD element but it is characterized by low noise, good uniformity and somewhat higher sensitivity in the blue region. The most important property, however, is the possibility to read out information non-destructively through the use of a high-impedance amplifier/comparator. As will be seen later, this property is essential for several of the things we want to accomplish.

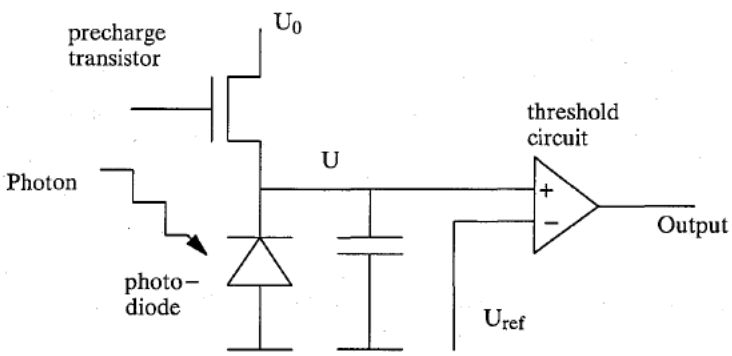


Figure 2.1. The photodiode element and sense amplifier

Figure 2.1 shows schematically the photodiode sensor and its thresholding amplifier. By using a high gain differential amplifier, the output of the sensor will be a binary signal suitable for further digital processing. At first sight it may seem a considerable restriction to forward only binary values from the sensor. Disregarding this for the time being, however, let us note that the "infinite gain" operation taking place directly at the output of the sensor device is very robust to component variations and ensures that good response uniformity is maintained throughout an array of SPE's.

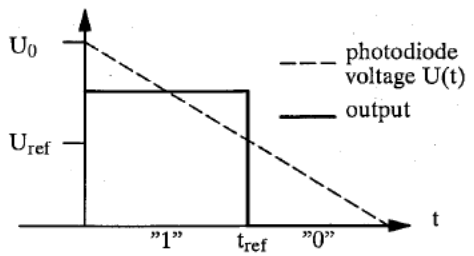


Figure 2.2. The voltage over the photodiode and the output from the comparator

The relationship between the light intensity I and the voltage U over the photodiode during exposure, can be approximated by the linear function:

$$U(t) = U_0 - \int_0^t kI(\tau)d\tau \tag{2.1}$$

In the case the intensity I(t) is constant over time t, we have

$$U(t) = U_0 - kIt \tag{2.2}$$

which is shown in Figure 2.2. Here, t is the time interval from the precharging of the photodiode to the sampling moment. k is a constant which describes the light sensitivity of the photodiode. U_0 is the precharge voltage. The relationship is valid as long as $U > 0$.

The time it takes for the voltage U to discharge to U_{ref} with a given intensity I is

$$t_{ref} = \frac{U_0 - U_{ref}}{kI} \tag{2.3}$$

It is obvious that we can use the device in Figure 2.1 to measure the light intensity in at least two ways. We may keep the exposure time constant and, after a complete integration time we may change the threshold level U_{ref}. The binary output reveals when an U_{ref} equal to the photodiode voltage is found. This corresponds to the action of an ordinary (CCD- or photodiode) sensor system in which a fixed exposure time is used after which the analog output is subjected to an A/D-converter. The maximum dynamic range will be about 1000:1.

Alternatively we may keep the threshold at a constant value and measure the time interval t_{ref}, shown in Figure 2.2. This is the NSIP approach. It yields a dynamic range in the order of 10^5:1 since valid exposure times may well range from 1μs to 100ms.

All NSIP algorithms are based on the observation of when the change of the binary output takes place. However, the output from a single photodiode such as the one in Figure 2.1, is not tested individually. Rather, all output data from an array of photodiodes are brought in for processing in parallel. In fact, the sensors are **interrogated** at even or uneven time intervals in the main NSIP loop during ongoing exposure. Hence, non-destructive read-out from the photodiode is essential to the whole idea.

Assume that the interrogation interval Δt are constant. Then, interrogation event number i where

i = 1, 2, .. , N

yields a linear relation to the interrogation points in time t_1, t_2, .. t_N

$$t_i = t_1 + (i - 1)\Delta t \tag{2.4}$$

which is shown in Figure 2.3.

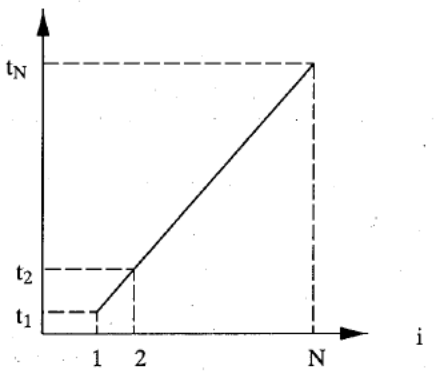


Figure 2.3. The relation between interrogation index and time

Suppose that a sensor element output changes its output at time t_{ref}, where

$$t_i \leq t_{ref} < t_{i+1} \tag{2.5}$$

If the time interval Δt_i is small we assume

$$t_{ref} = t_i \doteq t(i) \tag{2.6}$$

For this sensor element we then have that

$$t(i) = t_1 + (i - 1)\Delta t = \frac{U_0 - U_{ref}}{kI} \tag{2.7}$$

This gives us the intensity I as a function of the index, i,

$$I(i) = \frac{U_0 - U_{ref}}{k(t_1 + (i - 1)\Delta t)} \tag{2.8}$$

Note that although the intensity is constant in time t for each sensor in (2.8) which is the I-value which causes switching at time t_{ref}, I is now a function of t_{ref} and hereby also of the index i as indicated in Figure 2.4. As a consequence, the output quantity i, which is the only one the system can make use of, has an inverse linear dependence on the pixel intensity I according to (2.8).

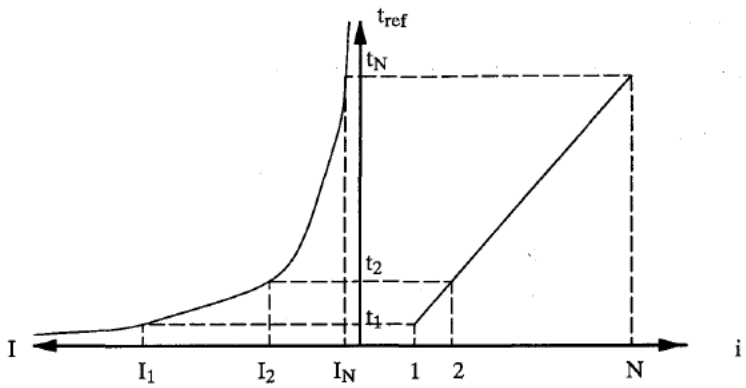


Figure 2.4. The relation between index, time, and intensity

2.2 LAPP1100 - a 1D architecture

The LAPP1100 [8] architecture is based on the traditional bit-serial SIMD architecture found in a number of different machines, [3], [9], [4], [10], [14], [16], and [18]. As shown in Figure 2.5, LAPP1100 is a 1D SIMD processor array with some extensions to handle image operations of global nature. The processor is register-accumulator oriented and typical operations use one of the registers as the first operand and the accumulator as the second operand. The result is stored into the accumulator destroying its previous value. Multi-level (gray-scale) operations are performed bitwise using the registers for intermediate storage (bit-serial operations). The size of the array is 128 and the memory consists of 13 1-bit registers per SPE. To be slightly more general in the description below, we set the array size to N and the number of registers to M.

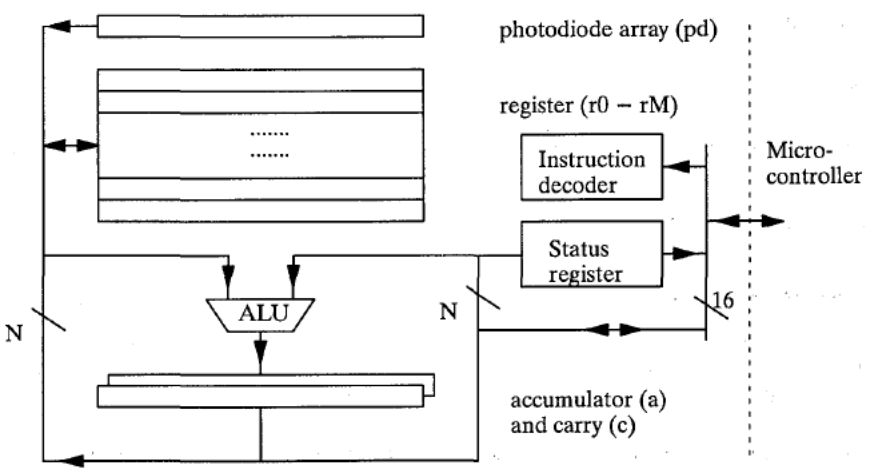


Figure 2.5. The LAPP processor (programming model)

In Figure 2.5 there are N 1-bit busses connecting the photodiode array, the registers, the ALU, and the accumulator. For conceptual reasons, the N busses are bundled together. In

the floor plan of the chip each individual SPE and its 1-bit bus forms a vertical slice, which is shown in Figure 2.6.

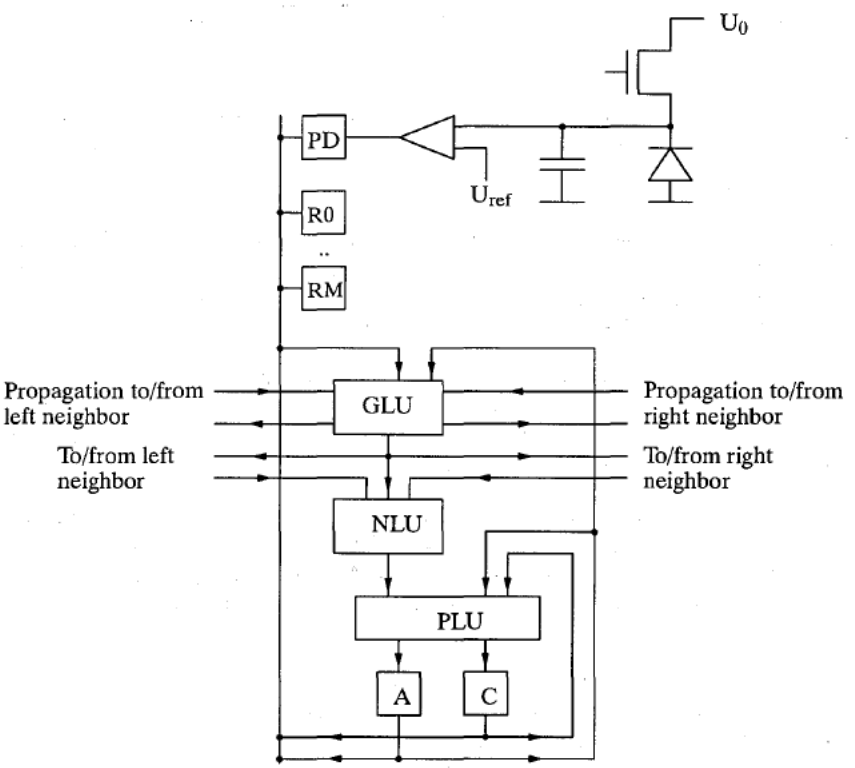


Figure 2.6. The SPE in LAPP

The ALU is purely combinational and permits point operations (PLU), neighborhood operations of type 3x1 template matching (NLU), and global operations (GLU to be explained). Operations belonging to different classes can be combined into one instruction. As an example, the instruction

```
and (010) r5
```

will access register 5 and will then apply local template matching with (010) to each position. All matching positions are set to **1** and then AND-ed with the previous value of the accumulator. The result is that only those **1**'s in the accumulator which correspond to isolated 1's in register 5 will remain as **1**'s in the accumulator. This instruction is performed in one clock cycle (currently 1 μs). In place of r5 one could write *pd* for the photodiode, *a* for the accumulator, or *c* for the carry register. The local templet can be omitted which is equivalent to using the template (x1x).

A typical GLU instruction is

```
lfill a
```

which sets all the accumulator bits to **1** if they are to the right of the leftmost **1**. The architecture of the GLU part will be discussed more in section 11.

One instruction is particular to the photodiode array. This instruction is

initpd

which pre-charges the photodiode as described in the previous section. Sampling (read-out) of the binarized photodiode output is done whenever *pd* is used as an operand. Thus

ld pd

will store the current value in the accumulator. The accumulator content will then depend on the exposure time which is given by the time elapsed between the "initpd" and the "ld pd" instructions. As was mentioned earlier the readout is non-destructive which makes it possible to do several "ld pd" instructions during the exposure.

The LAPP chip is equipped with two 8-bit output status registers for global data. The first, POS is only valid when there is a single position set to **1** in the accumulator. The content of the POS is then indicating this position. The second register, COUNT delivers a count of the number of PEs which has a **1** in its accumulator registers.

The current LAPP chip implementation uses an external 16-bit wide bus to communicate with a controlling micro-processor (not shown in Figure 2.5). The micro-processor issues instructions to the LAPP processor and uses the status registers to read out the result. A list of the instructions used in this thesis follows below. A Pascal-like notation will also be used throughout this part of the thesis.

ld pd	Load accumulator with comparator output
ldi pd	Load accumulator with inverse value of comparator output
initpd	Precharge (initialize) the photodiode
lfill a	Set all accumulator bits to **1** which are to the right of the leftmost **1**
st r0	Store the content of the accumulator in register 0
clr a	Set the accumulator to 0
or r0	Or the accumulator with reg 0 and put the result in the accumulator
xor r0	Exclusive-Or the accumulator with reg 0 and put the result in the accumulator
and r0	And the accumulator with reg 0 and put the result in the accumulator

We like to make the following final comments. The length of the programming examples to follow are deliberately kept to a minimum. Even so, NSIP programs in general often turn out to be very compact. It should be noted that the LAPP architecture is just one out of many possible architectures where the NSIP concept is applicable. We have chosen LAPP because of its simplicity and because it has been tested and found to work well in practice. In the sequel most of the operations and algorithms will be described and exemplified using a linear array. However, one should bear in mind that the principal results are applicable also to 2D NSIP systems which will be dealt with in section 10 and forward.

3 Near-Sensor Image Processing algorithms

3.1 General

Practically all NSIP algorithms have the structure illustrated by the flow chart in Figure 3.1. First, the photodiode in each SPE is initialized (=precharged). Then, a program loop is executed and for each pass the outputs from the comparators are interrogated and some logical operations are performed. When a certain condition is met the loop is terminated and a final feature extraction is performed.

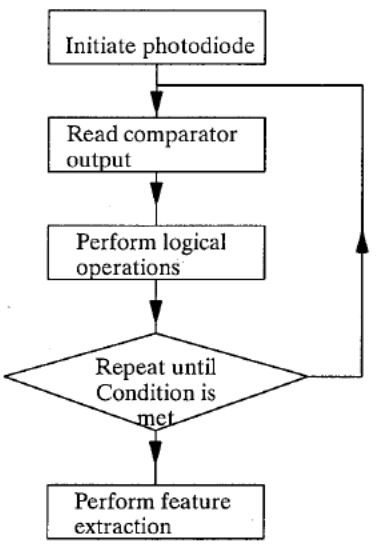


Figure 3.1. The flow chart for NSIP algorithms

In pseudo PASCAL Figure 3.1 correspond to:

Initialize photodiode
REPEAT
    Interrogate the output and perform operations
UNTIL *Condition is met*
Perform feature extraction

The interrogation/execution loop is the foundation of the NSIP concept and easy to recognize in the routines which we will describe below.

3.2 Finding the position of maximum intensity

This is an extremely simple example of NSIP. After precharge which takes place at time t=0, we interrogate the array of the photodiodes repeatedly. For a one-dimensional row of SPE's the initial status is shown in Figure 3.2(a). Then, the photodiodes will be discharged at a rate which is proportional to the incoming light. At a certain point in time t>0 the voltage over the photodiode array may look like Figure 3.2(b). During the discharge process we continu-

ously interrogate the output from all comparators to see if the threshold has been crossed somewhere. When this occurs, the loop is terminated and we obtain the position of the pixel with the highest intensity by performing an LFILL operation on the accumulator followed by a COUNT, see Figure 3.2(c).

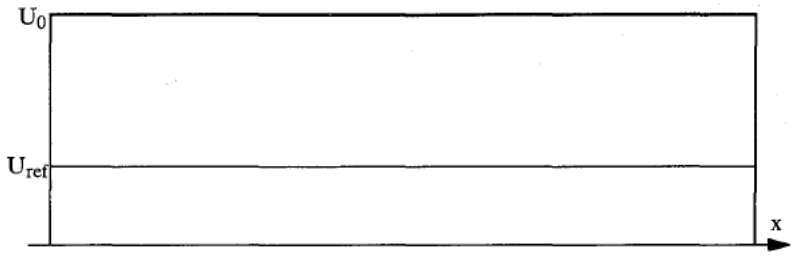


Figure 3.2(a). Photodiode array status at t=0

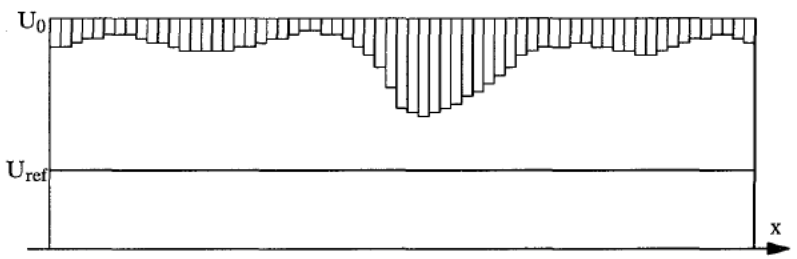


Figure 3.2(b). Photodiode array status at t>0

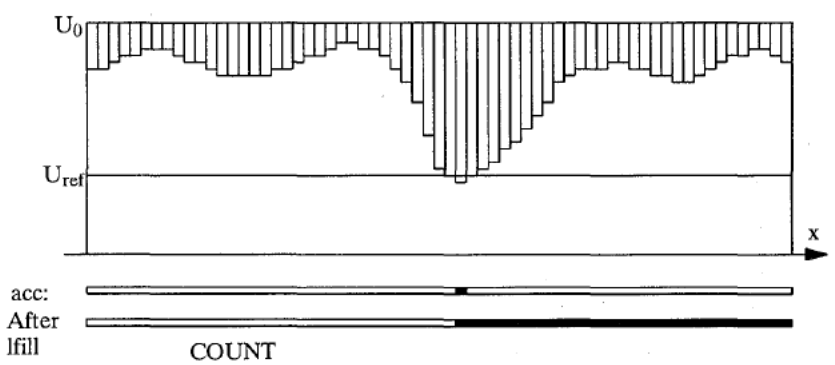


Figure 3.2(c). Photodiode array status when brightest intensity reaches U_{ref}

Using NSIP syntax this procedure can be described as:

```
initpd                          ; Initialize the photodiode
REPEAT
   ldi pd                       ; Load accumulator with inverse value of output
UNTIL COUNT <> 0
lfill a                         ; Fill to the right
answer:=COUNT
```

The "initpd" statement precharges the voltage over the photodiodes in the array to U_0. The "ldi pd" instruction loads the inverted values of the photodiode comparator into the accumulator. Thus, zeroes will be loaded as long as none of the diodes have reached the threshold level. The "UNTIL COUNT<>0" is the conditional branching operation which makes use of the global operation "COUNT". The "lfill" is a global instruction which sets all bits to the right of the leftmost "1". This will work as a tie-breaker in case several "1"s occur during the same sampling period but also, as a means to obtain the position of the leftmost "1" by reading the status variable COUNT. The status of the array after the termination of the loop is illustrated in Figure 3.2(c). At the bottom the accumulator content is shown at the instance of the first threshold passage and after the "lfill" operation.

This technique of finding the position of the maximum intensity is used in the range from sheet-of-light application to be described in section 14 below.

Finding the position of the **minimum** intensity requires the following changes of the max finding program:

```
initpd                          ; Initialize the photodiode
REPEAT
   st r0                        ; Store previous value of output
   ld pd                        ; Load accumulator with non-inverse value
UNTIL COUNT = 0                 ; Until all have passed
lfill r0                        ; Fill previous readout to the right
answer:=COUNT
```

For comparison, in a conventional image processing system, the position of the maximum intensity is found in the following way. First, we will acquire the image by exposing the photo-sensitive sensor (CCD/photodiode/..) a given time. This exposure time should be set to ensure that the maximal value is not over- or underexposed. The analog values which we get from the sensor are then A/D-converted with an appropriate precision to ensure that we can distinguish between the largest and the second largest intensity. We then scan the sensor values, comparing each value with the maximum value found so far and exchange that value and its corresponding position when we find a larger value.

Note that in the traditional method we have to pre-decide an exposure time. In NSIP the exposure time depends on the light condition as it is exposed until at least one position accumulates enough light energy. Thus, tolerance to various light conditions has been traded for data-dependent non-fixed exposure time. To obtain such adaptation in traditional systems an outer control loop is needed to modify the exposure time and maintain a reasonable signal-to-noise ratio. All this comes for free in with NSIP.

In the NSIP max finding example, the self-adaptivity to various light conditions is not limitless, however. Extremely low intensity levels would require extremely long and data-dependent operation times which might be prohibited in a real-time application. To a certain extent long operation times can be shortened by decreasing $U_0 - U_{ref}$ but only at the cost of lower signal-to-noise ratio and reduced possibility to distinguish between high intensities. In fact, for any NSIP-application the global controllable system parameter $U_0 - U_{ref}$ should be optimized in a trade-off between speed and noise sensitivity. A more complete error and parameter optimization will be presented below in Section 7.

3.3 Detecting positive gradients

This operation is equivalent to computation of the gradient (1D gradient = derivative) followed by thresholding with the threshold set to 0. The gradient kernel to be used is $\boxed{1\,|\,-1}$.

The photodiodes are precharged at time t=0 and then exposed to the image. Then, the processor array executes a program which interrogates each pixel position to see if the voltage has passed the threshold and if its left neighbor has not. If this is the case a register flag is set in that particular SPE. Figure 3.3(a) shows the status at a time t>0 when there is one positive gradient which has just barely passed the threshold U_{ref}. Figure 3.3(b) shows the final result when all the voltages have passed the threshold and all the positive gradient pixels have been ORed together.

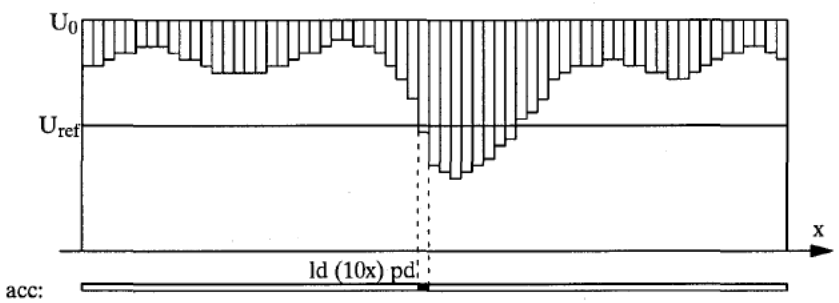


Figure 3.3(a). A positive gradient is detected

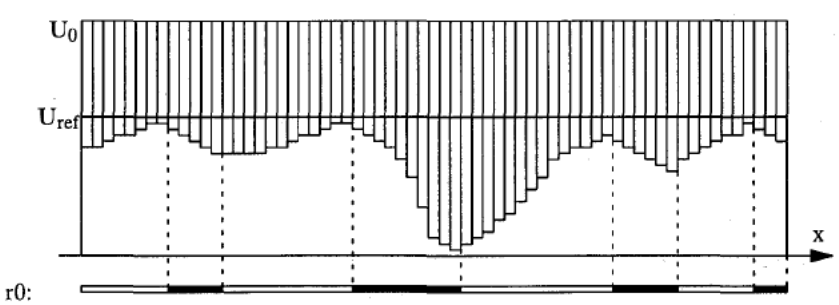


Figure 3.3(b). The result when all pixels are below the threshold voltage

This operation is insensitive to various light intensities since it only measures the differences, and not the absolute value, of neighboring pixels. However, for very dark pixels the

time-out for the total operation might prevent recording of their gradients. Also, if two neighbors have almost the same intensity they might switch their binary outputs so close in time that their gradient escapes detection.

The following NSIP program will perform detection of positive gradients:

```
clr a
st r0
initpd
REPEAT
   ld (10x) a
   or r0
   st r0
   ld pd
UNTIL COUNT = 0
```

The first two instructions clear the content of the accumulator and register 0. In the loop a "1" is accumulated in register 0 for all those pixels which have passed the threshold while their left neighbor has not. The "x" used in the templet field in the ld instruction means "don't care". The loop continues until all photodiodes have passed the threshold (COUNT=0).

3.4 Median and rank order filtering

Traditionally, median filtering is performed as a series of comparisons and multiplexing of grayscale values. Figure 3.4(a) shows how a three-way median operation can be performed with one max/min-, one max-, and one min-operation, where the max/min-operation is shown in Figure 3.4(b).

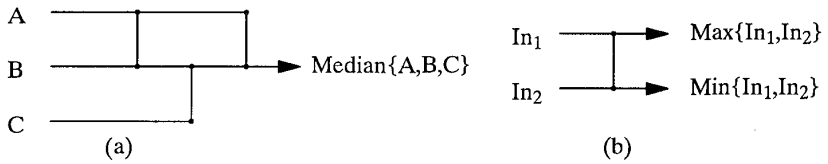


Figure 3.4. Three-input median operation

In NSIP we use the fact that for each neighborhood of three, the second pixel to cross the threshold is the median pixel. To keep track of the events we use three binary registers r_3, r_4, and r_5, which are set to 1 if the center-, the left-, or the right pixel is the median. The incoming value of the comparator output will be stored in r_1 and the result of the previous interrogation in r_2.

Equations (3.1), (3.2), (3.3), and (3.4) are used to compute the median. As they are carried out sequentially we will favor the center pixel to the neighbor pixels, and the left pixel before the right pixel as a tie-breaking rule. Equation (3.1), indicates that the center pixel is the median if the center pixel has passed the threshold since the last interrogation ($\overline{r_2}\&r_1=1$) and at least one of the neighbors has passed ($(1xx)r_1\&(xx1)r_1=\overline{(0x0)r_1}=0$) and no median pixel has yet been determined ($\overline{r_4}$ & $\overline{r_5}$). Equations (3.2) and (3.3) perform corresponding operations for the left and the right neighbor respectively.

$$r3 := \overline{r2} \;\&\; r1 \;\&\; \overline{(0x0)r1} \;\&\; (\overline{r4} \;\&\; \overline{r5})) + r3 \quad (3.1)$$

$$r4 := \overline{(1xx)r2} \;\&\; (1xx)r1 \;\&\; \overline{(x00)r1} \;\&\; (\overline{r3} \;\&\; \overline{r5})) + r4 \quad (3.2)$$

$$r5 := \overline{(xx1)r2} \;\&\; (xx1)r1 \;\&\; \overline{(00x)r1} \;\&\; (\overline{r3} \;\&\; \overline{r4})) + r5 \quad (3.3)$$

$$r2 := r1 \quad (3.4)$$

The program starts by first resetting a number of registers. The loop is then performed according to (3.1), (3.2), (3.3), and (3.4).

```
clr a
st r1..5
REPEAT
   ld r1
   st r2
   ldi pd
   st r1
   ldi r2
   and r1
   andi (0x0) r1
   andi r4
   andi r5
   or r3
   st r3                    ; Store center pixel pointer
   ldi (1xx) r2
   and (1xx) r1
   andi (x00) r1
   andi r3
   andi r5
   or r4
   st r4                    ; Store left pixel pointer
   ldi (xx1) r2
   and (xx1) r1
   andi (00x) r1
   andi r3
   andi r4
   or r5
   st r5                    ; Store right pixel pointer
   ld r1
UNTIL COUNT=N
```

This NSIP-operation results in a neighborhood pointer to the position of the median. A different problem is to generate the median value in the neighborhood in the same representation as the input, i.e. a signal which is 0 before the median pixel has passed the U_{ref}, and 1 thereafter. Again, assume that we use a 1x3 neighborhood. The Boolean function which sets the output ($=r_2$) to one when the median pixel in the neighborhood passes is

$$r2 = (1xx)\ r1\ \&\ r1 + r1\ \&\ (xx1)\ r1 + (1xx)\ r1\ \&\ (xx1)\ r1 = (11x)\ r1 + (x11)\ r1 + (1x1)\ r1 \quad (3.5)$$

Figure 3.5 shows the function of Equation (3.5) graphically. The t-value when r_2 switches represents the median for the neighborhood over the equation

$$I_{median} = \frac{U_0 - U_{ref}}{kt_{median}} \tag{3.6}$$

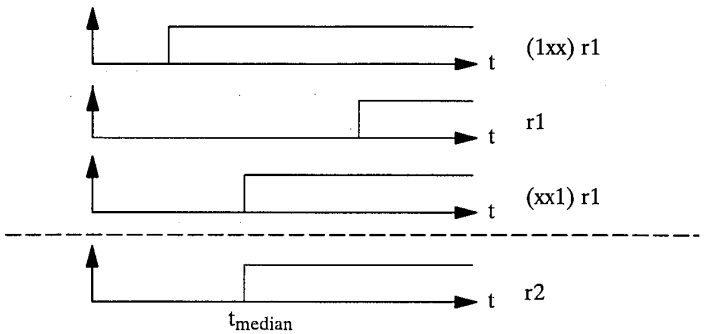


Figure 3.5. The median as a logic function of the three inputs

The NSIP program for this procedure is as follows:

```
initpd
ldi pd
REPEAT
   st r1
   ld (11x) r1              ;
   or (1x1) r1              ; from Equation (3.5)
   or (x11) r1              ;
   st r2
   ..
   ldi pd
UNTIL COUNT=N
```

Here, r0 holds the comparator output and r1 the median. As will be shown in section 3.6 this median signal can be used as input to other NSIP operations interwoven in the same program.

In the same manner we can obtain any rank order function. For instance, a max function is implemented by

```
r2 = (1xx) r1 + r1 + (xx1) r1 = (000) r1          (3.7)
```

and a corresponding min function by

```
r2 = (1xx) r1 & r1 & (xx1) r1 = (111) r1          (3.8)
```

The right hand side of (3.5), (3.7), and (3.8) are all positive Boolean functions. (A positive Boolean function may always be expressed with non-inverted input variables.) Evaluation of such functions can be done with so called *stack filters* [17] which require the input vari-

ables to be monotonously growing during evaluation. Thus, a binary variable may not switch back from value 1 to value 0 once it has been set. Obviously, this condition is fulfilled in our case as illustrated in Figure 3.5 and therefore, the NSIP implementation may be called stack filter. However, the signal representation illustrated in Figure 3.5 is also equivalent with the *Umbra transform* [15] which will be discussed in section 8 of this treatise.

3.5 Histogramming

The number of SPE's where the voltage has dropped below U_{ref} varies over time, for instance as in Figure 3.6. This is the integral of the histogram of the image $t_{ref}(x)$ which will be referred to as Inthist. To obtain the histogram we should differentiate the Inthist data.

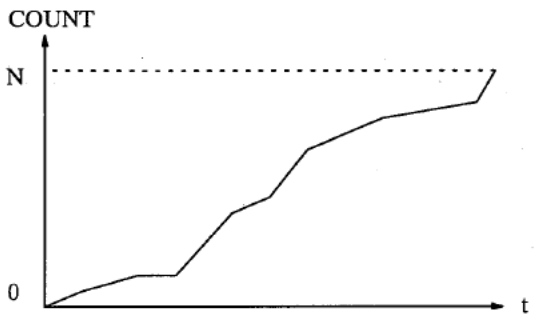


Figure 3.6. The content of COUNT as a function of time = Inthist

Program for differentiating the inthist:

```
Old_Count:=0
initpd
FOR i:=1 TO n DO
    ldi pd
    hist[i]:=COUNT − Old_Count
    Old_Count:=COUNT
END−FOR
```

An alternative is to count only those SPE's for which the voltage passed the U_{ref} since the last interrogation. Obviously, this will give us the histogram directly.
Program for histogramming:

```
clr a
st r0
initpd
FOR i:=1 TO n DO
    ldi pd
    xor r0                    ; accumulator is 1 only when passing threshold
    hist[i]:=COUNT
    or r0
    st r0
END−FOR
```

The variable n corresponds to the number of bins, and hist[i] is the resulting histogram. In each SPE r_0 is used to signify that this SPE has already contributed to the histogram. The "xor r0" instruction will set those pixels which contribute to the most recent bin.

It should be observed that histograms created by the above programs are non-linear in light intensity. This is due to the non-linear relation (2.3) between the light intensity, I, and the time, t_{ref}. Different schemes to obtain linear and non-linear relations are discussed in sections 5 and 6. It should also be noted that storage of the histogram data takes place in the control unit (host computer), not in the NSIP array.

3.6 Combining operations

Image processing tasks often require a combination of several image operations. As an example, Figure 3.7 shows a sequence of operations which measures the area occupied by positive gradients. The image is first subjected to median filtering to suppress noise. We then detect where we have positive gradients. Finally, we count the number of these pixels to get the total area.

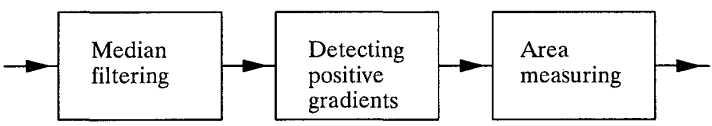


Figure 3.7. A three stage image processing operation

The median filter routine is obtained from section 3.4, Equation (3.5), which delivers an output set to one as soon as the median pixel has passed. This is used by the positive gradient detector in section 3.3. The output from the combined NSIP routine is a count which represents the total area of positive gradients.

```
clr a
st r1
initpd
REPEAT
   ldi pd                ; Input
   st r0
   ld (11x) r0
   or (1x1) r0
   or (x11) r0           ; Median = ab+ac+bc
   st r0

   ld (01x) r0           ; Positive gradient ?
   or r1
   st r1

   ld r0
UNTIL COUNT=n
ld r1
C := COUNT               ; Area/length
```

The above example underscores some of the principles of NSIP. The first processing stage is the sensor and comparator which delivers an information about the t_{ref}-value in its special "thermometer" representation. This is input to the median filter which has the same representation as output which is fed to the positive gradient detector. The output from the gradient detector is not delivered in thermometer representation, however. Instead, the output is accumulated over time and when the interrogation loop is finished we take a global count of the result.

The general structure of a composite NSIP program is as follows:

```
Initializations
initpd
REPEAT
   ldi pd                        ; Input
   st r0
   r1 := Operation_1(r0)
   r2 := Operation_2(r1)
   ...
   rm := Operation_m(rm-1)
   ld ri                         ; Termination control operand
UNTIL COUNT=n
ld rm
C := COUNT                       ; Feature extraction
```

All the operations in the inner loop are Boolean and the data transfers between the operations are implicit inside the loop.

4 Adaptive thresholding

4.1 Using the maximum intensity

Adaptive thresholding means that with a given threshold we want to find an exposure time so that the binarized image is optimal with respect to some parameter or quality. In the first case to be presented here, optimal thresholding is defined by the following procedure, see Figure 4.1. Initially U_{ref} is set to 10% of U_0. When the first pixel in the image passes this threshold U_{ref} is changed to 55% of U_0. The binary image is then red out.

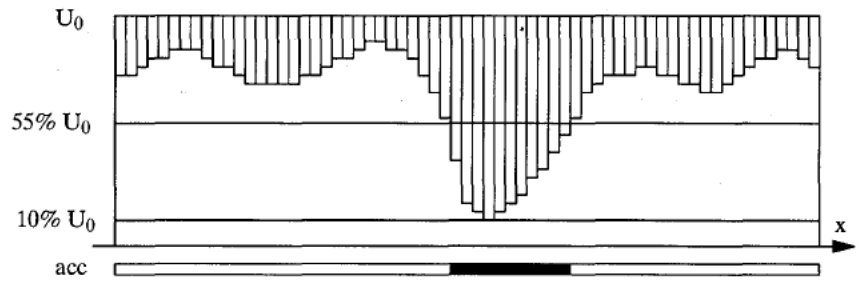


Figure 4.1. The result from the adaptive exposure

A generalization of the same idea would be to set the initial threshold to A % of U_0 and the read-out threshold to (50+A/2) %. In an ideal noise free situation the binarized result will not depend on A. Higher thresholds will automatically bring about shorter integration times. The advantages and disadvantages for high and low thresholds were discussed in Section 3.2.

Program for performing the operation in Figure 4.1:

```
ref:=0.1 * U0
initpd
REPEAT
   ldi pd
UNTIL COUNT<>0
ref:=0.55 * U0
ldi pd
```

The variable "ref" holds the value of the reference voltage. First, a low value is used to ensure that the highest intensity pixel exposes the photodiode array up to 90 % of the saturation level. When this has happened the reference is set to the middle of the amplitude range and a new sample of the image line is read into the accumulator.

4.2 Pattern oriented thresholding

Another variation of adaptive thresholding is to interrogate the photodiode array until the pixels of an expected object or a specific pattern has passed the threshold and comes forward as a binary output. For example, if we want to detect the position of the object with the size S of connected pixels in a single-object image, we can use the following simple binarizer.

```
initpd
REPEAT
   ldi pd
UNTIL COUNT>S
ld pd
lfill a                    ; Fill to the right
answer:=COUNT
```

In a more complex case we have noise or unwanted light flashes in the image. Then several smaller objects (noise spikes) are to be expected. To avoid these we interrogate until an object larger than a predefined size shows up. The size of an object is obtained with the operations *ld (01x) lfill* and *mark* which together reset all pixels except for the pixels in the leftmost object. The net effect of screening out small objects is a kind of median filtering.

```
initpd
REPEAT
   ldi pd
   st r0
   ledge r0                ; Get no of objects
   Max_size:=0
   WHILE COUNT > 0
      ld (01x) lfill r0    ; Set leftmost pixel
      mark r0              ; Mark leftmost object

      Max_size := max(Max_size,COUNT)

      xor r0               ; Remove leftmost object from image
      st r0
      ledge r0             ; Get no of objects
   END-WHILE
UNTIL Max_size > constant
ld pd
```

Several other types of assumption and a priori information about the object can be utilized in the NSIP concept. It is interesting to note that a device which exploits a similar technique to achieve robust thresholding was described by Lyon [13] in his *Optical mouse*. Here, a hard-wired logic network interacts with the photodiodes to achieve a particular sensitivity to the periodic pattern which is printed of a highly reflective board.

4.3 Histogram based thresholding

In image processing it is very common to assume a bimodal histogram and set the threshold at the min-point between the two maxima. Figure 4.2 shows a typical bimodal histogram and ideally, the histogram contains two different gaussian distributions which correspond to the background and the object. The min-point has a positive zero-crossing of the first derivative of the histogram [1].

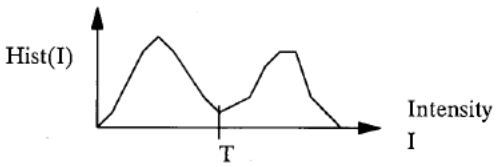


Figure 4.2. A histogram and its optimal thresholding

In a well-behaved bimodal histogram we can easily compute the first derivative of the histogram by approximating the derivation with some differentiating operator. In the result *Hist'(I)* the point T is located where the sign has its positive zero crossing, i.e. where the sign changes from minus to plus, see Figure 4.3.

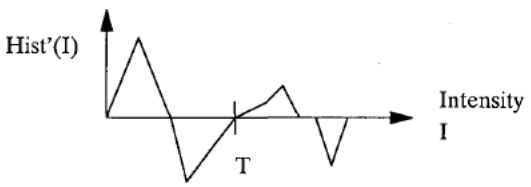


Figure 4.3. Zero-crossing of the first derivatives

In the case of a discrete histogram the choice of the derivative estimator is critical. For the very noisy histogram shown in Figure 4.4, the corresponding convolution kernel must be wide and large, suppressing high frequencies to avoid false zero-crossings.

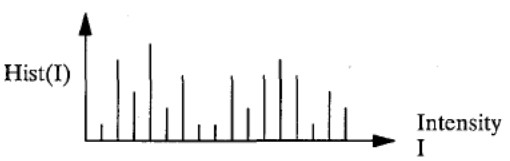


Figure 4.4. A noisy histogram

Large convolution kernels may create an excessive amount of computation. To avoid this we propose the following differentiating kernel $g_n(I)$

$$g_n(I) = \underbrace{1\ 1\ \ldots\ 1}_{n}\ \underbrace{-1\ -1\ \ldots\ -1}_{n} = 1\ -1 * \underbrace{1\ 1\ \ldots\ 1}_{n} * \underbrace{-1\ -1\ \ldots\ -1}_{n} \quad (4.1)$$

We also propose to use the Inthist instead of Hist as input to the thresholding function. As was shown in section 3.5 the Inthist is as easy as Hist to compute. We then propose to compute

$$Hist'(I) \doteq g_n * Hist(x) = [1\ -1] * g_n * Inthist(I) = \quad (4.2)$$
$$[\ 1\ \underbrace{0\ 0\ \ldots\ 0}_{n}\ -2\ \underbrace{0\ 0\ \ldots\ 0}_{n}\ 1\ ] * Inthist(x)$$

where we allow ourselves to employ the simple [1 −1] kernel to estimate the second derivative in (4.2). To compute Hist'(I) in this manner in the host computer is trivial.

5 Variable sampling time

5.1 General

The result from the comparator in the SPE as a function of time is the thick line in Figure 5.1. If we interrogate the comparator output we will first get a number of 1's followed by 0's, where the number of leading 1's is a measure of the intensity. This is how we made use of the intensity in the previous sections.

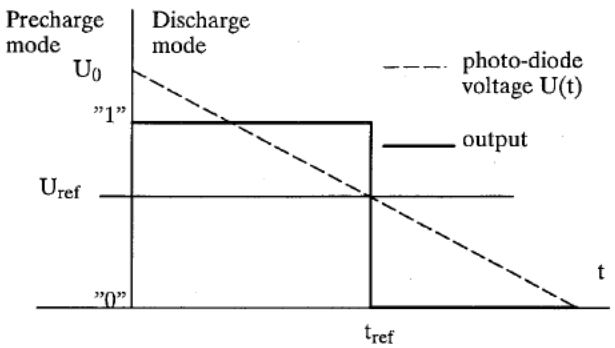


Figure 5.1. Output from an NSIP element

In the analog domain we observe the following. The voltage U(t) over the photo-diode as a function of time is a linear function shown as a dotted line in Figure 5.1 and can be written as

$$U(t) = U_0 - kIt \tag{5.1}$$

As was explained in section 2.1 above, the time t_{ref} from the precharge of the photodiode until the voltage reaches the threshold level and thereby changes the sign of the comparator is inversely proportional to the illumination, so that

$$t_{ref} = \frac{U_0 - U_{ref}}{kI} = \frac{K}{I} \qquad where\ K = \frac{U_0 - U_{ref}}{k}. \tag{5.2}$$

In the digital domain, the time variable t is mapped by the interrogating program onto a new quantity i, the interrogation cycle number. Obviously, by introducing a **variable interrogation cycle time** we will obtain a relation between I and i quite different from the relation t_{ref} and I in (5.2).

An NSIP program contains a loop with a sequence of instructions which are repeated until a certain termination condition is meet. As pointed out before, a typical NSIP program has the following structure:

REPEAT
  compute
UNTIL *condition*

Assume that the *compute* loop cycle time is t_d and that the discrete mapping between i and t is given by the time sampling set $(t_1, t_2, .., t_i, ... , t_N)$. Given a fixed threshold level U_{ref}, to

obtain a specific linear or a non-linear relation between time t and the interrogation cycle i, we insert a variable extra delay in the loop. These delays are called δt_i and may be stored in a Look-Up Table (LUT). They are defined as

$$\delta t_i = t_{i+1} - t_i - t_d = \Delta t_i - t_d \tag{5.3}$$

This will give us the following program structure:

```
wait( t1)
FOR i=1 TO N−1
   compute                    ; execution time = td
   wait(δti)
END−FOR
```

Thus, to realize a certain relation between i and I in a NSIP system we need to know the time it takes to process one *compute* t_d, determine the number of quantization levels N which is also the number of times the interrogation loop will be executed, and, most importantly, determine the mapping functions $I=g(i)$ and $i=g^{-1}(I)$. These functions are given by the parameters N, t_d, t_1, and a set (δt_1, δt_2, .., δt_i, .., δt_{N-1}) stored in a LUT.

5.2 Linear mapping

Figure 5.2 and Equation (5.2) show the intensity as a function of the time it takes to reach the threshold. A linear mapping between time t and loop index i is to interrogate at N points in time which are equidistant in intensity from I_N to I_1 so that with

$$\Delta I \doteq \frac{I_1 - I_N}{N - 1} \tag{5.4}$$

The intensity I_i which changes the sign of its comparator at interrogation event i is given by the following linear function.

$$I_i \doteq g(i) = I_1 f(i) = I_1 - (i - 1)\Delta I = I_1 - (i - 1)\frac{I_1 - I_N}{N - 1} \tag{5.5}$$

For the case N=5, Figure 5.2 illustrates that we want to find the sampling points t_1 to t_5.

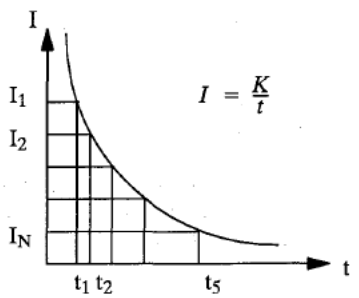


Figure 5.2. Equidistant sampling of I prolongs the time between two samples

If the photo-diode is precharged to U_0 at t=0, the illumination intensity I_1 will cause the voltage to have discharged to the threshold voltage U_{ref} at $t=t_1$ so that

$$I_1 = \frac{U_0 - U_{ref}}{kt_1} = \frac{K}{t_1} \tag{5.6}$$

More generally, the intensity I_i that will reach the threshold at time t_i is

$$I_i = \frac{K}{t_i} \tag{5.7}$$

Using (5.5) and (5.7) we get

$$t_i = (N-1)\frac{t_1 t_N}{t_1(i-1) + t_N(N-i)} \tag{5.8}$$

In the special case where $I_N = \Delta I$, which means that $I_1 = N^* I_N$, (5.8) simplifies to

$$t_i = t_1 \frac{N}{N+1-i} \tag{5.9}$$

Comparing (5.9) with (5.7) we see that a positive change in the variable i maps on t_i as a negative change in I_i. This is illustrated graphically in Figure 5.3.

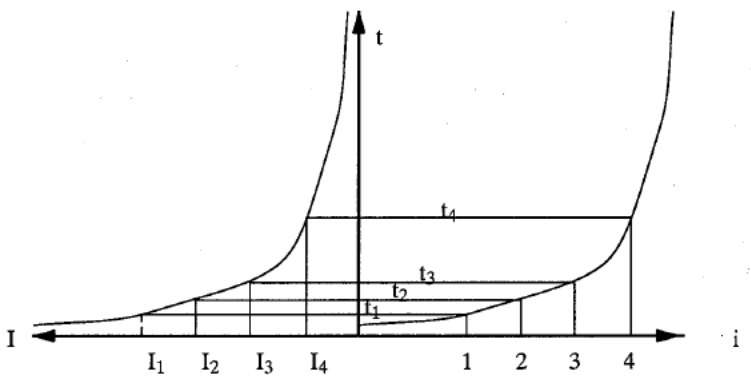


Figure 5.3. The relation between index, time, and intensity

In this case, according to (5.3), the difference between two time instances Δt_i, to be stored in a LUT as δ_i is

$$\Delta t_i = t_{i+1} - t_i = t_1 \frac{N}{(N-i+1)(N-i)} \tag{5.10}$$

The iteration number that requires the fastest loop (shortest delay) is the first which is evident from Figures 5.2 and 5.3. This means that the value of Δt_1 to be selected must be equal to or larger than t_d which is the time to complete the loop without any delays.

$$\Delta t_1 = t_2 - t_1 = \frac{t_1}{N-1} \geq t_d \tag{5.11}$$

For the case (5.9), the time t_N, which is the time for the lowest detectable intensity I_N to reach the threshold voltage, is

$$t_N = t_1 N \tag{5.12}$$

If the first sample should be able to detect the intensity I_1 at time t_1, the number of possible levels N according to (5.11) is bounded by

$$N \leq \frac{t_1}{t_d} + 1 = \frac{t_N}{N t_d} + 1 \Leftrightarrow N^2 - N \leq \frac{t_N}{t_d} \tag{5.13}$$

The quantity t_N might be a natural parameter to use since it may correspond to the maximal time to be allowed for the task.

To conclude, by making the rather natural choice, $I_N = \Delta I$, we have only two adjustable parameters, N and t_N. Another one is t_d given by the actual NSIP algorithm. If either t_N or N is decided the other one is constrained according to (5.13).

As an example, if we have a system with a minimum loop delay of $t_d = 10$ μs it is possible to make one of the following choices. We may set N to 100 for which (5.13) will give us $t_1 \geq 1$ms and (5.12) $t_N \geq 100$ ms. Alternatively, we may decide to make room for $N \leq 200$ which gives us $t_1 = 2$ ms and $t_N \geq 400$ ms.

5.3 Non-linear mapping

We define a more general mapping from interrogation index i to intensity I_i as

$$I_i = I_1 f(i) \tag{5.14}$$

In the special case of linear mapping (5.5), f(i) is

$$f(i) = \frac{N - i + 1}{N} = 1 - \frac{i - 1}{N} \tag{5.15}$$

The constraints for f(i) is that $f(1) = 1$ and $f(i+1) < f(i)$, because I decreases and i increases over time. In the same manner as (5.5) and (5.7) gave us (5.8), (5.14) will give us

$$t_i = \frac{K}{I_i} = \frac{t_1}{f(i)} \tag{5.16}$$

Assume that we would like to map I_i logarithmically onto i, i.e.

$$I_i = I_1 f(i) = I_1 e^{-A(i-1)} \tag{5.17}$$

From (5.16) we get

$$t_i = t_1 e^{A(i-1)} \tag{5.18}$$

and

$$t_N = t_1 e^{A(N-1)} \tag{5.19}$$

which gives us an N value of

$$N = \frac{1}{A}\ln\frac{t_N}{t_1} + 1 \tag{5.20}$$

Again, the constraint is that the time difference between any two sampling events must be equal to or greater than t_d.

$$t_{i+1} - t_i = t_1 e^{Ai} - t_1 e^{A(i-1)} = t_1 e^{Ai}(1 - e^{-A}) \geq t_d \tag{5.21}$$

We notice that the difference is smallest for i=1, due to the factor e^{Ai}. This means that

$$t_1(e^A - 1) \geq t_d \tag{5.22}$$

which can be viewed both as a constraint for t_1 and for A:

$$t_1 \geq \frac{t_d}{(e^A - 1)} \tag{5.23}$$

$$A \geq \ln(1 + \frac{t_d}{t_1}) \approx \frac{t_d}{t_1} \tag{5.24}$$

5.4 Data dependent mapping

The interrogation program in the section 5.2 has the following structure

```
initpd
initiate(time)
FOR i:=1 TO N DO BEGIN
   REPEAT
   UNTIL time ≥ t1 * N / (N−i+1)
   compute_operation
END
```

Thus, in section 5.2 the delay in the REPEAT-loop is variable but predetermined and data-independent. In this section we will see some cases where the delay is made dependent on the incoming sensor data.

Histogram equalization is a procedure which quantizes the pixel intensities so that the same number of pixels ends up in each gray-level bin. Using conventional methods this is a rather complex operation. As was shown in Figure 3.6 in section 3.5 the count of the output from the comparators over time is the Inthist of the image. This means that we should be able to obtain equalization by just observing when the appropriate number of pixels have passed the threshold. Given an image size of M and a bin number of N, this is an NSIP program for positive gradient detection, as described section 3.3, performed on a histogram equalized image.

```
FOR i=1 TO N
   REPEAT
      ldi pd
   UNTIL COUNT>i*M/N
   ld (01x) a                    ; ldi pd : ld (01x) a = ld (10x) pd
END−FOR
```

Note that the complete equalized image never exists in the NSIP program. Instead, the outer loop uses one quantization level at a time of the equalized image to execute (in this example) positive gradient detection.

Figure 5.4 shows graphically an equalization procedure with N=7. The count bins on the vertical axis will all contain the same number of pixels. This is accomplished by sampling (read-out) at times (t_1, .., t_7) when the appropriate number of pixels have passed the threshold.

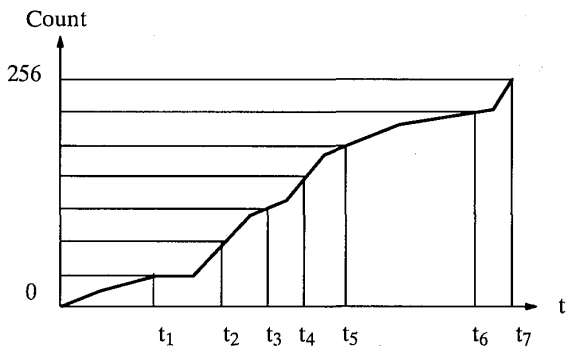


Figure 5.4. Histogram equalization

By changing the UNTIL condition in the REPEAT-loop of the previous program we can modify the implicit flat equalized histogram to something of our own choice. In Figure 5.5 is shown the Inthist for a perfect equalized case (a) and a case which allocates more dynamic range to lower intensities.

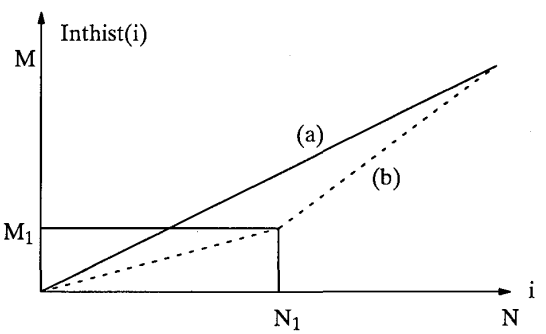


Figure 5.5. Different specifications of the Inthist

An NSIP program for this modified equalization without the gradient detection can be written as

```
FOR i=1 TO N1
   REPEAT
      ldi pd
   UNTIL COUNT>i*M1/N1
   compute
END-FOR
FOR i=N1+1 TO N
   REPEAT
      ldi pd
   UNTIL COUNT>M1+(i-N1)*(M-M1)/(N-N1)
   compute
END-FOR
```

The graphic description of this operation, is shown in Figure 5.6.

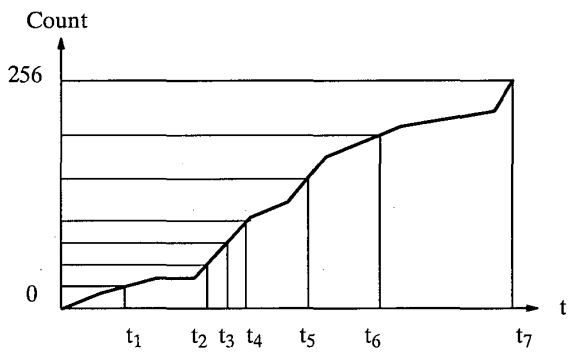


Figure 5.6. Modified histogram equalization

In fact, the Inthist could be modified to any monotonously growing function f(i) using the following program.

```
FOR i=1 TO N
   REPEAT
      ldi pd
   UNTIL COUNT>f(i)
   compute
END-FOR
```

A even more general approach to use the COUNT operation for data dependent mapping of the grayscale of the input image is to wait, within each iteration, until a certain condition to be satisfied. In the case of the histogram equalization, this condition is simply that the right amount of pixels have passed the threshold. An example of a more sophisticated but totally arbitrary case is to leave the inner REPEAT-loop when 5 new left edge pixels have passed the threshold. This can be written as

```
FOR i:=1 TO N DO BEGIN
   sum:=0
   REPEAT
      st pd,r0
      ld (01x) r0
      sum:=sum+COUNT
   UNTIL sum>5
   ld r0
   compute_operation
END
```

In a general case, the number which must satisfy the condition is a function f(i). In this case the program may look like:

```
FOR i:=1 TO N DO BEGIN
   sum:=0
   REPEAT
      st pd,r0
      compute_condition
      sum:=f(sum,COUNT)
   UNTIL sum>f(i)
   ld r0
   compute_operation
END
```

6 Variable threshold

6.1 General

In the basic NSIP formula, which defines the intensity I for which the output switches at time t,

$$I(t) = \frac{U_0 - U_{ref}}{kt} \tag{6.1}$$

the reference voltage U_{ref} has so far been considered to be constant. However, in reality this global parameter can easily be put under global control by the program executed in the host computer. The Equation (6.1) should then be written

$$I(t) = \frac{U_0 - U_{ref}(t)}{kt} \tag{6.2}$$

Assume that we want to achieve a pre-specified I(t). Then, the function $U_{ref}(t)$ which delivers this effect is

$$U_{ref}(t) = U_0 - k\ t\ I(t) \tag{6.3}$$

where the reference voltage is physically constrained as

$$0 \leq U_{ref}(t) \leq U_0 \tag{6.4}$$

Now, the time t will also be mapped to the interrogation index i by a function t(i) and the NSIP program can be written as:

wait(t_1)
FOR i=1 TO N
  Set voltage to $\tilde{U}_{ref}(i)$
  ldi pd
  compute
END–FOR

where $U_{ref}(t(i)) = \tilde{U}_{ref}(i)$.

As in (5.14), the specified intensity function I(t) can be written as a function f(t) multiplied with the intensity at the starting time, $I(t_1)=I_1$, as

$$I(t) = I_1 f(t) \tag{6.5}$$

Inserted in (6.3) (6.5) gives us

$$U_{ref}(t) = U_0 - k\ t\ I_1 f(t) = U_0 - \frac{t}{t_1}(U_0 - U_1)f(t) \tag{6.6}$$

As mentioned above, the executable NSIP program needs the function $\tilde{U}_{ref}(i)$ rather than $U_{ref}(t)$. However, if we assume that t(i) is linear (constant Δt_i) this mapping is trivial and for the time being we may consider $U_{ref}(t)$ as our main target.

6.2 Linear mapping

A function f(t) which delivers a general linear mapping between the intensity I and the time t can be written as

$$f(t) = 1 - \frac{t - t_1}{t_N - t_1} \frac{I_1 - I_N}{I_1} \tag{6.7}$$

and illustrated in Figure 6.1.

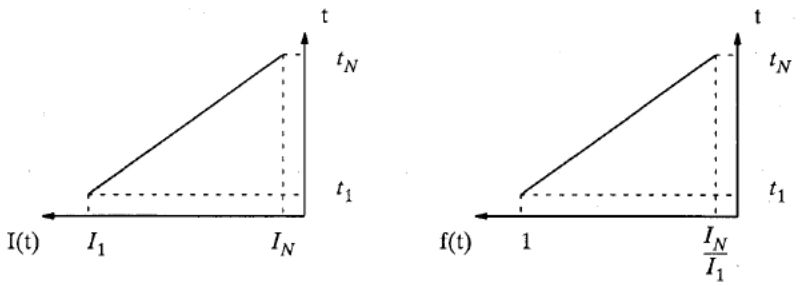


Figure 6.1. A mapping between time and intensity and its corresponding f(t)

From (6.7) we get

$$U_{ref}(t) = U_0 - \frac{U_0 - U_1}{t_1}\left(t + \left(t_1 t - t^2\right)\frac{I_1 - I_N}{t_N - t_1}\frac{1}{I_1}\right) \tag{6.8}$$

which is a second order polynomial with a parabola shape as shown in Figure 6.2.

There are several degrees of freedom in (6.8) but we have at least (6.4) as a limiting condition. It would seem optimal to place the min-point at U=0 which means that the derivative of $U_{ref}(t)$ should be zero when $U_{ref}(t)$ is zero. The derivative of (6.6) is

$$\frac{d}{dt}U_{ref}(t) = -\frac{U_0 - U_1}{t_1}\left(f(t) + t\,f'(t)\right) \tag{6.9}$$

which is zero when

$$f(t) + t\,f'(t) = 0 \tag{6.10}$$

From (6.7) we find that minimum occurs at

$$t_m = \frac{t_1}{2} + \frac{I_1}{I_1 - I_N}\frac{t_N - t_1}{2} \tag{6.11}$$

and using $U_{ref}(t_m)=0$ we get

$$U_1 = U_0\left(1 - \frac{t_1}{t_m f(t_m)}\right) \tag{6.12}$$

Assume that we use the linear mapping of time onto the index i so that

$$t(i) = t_1 + (i - 1)t_d \tag{6.13}$$

Substituting t_N, I_1, and I_N using (6.13), in (6.8) gives us

$$U_{ref}(t) = U_0 - \frac{t_1 N + \left(N t_d - t_1\right)(i - 1) - (i - 1)^2 t_d}{t_1 N}\left(U_0 - U_1\right) \tag{6.14}$$

The formula for the first threshold, (6.12), is now

$$U_1 = U_0\left(\frac{Nt_d - t_1}{Nt_d + t_1}\right)^2 \tag{6.15}$$

We also know that $U_1 = U_0 - kI_1t_1$ which gives us

$$(Nt_d + t_1)^2 = 4Nt_d\frac{U_0}{kI_1} = 4Nt_d\, t_I \tag{6.16}$$

where t_I is defined as the time it takes for the highest detectable intensity, I_1, to reach the zero level, which is equal to t_1 if $U_1 = 0$.

$$t_I = \frac{U_0}{kI_1} = \frac{U_0}{U_0 - U_1}t_1 \tag{6.17}$$

Thus, using (6.16), t_1 can now be written as

$$t_1 = \sqrt{4Nt_d\, t_I} - Nt_d \tag{6.18}$$

which yields

$$N = \frac{2t_I - t_1 + 2\sqrt{t_I^2 - t_It_1}}{t_d} \approx 4\frac{t_I}{t_d} \quad \text{for } t_I \gg t_1 \tag{6.19}$$

Hence, the maximum number of levels N is four times the quotient between the time it takes for the highest intensity to completely discharge the photodiode, t_I, and the loop time, t_d.

Figure 6.2 shows a case with N=8. The intensities I_1, $¾I_1$, $½I_1$, and $¼I_1$ are marked in the Figure and one can easily see that if t_d is doubled then N is halved, as in Equation (6.19).

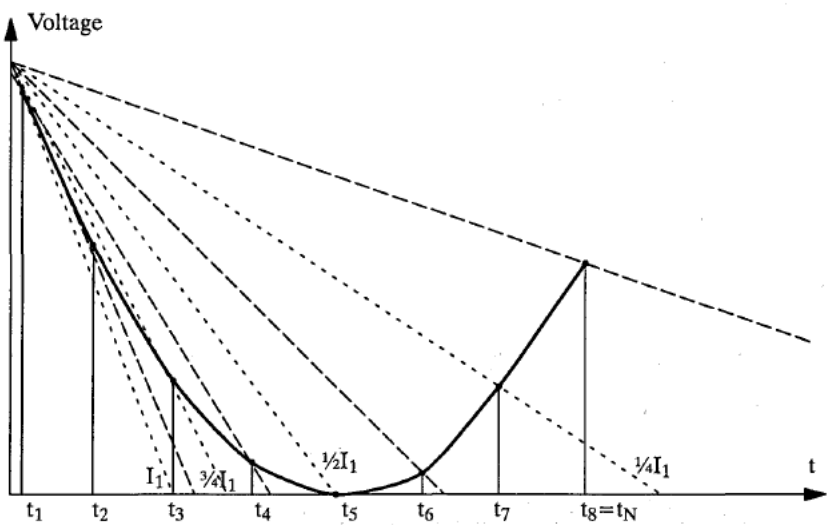


Figure 6.2. Illustration of where the samples are taken

6.3 Non-linear mapping

The general formula (6.6) for $U_{ref}(t)$ will now be used to obtain a pre-specified non-linear function I(t). We notice that U_i still must satisfy (6.4) and to make sure that we have a thermometer output from the comparators, i.e. that once the comparator have changed from 1 to 0 it will stay there, we must also satisfy the requirement of being monotonically which means that

$$\frac{d}{dt}I(t) = \frac{d}{dt}\left[I_1 f(t)\right] < 0 \Rightarrow \frac{d}{dt}f(t) < 0 \tag{6.20}$$

As an example let us prescribe the following **exponential relation between I and t.**

$$I(t) = I_1 e^{-A(t-t_1)} = I_1 f(t) \tag{6.21}$$

In order to use the maximum dynamic we use relation (6.10) which gives us

$$e^{t_1}\left(e^{-At_m} - At_m e^{-At_m}\right) = 0 \quad \Rightarrow \quad t_m = \frac{1}{A} \tag{6.22}$$

When inserted into (6.6) we get

$$U_{ref}\left(\frac{1}{A}\right) = U_0 - \frac{U_0 - U_1}{t_1 A} e^{At_1 - 1} = 0 \tag{6.23}$$

and

$$U_1 = U_0\left(1 - t_1 A e^{1-At_1}\right) \tag{6.24}$$

By combining (6.6), (6.21), and (6.24), we finally get the reference voltage

$$U_{ref}(t) = U_0\left(1 - tAe^{-At+1}\right) \tag{6.25}$$

7 Error analysis

7.1 Error model

The basic NSIP sensor element is shown again in Figure 7.1. For the present purpose we will model its function slightly more detailed as follows. The incoming light intensity I creates a photodiode current Ig, where g is the photoelectric gain. This current discharges the capacitor C so that the voltage U varies over time as

$$U(t) = U_0 - \frac{gI}{C}t \tag{7.1}$$

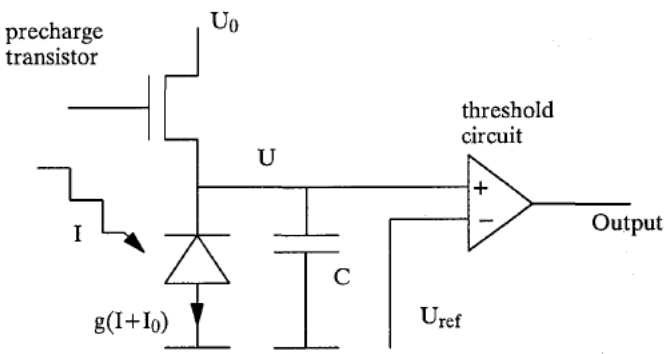


Figure 7.1. An NSIP sense element

The photodiode has a dark current which we prefer to model as an (unwanted) extra light intensity contribution I_o.

$$U(t) = U_0 - \frac{g(I + I_0)}{C}t \tag{7.2}$$

Thus, the time it takes to reach a certain voltage U for a certain intensity is

$$t(I) = \frac{C}{g(I + I_0)}(U_0 - U) \doteq \frac{C}{g(I + I_0)}U_d \tag{7.3}$$

where U_d is the discharge voltage. Due to imperfectness of the manufacturing process the photodiodes will have different characteristics. To map these effects on the time axis we differentiate (7.3) with respect to the five parameters C, g, I, I_0, and U_d, and take the absolute value of each term.

$$\Delta t = t\left(\frac{\Delta C}{C} + \frac{\Delta g}{g} + \frac{\Delta U}{U_d} + \frac{\Delta I}{I + I_0} + \frac{\Delta I_0}{I + I_0}\right) \tag{7.4}$$

We interpret the differentials in (7.4) as follows.

ΔC	deviation in uniform capacitance C
ΔU	deviation in comparator offset plus voltage drop over the precharge transistor
Δg	deviation in photoelectric gain
ΔI	deviation in incoming light
ΔI_0	deviation in dark current

The deviation in incoming light will hereafter be omitted since we are only modeling the measuring system and not unwanted noise in the image itself. The fourth term in (7.4) may therefore be omitted. Also, we replace $I+I_0$ using (7.3) which gives us

$$\Delta t = t\left(\frac{\Delta C}{C} + \frac{\Delta g}{g} + \frac{\Delta U}{U_d} + \frac{\Delta I_0}{CU_d} gt\right) \doteq t\frac{\Delta U_{tot}}{U_d} \tag{7.5}$$

where Δt is the uncertainty in time and ΔU_{tot} is the total uncertainty in voltage at the comparator input at time t. The uncertainty in time Δt increases linearly with time, except for the contribution of the dark current which grows quadratically. Normally, it is possible to reduce the uncertainty by using a low threshold voltage U_{ref}, which in (7.5) means a large U_d. Very long exposure times will be prohibited by the variation in the dark current, which grows quadratically with t.

7.2 Variable sampling time

To avoid ambiguity, we will now change the notation for the interrogation loop index from i to n. Also, for the time being we will allow ourselves to treat n as a continuos variable.

No assume that the loop index n is a monotonous function n(t) so that

$$\Delta t = \Delta n\frac{dt}{dn} \tag{7.6}$$

Let us now identify the quantity Δt in (7.6) with the uncertainty Δt in (7.5). The combination yields

$$\Delta n = \frac{dn}{dt}\, t\left(\frac{\Delta C}{C} + \frac{\Delta g}{g} + \frac{\Delta U}{U_d} + \frac{\Delta I_0}{CU_d} gt\right) \tag{7.7}$$

Equation (7.7) expresses the amount of uncertainty, i.e. error liability in the loop index as a function of time t, the mapping function n(t), the physical parameters C and g of the sensor model, the dark current I_0, and the globally controlled parameter U_d. The latter is assumed constant over time.

For best performance it seems reasonable to require that Δn is constant over time. This is not withstanding the obvious fact that Δn is decreasing with increasing $U_d = U_0 - U_{ref}$. Let U_{ref} be constant over time. To simplify the formulas, let

$$a = \frac{\Delta C}{C} + \frac{\Delta g}{g} + \frac{\Delta U}{U_d} \tag{7.8}$$

$$b = \frac{\Delta I_0}{CU_d} g \tag{7.9}$$

so that

$$\Delta n = \frac{dn}{dt} t(a + bt) \tag{7.10}$$

which is constant if

$$\frac{dn}{dt} = \frac{1}{t(a + bt)} = \frac{1}{at} - \frac{b}{a^2} \frac{1}{1 + \frac{b}{a}t} \tag{7.11}$$

The differential equation (7.11) has the solution

$$n(t) = \frac{1}{a} \ln t - \frac{b}{a^2} \ln\left(1 + \frac{b}{a}t\right) \tag{7.12}$$

For the case with no dark current (b=0), n(t) is a simple logarithmic function as illustrated by curve (b) in Figure 7.2. With the dark current, the mapping function will be like curve (a). The curve (c) illustrates the linear mapping, described in section 5.

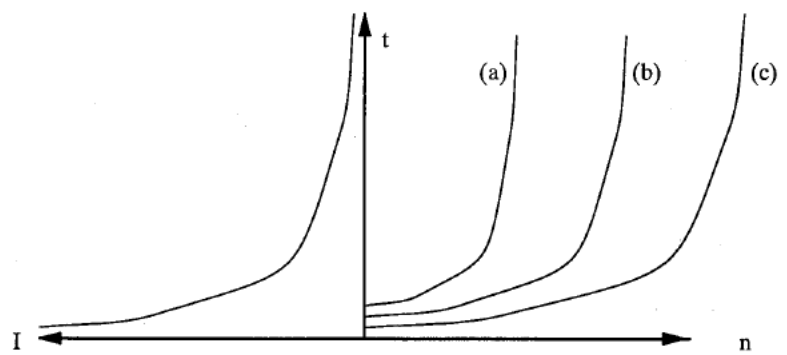


Figure 7.2. The relation between index, time, and intensity for different variations

7.3 Time variable threshold

For a fixed threshold voltage U_{ref}, we have an uncertainty in time of Δt given by (7.5) and (7.7). With a variable U_{ref}, let the derivative of $U_{ref}(t)$ in Figure 7.3 be $U_{ref}'(t)$. As before the slope of U(t) is

$$U'(t) = -\frac{Ig}{C} = -\frac{U_d}{t} \tag{7.13}$$

Let the differential ΔU_{tot} be defined as in (7.5), i.e.

$$\Delta U_{tot} = U_d\left(\frac{\Delta C}{C} + \frac{\Delta g}{g} + \frac{\Delta U}{U_d} + \frac{\Delta I_0}{CU_d} gt\right) \tag{7.14}$$

Then, from Figure 7.3, in the vicinity where $t=t_{ref}$, $U=U_{ref}$, and $U_d=U_0-U_{ref}$, we have

$$\Delta t = t\frac{\Delta U_{tot}}{U_{ref}' - U'} = t\frac{\Delta U_{tot}}{t\,U_{ref}' + U_0 - U_{ref}} \tag{7.15}$$

which is a generalization of (7.5). Combining (7.14) and (7.15) yields

$$\Delta t = \frac{t\left(\frac{\Delta C}{C} + \frac{\Delta g}{g}\right)(U_0 - U_{ref}) + t\Delta U + t^2\Delta I_0\frac{g}{C}}{t\,U_{ref}' + U_0 - U_{ref}} \tag{7.16}$$

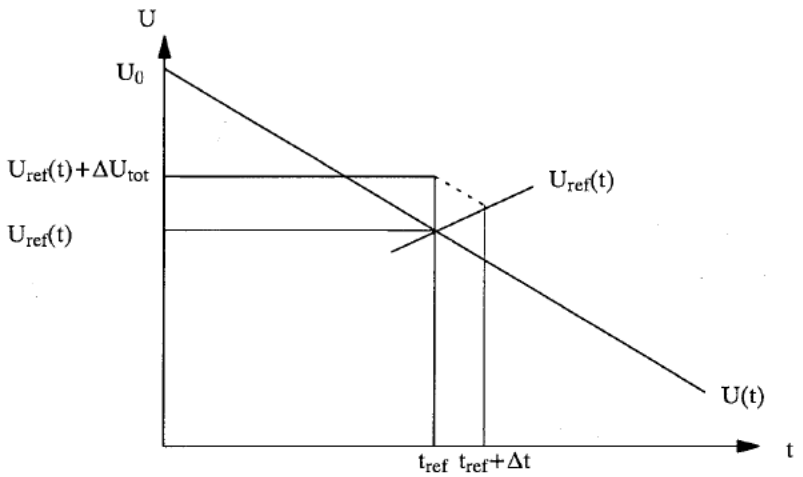


Figure 7.3. The effect on the uncertainty in time when moving U_{ref}

It would be tempting to find a function $U_{ref}(t)$ which would make Δt constant in (7.16). This is equivalent to find a solution to the following differential equation

$$\left(t\,U_{ref}' + U_0 - U_{ref}\right)\cdot const = t\left(\frac{\Delta C}{C} + \frac{\Delta g}{g}\right)(U_0 - U_{ref}) + t\Delta U + t^2\Delta I_0\frac{g}{C} \tag{7.17}$$

Lacking a closed form solution to this problem, we will evaluate some particular viable cases. The case of constant U_{ref} is already treated above and is a special case of (7.17). The second simplest form of $U_{ref}(t)$ function is a straight line.

$$U_{ref} = \frac{t - t_1}{t_N - t_1}U_0 \tag{7.18}$$

as illustrated in Figure 7.4. From (7.18) we have

$$U_{ref}' = \frac{1}{t_N - t_1}U_0 \tag{7.19}$$

and inserted in (7.16) we obtain

$$\Delta t = t\left(\frac{\Delta C}{C} + \frac{\Delta g}{g}\right)\frac{t_N - t}{t_N} + t\frac{\Delta U}{U_0}\frac{t_N - t_1}{t_N} + t^2\frac{\Delta I_0}{U_0}\frac{g}{C}\frac{t_N - t_1}{t_N} \tag{7.20}$$

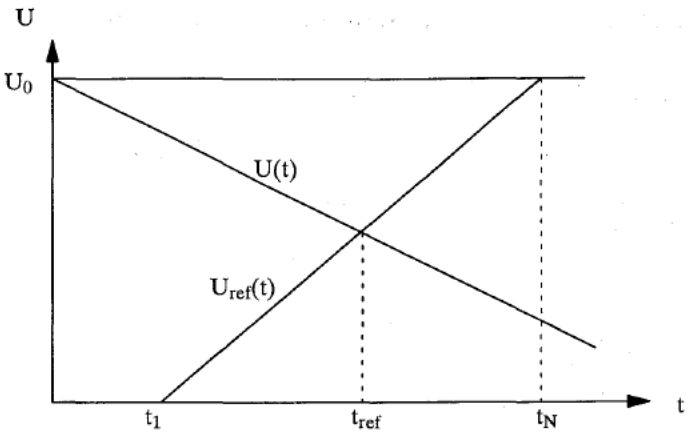


Figure 7.4.

The uncertainty for $t=t_1$ is

$$\Delta t = \frac{t_N - t_1}{t_N} t_1 \left(\frac{\Delta C}{C} + \frac{\Delta g}{g} + \frac{\Delta U}{U_0} + t_1 \frac{\Delta I_0 g}{U_0 C} \right) \tag{7.21}$$

and for $t=t_N$

$$\Delta t = (t_N - t_1) \left(\frac{\Delta U}{U_0} + t_N \frac{\Delta I_0 g}{U_0 C} \right) \tag{7.22}$$

$\Delta t(t_1)$ is equal to $\Delta t(t_N)$ for

$$\frac{\Delta C}{C} + \frac{\Delta g}{g} = \frac{t_N - t_1}{t_1} \frac{\Delta U}{U_0} + \frac{t_N^2 - t_1^2}{t_1} \frac{\Delta I_0 g}{U_0 C} \tag{7.23}$$

For $t_N=N^*t_1$ we have

$$\frac{\Delta C}{C} + \frac{\Delta g}{g} = (N-1) \frac{\Delta U}{U_0} + (N^2 - 1) t_1 \frac{\Delta I_0 g}{U_0 C} \tag{7.24}$$

which gives us a relation between the different uncertainty contributions. Another relation is obtained in the case when $t_N=t_1+(N-1)t_d$.

$$\frac{\Delta C}{C} + \frac{\Delta g}{g} = \frac{t_d (N-1)}{t_1} \frac{\Delta U}{U_0} + \frac{(N-1)^2 t_d^2 - 2(N-1) t_d}{t_1} \frac{\Delta I_0 g}{U_0 C} \tag{7.25}$$

Thus, from (7.24) we see that the relative uncertainty in the photoelectric gain Δg and the capacitor ΔC should be in the order of N times greater than the uncertainty in the comparator ΔU and a factor N^2 time greater than the uncertainty in the dark current ΔI_0 in order to have a constant total uncertainty in read-out time Δt. Equation (7.25) can be interpret in the same manner, given the quotient t_d/t_1.

8 Grayscale morphology

Grayscale morphology is an extension of binary morphology [15]. In binary morphology various operators are applied to a binary image. Typical operators are expansion, erosion, and thinning. In grayscale morphology these operators are generalized to operate on grayscale images where the image is viewed as a terrain map and the pixel value is interpreted as height. Such representation of the pixel value is called the **Umbra** transform. An umbra transform $\mathcal{U}$ of a function f(x) is defined as

$$\mathcal{U}(f(x)) = u(x,y) = \begin{cases} 1 & y \geqq f(x) \\ 0 & y \leqq f(x) \end{cases} \tag{8.1}$$

which is illustrated in Figure 8.1 where (a) represents the original function and (b) the u(x,y) function. Note that u(x,y) is a binary 2D function. In a similar fashion, the Umbra transform of a 2D function f(x,y) defines a binary 3D function u(x,y,z).

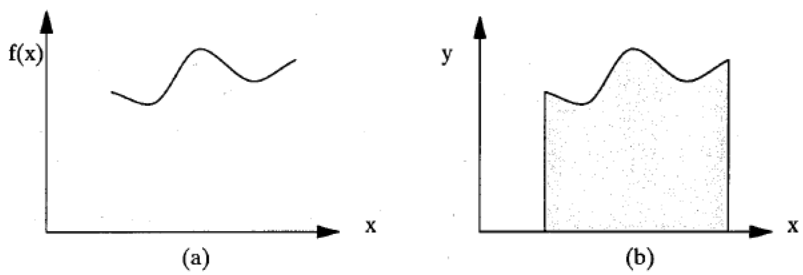


Figure 8.1. The Umbra transform

NSIP is particularly well adapted for grayscale morphology. The reason is that the output from the individual pixel comparators as a function of time corresponds to the Umbra representation of an image. Figure 8.2(a) shows a coarsely digitized 1D grayscale signal, where the Y-axis represents light intensity. The Umbra representation of this signal is shown in Figure 8.2(b). At time t=0, no pixels have passed the threshold. When t=1 the comparator with the pixel corresponding to the highest intensity in Figure 8.2(a) is set. At t=2 another comparator is set, and at t=3 three new comparators are set, etc. As pointed out in section 3.4 this is the same as the stack or thermometer representation [17].

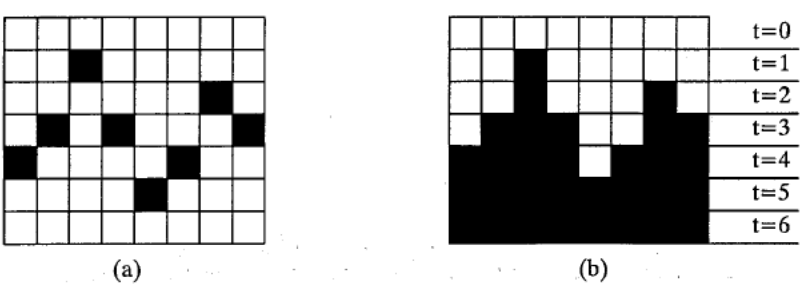


Figure 8.2. The comparator output viewed as an Umbra transform

In the following sections we will present some grayscale morphological operations and show how they are implemented in a NSIP system.

As described in section 3.4 we can perform any type of rank order filtering, for instance max or min, using NSIP techniques. These operations applied on an umbra transformed image is grayscale *dilation* and *erosion* respectively.

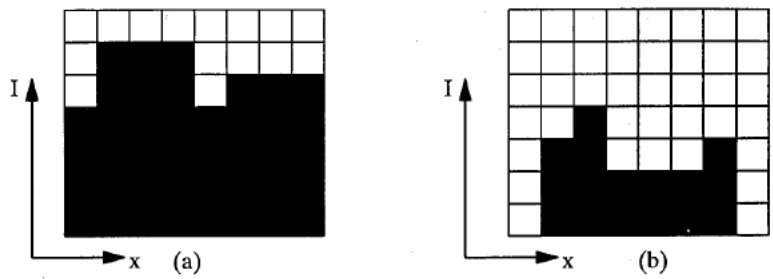


Figure 8.3. Dilation and erosion

Figures 8.3(a)&(b) show the result after dilation and erosion, using the image input of Figure 8.2(a) and the structure element shown in Figure 8.4. The structure element works in the following way. For dilation, the structure element will set the pixel to **1** if the center pixel is already set (white circle) or if either one or both of the two neighbors is set (black circles). For erosion, the structure element will keep the pixels set (in the white circle position) if both the two neighbors also is set (black circles), otherwise it will be changed to **0**.

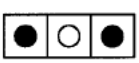


Figure 8.4. A structure element

Program for grayscale dilation:

```
REPEAT
   ldi pd                    ; invert pd-value
   st r0
   ldi (000) r0              ; dilation operation
   st r1
   ..
   ld r0
UNTIL COUNT=N
```

The operation *ldi (000) r0* performs a dilation since

$$a + b + c = \overline{\bar{a}\&\bar{b}\&\bar{c}} \tag{8.2}$$

In the same way as for binary images *closing* and *opening* are combinations of erosion and dilation. Figures 8.5(a)&(b) show the result of closing and opening applied to the signal in Figure 8.2(a).

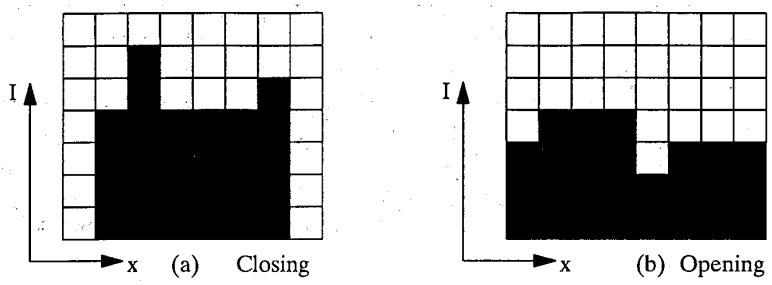


Figure 8.5. Closing and opening

Program for grayscale closing:

```
REPEAT
   ldi pd
   st r0
   ldi (000) r0          ; dilation
   ld (111) a            ; erosion
   st r1
   ..
   ld r0
UNTIL COUNT=N
```

The input data to be operated upon by the structuring element of Figure 8.4 are all coming from the same interrogation event. Figure 8.6 illustrates three different structuring elements which utilizes input data from three successive cycles i, i+1, and i+2.

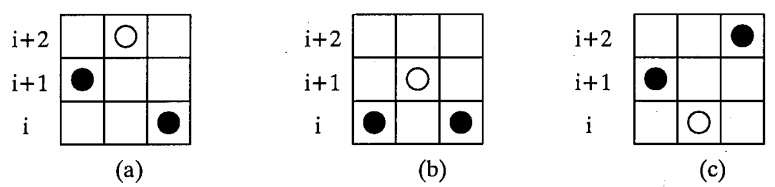


Figure 8.6. Structure elements both in the x and time dimension

If we erode the image of Figure 8.2 with the three operators in Figure 8.6, the result is as shown in Figure 8.7, where the dark area represents the result and the shaded pixels the eroded area.

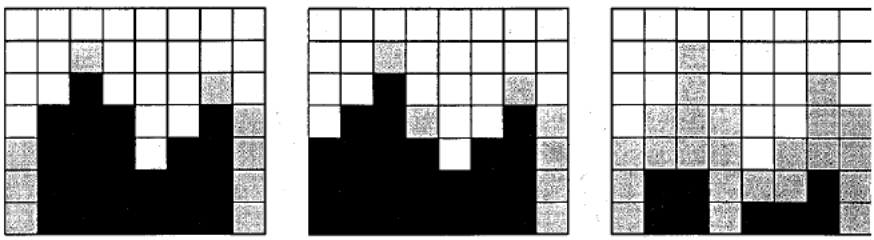


Figure 8.7. The effect of dilation and erosion using the structure elements in Figure 8.6

An approximation of the Rolling ball operator (Sternberg [15]) is shown in Figure 8.8. The following NSIP program using this element for dilation consists of two parts as shown below. The first part computes the output, the second part rearranges register data for the next interrogation and computation cycle. The vertical size of the operator determines the number of registers to be employed since this size indicates how many cycles back in time the output is depending on.

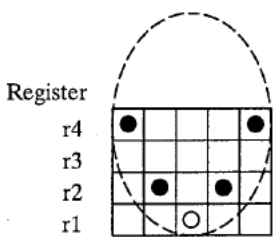


Figure 8.8. A "Rolling ball" operator

```
clear register r2..r4
REPEAT
   ldi pd
   st r1
   or (1xx) r4              ; Compute dilation
   or (xx1) r4
   ori (0x0) r2
   st r0                    ; Dilation output
   ..
   ld r3                    ; Shift rows
   st r4
   ld r2
   st r3
   ld r1
   st r2
   ld r1                    ; Load input
UNTIL COUNT=N
```

9 Linear convolution

In this section we will utilize grayscale pixels where the grayscale is represented by the interrogation index i. As described in section 2.1 we do not automatically get a linear relation between the time t and the intensity I, but using the strategies described in sections 5 & 6 we can make the relation linear. For the sake of simplicity, however, we omit the mapping procedures here and assume that they can be introduced wherever wanted.

The time, or rather the interrogation index, elapsed from the precharging of a photodiode till the diode passes the threshold is measured as follows (also shown in the program below). A stack of registers is used as a counter and the incrementing bit from the photodiode, is placed in the accumulator. If the counter is incremented more than 256 times the counter would normally wrap around back to zero. This is prevented by a few extra lines of code and an overflow bit r8 which is set when the counter reaches 256. The number of instructions will average to about 3 per intensity level if Gray-code representation is used.
Program for obtaining gray level pixels in natural binary code:

```
clear register r0..r8
initpd
ld pd
REPEAT
   increment(r0..r7)
   ld c                    ; Generate overflow ...
   or r8
   st r8                   ; stored in r8
   ld pd
UNTIL COUNT = 0
```

The result in r0..r7 will have an inverted gray-scale since a high intensity corresponds to a low interrogation index and vice versa. To produce a non-inverted result, we can either decrement a fixed value (in this case 255 with underflow check), or we can change *ld pd* to *ldi pd* which means that the counter counts from passing the threshold until exit from the loop.

In section 3.3 we described a positive gradient detection routine where the left edges in the input image were marked. We will now modify this routine to measure the steepness of the gradient. As long as the center pixel has not passed the threshold the counter is idle, which correspond to the situation in Figure 9.1(a). When the pixel passes the threshold, corresponding to (b), we continue to increment as long as the left neighbor has not passed the threshold, (c).

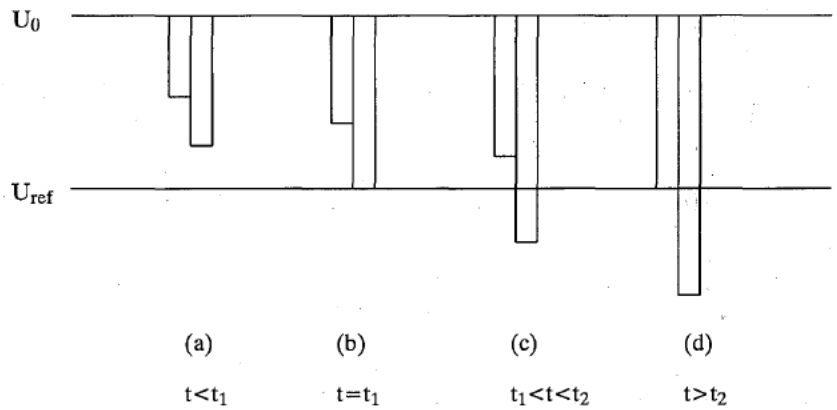


Figure 9.1. Four different time instances during discharge of the photodiode

Program for computation of positive gradients.

```
clear register r0..r4
initpd
ld pd
REPEAT
   ld (10x) a
   increment(r0..r3)
   r4 = overflow check(carry,r4)
   ld pd
UNTIL COUNT = 0
```

Compared to the previous program we use a smaller number of bits in the counter (r_0–r_4) since the dynamic range is expected to be less for the gradient than for the absolute values. An overflow check is inserted.

By adding the instruction

```
or (01x) pd
```

before *increment(r0..r3)* the counter will be incremented as long as there is a left or a right edge. Hence, this modified routine computes the absolute value of the gradient.

Figure 9.2 shows a 1D example of an Laplacian convolution kernel.

1	−2	1

Figure 9.2. A 1D Laplacian operator

The straight-forward way to compute this convolution is to do the following in the interrogation loop. Increment the stack of registers by 1 if the left neighbor has passed the threshold. Increment this value by 1 if the right neighbor has passed the threshold. Decrement this value by 2 if the center pixel has passed the threshold.
Program for Laplacian.

```
clear register r0..r7
initpd
ld pd
REPEAT
   ldi (1xx) pd
   increment(r0..r7)
   ldi (xx1) pd
   increment(r0..r7)
   ldi pd
   decrement(r1..r7)          ; −2 = −1 shifted one step
   ld pd
UNTIL COUNT = 0
```

In most cases it is faster to do the convolution by performing a delta computation ahead of the incrementation. As long as none of the pixels in a neighborhood has passed the threshold no increment should take place, i.e. $\Delta:=0$. If one of the neighbors has passed the threshold but not the center pixel or the other pixel, then $\Delta:=1$. For the similar obvious reasons it is understood that the total incrementation to be done in the loop can take the values $\Delta:=-2, -1, 0, 1, 2$. Thus, the delta value can be represented by 3 bits. Figure 9.3 shows four different time instances and the corresponding delta values.

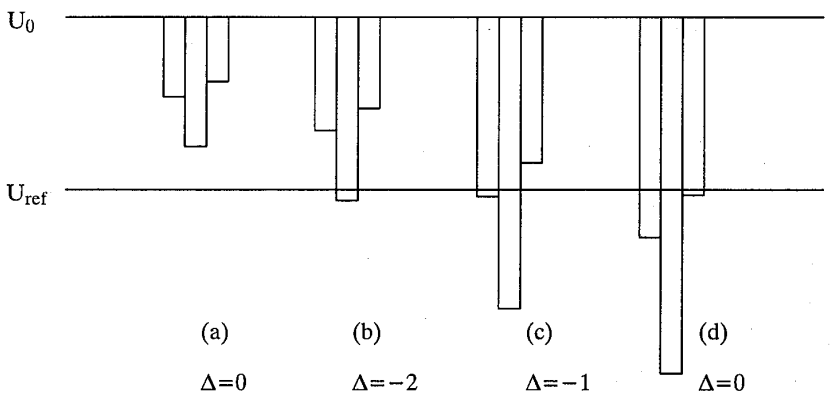


Figure 9.3. An illustration of the delta-increment method

Program for Laplacian using Δ-increment.

```
initpd
ldi pd
REPEAT
   st r0
   generate delta value in r1..r3 given the comparator value in r0
   add(r1..r3,r4..r11)
   ldi pd
UNTIL COUNT = 256
```

The word-length of the final value has been set to 8 bits. The input data is stored in r0, the delta value in r1..r3, and the final value in r4..r11.

The number of bits b_1 for the delta value is determined from the coefficients in the kernel. Given the arbitrary n-sized convolution kernel

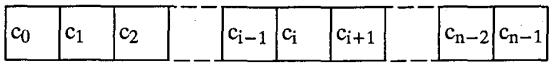


Figure 9.4. An arbitrary 1D convolution kernel

the formula for the number of bits b_1 is

$$b_1 = \left[{}^2\log\left[\sum_i |c_i|\right]\right] + 1 \tag{9.1}$$

To see when the delta-increment method is more efficient than the straight-forward method, we assume the following parameters.
i) An addition takes **3b** cycles.
ii) Each of the coefficient can be described by $\mathbf{b_2}$ bits.
iii) There are **n** coefficients.
iv) The final result contains $\mathbf{b_3}$ bits.

Then, with the straight-forward method, the cycle count is

$$S_{S\text{-}F} = 3nb_3 \tag{9.2}$$

For the delta-increment method the count yields

$$S_{\Delta} = 3(n-1)(\log n + b_2) + 3b_3 \tag{9.3}$$

The difference

$$S_{S\text{-}F} - S_{\Delta} = 3(n-1)b_3 - 3(n-1)(\log n + b_2) \tag{9.4}$$

is equal to or greater than zero when

$$b_3 \geq \log n + b_2 \tag{9.5}$$

From (9.1) we know that $b_1 = b_2 + \log n$. Thus, as the word-length of the final result must be larger than the Δ-value, the delta-increment method is always faster.

10 2D NSIP model

10.1 Programming model

The 2D NSIP system to be described below is largely a generalization of the LAPP1100. A floorplan for a possible 2D NSIP chip is shown in Figure 10.1. The SPE's are distributed along the 2D array in a square grid. As in the 1D case the 2D NSIP SPE consists of a photo sensor, a analog comparator, a digital processor, and memory in the form of binary registers.

The photo sensor (photodiode) and the comparator work in the same way as in the 1D system. However, the rest of the VLSI implementation is different as the area available for each SPE has to be smaller. Therefore, the processing power is likely to be smaller and we may have to accept that some of the single cycle instructions in LAPP 1100 have to be performed in a number of cycles in the 2D case.

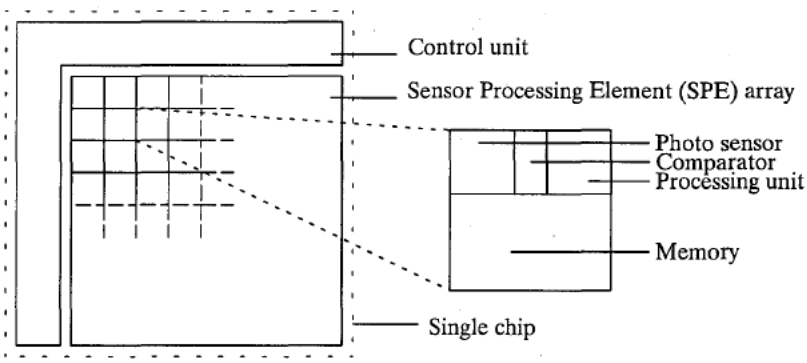


Figure 10.1. A floor plan of a 2D NSIP architecture

Figure 10.2 shows the logic function of the SPE which is a 2D generalization of Figure 2.6. With a few exceptions, the instructions for the PLU (Point Logic Unit) will be the same. The NLU (Neighborhood Logic Unit) in receives data from its four nearest neighbors and the output from the GLU. Therefore, the NLU mask which controls the operation has five positions. An NLU-mask in an instruction like

ld (abcde) pd

corresponds to the neighborhood shown in Figure 10.2. Each parameter (a,b,c,d,e) takes one of the values (0, 1, X (=don't care)).

This means that the instruction

ld (x10xx) pd

will perform the same operation as

ld (10x)

in the 1D case.

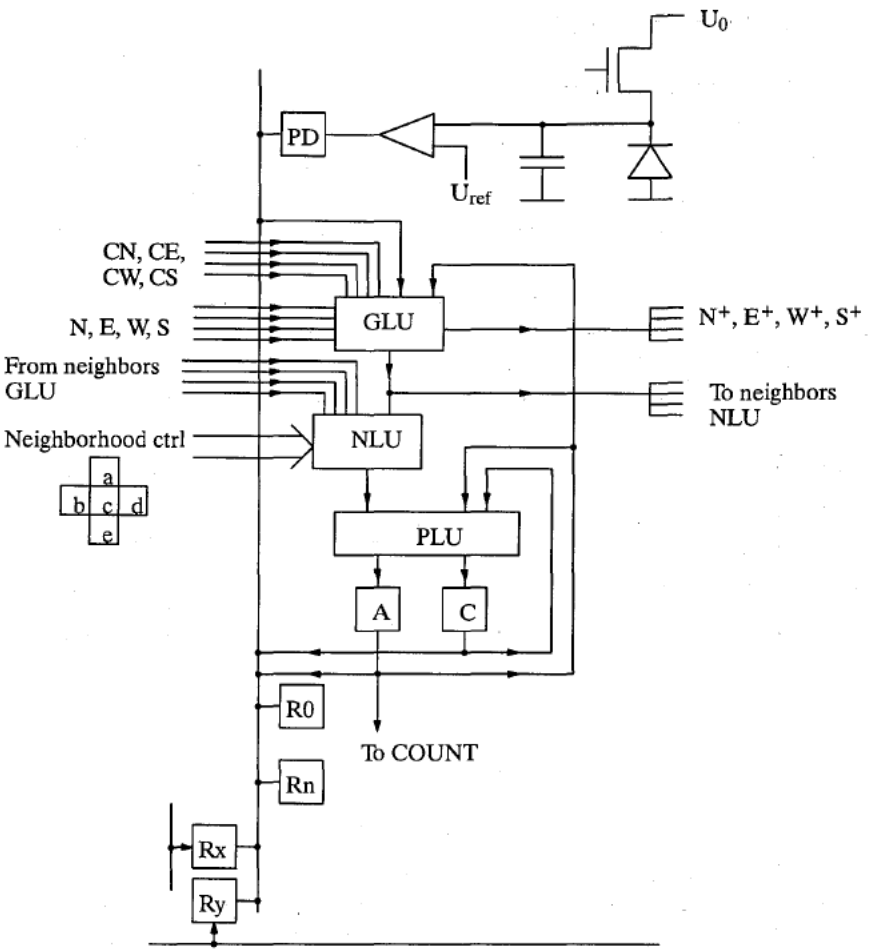


Figure 10.2. The programmers model of one SPE

The 2D GLU (Global Logic Unit), shown in Figure 10.2, performs global operation with the bus and the accumulator as input operands. The output is the result after the global operation, the bus, or the accumulator. Unlike in the 1D LAPP1100, the GLU neighbor connections are controlled in a way which makes it possible to select certain propagation directions across the array. The architecture differs a great deal from the 1D case and is described in more detail in section 11.

The memory size will will not be decided here although it is reasonable to assume a maximum number of bits between 16 and 32. The memory includes two special registers Rx and Ry, which are connected to a horizontal bus and a vertical bus respectively.

It should be emphasized that the diagram of Figure 10.2 is strictly a programmers model. What the programmer sees as one single-step operation will often be executed as a multi-

step sequence in a vastly different manner. Several of the connections and functional units in Figure 10.2 will actually be implemented by the same piece of hardware as will be shown in section 13 below.

10.2 Generation of coordinates

As shown by Figure 10.2, each SPE is connected to one vertical and one horizontal bus. A similar two-dimensional data broadcasting network exist in the DAP array processor [14]. It would be quite feasible to connect the bus-connections available at the array edge to a broad-bandwidth data channel for all types of communications to and from the array. Note that if we want to load data into a specific row of the array we can use the vertical buses and do the selection with a proper control word on the horizontal bus.

We propose a somewhat more limited use of the buses and the Rx, Ry registers. In Figure 10.3 is shown a 16x16 array with 16 horizontal and 16 vertical buses. At the left and upper edge of the array are two Read-Only-Memories (ROM), four bits wide, which contain nothing but the row and column number in ordinary binary code. Clearly, if these numbers were transported into the array, the combined result is that each SPE would have access to its own coordinates. Thus, as soon as our application contains instructions or algorithms which depend on the position inside the array, we may expect to be able to make use of this feature. Actually, as will be shown below, in many cases it is not necessary to make the full (x,y)-coordinate available to the SPE's.

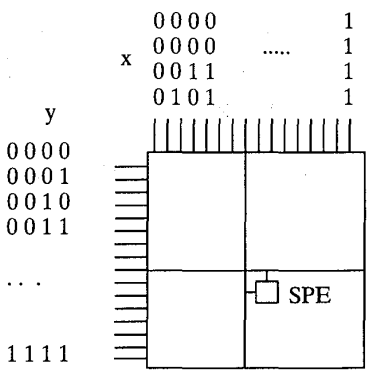


Figure 10.3. Position constants

In the following programs, the bit-planes in the ROMs to be connected to the horizontal and vertical buses, are set with the instructions

setrx <bit-plane x>
setry <bit-plane y>

The content of the buses will be available in memory cells rx and ry in the SPE.

In the first example we want to select a region-of-interest which consists of the lower right hand corner, shown in Figure 10.4. Assuming that (x,y)=(0,0) in the upper left hand corner, this is equivalent to say that only SPE's which have the MSB set in **both** x and y should be set. The following program produces such a mask array.

```
setrx n-1          ; MSB in x
setry n-1          ; MSB in y
ld rx
and ry
st a,r0            ; Store mask in r0
```

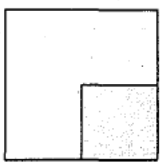
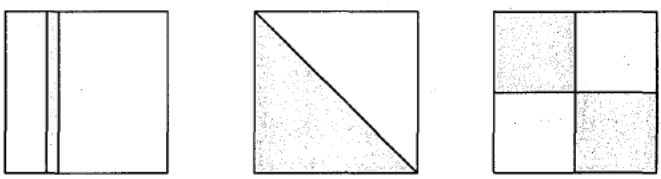


Figure 10.4. Region of interest. SPE's in shaded area are selected

By using more sophisticated formulas than the previous, it is possible to achieve regions-of-interests as shown in Figure 10.5.



(a) (b) (c)

Figure 10.5. Different regions of interest patterns

The vertical line in Figure 10.5(a) is achieved with the following program.

```
clr a
FOR i=0 TO n-1
   setrx i
   IF Pattern[i] = 1 THEN
      and rx
   ELSE
      andi rx
   END-IF
END-FOR
st a,r0                    ; Store mask in r0
```

Here, the Pattern[i] is a bit vector which defines the position of the column. For instance, if we want to use column 34, the vector Pattern will look like [0, 1, 0, 0, 1, 0, 0, 0].

Figure 10.5(b) shows a region where the y-position is greater than the x-position each SPE. This is achieved in the following program by subtracting the y-position from the x-position. If the final carry bit (=borrow) is set, the result x−y is negative and the SPE belongs to the region-of-interest.

```
clr a
st r1
xori a                          ; Carry = 1
st r1
FOR i=0 TO n-1
   setrx i
   setry i
   r0=rx&ry+rx&r1+ry&r1         ; Carry
   r1=r0                        ; New carry
END-FOR
                                ; Mask in r0
```

Figure 10.5(c) is a union (=OR) between two regions. The following program computes the mask for each individual region and then OR them together.

```
setrx n-1                       ; MSB in x
setry n-1                       ; MSB in y
ld rx
and ry
st a,r0                         ; lower right hand corner in r0
setrx n-1                       ; MSB in x
setry n-1                       ; MSB in y
ldi rx
andi ry                         ; upper left-hand corner in accumulator
or r0                           ; upper left-hand OR lower right hand
st a,r0                         ; Store mask in r0
```

In the same manner, the region in Figure 10.6 is obtained by ANDing a series of simple regions.

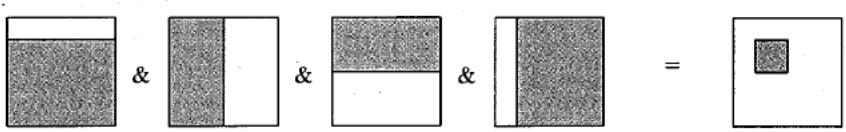


Figure 10.6. Constructing a region-of-interest as an arbitrary rectangle

Obviously, the more irregular the region-of-interest is, the longer is the program to produce it. In the worst case, we might have to define and generate the region as a set of individual 1-pixel regions that are ORed together. The complexity of such a program is $\mathbf{O(N^2 2logN)}$ since there are $\mathbf{O(N^2)}$ pixels each of which requires **2logN** accesses from the ROMs. Fortunately, mask to be used in practice are much simpler to produce.

11 Global operations

11.1 MARK and FILL

Before we discuss two-dimensional global operations, let us reconsider the one-dimensional case as it was presented for MAPP2200 in section 1 in part II of this thesis.

The instructions performed by the Global Logic Unit, the GLU-instructions, have proved to be a very powerful tool both in real applications for LAPP and MAPP, and in our NSIP studies. For instance, the pattern adaptive thresholding in section 4.2 used the GLU instructions *mark* and *fill* to compute the size of the object in the image, and the max detector in section 3.2 used the instruction *lfill* to determine the position of the maximum intensity before the feature extraction.

The basic GLU-instruction in both MAPP and LAPP is *mark*. Other GLU instructions can be derived from *mark*. The mark operation keeps those objects in the operand which overlap in any position any object in the accumulator. The result is placed in the accumulator. For example, given an accumulator and r0 content as in Figure 11.1(a), the instruction *mark r0* will result in an accumulator content as in (b).

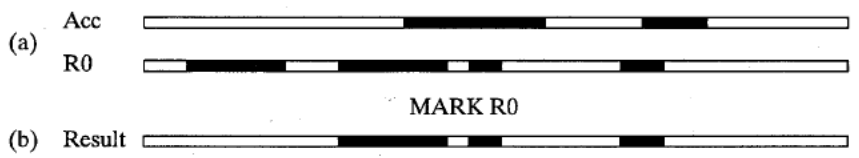


Figure 11.1. The MARK operation

The net that performs these operations is shown in Figure 11.2. Here, Mask and Image represent two register, which in the LAPP1100 case is the accumulator and the operand bus, respectively. First the Mask (accumulator) is applied and then the Image (bus). Positions, i.e. those SPEs, where both the Mask and Image are set will give a "1" at the Result output. This combination will also generate a propagation to the left and to the right, thus setting all the Result bits in those SPEs which are connected to a set Result bit. The Result bit is loaded to the accumulator but not until the next clock cycle.

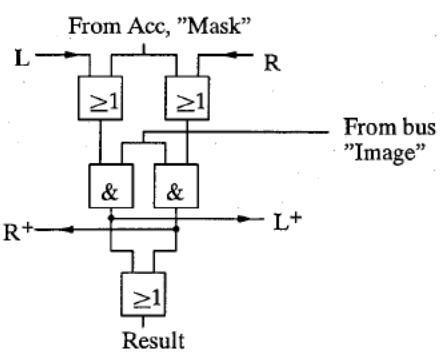


Figure 11.2. The GLU−net in LAPP1100

It might seem possible to reduce the logic of Figure 11.2 to Figure 11.3. However, with a common wire for left and right propagation, latches and oscillations may appear. These effects should also be avoided in the 2D-case.

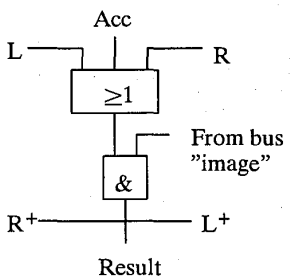


Figure 11.3. Reduced GLU-net in LAPP1100

This reduced GLU-model of Figure 11.3 can be generalized to two dimensions, see Figure 11.4. The four nearest neighbors are connected. The connections are controlled by four control signals (CN,CE,CW,CS). Thus, while we distribute data to neighbors from one point as in Figure 11.3 we safeguard ourselves with extra gating. The potential problems with two-dimensional bi-directional propagation across the array will be discussed below in section 11.2.

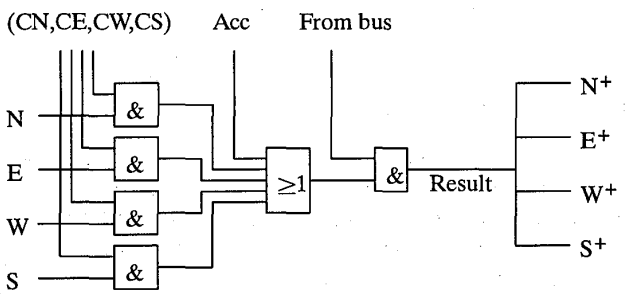


Figure 11.4. A 2D MARK-net with input controls

The gates for direction control can either be placed at the input, as in Figure 11.4, or at the output, as in Figure 11.5. Actually, these two nets are the same. This is illustrated in Figure 11.6 where Figure 11.6(a) corresponds to the net in Figure 11.4 and Figure 11.6(b) to the net in Figure 11.5. As the control bits are addressed individually (represented by the small boxes on the arrows) it does not matter if the direction control is performed on the incoming or the outgoing signal.

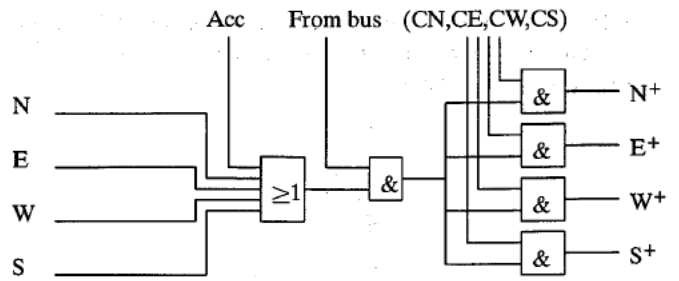


Figure 11.5. A 2D MARK-net with output controls

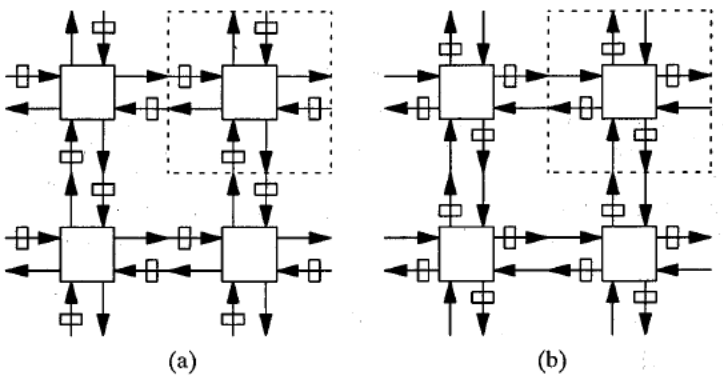


Figure 11.6. The equivalence of input and output control

To perform the 2D operation *mark*, all the bits in the direction control vector (CN,CE,CW,CS) are set to 1, the mask bit is placed in the Mask register (which we will assume to be the accumulator) and the image bit is taken from the bus (which is typically a register). This will initiate a 2D-propagation equivalent to Figure 11.1 and when it is finished, the Result bit can be stored in the accumulator.

In the same manner we can obtain *fill* by just bringing in the inverted value from the bus. As in LAPP and MAPP (see section 1 in part II of this thesis) a fill operation preserves holes rather than objects as determined by the mask, see Figure 11.7.

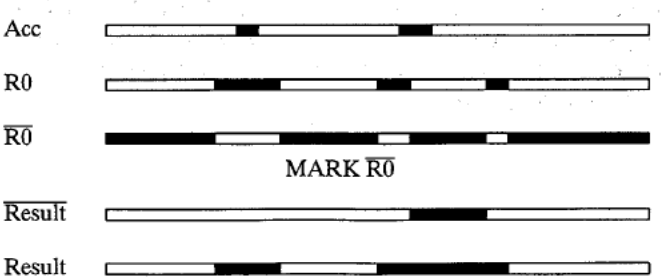


Figure 11.7. The FILL operation

Both **MARK** and **FILL** operations may use instruction specific mask rather than the accumulator content. In the 1D-case we had three choices (lfill, rfill, and lrfill) for these fill operations. Conceptually, it is convenient to think of the fourth choice as the one which was defined by the image in the accumulator. In the 2D-case we have 15 border fill alternative as listed in Table 11.1. They are all controlled by the vector (CN,CS,CE,CW) and they turn out to have some non-trivial effects compared to the 1D-case.

no	CN	CS	CE	CW	Name
0	0	0	0	0	
1	0	0	0	1	rfill
2	0	0	1	0	lfill
3	0	0	1	1	lrfill
4	0	1	0	0	tfill
5	0	1	0	1	trfill
6	0	1	1	0	tlfill
7	0	1	1	1	tlrfill
8	1	0	0	0	bfill
9	1	0	0	1	brfill
10	1	0	1	0	blfill
11	1	0	1	1	blrfill
12	1	1	0	0	tbfill
13	1	1	0	1	tbrfill
14	1	1	1	0	tblfill
15	1	1	1	1	tblrfill (hole_fill)

Table 11.1. Border FILL

Two rather powerful operations can be carried out using the functions in Table 11.1 and will be described below. The first one expands each object to the smallest possible rectangle which fully covers the object. The second one removes such holes which are enclosed by an object and do not touch the image border.

Figure 11.8 shows the operation steps required to expand an object into a circumscribing rectangle. Here, (a) is the original image. First we perform the operation 9 in Table 11.1 which is a *brfill* and set the shaded area in (b). Similarly, the shaded areas in (c), (d), and (e) in Figure 11.8 are set by operations 6, 10, and 5 in Table 11.1. Each of these operations will produce a corner of the final rectangle. When these images are ORed together we obtain the final result in Figure 11.8(f) which is the smallest possible rectangle which covers the object.

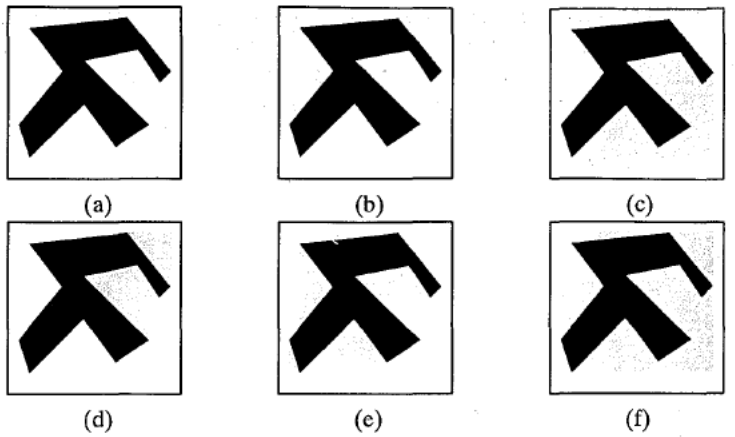


Figure 11.8. Smallest possible rectangle

Normally, this operation works regardless of the number of objects in the image. However, in some cases the rectangles of two objects are overlapping, see Figure 11.9(a), and in yet other cases one rectangle might cover another rectangle completely, see Figure 11.9(b). In the worst case, two objects are connected in a way that they will be regarded as one object by the operator, see Figure 11.9(c).

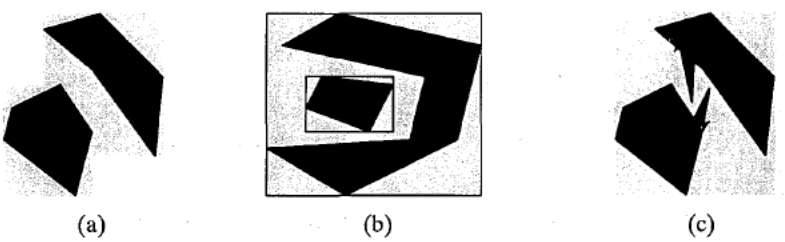


Figure 11.9. Objects which intersects each others rectangles

To remove holes in objects we first use the *tblrfill* operation. As shown in Figure 11.10 this operation sets the background pixels which are connected to the image border.

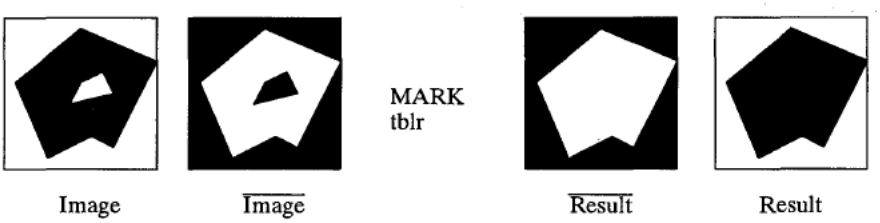


Figure 11.10. Hole filling

A very useful operation is **thresholding with hysteresis**. A 2D NSIP system with the GLU-function MARK is extremely well suited for this task. Two binary images are produced for the same scene using two different threshold. The first one is set so low (i.e. high U_{ref}) so

that all parts of the object come through plus some unwanted noise as illustrated by Figure 11.11(a). The second image image is taken with a high threshold (i.e. a low U_{ref}) that only the brightest parts the intensity objects will come through. This is the binary image of Figure 11.11(b). The final result is obtained with the operation MARK.

(a) (b) MARK(a,b)

Figure 11.11. Thresholding with hysteresis

Program for thresholding with hysteresis.

```
ref:=0.5 * U0          ; High Uref
ldi pd
st a, r0
ref:=0.1 * U0          ; Low Uref
ldi pd
mark r0
```

Pure unmasked propagation is of course also possible with the GLU. It is implemented as a special case of the MARK-operation by simply setting all accumulator bits to 1 along the array. In Figure , once a "seed" has been injected into the global net it will continue to propagate according to what is allowed by the control vector (CN,CE,CW,CS). The possibilities are given by Table 11.2 and the case when the input image contains a single 1-pixel is shown in Figure 11.12.

no	CN	CS	CE	CW	Direction	Name
0	0	0	0	0	No propagation	
1	0	0	0	1	W	wprop
2	0	0	1	0	E	eprop
3	0	0	1	1	E-W (Horizontal)	hprop
4	0	1	0	0	S	sprop
5	0	1	0	1	SW	swprop
6	0	1	1	0	SE	seprop
7	0	1	1	1	~N	hsprop
8	1	0	0	0	N	nprop
9	1	0	0	1	NW	nwprop
10	1	0	1	0	NE	neprop
11	1	0	1	1	~S	hnprop
12	1	1	0	0	N-S (Vertical)	vprop
13	1	1	0	1	~E	vwprop
14	1	1	1	0	~W	veprop
15	1	1	1	1	All	vhprop

Table 11.2. PROP operations

In Figure 11.12, the numbers below each square corresponds to an entry in Table 11.2. The single initially 1, the seed, is placed as in plot number 0.

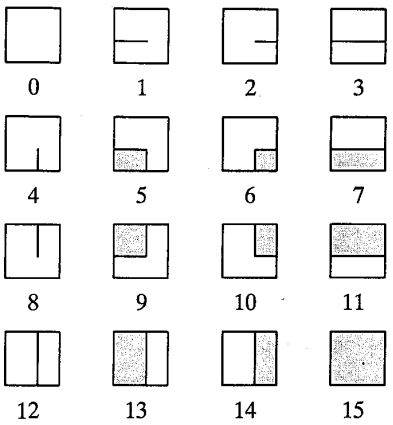


Figure 11.12. PROP propagations

11.2 Safe and unsafe propagation

Consider the one-dimensional GLU-net shown in Figure 11.13. The operation to be perform is LRPROP. The two processors shown which receive nothing but zeroes from left and right and from the mask, should therefore deliver zero at their outputs. However, because of clock-signals and external noise sources we can never rule out the possibility of short false spikes, say, at point (a). Because of the double directed propagation this spike may perpetuate to (b), (c), and (d). Then, the two SPE's latch into a false stable state with both outputs equal to one. Even worse, this false condition will be propagated throughout the array.

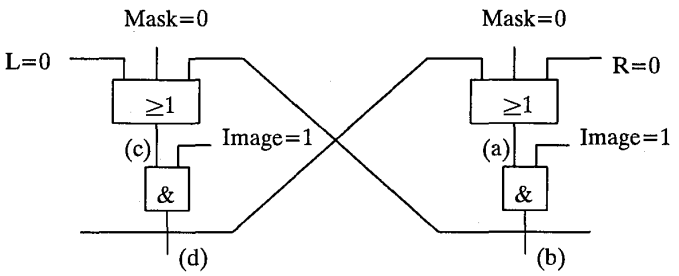


Figure 11.13. Stable error

The underlying physical reason for this effect is the positive feed-back between the two SPEs. No such feed-back can be allowed. For the 2D-case the implication is that we allow the propagation directions for MARK and FILL as shown in Table 11.3.

no	CN	CS	CE	CW	MARK	FILL
1	0	0	0	1	rmark	rfill
2	0	0	1	0	lmark	lfill
4	0	1	0	0	tmark	tfill
5	0	1	0	1	trmark	trfill
6	0	1	1	0	tlmark	tlfill
8	1	0	0	0	bmark	bfill
9	1	0	0	1	brmark	brfill
10	1	0	1	0	blmark	blfill

Table 11.3. Safe GLU operations

It should be noted that the **propagation** operations in Table 11.2 are different in purpose and function. Here, all Mask bits in Figure 11.13 are set to 1 and the result is not dependent on this operand at all. Instead, these propagations just expand the object unconstrained up to the image border. Clearly, the propagation distance in this case is limited to 2N SPEs, the worst case being propagation from one image corner to the diagonally opposite one.

The propagation that take place in **mark**- and **fill**-operations is constrained to the input image. For a given image size of **N*N** pixels, the longest and most difficult object to propagate though is the Meander curve shown in Figure 11.14.. The length of this single one-pixel wide object is $N^2/2$. Thus, even if we could allow the propagation to take place asynchronously within one single instruction as in the pure propagation instructions, we would have to allow this instruction an execution time of $O(N^2)$. But since we can only use the restricted directions for **mark** and **fill** we also have to divide the total procedure into an iterative sequence of mark/fill steps as shown by the following program.

```
REPEAT
   ld r0
   st r1                    ; Store previous row
   ld r2                    ; Mask bit
   trmark r0
   st r0
   ld r2                    ; Mask bit
   tlmark r0
   st r0
   ld r2                    ; Mask bit
   brmark r0
   st r0
   ld r2                    ; Mask bit
   blmark r0
   st r0
   xor r1
UNTIL COUNT=0
```

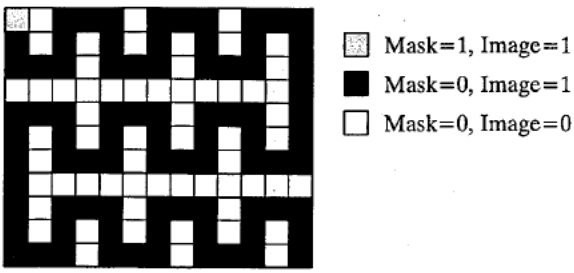


Figure 11.14. The Meander curve

Suppose that the control bits (CN,CE,CW,CS) are stored in register positions within each SPE. Since we may also bring in the SPE-position number, we would then be able to configurate arbitrary propagation paths across the array. For instance, by using the last bit in the row (y) and column (x) number and take and XOR on these bits, we get a checker board pattern image C as shown in Figure 11.15.

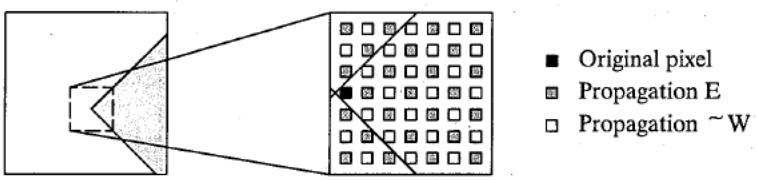


Figure 11.15. NE propagation tilted 45 degrees

Then, if CN:=C, CE:=1, CW:=0, and CS:=C, the result will be a propagation within a 90° cone directed eastwards. The global propagation goes both north and south but is still safe because the latch condition of Figure 11.13 do not exist.

Diagonal propagation only is even simpler to obtain from the checker board pattern by CN:=C, CE:=$\overline{C}$, CW:=0, and CS:=0, shown in Figure 11.16.

Figure 11.16. Diagonal line propagation

As a final example, let the LSB position in row number be the image B as in Figure 11.17. then, an almost 180° propagation cone angle can be obtained by CN:=0, CE:=B, CW:=$\overline{B}$, and CS:=1.

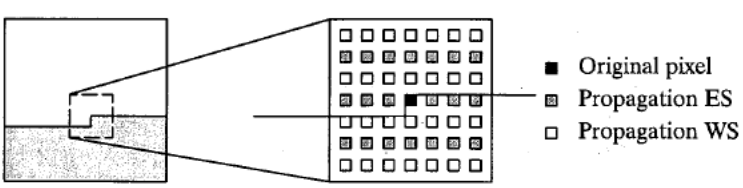


Figure 11.17. Almost 180 degrees of propagation

12 Feature extraction

12.1 Finding a position

In NSIP acquiring and processing of an image is performed at the same time. To perform a certain processing task may then require global data from the array at two stages in the process. The first case is when global data is used for termination of the interrogation loop. The second case is for delivering features of the final result.

In the 2D-NSIP chip which we have proposed here, the only available global data is COUNT which is constantly available and delivered to the host and its program control after every instruction. COUNT is a number which tells how many SPEs which have a value of **1** in their accumulators. As we have seen in numerous examples above, this is a typical termination parameter for a NSIP loop. However, as will be shown below, the COUNT operation is also extremely useful for feature extraction. In the examples to follow, we assume that the NSIP loop has been terminated and left a result in form of one or several images where each one occupies one bit-plane of memory in the SPE array.

Many of the features to be extracted from an image are answers to questions "How many?" or "Where?". Obviously, features that answers the question *How many* is to be extracted by the COUNT operation. As was shown above, the *Where* question in the 1D LAPP was answered by a combination of PROP and COUNT operations. In the proposed 2D-architecture the global propagation operations in four possible directions serve the same purpose.

To extract the position of a single 1-pixel is fairly simple. As can be understood from Figure 12.1 we may extract values for areas A_1 and A_2 with the following program, where the GLU operations are taken from Table 11.2 and Figure 11.12.

```
ld swprop r0
A1 := COUNT
ld seprop r0
A2 := COUNT
```

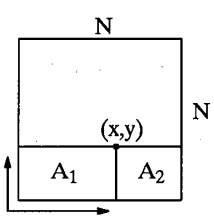


Figure 12.1. Position acquiring using COUNT

In the host computer we may then compute

$$y = \frac{A_1 + A_2}{N} \tag{12.1}$$

$$x = \frac{A_1}{y} = N\frac{A_1}{A_1 + A_2} \tag{12.2}$$

We refer to a situation illustrated in Figure 12.2 with three points of which 1 and 2 have the same y-coordinate. In the first step, we perform a *hsprop* which produces the result in Figure 12.2(b). A count gives us the y-coordinate for 1 and 2. A neighborhood condition (0x1xx) selects the upper edge of this result which is ANDed with the original to obtain 12.2(c). We may then propagate west and COUNT to obtain the x-coordinate for 2. The result is shown in Figure 12.2(d). A neighborhood condition (xx10x) selects the rightmost pixel of the line and by XOR with the original image we may obtain an image without point 2. From here we may repeat the procedure and bring out the coordinates for the remaining points. This following program performs this procedure.

```
hsprop r0
y := COUNT / N
ld (0x1xx) a
and r0
wprop a
x := COUNT
ld (xx10x) a
xor r0
....
```

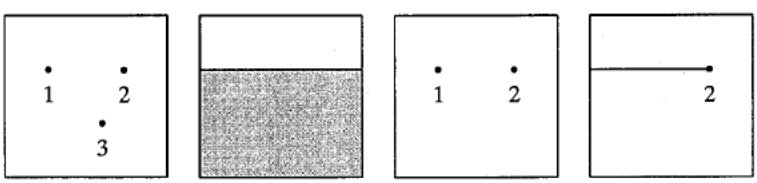


Figure 12.2. Select rightmost uppermost pixel

12.2 Moments

As mentioned in part II of this thesis, moments are often used in image processing to characterize and recognize binary objects and shapes.

Let P_{xy} be the value of pixel at position (x,y). Then, the zero-th order moment is

$$m_{00} = \sum_{y=0}^{N-1}\sum_{x=0}^{N-1} P_{xy} \tag{12.3}$$

and it is computed as follows in a 2D NSIP system, where we assume that the binary image is stored in r0.

ld r0
$m_{00} := COUNT$

The first order moments in the x-direction, m_{10}, is computed as

$$m_{10} = \sum_{y=0}^{N-1} \sum_{x=0}^{N-1} xP_{xy} \tag{12.4}$$

Since

$$x = \sum_{i=0}^{n-1} x_i 2^i \tag{12.5}$$

(12.4) can be written as

$$m_{10} = \sum_{y=0}^{N-1} \sum_{x=0}^{N-1} P_{xy} \sum_{i=0}^{n-1} x_i 2^i = \sum_{i=0}^{n-1} 2^i \sum_{y=0}^{N-1} \sum_{x=0}^{N-1} x_i P_{xy} \tag{12.6}$$

We identify the inner product as an AND between the image P_{xy} and a position dependent binary constant. The two summations is equivalent to a COUNT readout. Finally, the COUNT values are to be weighted by a factor 2^i and summed which takes place in the host. Program to compute m_{10}:

```
m10 := 0;
FOR i:=0 TO n-1 DO BEGIN
   setrx i;
   ld r0
   and rx
   m10 := m10 + COUNT*2^i   ; m10 is accumulated and stored in the host
END
```

The moments of second order, m_{11}, m_{20}, and m_{02}, can be derived in the same manner as for the first order moments. For instance, the formula for m_{20}

$$m_{20} = \sum_{y=0}^{N-1} \sum_{x=0}^{N-1} x^2 P_{xy} \tag{12.7}$$

can be rewritten using binary weights as

$$m_{20} = \sum_{y=0}^{N-1} \sum_{x=0}^{N-1} P_{xy} \sum_{i=0}^{n-1} x_i 2^i \sum_{j=0}^{n-1} x_j 2^j = \sum_{i=0}^{n-1} \sum_{j=0}^{n-1} 2^{i+j} \sum_{y=0}^{N-1} \sum_{x=0}^{N-1} x_i x_j P_{xy} \tag{12.8}$$

Again, the inner product and summation can be identified as an AND between the image and position dependent constants x_i and x_j, followed by a COUNT readout. The value we get for each summation is then weighted with a factor 2^{k+l}.
Program to compute m_{20}:

```
m20 := 0;
FOR i:=0 TO n-1 DO BEGIN
    setrx i;
    ld r0
    and rx
    st r1
    FOR j:=0 TO n-1 DO BEGIN
        setrx i;
        ld r1
        and rx
        m20 := m20 + COUNT*2^(i+j)
    END
END
```

The second order moment m_{11} can be derived in almost the same manner.

$$m_{11} = \sum_{y=0}^{N-1}\sum_{x=0}^{N-1} xyP_{xy} = \sum_{y=0}^{N-1}\sum_{x=0}^{N-1} P_{xy}\sum_{i=0}^{n-1} x_i 2^i \sum_{j=0}^{n-1} y_j 2^j = \sum_{i=0}^{n-1}\sum_{j=0}^{n-1} 2^{i+j}\sum_{y=0}^{N-1}\sum_{x=0}^{N-1} x_i y_j P_{xy} \tag{12.9}$$

Moments calculations for gray level images requires an extra inner bit loop. If

$$P_{xy} = \sum_{k=0}^{b-1} P_{xyk} 2^k \tag{12.10}$$

m_{10} can be computed as

$$m_{10} = \sum_{y=0}^{N-1}\sum_{x=0}^{N-1} P_{xy}\sum_{i=0}^{n-1} x_i 2^i = \sum_{i=0}^{n-1}\sum_{k=0}^{b-1} 2^{i+k}\sum_{y=0}^{N-1}\sum_{x=0}^{N-1} x_i P_{xy,k} \tag{12.11}$$

With 8-bit gray-level pixels stored in r0–r7 (MSB in r7), the following program computes (12.11).

```
m20 := 0;
FOR i:=0 TO n-1 DO BEGIN
    setx i;
    FOR j:=0 TO 7 DO BEGIN
        ld r0 + i
        and rx
        m20 := m20 + COUNT*2^(i+j)
    END
END
```

12.3 Shape factor

A shape factor for a binary object very often represents the compactness of the object where the circle is defined to be the most compact object. One such shape factor can be computed

from the distance transform of the object [5]. If the object is defined as in Figure 12.3(a) the distance transform in city-block metric gives the result shown in Figure 12.3(b).

(a)

	1	1	1	1	1
1	2	2	2	2	1
1	2	3	3	2	1
1	2	3	3	2	1
1	1	2	2	2	1
		1	1	1	

(b)

Figure 12.3. Distance transform of a binary object

The average distance to the edge is obtained as (12.12) and the final shape factor as (12.13).

$$\bar{d} = \frac{\sum_A \left[d(x,y) - \frac{1}{2} \right]}{A} = \frac{\sum_A d(x,y)}{A} - \frac{1}{2} \tag{12.12}$$

$$Shape = \frac{1}{9\pi} \frac{A}{\bar{d}^2} \tag{12.13}$$

To implement this shape factor in a 2D NSIP system we do not have to generate a distance map and then calculate the $\bar{d}$. Instead, we calculate the contributions to $\bar{d}$ iteratively while the distance map is generated. The iteration starts by setting the object pixels which are connected to the background, i.e. the edge pixels. The number of such pixels are then collected using COUNT and the sum is weighted by the factor 1 (in the host). All the object pixels which now have been counted are set to 0, i.e. turn into background pixels. We again select all pixels belonging to the reduced object which are connected to the background, count them and add the count to the previous sum weighted by a factor of 2. This procedure is repeated until we have reached the center of the object, i.e. when the object is annihilated. The following program shows this procedure. The image is in r0.

```
A = COUNT                    ; Area of the object
d = 0
dist = 0
REPEAT
   ldi (11111) r0            ; Select border pixels
   and r0                    ;
   tmp = COUNT
   d := d + dist * tmp       ; Add one layer to the total sum
   xor r0                    ; Remove border pixels
   st r0                     ;
   dist = dist + 1
UNTIL tmp = 0
d = d / A - 0.5
Shape_factor := A / (9 * pi * d * d )
```

13 VLSI realization of 2D NSIP

The transistor level design in this section has been made by
Per Ingelhag, LSI Design, Dept. of Physics, Linköping University, Sweden

We will now go beyond the programmers model in Figure 10.2 and propose a VLSI implementation of the sensor-processor element. As was mentioned earlier, this implementation may look vastly different from the programmer model. In the first place, we allow this hardware to spend several cycles to perform something which the programmer considers as one single unbreakable instruction cycle. Secondly, the VLSI design in CMOS presents so many possibilities to handle logic and storage function that the traditional gate-level synchronous digital design is no longer the optimal strategy for VLSI.

Nevertheless, Figure 13.1 is such a gate-level design for the SPE. This design should not be viewed as the final word, but rather as one of many possible solutions. At the top of Figure 13.1 is the sensor circuit which connects to the bus over the PD-register. There are no differences here between Figure 13.1 and Figure 10.2 here. Also connected to the bus are the registers R0, R1, .. , R9, Rx, and Ry just as in Figure 10.2. The largest differences are in the implementation of the ALU-functions. In Figure 13.1 the (CN,CE,CW,CS)- controlled GLU-functions and the NLU-functions are totally integrated and are implemented by the same net, i.e. the five two-input AND-gates followed by the five-input OR-gate in the center of Figure 13.1. The only difference between the neighborhood connections for GLU- and NLU-functions is found below this network. The GLU propagates the neighborhood logic result to the neighbors of this SPE. The NLU signal gates the data from the bus of this SPE to the neighbors. The result feeds through the AND/OR network but stops there and stays intermediately in G before it is forwarded to the final destination. The short term register G comes for free in CMOS.

An NLU-operation may require several cycles. We use the instruction ld (11x00) as an example. The basic logic function to be performed is

$$G = pd_N \cdot pd_W \cdot \overline{pd_S} \cdot \overline{pd_E} = \overline{\overline{pd_N} + \overline{pd_W} + pd_S + pd_E} \qquad (13.1)$$

It is the latter of these expressions that will be implemented. It will be done in four steps.

i) Set G-register to 1 (precharge)
ii) Evaluate $\overline{pd_N}+\overline{pd_W}$ (if any of these terms are 1, G is set to zero)
iii) Evaluate pd_S+pd_E (if any of these terms are 1, G is set to zero)
iv) Invert and move final G-value to bus

In the second machine cycle the PD-register in Figure 13.1 is gated out to the bus and via the NLU gate this bit is fed in inverted form to the neighbors. Simultaneously, by using control signals (to be described shortly) the registers RN and RW are set to 1. The first partial evaluation takes place. In the third machine cycle the non-inverted pd-value is propagated to the neighbors while the RE and RS registers are set to 1. The second partial evaluation takes place. In the fourth machine cycle the result is inverted and gated to the bus. We notice that no accumulator is visible. This is due to one of the features of CMOS. The bus itself works fine as the accumulator.

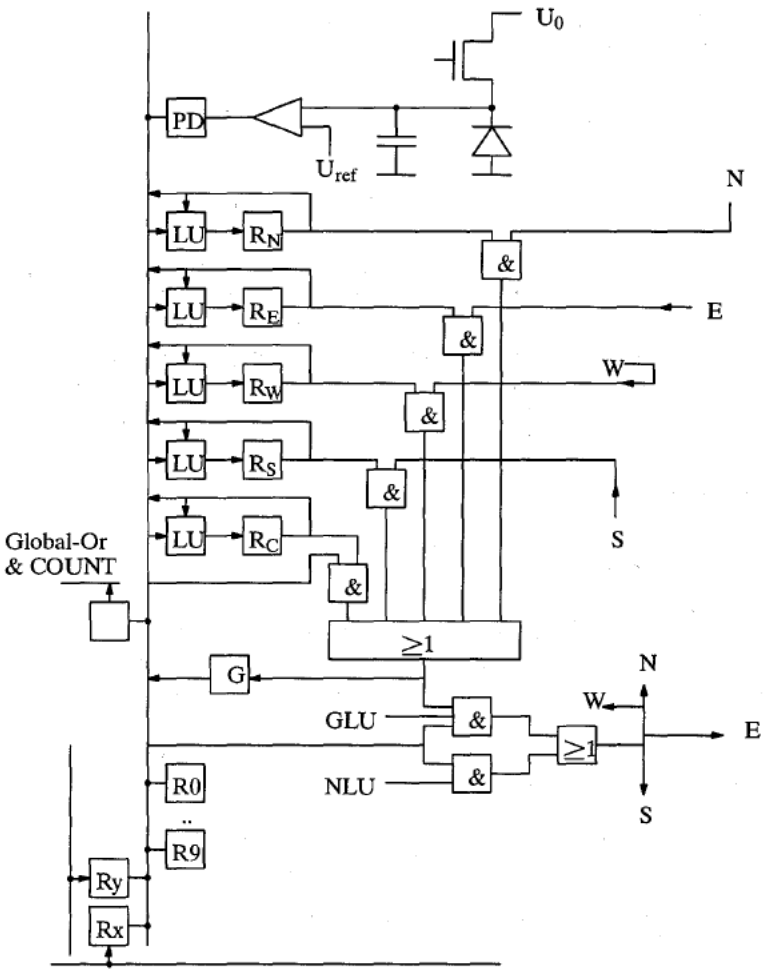


Figure 13.1. One SPE

As was described in section 11, when performing a GLU instruction we first load the direction bits (CN,CE,CW,CS) in the corresponding LUs. The mask bit, called "acc" in Figure 11.4, is loaded into the lowermost LU in Figure 13.1, and the image bit is placed on the bus. When the GLU signal is activated, the propagation starts.

The PLU-functions (AND, OR, XOR, ADD, etc.) are implemented by the logic unit LU which are found between the bus and the direction register RN, RE, RW, RS, and RS. Hence, PLU, NLU, and GLU functions cannot work concurrently as in LAPP. For each of the units there are five control signals (C1,C2,C3,C4,C5) which have the following effect on the register content and the bus, see Table 13.1.

Control	Operation
C_1=0	R := R + B
C_2=1	R := R & B
C_3=0	R := R + $\overline{B}$
C_4=1	R := R & $\overline{B}$

Table 13.1. Some Possible logic function in the LU

Different operations can be performed in different LUs at the same time, which makes it possible to accomplish more sophisticated operations. The cycle counts for some typical operations are shown in Table 13.2. The micro-codes to perform these operations can be found in [11]. The XOR-function can be implemented by employing two LU-functions performing A&$\overline{B}$ and $\overline{A}$&B respectively and exploiting precharge logic for combining the result. The half adder and the full adder function may be put together in a similar manner.

Operation	Cycles
XOR	3
Half adder	2b+1
Full adder	4b+1

Table 13.2. Typical arithmetical performances

A CMOS implementation of an LU and its register is shown in Figure 13.2. The transistor count for each such unit is 10.

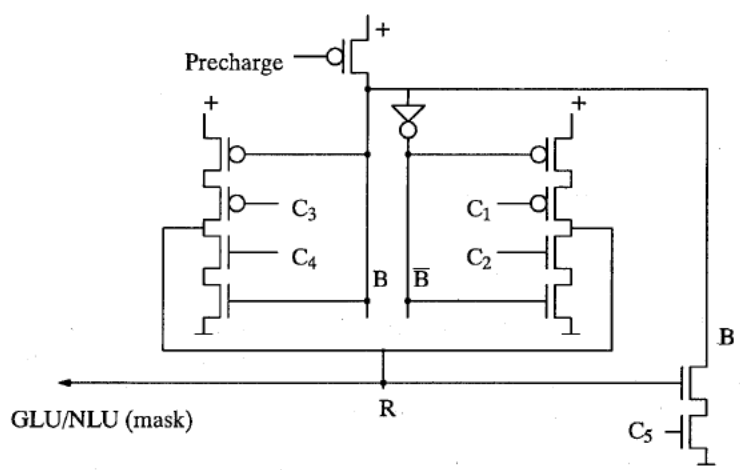


Figure 13.2. Logical Unit, LU

The CMOS implementation of the NLU/GLU function is shown in Figure 13.3 having the transistor count of 20.

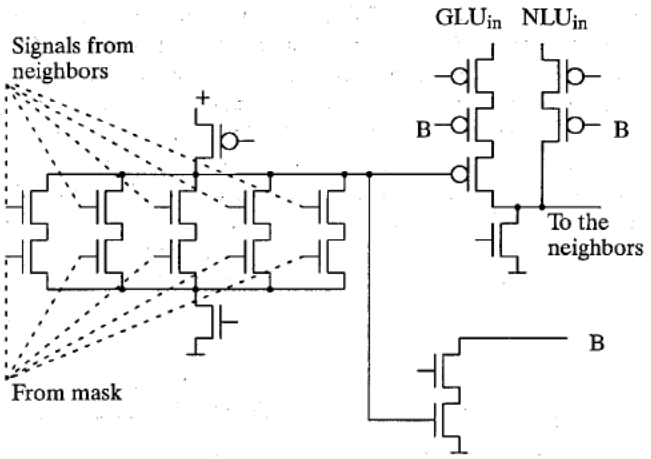


Figure 13.3. NLU and GLU

The registers R0, .., R9 are implemented as dynamic memory cells with three transistors per cell as shown in Figure 13.4. The transistor count for this unit is 33.

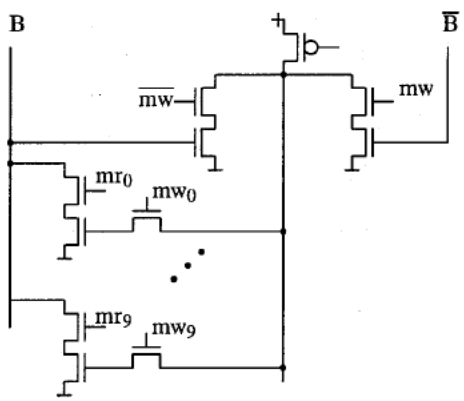


Figure 13.4. Dynamic memory

Additional circuitry in Figure 13.1 amount to approximately 6 transistors. The rough total estimate for the whole SPE is therefore as shown in Table 13.3.

Unit	Transistors
Memory	33
LE * 5	50
NLU/GLU	20
Misc	6
	109

Table 13.3. Transistor count

In section 12 we described the usefulness of COUNT for feature extraction. In LAPP this operation is carried out by a special net which occupies a substantial part of the chip. In 2D however, a straight-forward combinatorial one-cycle implementation of this network would occupy to much space. One way to implement the COUNT operation is to use fast shift registers, one for each column of the array, shown in Figure 13.5. The data could then be added together in a combinatorial or a systolic net. Even with a high speed shift register (200MHz) it would take some time (for a 256x256 array, 2*256/200=5μs). Fortunately, in most application we are not interested in the actual COUNT value when we are inside the NSIP loop, but rather if none or all pixels have passed the threshold. This question can be answered by a global-OR signal which can be implemented by a pre-charged wired-OR.

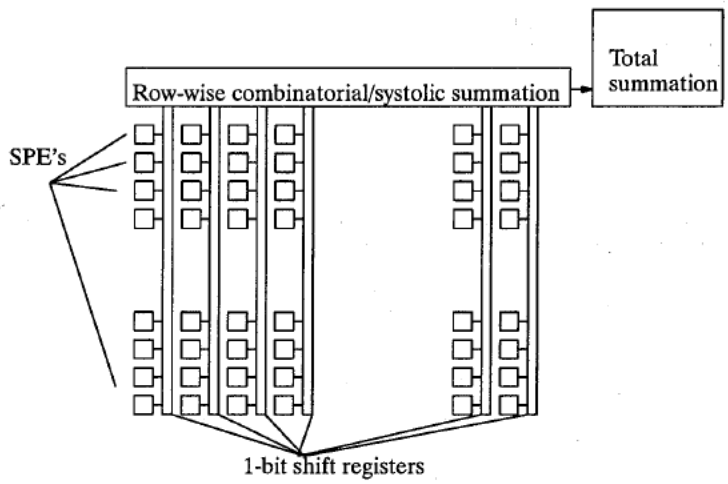


Figure 13.5. COUNT realized as 1−bit shift registers

14 NSIP Sheet-of-light range images

14.1 Implementation in NSIP

A scene which is illuminated with a sheet-of-light, may result in an illumination of the sensor array as shown in Figure 14.1. For each x-coordinate the y-coordinate is a measure of the distance from the laser to the object. A program which detects the maximum intensities for each column can be implemented in the general NSIP architecture by the following program.

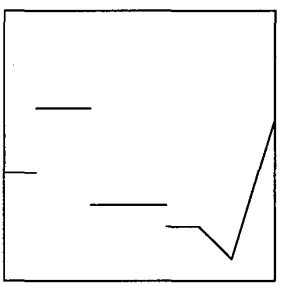


Figure 14.1. A scene illuminated with a sheet-of-light as acquired by the sensor

```
clr a
st r1                    ; inhibit column
st r0
REPEAT
   ldi pd                ; Read from photodiode
   xor r1                ;
   or r0                 ;
   st r0                 ; Conditional storage
   vprop  a              ; Has any pixel passed the threshold?
   st r1
UNTIL COUNT = N
sprop  r0
st a ,r0
```

The vertical propagation instruction *vprop* insures that the first SPE in a column which passes the intensity threshold blocks the possibility to set r1 in all other SPEs. When the program has terminated the image in Figure 14.1 will result in Figure 14.2. Here, shaded area represents set bits in r0.

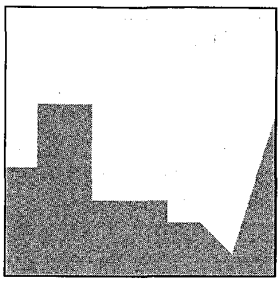


Figure 14.2. The contents of R0 after the loop has terminated

The result can then be read out, column by column, using COUNT as is shown in the following program.

```
setx 0
ldi rx
FOR i=1 TO n-1
   setx i
   andi rx
END-FOR
st a,r1                    ; set r1 in leftmost column to 1
FOR i=0 TO N
   ld r0
   and r1
   Range = COUNT
   ld (x1xxx) r1           ; shift readout to next column
   st r1
END-FOR
```

The max detection program executes as long as needed, or until a certain time-out arrives from the host. In any case, this part of the total procedure will not set any limit on speed as long as the NSIP-loop can be executed fast enough. Very likely, it is the propagation instructions that will dominate. A conservative estimation is 2μs for the whole loop and with N equal to 100, exposure times as short as 200μs could be allowed. The read-out program contains 256 count operation. With 5μs per COUNT this takes about 1300μs and a maximum frame rate would then be approximately 670 frames/s or 172,000 rangles/s. If the column shift register could be controlled by an independent I/O-system the two procedures would be pipe-lined so that the exposure of one frame takes place while the previous result is shifted out and counted. The maximum speed would then be 770 frames/s and around 200,000 rangles/s. The duty cycle for exposure is then 100%.

14.2 Special NSIP architecture for sheet-of-light

A full 256x256 2D NSIP chip is not likely to be designed and manufactured in the immediate future. In the wake of this prospect however, the basic idea of NSIP may be implemented in some special purpose application where fewer transistors per SPE are needed. One such application is the sheet-of-light camera sensor.

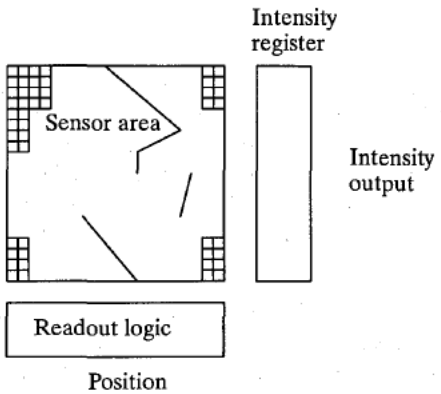


Figure 14.3. Special architecture for sheet-of-light range imaging, using NSIP technique

A proposal for this application is shown in Figure 14.3 and 14.4. A full description is given in [12]. Note that in Figure 14.3 the jagged line runs in the vertical direction. To understand this function see Figure 14.4 which shows the design of an individual SPE. In each row all SPEs have a common inhibit bus and all the SPE's are precharged at the same time. As soon as the first SPE passes the threshold, the inhibit bus goes low because of the (a) transistor. This inhibits all other SPE's in the same row and only the inhibiting SPE is able to introduce a new state at the inverter input (b). The data is transferred to the readout register when the exposure time is over and then read, one row at a time, to the position unit. This unit performs a 1D lfill operation followed by a COUNT which gives us the position of the maximum intensity.

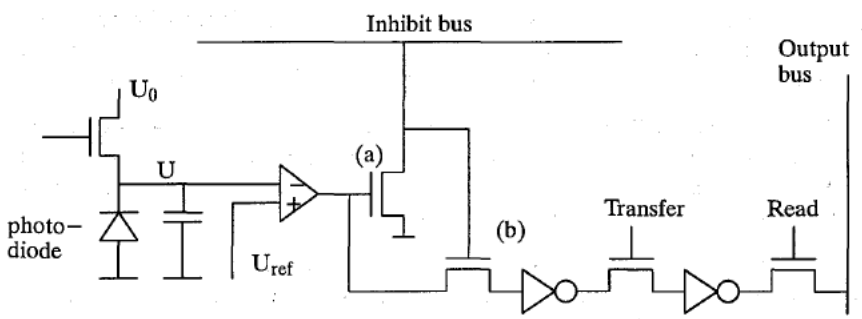


Figure 14.4. An SPE in the special architecture

The intensity, or rather the time to inhibit, can be registered for each column. Saved in a shift register these data may be shipped out from the chip together with the position for each row. Ideally, only one single pixel per row should be set when read out from the array. In practice, other cases may occur. Various 1D filtering technique were presented in part II of this thesis. Several of them are suitable for hardware implementation in Figure 14.4 as described in [12].

15 Conclusion

The idea of Near-Sensor Image Processing originates from the development of LAPP1100. In those days, the VLSI constraints prevented the designers from incorporating a traditional A/D-converter. The solution was to keep the comparator of the potential A/D-converter and take a number of binary samples for different threshold voltages. Thereby, a limited form of grayscale processing capacity was obtained. It was then discovered that many image processing problems could be solved using a constant threshold, and the ideas behind NSIP was born.

Our motivations for developing the NSIP concept further is that each SPE seems to require less space than a corresponding SPE with an ADC. And, as we have proved in this thesis, most of the typical low-level image processing operations can be implemented rather easily on an NSIP system. In many cases, for instance grayscale morphology, the NSIP implementation is much simpler to implement than conventional methods. We have also described how other operations come very natural in the NSIP concept. Among them are locating the maximum intensity, gradient thresholding, rank order filtering, histogramming, and adaptive thresholding.

The time from the pre-charge of the photodiode until the voltage passes the reference level is the available intensity measure in NSIP and it is inversely proportional to the light intensity. We have presented in this paper two different approaches to linearize the measurement. In the first approach we use variable sampling time to achieve different mappings between time and the NSIP loop index. The same idea could be used for histogram equalization. In the other approach we showed that by varying the threshold voltage and keeping the equidistant sampling, we can effectively perform both linear- and non-linear mappings.

For a given light intensity, the uncertainty in time Δt of the comparator threshold crossing depends on several parameters in the photodiode-comparator circuitry. We have modeled the various error sources and also tried to describe their effects on the uncertainty Δt when a globally controlled parameter such as the threshold voltage U_{ref} varies over time.

For two-dimensional NSIP the 1D LAPP1100 architecture needs some modification. We have given a programmers 2D model as well as a gate-level CMOS design of the individual SPE. We have also shown in a few examples how such important 2D features as moments and shape factor can be obtained in this architecture.

The (highly preliminary) CMOS design needs 109 transistors per SPE for its digital part. A 256x256 array would then require more than 7 million transistors. This is not possible to implement today but maybe in a not too distant future. Even so, as was shown by the sheet-of-light sensor example, the basic NSIP-idea may be viable in other more special purpose applications.

16 References

[1] Åström A., *A Smart Image Sensor. Evaluation and Description of PASIC.*, Lic Thesis, Linköpings University 1990.

[2] Åström A., Forchheimer R., Ingelhag P., *An Integrated Sensor/Processor Architecture Based on Near-Sensor Image Processing*, To appear in Proc of CAMP93, New Orleans, USA.

[3] Batcher K.E., *Design of a Massively Parallel Processor*, IEEE Computer, Vol C29, 1980, pp. 836–840.

[4] Chen K., Åström A., Danielsson P.E., *PASIC a smart Sensor for Computer Vision*, Proc of Pattern Recognition, Atlantic City, June 1990, pp. 286–291.

[5] Danielsson P.E., *A New Shape Factor*, Computer Graphics and Image Processing, Vol 7, No 2, 1978, pp 292–299.

[6] Danielsson P.E., *SIMD–arrays. A GAPP–exercise*,1986. Lecture notes, Dept of EE, Linköping University.

[7] Forchheimer R., Ingelhag P., Jansson C., *MAPP2200, a second generation smart optical sensor*, SPIE vol 1659 (1992)

[8] Forchheimer R., Ödmark A., *Single chip linear array processor,* in *Applications of digital image processing,* SPIE, Vol. 397, (1983).

[9] Fountain T.J., *Processor Arrays Architecture and Applications*, Academic Press, London, 1987, pp.49–60.

[10] *Geometric Arithmetic Parallel Processor*, Data sheet NCR45CG72, NCR corporation, (1984).

[11] Ingelhag P., Åström A., *A VLSI realization of a 2D Architecture for Near–Sensor Image Processing*, Internal report, to be published.

[12] Johanneson M., Åström A, Danielsson P–.E. , *A Range Image Architecture Using Sheet of Light*, Proc of IAPR Workshop of Machine Vision Applications, Japan, 1992.

[13] Lyon R.F., *The optical Mouse, and an Architectural Methodology for Smart Digital Sensors.* Palo Alto Research Center.

[14] Reddaway S.F, *DAP – a Distributed Processor Array*, First annual symposium on computer architecture, Florida 1973, pp 61–65.

[15] Sternberg S. R., *Grayscale Morphology*, Computer Vision, Graphics, and Image Processing, Vol 35, 1986.

[16] Tucker L.W., Robertson G.G., *Architecture and Application of the Connection Machine*, IEEE Computer, August 1988, pp 26–38.

[17] Wendt P., Coyle E., Gallagher Jr N., *Stack filters*, IEEE trans ASSP−34, no 4, pp. 898−911, Aug 1986.

[18] Wilson S.S., *One dimensional SIMD architectures − the AIS−5000, Multicomputer Vision*, S. Levaldi, Ed, Academic Press, London, pp. 131−149.

[19] Bernard T., Zavidovique B, Devos F., *A Programmable Artificial Retina*, IEEE Journal of Solid-State Circuits, vol 28, no 7, July 1993.