USC-SIPI REPORT #186

Digital Image Processing on VLSI

by

Bing J. Sheu and Oscal T.-C. Chen

August 1991

Signal and Image Processing Institute
UNIVERSITY OF SOUTHERN CALIFORNIA
Department of Electrical Engineering-Systems
Electrical Engineering Building
University Park/MC-2564
Los Angeles, CA 90089 U.S.A.

Digital Image Processing on VLSI

Edited by Bing J. Sheu, Ph.D.
and Oscal T.-C. Chen

Table of Contents

(EE599 Term Projects, Summer 1991)

DESIGN OF A DIGITAL VLSI NEUROPROCESSOR FOR SIGNAL AND IMAGE PROCESSING

Chia-Fen Chang and Bing J. Sheu
Department of Electrical Engineering
Signal and Image Processing Institute
University of Southern California
Los Angeles, CA 90089-0271

Abstract--An efficient processing element for data/image processing has been designed. Detailed communication networks, instruction sets and circuit blocks are created for ring-connected and mesh-connected systolic arrays for the retrieving and learning phases of the neural network operations. 800 processing elements can be implemented in 3.75 cm x 3.75 cm chip by using the 0.5 μm CMOS technology from TRW, Inc. This digital neuroprocessor can also be extended to support fuzzy logic inference.

I. INTRODUCTION

The techniques for designing a VLSI neuroprocessor have been diversed into two approaches: one is digital design, the other is mixed signal design. Comparatively, a digital design could provide better programmability which is desirable for performing different networks. Moreover, better expandability is also provided to solve large-scale scientific problems.

In the digital design, a systolic system is easy to be implemented because of its regularity and is easy to be reconfigured due to its modularity [1]. A systolic configuration of the connectionist networks can greatly speed up the system throughput in terms of data pipelining. The same architecture is shared between the feedforward propagation phase and the feedback learning phase, which suggests no requirement in reconfigurability [2]. A redundant path can be attached to achieve fault tolerant design.

This paper presents a digital implementation of an efficient processing element (PE). The design is aimed at a wide class of neural network models. Microprogramming technique which results in portability is used. Since the PE is scalable and reconfigurable, multi-chip configuration can be used to increase the network to solve complex problems. The design results can be utilized in the large-volume production and be elastically expanded to Wafer Scale Integration (WSI) to achieve high-performance computing.

II. COMMUNICATION NETWORKS AND SYSTEM OVERVIEW

A custom-designed PE has been developed. This PE can be directly applied to a ring systolic array [3], a mesh-connected array [4], a cellular neural network [5], or a fuzzy control system in which forward processing and

This work was partially supported by DARPA under Contract J-FBI-91-194 and TRW, Inc., NKK Corp.

learning operations are expressed in terms of matrix and vector computations. In this writing, the ring and mesh-connected system will serve as two examples for describing the operations of the designed PE.

A. Ring Systolic Array

The overall system architecture of a ring systolic array used in our system is given in Figure 1. The host computer serves as the interface between the user and the systolic array. It provides the problem-specific parameters, such as input patterns, initial weights, convergence-controlling parameters, etc. The ring controller specifies and monitors the executions in each PE and also performs loading and receiving data to and from each PE. The control line is designed in a broadcast fashion since all commands which are performed in all the PE's are the same during each clock cycle. The control line is operated in a broadcast fashion since all commands which are performed in all the PE's are the same during each clock cycle. To provide an adequate dynamic range for the weights and activation calculations, a word length of 16-bit wide is adopted based on extensive simulations and analysis of the finite word length effects.

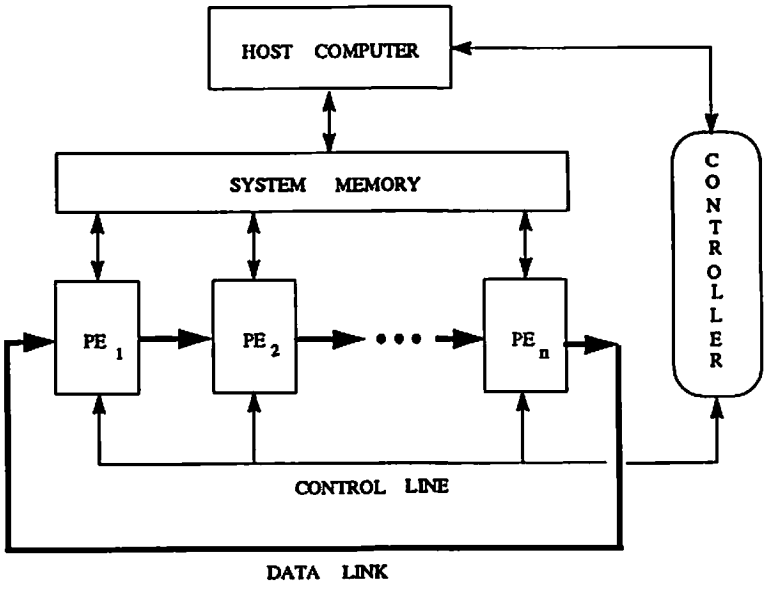


Figure 1 The ring systolic system architecture.

B. Mesh-Connected Array

A mesh-connected systolic architecture requires two-way communication between each PE and all of its nearest neighbors. In our system as shown in Figure 2, the I/O operations occur only on the right- and left- most columns. The top and bottom rows inside the processor array are connected in a wrap-around fashion. An array controller is used to control all of the operations in mesh-connected processors. The system memory is a two-port memory which is shared by the processor array. It receives data and instructions from the host computer when a host call is received by the array controller. It also loads data into the host computer following a system call. Data are only transmitted to (from) this system memory from (to) the right- and left- most columns.

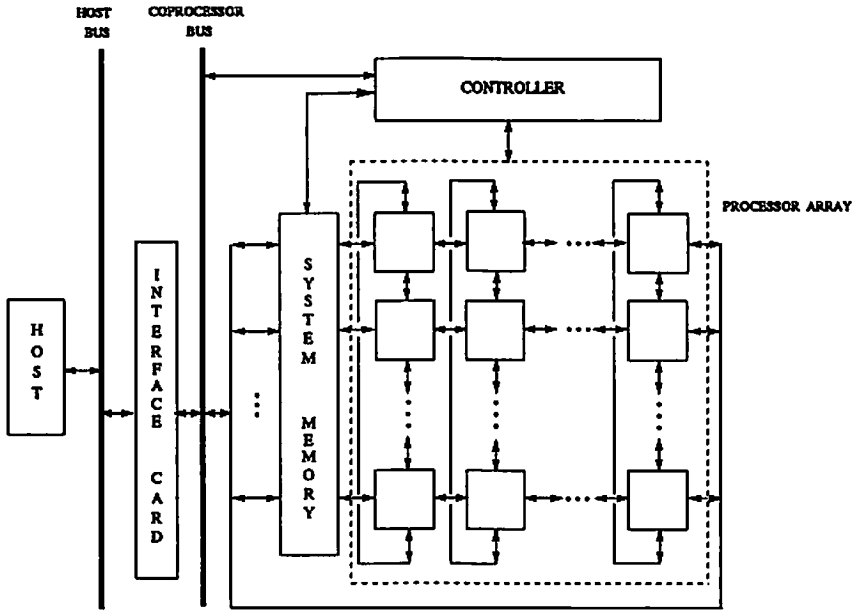


Figure 2 The overall system of a mesh-connected systolic array.

The presented PE is designed to be used either as a neuron processor or a synaptic processor. The mapping of a neural network into a mesh-connected network was discussed by Przytula, Lin and Kumar [4]. Consider a mesh of N x N processors, two cases must be addressed. In the first one, a neural network of n neurons and w synaptic connections are assigned one-to-one to $n+w$ processors, where $N^2 \geq n+w$. Based on the example of a multilayer perceptron with back-propagation learning, the data routing for a single iteration of the retrieving phase requires $24(N-1)$ elemental shift operations whereas the learning iteration takes $60(N-1)$ shifts. For the second mapping, $N^2 \leq n+w$, the original neural net has to be partitioned to suitably utilize the provided hardware.

Benes network was used for interprocessor routing. All of the information regarding processor assignments, partition strategies and data routing are supplied by the procedure from the host computer. A translator (or compiler) is needed to transform this high-level procedure into microcodes which are recognizable by the array controller.

III. DETAILED PE DESIGN

The ring systolic architecture is simpler in structure and clearer in data routing. In the following, we will take this architecture for describing the main functions of our custom- designed PE.

Operations in both retrieving and learning phases of the popular error backpropagation can be formulated as matrix-vector multiplication (MVM), outer-product updating (OPU) and vector-matrix multiplication (VMM) problems. The formulas for these three operations are as follows:

MVM:

$$S_i(l+1) = \sum_{j=1}^{N_l} w_{ij}(l+1)a_j(l) \tag{1}$$

$$a_i(l+1) = f_i(S_i(l+1),\theta_i(l+1)) \tag{2}$$

where l represents the lth layer. The net input value $S_i(l)$, along with the external input $\theta_i(l)$, will determine the new activation value $a_i(l)$ by the nonlinear activation function $f_i(l)$. The weight values w_{ij}'s are stored in the on-chip cache memory whereas $a_i(l)$'s are routed through all the processors. After n steps, all the PE's accomplish their jobs simultaneously.

OPU:

$$\Delta w_{ij} = \Delta w_{ij} + g_i(l+1)a_i(l) \tag{3}$$

$$w_{ij} = w_{ij} - \eta\Delta w_{ij} \tag{4}$$

where

$$g_i(l+1) = e_i(l+1)f'(l+1) \tag{5}$$

the weight value w_{ij}, is updated by the value Δw_{ij} and the updating rate η. These w_{ij}'s and Δw_{ij}'s are stored in the on-chip cache memory, $g_i(l)$'s are stored in the registers. whereas $h_i(l)$'s are routed through all the processors.

VMM:

$$e_i(l) = \sum_{j=1}^{N_l} g_j(l+1)w_{ij} \tag{6}$$

where e_i is the back-propagated corrective signal. The weight values w_{ij}'s are stored in the on-chip cache memory while $g_j(l+1)$'s are routed through all the processors.

The three operations, MVM, OPU and VMM, are basic steps for many signal and image processing applications. For instance, FFT, convolution and Viterbi decoding are mainly composed of these operations. These three matrix and vector computations are supported by the PE.

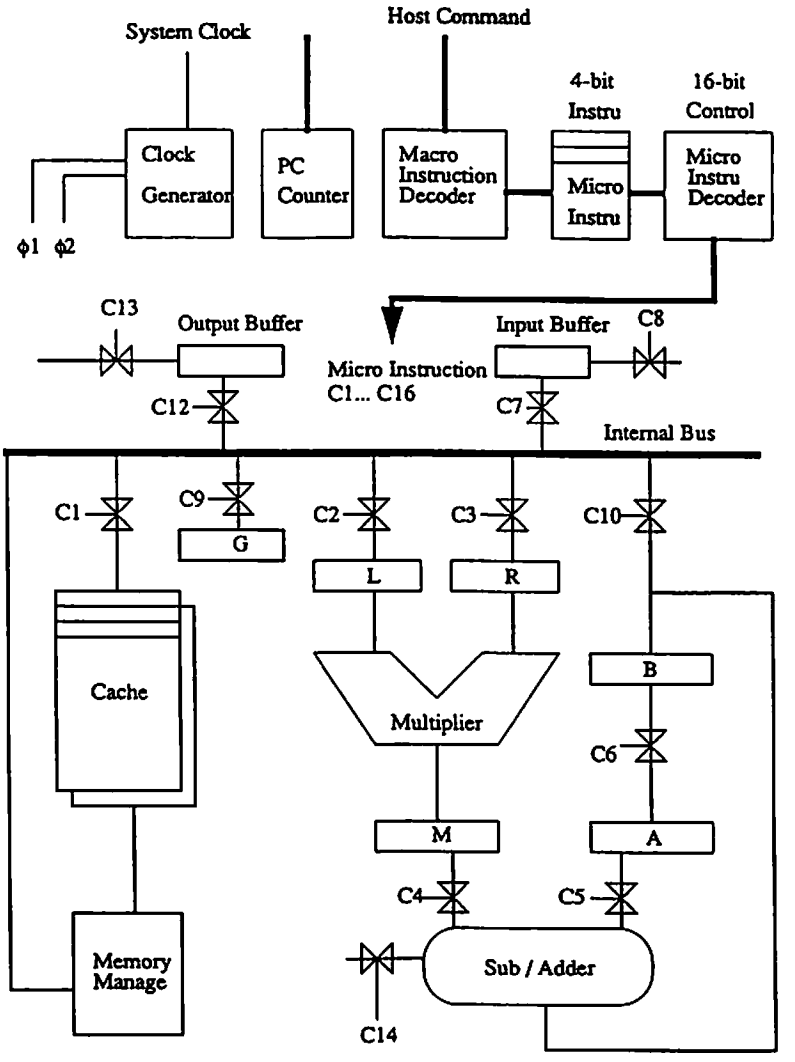


Figure 3 The building blocks inside a PE.

Figure 3 shows the building blocks inside a processing element. Two on-chip cache memories are included in each PE for faster processing. One is instruction cache and the other is data cache. The advantage of separating instruction cache and data cache is that, unlike data, instructions do not change, so the contents of an instruction cache need never be written back to the main memory. Besides the on-chip cache memory inside each PE, there is also a system memory for the PE's. Each PE has its own accessible memory region which is defined initially by the system designer and can then be expanded or shrunk by the user. After the initialization of the systolic array system, the initial parameters (e.g., weights) will be down-loaded into the system memory.

Step	Micro instru description	c1	c2	c3	c4	c5	c6	c7	c8	c9	c10	c11	c12	c13	c14
1	load $a_j(l)$ to L from G		1							1					
2	read $h_j(l+1)$ to buffer								1						
3	$h_j(l+1) \rightarrow$ R,buffer			1				1					1		
	Do multiplication														
4	read Δw_{ij} from cache	1									1				
5	$\Delta w_{ij} \rightarrow$ A						1								
6	Do addition $\rightarrow$ B				1	1									
7	restore Δw_{ij} to cache,R	1		1							1				
8	load η from G		1							1					
	Do multiplication														
9	load w_{ij} from cache	1									1				
10	$w_{ij} \rightarrow$ A						1								
11	Do addition				1	1									

(a)

Step	Micro instru description	c1	c2	c3	c4	c5	c6	c7	c8	c9	c10	c11	c12	c13	c14
1'	Group 1 , 5		1				1			1					
2'	Group 2 , 3 , 10			1				1	1				1		
3'	Group 4 , 6	1			1	1					1				
4'	Group 7 , 11	1		1							1				
5'	Step 8		1							1					
6'	Step 9	1									1				

(b)

Figure 4 (a) The original microinstructions for performing OPU operations.

(b) The associative reduced microinstructions.

During the processing, data will flow between the system memory and the cache memory. A DMA controller is used for concurrent I/O and CPU operations.

The system clock is synchronized for each PE in a systolic array. Microprogramming technique is used in the design since it has advantages of being easy to design, maintain, and expand; it also has great portability and compatibility. However, it has the disadvantage in speed when compared with a dedicated-control finite state machine, high speed memory has been used to improve the performance. C1 to C16 in Figure 3 represent the microinstructions. For a particular problem, a systolic procedure is first implemented by a high level language in the host computer, then is sent to the ring controller. Upon receiving the macro commands, the controller decodes them into microinstructions. The microinstructions for performing OPU operations are shown in Figure 4(a). Since there are many overlapping instructions between these steps, they can further be reduced into a compact instruction-step design, which is shown in Figure 4(b). Notice that multiplication and addition have to be separated by at least one clock.

In order to increase the speed of the processor, the internal bus is segmented into multiple local bus lines. By doing so, more steps are overlapped together because they do not occupy the same bus. The speed of the PE is limited by the multiplier. The Wallice multiplier is used. Since a Wallice structure can be separated into 4 x 4 multiplier blocks and Wallice tree blocks, a pipeline procedure can be added here and the speed of the PE can be increased.

The controller includes four major modules: a memory access module, a fault detection and recovery module, an I/O module and a control module. The memory access module is used to down-load data to each local memory. The fault detection and recovery module is used for detecting faulty PE's, correcting and recovering from errors by a spare PE. The I/O module receives data and macrocodes from the host, up-loads data and faulty signals to the host computer. The control module controls operations of each PE, broadcasts and collects data.

A two-level microprogramming is used to speedup the network response because most operations are repetitive and commonly used. Moreover, this two-level control design technique can reduce the total size of the control memory needed, this translates to smaller chip area in the case of one-chip CPU's or PE's [6]. The two-level microprogram and supporting hardware are shown in Figure 5. There are four major operations inside a microprogram:

Next Address Operation: next operation is at the next address. Opcode = 00.

Conditional Branch Operation: next operation's address depends on the condition in the microcode. Opcode = 01.

While Do Operation: could be done by combination of next address and conditional branch operations, and use a dedicated counter to count the iteration number. Opcode = 11.

Stop Operation: for the last operation in the nanoprogram, test if repetition, that is, content of the dedicated counter for the dimension (i.e., number of operations) of a macroinstruction from the host, is enough. If yes, the microprogram goes to next operation, else the nanoprogram goes to the first nanooperation and trigger the dimension counter to count down. Opcode = 10.

The implementations of these four operations is shown in Figure 6.

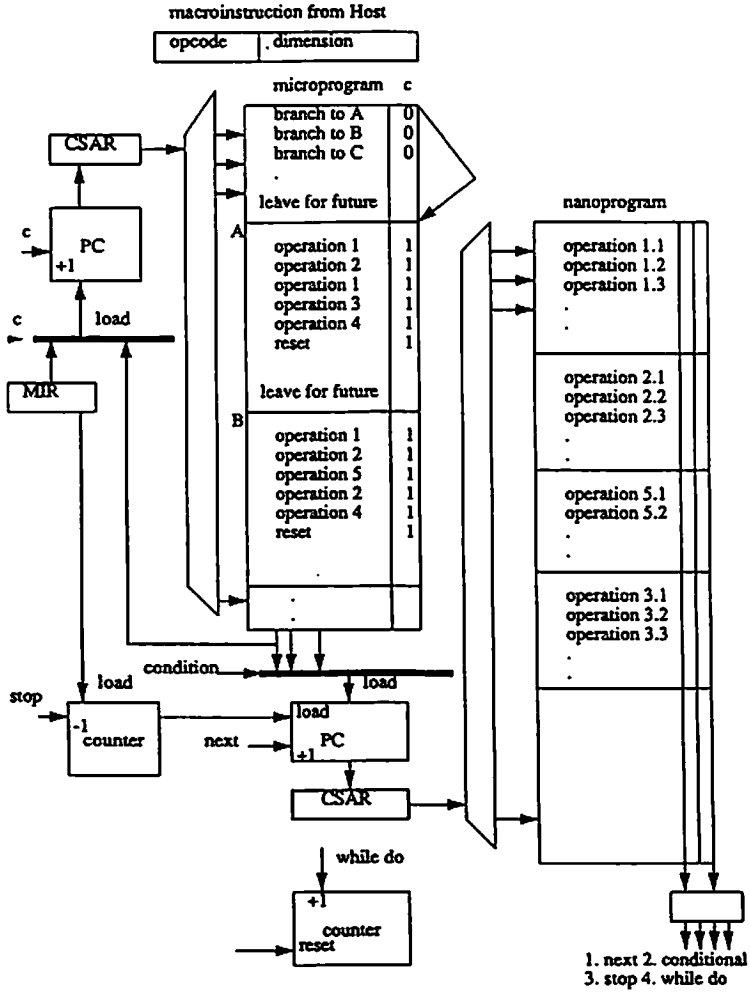


Figure 5 The two-level microprogram.

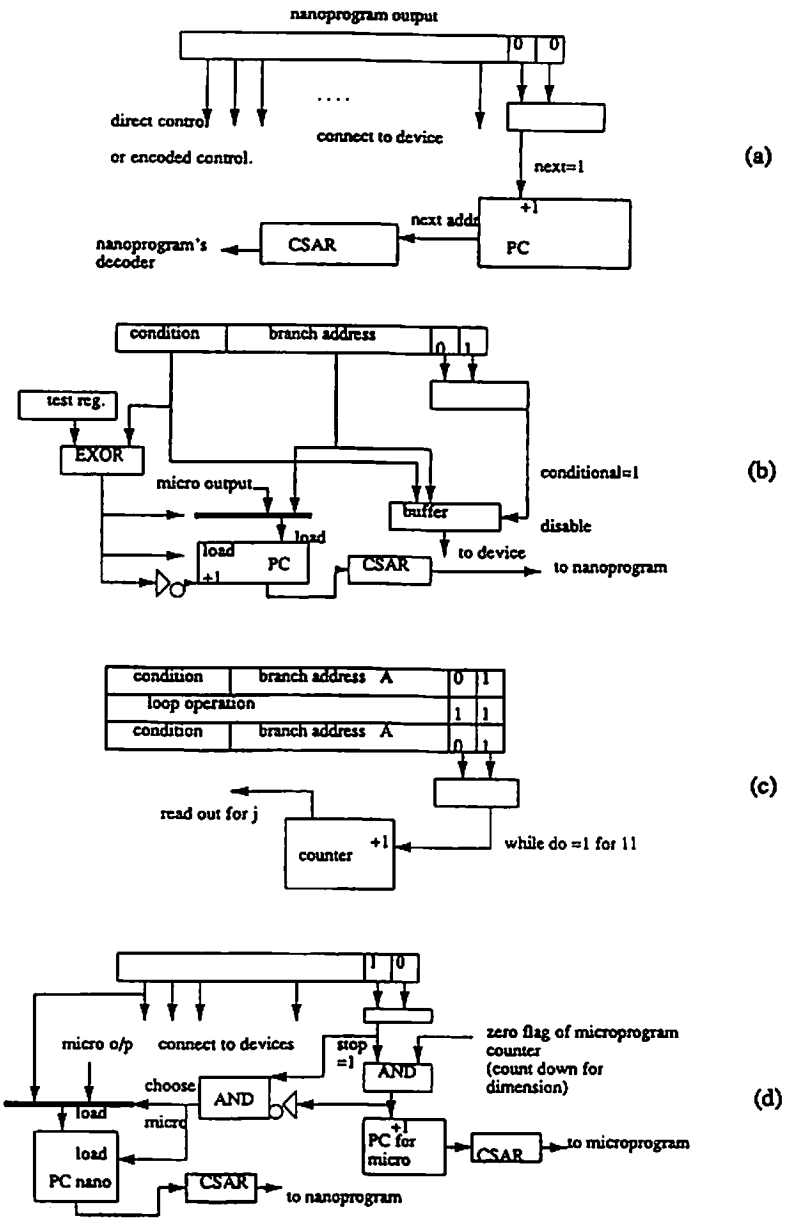


Figure 6 Implementations of the four major operations inside a microprogram.
(a) Next address operation. (b) Conditional branch operation.
(c) While do operation. (d) Stop operation.

From our calculation, 800 processing elements can be implemented in a 3.75 cm x 3.75 cm chip by using the 0.5 μm CMOS technology provided by TRW, Inc. Data buses occupy 20% area of the whole chip. The system memory is supported off-chip.

For the fault tolerant algorithm, the weighted check sum method is used to detect and correct errors [7]. Moreover, parity bits are used to detect one or two adjacent faulty bits in the microprograms and the counters. After completion of error detection and correction, recovery operation can be accomplished by dedicated microprogram.

IV. CONCLUSION

This microprogram-controlled PE is designed for running various applications in the field of data and image processing. Ring and mesh have been the target communication networks during the design. For a neural network of large size, either partition the network to fit a small hardware size or use multiple chips is achievable.

V. ACKNOWLEDGEMENT

Valuable discussions with Professors Alvin Despain, Jerry Mendel, Sun-Yuan Kung and Mr. Te-Ho Chen are highly appreciated. James Cable, Advanced Technology Manager, and Mark Miscione, VHSIC Program Manager of TRW, Inc. provided the advanced CMOS technology support.

REFERENCES

[1] H. T. Kung, "Why Systolic Architecture?", *IEEE Computer*, 15(1), 37-42, 1982.

[2] N. Morgan, *Artificial Neural Networks Electronic Implementations*, IEEE Computer Society Press, 1990.

[3] J.-N. Hwang, J. A. Vlontzos and S.-Y. Kung, "A Systolic Neural Network Architecture for Hidden Markov Models," *IEEE Trans. on ASSP*, vol. 37, no. 12, pp. 1967-1979, Dec. 1989.

[4] K. W. Przytula, W.-M. Lin and V. K. Prasanna Kumar, "Partitioned Implementation of Neural Networks on Mesh Connected Array Processors," *VLSI Signal Processing, IV*, IEEE Press, pp. 106-115, 1991.

[5] L. O. Chua and L. Yang, "Cellular Neural Networks: Theory," *IEEE Trans. on Circuits and Systems*, vol. 35, no. 10, pp. 1257-1272, Oct. 1988.

[6] J. P. Hayes, *Computer Architecture and Organization*, McGraw-Hill, Inc., 1988.

[7] M. Abramovici, M. A. Breuer and A. D. Friedman, *Digital Systems Testing and Testable Design*, Computer Science Press, 1990.

MASK-BASED ALGORITHMS FOR A MESH-CONNECTED SYSTOLIC PROCESSING ARRAY

A single instruction multiple data (SIMD) image processing computer has been conceptually designed based on a mesh-connected array of chips, each of which is a mesh-connected systolic array of processing elements. This design has been simulated, and the simulation used to develop algorithms for low-pass filtering, Sobel masking, and edge detection. The processing element instruction set has been developed, and the functional allocation between processing array and controller accomplished. Finally, the simulated instruction cycle count has been used to estimate real-time performance.

Stephen R. Smith Jr.
567-94-5808
EE599, Special Projects
c/o Northrop Corporation
1 Northrop Ave. E610/90
Hawthorne, CA 90250
(213)332-8241

Multiprocessor architectures for image processing applications have been extensively studied. Many schemes have been developed to apply VLSI technology to special purpose image processing tasks; median filtering [1] and Sobel filtering [2] are but two. The limitation in these designs however is precisely in their "specialness"; such designs have very limited flexibility. An alternate scheme is to employ chips consisting of multiple processing elements (PEs), each of which may be programmed to at least some variety of tasks. Examples are the pipeline architecture developed by Denayer et. al. [3] and the neuro-processor proposed by Chang and Sheu [4]. Both designs are characterized by either ring architectures or for the latter, mesh-connected architectures with internal left-to-right wrapping (i.e. the "left-most" PEs on the chip are connected within the chip to the "right-most" PEs). However, it is conceivable that a chip could be designed to be combined with others into a fully-expandible mesh-connected array. This paper explores such a design.

A custom-designed PE was described by Chang and Sheu [4]. This design was adapted for use in an extensible mesh-connected chip as shown in Figure 1. First, four additional control lines were added to allow selective connection between a PE's input register and any of its four-neighbor's output registers. (These same control lines equivently permit selective routing of a PE's output register to its neighbors.) Second, a "compare unit" was added to permit limited logic control. This compare unit is conceptually identical to the compare and swap unit CSU1 developed by Karaman, et. al.[1], except that the hitherto internal "equal" state is exported. Third, the "equal" state output from the CSU1 is combined with the "if control" line to act as a consent switch for all the external control lines. For the equal state to affect an instruction's execution, the external control must be asserted. If then the equal ste is asserted also, the commanded instruction procedes; otherwise no operation occurs. This feature provides limited capability for data-dependent execution.

The instruction set for the PE is shown in Figure 2. The load, store, read, and output instructions are straightforeward. The compare operation really involves the use of the data in the output registers, since the actual operation of the CSU1 procedes as soon as data is available. Similarly, the convolve operation is a permission signal to the add unit since the multiply unit (the time limiter in the PE) runs whenever it has data. The clear instruction is added for convenience; it is equivalent to dedicating one address of cache memory to storing a zero. The keys to the array's

Figure 1: Processing Element Architecture

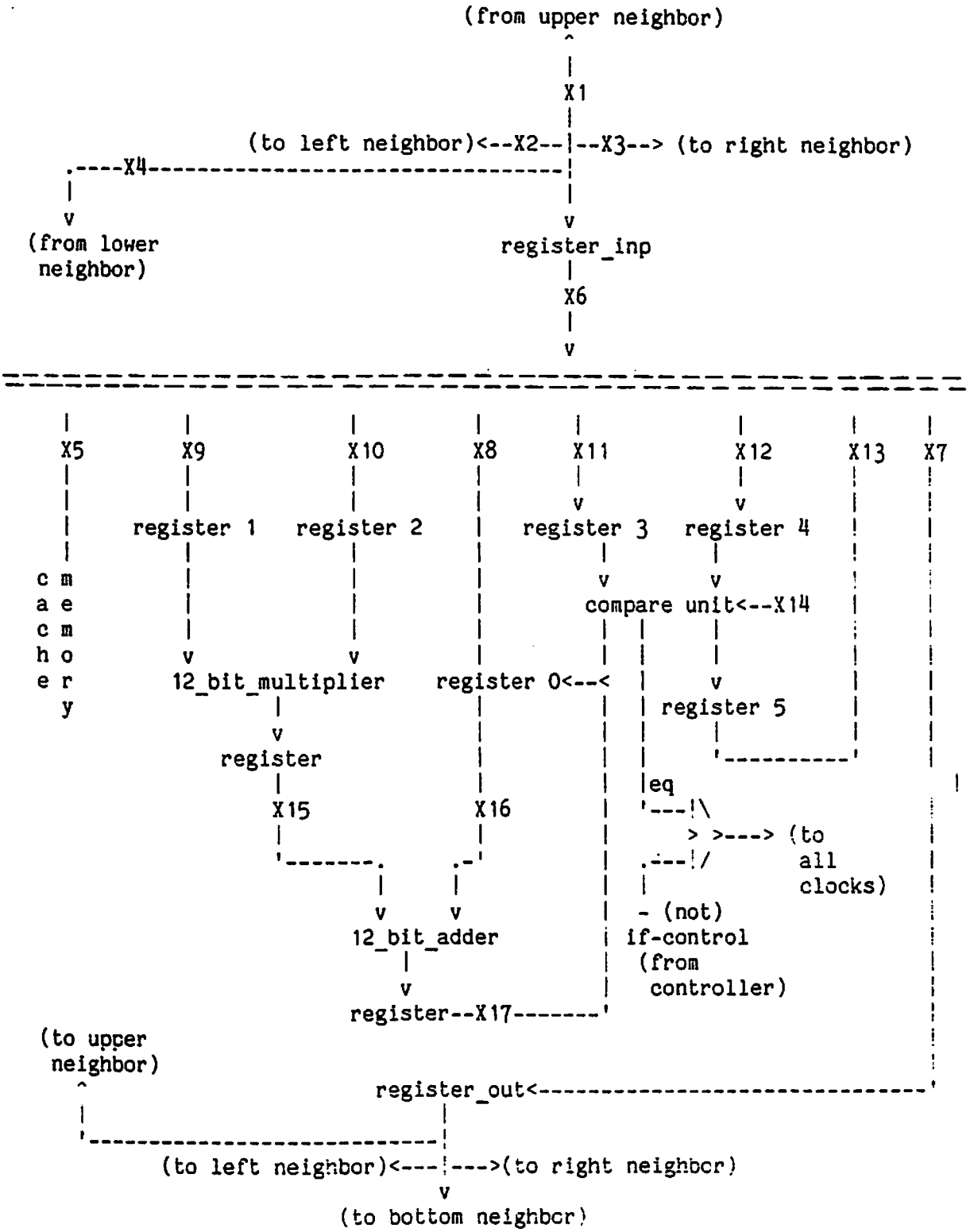


Figure 2: Processing Element Instruction Set

INSTRUCTION	ARGUMENTS	X5	X6	X7	X8	X9	X10	X11	X12	X13
LOAD	R1 MEM[i]	X	(1)	(1)	(1)	(1)	(1)	(1)	(1)	
STORE	R1 MEM[i]	X	(1)	(1)	(1)	(1)	(1)			(1)
OUTPUT	R1		(1)	X	(1)	(1)	(1)			(1)
READ REGISTER	R1 R2(to)	(1)	(1)	(1)	(1)	(1)	(1)	(1)	(1)	(1)
		(2)	(2)	(2)	(2)	(2)	(2)	(2)	(2)	
CLEAR	R1	(1)	(1)	(1)	(1)	(1)	(1)	(1)	(1)	

X: Required clock setting
(1): Optional clock setting determined by argument 1
(2): Optional clock setting determined by argument 2

INSTRUCTION	ARGUMENTS	X1	X2	X3	X4	X14	X15	X16	X17	X18
RESET	none									X
COMPARE	none					X				
CONVOLVE	none						X	X	X	
MDRU	none	X								
MURD	none				X					
MRRL	none		X							
MLRR	none			X						

X: Required clock setting
MDRU: Move Down, Receive Up ([i][j]->[i-1][j]; [i][j]<-[i+1][j])
MURD: Move Up, Receive Down ([i][j]->[i+1][j]; [i][j]<-[i-1][j])
MRRL: Move Right, Receive Left ([i][j]->[i][j+1]; [i][j]<-[i][j-1])
MLRR: Move Left, Receive Right ([i][j]->[i][j-1]; [i][j]<-[i][j+1])

performance are the move instructions. Each instruction connects a PE's input to another's output; thus each PE simultaneously sends and receives a datum. For image processing applications, these are typically used to route image gray-scale values from one PE to another.

Chang and Sheu suggested that using 0.5 um technology 800 PEs could be placed on a single 3.75 cm x 3.75 cm chip, with 20% overhead due to interconnections [4]. This suggests that each PE occupies 1.4 mm2. For my design, the CSU1 adds less than .01 mm2, and the increased internal signal paths add about 20%. The extimated PE size is then 1.68 mm2. Since my chip has twice the data paths of the Chang chip, the estimated data overhead is 40%. The configuration of PEs for the simulated chip is 24 x 34. Thus, this chip has an estimated size of 3.67 cm x 5.20 cm. This is quite large, yet not outlandish when compared to the Chang and Sheu design.

The simulated image processing computer has 12 chips in a 4 x 3 array, as shown by Figure 3. The right- and left-edge chips are connected so that the data wraps around. Data enters from a 1 x 136 set of registers at the top of the array, and exits to a similar set of registers at the bottom of the array. The simulated controller has an expanded instruction set compared to the PE. Program control is provided via loop, loop end and halt operations; image-oriented operations for loading each PE with a selected pixel, writing a line of computed output to the output registers, or for reading an image line into the computer are provided; initialization of a cache location is available; and control of the "if" line is provided. The controller instructions are shown in figure 4.

Three algorithms were developed for execution on the simulated computer. These were lowpass filtering, Sobel masking, and edge detection using horizontal and vertical Sobel operators combined with thresholding. The first algorithm was the simplest. First the mask was initialized to the desired value. (Since the computer cannot handle fractions, an integer mask was used, and the output rescaled by the simulated SISD host computer.) A loop was established to assure the entire image was operated on. Each PE was assigned to compute one output pixel. Thus, if the PE's equivalent location in the output image is [i][j], that location in the input image plus all 8-neighbors must be routed through the PE. The initial value placed in the PE at subimage initialization time was [i-1][j]. Prior to any other processing, the value at [i-1][j+1] was moved into the PE. This meant, by the way, that the results from the PEs in columns 0 and

Figure 3: Chip Array

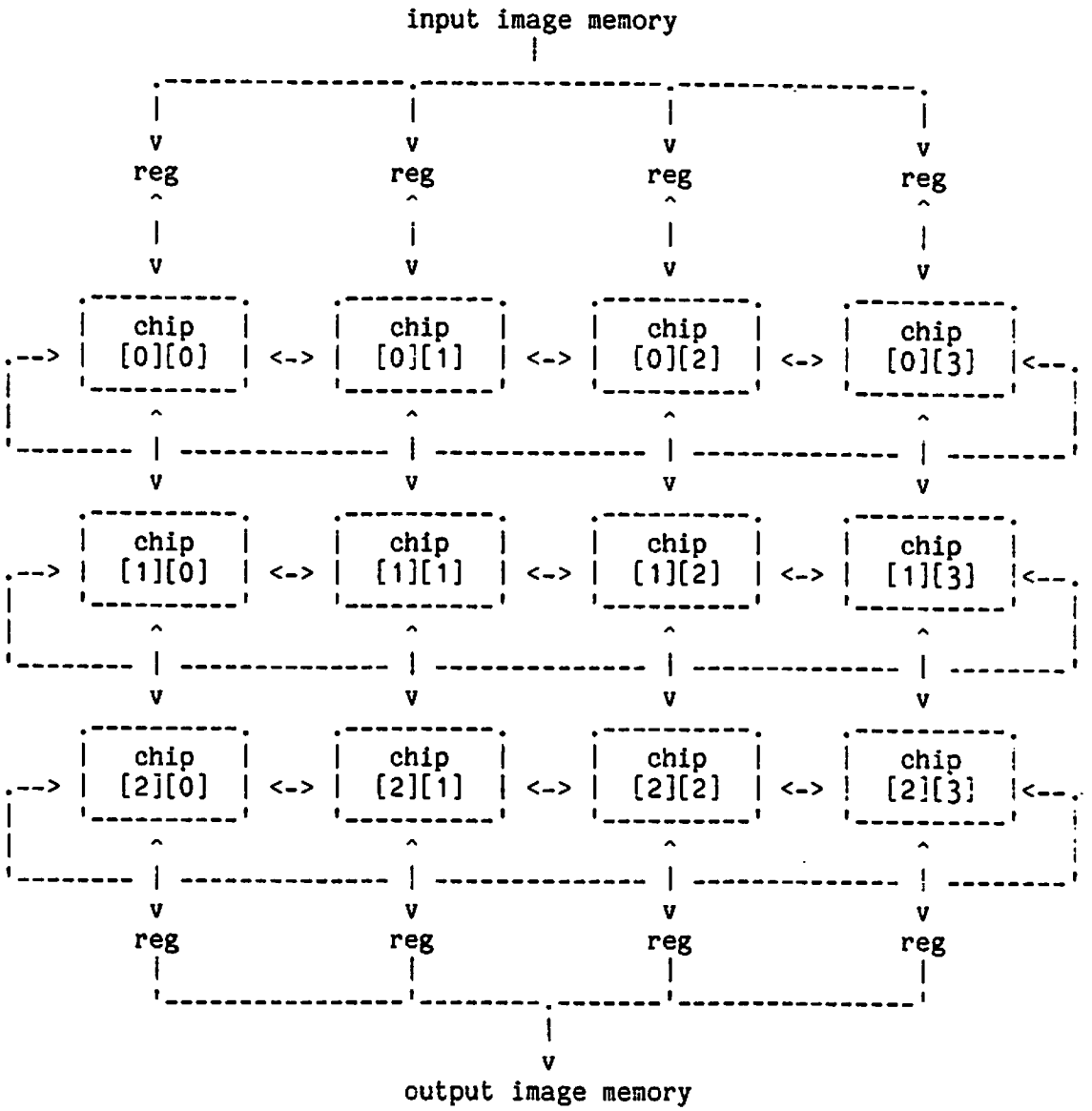


Figure 4: Controller Instruction Set

INSTRUCTION	OP-CODE	ARGUMENTS	COMMENTS
RESET	-1	none	Resets PE equal signal from CSU1
LOAD	0	R1 i	Loads register R1 from cache[i]
STORE	1	R1 i	Stores R1 into cache[i]
OUTPUT	2	R1	Outputs from R1
READ	3	R1 R2	Reads R2 from R1
CLEAR	4	R1	Sets R1 to 0
COMPARE	5	none	r0<-min(r3,r4); max(r3,r4)->r5
CONVOLVE	6	none	r0<-r0 + r1 * r2
MDRU	8	n	Performs n MDRUs
MURD	9	n	Performs n MURDs
MLRR	10	n	Performs n MLRRs
MRRL	11	n	Performs n MRRLs
WDRU	16	n m	Writes out line; reads in 1 line
LOOP	17	none	Starts loop through image
LEND	18	none	Ends instructions w/in loop
HALT	19	none	Normal stop
IFEQ	20	none	Sets if control to PEs
CINI	24	i n	Initializes cache[i]<-n
IMLD	25	x y m	Reads subimage from (x,y)
IMRD	26	n m	Reads next line of image

R1,R2: Register numbers used as arguments 1 and 2, respectively
i: Refers to cache location i
n: A number, used as an iteration counter
m: Mask size, used to compensate when reading image lines
x,y: An input image location; lower left hand corner of subimage

135 were unusable. Then each image location was convolved with its corresponding mask value in the following order: [i-1][j+1], [[i-1][j], [i-1][j-1], [i][j-1], [i][j], [i][j+1], [i+1][j+1], [i+1][j], [i+1][j-1]. The convolved value was then output, and the process repeated for the next subimage. The results are shown in Figure 5.

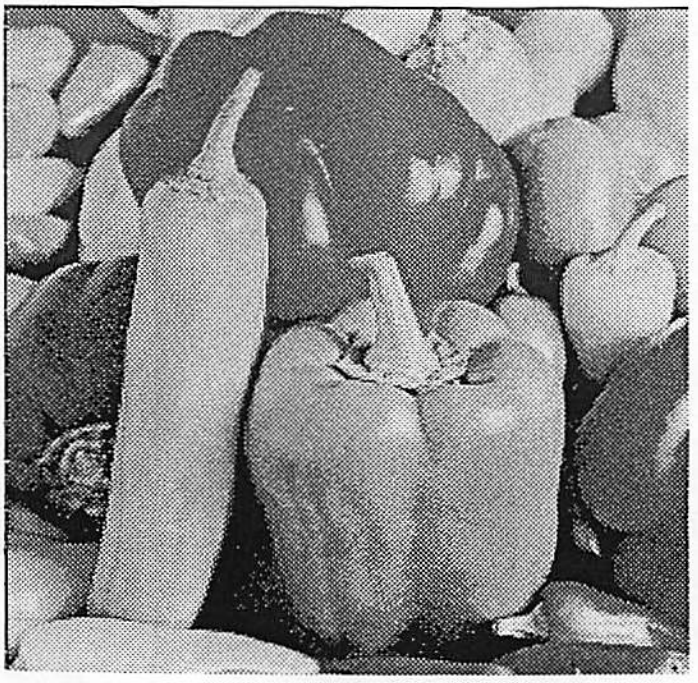
The second algorithm, the Sobel mask, was similar. Exactly the same data routing scheme was used. The only addition was the computation of absolute value using the CSU1 to output the larger of the value and its negative. The results are shown in figure 6.

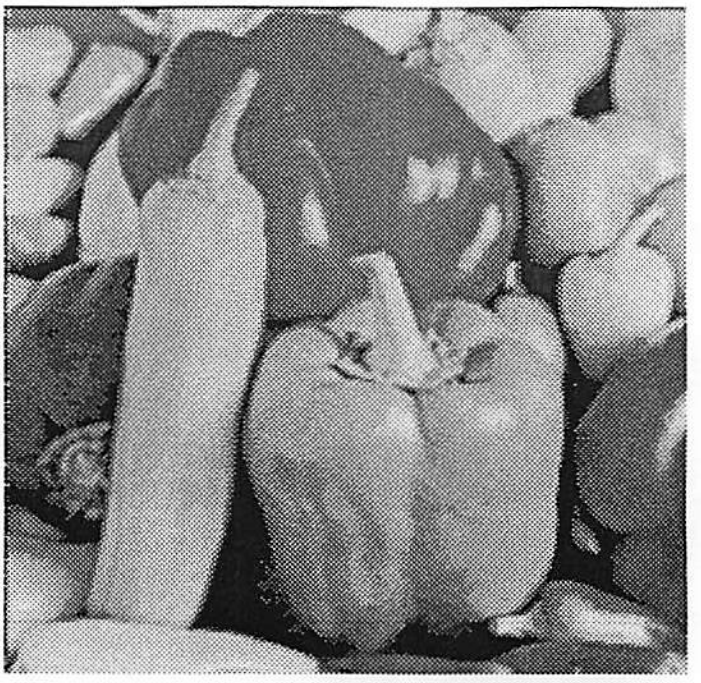
The final algorithm was the most complicated. Only the input image locations corresponding to non-zero mask values were used. The data routing order was as follows: [i-1][j+1] (used for both Sh and Sv); [i-1][j] and [i-1][j-1] (used for Sh); [i-1][j-1], [i][j-1], [i][j+1], [i+1][j+1], [i+1][j-1] (used for Sv); [i+1][j-1], [i+1][j], [i+1][j+1] (used for Sh). The magnitude of the output pixel was computed by |Sv| + |Sh|. Finally the if control was used to selectively output 0, if the magnitude was below threshold, the magnitude if it was less than the maximum grayscale value, or the maximum grayscale value. The results are shown in Figure 7.

The simulated number of instruction cycles is shown in Figure 8. The number of cycles required for mask initialization is also identifed in Figure 8; the remainder provides some indication of predicted real-time performance. Any data-moving operation was assigned one cycle; compares were assigned 2; and a convolution was assigned 3. These numbers are fairly arbitrary, and were intended to support an estimate rather than be precise. For purposes of estimation, a clock-speed of 40 MHz was chosen; this is a conservative number for 0.5 um technology. The results (again shown in Figure 8) indicate that real-time operation on a 1024 x 1024 (equivalent to 16 of the 256 x 256 images actually tested) is feasible.

In summary, the proposed conceptual design, while somewhat large, provides significant flexability for use in a number of image processing applications. The major cost to the system is due to data flow. Thus the ability to perform a variety of tasks on a single subimage is what provides the flexibility and power of this design. The significant margin available for additional real-time image processing suggests that a computer of this type may have considerable value.

Figure 5: Lowpass Filtering Results





ORIGINAL IMAGES

OUTPUT IMAGES

Figure 6: Sobel Masking Results

ORIGINAL IMAGES

OUTPUT IMAGES

Figure 7: Edge Detection and Thresholding Results

ORIGINAL IMAGES

OUTPUT IMAGES

Figure 8: Timing Estimates

TOTAL CYCLES	INIT CYCLES	IMAGE CYCLES 256X256	IMAGE CYCLES 1024X1024	EST'M'TED TIME (ms)
2907	164	2743	43888	1.097
3131	308	2823	45168	1.129
3681	466	3215	51440	1.286

Note 1: Line 1 is from the lowpass filtering results; line 2 is from the Sobel mask results; and line 3 is from the edge detection results.

Note 2: To support real-time operation, a maximum time of 33.3 ms is required. All three algorithms meet this easily. (40 MHz)

REFERENCES

[1] M. Karaman, L. Onural and A. Atalar, "Design and Implementation of a General-Purpose Median Filter Unit in CMOS VLSI", *IEEE Journal of Solid State Circuits*, 25(2), 505-513, 1990.

[2] N. Kanopoulos, N. Vasanthavada and R. Baker, "Design of an Image Edge Detection Filter Using the Sobel Operator", *IEEE Journal of Solid-State Circuits*, 23(2), 358-366, 1988.

[3] T. Denayer, E. Vanzieleghem and P. G. A. Jespers, "A Class of Multiprocessors for Real-Time Image and Multidemensional Signal Processing", *IEEE Journal of Solid-State Circuits*, 23(3), 630-638, 1988.

[4] C. F. Chang and B. J. Sheu, "Design of a Digital VLSI Neuroprocessor for Signal and Image Processing", pre-publication handout, 1991.

APPENDIX 1:

MACHINE LANGUAGE ALGORITHMS

```
/* This is a 3x3 mask algorithm for the systolic array. The intent is to
 * perform lowpass filtering, but the PEs can't divide. Therefore an integer
 * mask is used, as shown below, and the host computer (represented by this
 * program) applies the necessary weighting factor of 1/16.
 *
 *           / 1    2    1\
 *    mask = | 2    4    2|
 *           \ 1    2    1/
 */

#include <stdio.h>
#include <sys/file.h>
#include <sys/types.h>
#include <sys/uio.h>

#define SIZED 256
#define MASKD 3
#define EBUF (MASKD-1)/2
#define BSIZE SIZED+MASKD-1

#include "controller.c"

main()

{
char ifile[80];
char ofile[80];
int buffer[SIZED][SIZED];
int output[SIZED][SIZED];
char line[SIZED];
int i,j,word,temp;
INSTRUCTION pq;
FILE *image, *out;
static INSTRUCTION prog[SIZED];

  /* Initialization phase */

pq.op_code=CINI; pq.arg1=0; pq.arg2=1;  /* cache[0]=1 */
prog[0]=pq;
pq.op_code=READ; pq.arg1=6; pq.arg2=1;  /* R1 = R6(input) = 1 */
prog[1]=pq;
pq.op_code=STOR; pq.arg1=1; pq.arg2=2;  /* cache[2]=1 */
prog[2]=pq;
pq.op_code=STOR; pq.arg1=1; pq.arg2=6;  /* cache[6]=1 */
prog[3]=pq;
pq.op_code=STOR; pq.arg1=1; pq.arg2=8;  /* cache[8]=1 */
prog[4]=pq;
pq.op_code=READ; pq.arg1=1; pq.arg2=2;  /* R2 = R1 = 1        */
prog[5]=pq;
pq.op_code=READ; pq.arg1=1; pq.arg2=0;  /* R0 = R1 = 1        */
prog[6]=pq;
pq.op_code=CONV; pq.arg1=1; pq.arg2=8;  /* R0 = R0 + R1*R2 = 2 */
prog[7]=pq;
pq.op_code=STOR; pq.arg1=0; pq.arg2=1;  /* cache[1]=2 */
prog[8]=pq;
pq.op_code=STOR; pq.arg1=0; pq.arg2=3;  /* cache[3]=2 */
prog[9]=pq;
pq.op_code=STOR; pq.arg1=0; pq.arg2=5;  /* cache[5]=2 */
prog[10]=pq;
pq.op_code=STOR; pq.arg1=0; pq.arg2=7;  /* cache[7]=2 */
```

```
prog[11]=pq;
pq.op_code=READ; pq.arg1=0; pq.arg2=2;  /* R2 = R0 = 2 */
prog[12]=pq;
pq.op_code=CONV; pq.arg1=6; pq.arg2=-1; /* R0 = R0 + R1*R2 = 4 */
prog[13]=pq;
pq.op_code=STOR; pq.arg1=0; pq.arg2=4;  /* cache[4]=4 */
prog[14]=pq;
pq.op_code=CLER; pq.arg1=0; pq.arg2=3;  /* R0 = 0 */
prog[15]=pq;
pq.op_code=OUTP; pq.arg1=7; pq.arg2=0;  /* no operation */
prog[16]=pq;

  /* Initialize the chip array with an image patch, compensating */
  /* for the lowpass mask size.                                  */

pq.op_code=IMLD; pq.arg1=0; pq.arg2=0; pq.arg3=MASKD;
prog[17]=pq;

  /* Convolution phase */

pq.op_code=LOOP; pq.arg1=0; pq.arg2=0;  /* arg1=0 => loop thru entire image */
prog[18]=pq;
pq.op_code=MLRR; pq.arg1=1; pq.arg2=0;  /* R6=upper right pixel value */
prog[19]=pq;
pq.op_code=LOAD; pq.arg1=1; pq.arg2=8;  /* R1=cache[8]=mask[i-1][j+1] */
prog[20]=pq;
pq.op_code=READ; pq.arg1=6; pq.arg2=2;  /* R2 = R6 */
prog[21]=pq;
pq.op_code=CONV; pq.arg1=0; pq.arg2=0;  /* R0+=R1*R2 */
prog[22]=pq;
pq.op_code=MRRL; pq.arg1=1; pq.arg2=0;  /* R6=upper center pixel value */
prog[23]=pq;
pq.op_code=LOAD; pq.arg1=1; pq.arg2=7;  /* R1=cache[7]=mask[i-1][j] */
prog[24]=pq;
pq.op_code=READ; pq.arg1=6; pq.arg2=2;  /* R2 = R6 */
prog[25]=pq;
pq.op_code=CONV; pq.arg1=0; pq.arg2=0;  /* R0+=R1*R2 */
prog[26]=pq;
pq.op_code=MRRL; pq.arg1=1; pq.arg2=0;  /* R6=upper left pixel value */
prog[27]=pq;
pq.op_code=LOAD; pq.arg1=1; pq.arg2=6;  /* R1=cache[6]=mask[i-1][j-1] */
prog[28]=pq;
pq.op_code=READ; pq.arg1=6; pq.arg2=2;  /* R2 = R6 */
prog[29]=pq;
pq.op_code=CONV; pq.arg1=0; pq.arg2=0;  /* R0+=R1*R2 */
prog[30]=pq;
pq.op_code=MDRU; pq.arg1=1; pq.arg2=0;  /* R6=middle left pixel value */
prog[31]=pq;
pq.op_code=LOAD; pq.arg1=1; pq.arg2=3;  /* R1=cache[3]=mask[i][j-1] */
prog[32]=pq;
pq.op_code=READ; pq.arg1=6; pq.arg2=2;  /* R2 = R6 */
prog[33]=pq;
pq.op_code=CONV; pq.arg1=0; pq.arg2=0;  /* R0+=R1*R2 */
prog[34]=pq;
pq.op_code=MLRR; pq.arg1=1; pq.arg2=0;  /* R6=center pixel value */
prog[35]=pq;
pq.op_code=LOAD; pq.arg1=1; pq.arg2=4;  /* R1=cache[4]=mask[i][j] */
prog[36]=pq;
pq.op_code=READ; pq.arg1=6; pq.arg2=2;  /* R2 = R6 */
prog[37]=pq;
```

```
pq.op_code=CONV; pq.arg1=0; pq.arg2=0;  /* R0+=R1*R2 */
prog[38]=pq;
pq.op_code=MLRR; pq.arg1=1; pq.arg2=0;  /* R6=middle right pixel value */
prog[39]=pq;
pq.op_code=LOAD; pq.arg1=1; pq.arg2=5;  /* R1=cache[5]=mask[i][j+1] */
prog[40]=pq;
pq.op_code=READ; pq.arg1=6; pq.arg2=2;  /* R2 = R6 */
prog[41]=pq;
pq.op_code=CONV; pq.arg1=0; pq.arg2=0;  /* R0+=R1*R2 */
prog[42]=pq;
pq.op_code=MDRU; pq.arg1=1; pq.arg2=0;  /* R6=upper right pixel value */
prog[43]=pq;
pq.op_code=LOAD; pq.arg1=1; pq.arg2=2;  /* R1=cache[2]=mask[i+1][j+1] */
prog[44]=pq;
pq.op_code=READ; pq.arg1=6; pq.arg2=2;  /* R2 = R6 */
prog[45]=pq;
pq.op_code=CONV; pq.arg1=0; pq.arg2=0;  /* R0+=R1*R2 */
prog[46]=pq;
pq.op_code=MRRL; pq.arg1=1; pq.arg2=0;  /* R6=lower center pixel value */
prog[47]=pq;
pq.op_code=LOAD; pq.arg1=1; pq.arg2=1;  /* R1=cache[1]=mask[i+1][j] */
prog[48]=pq;
pq.op_code=READ; pq.arg1=6; pq.arg2=2;  /* R2 = R6 */
prog[49]=pq;
pq.op_code=CONV; pq.arg1=0; pq.arg2=0;  /* R0+=R1*R2 */
prog[50]=pq;
pq.op_code=MRRL; pq.arg1=1; pq.arg2=0;  /* R6=lower left pixel value */
prog[51]=pq;
pq.op_code=LOAD; pq.arg1=1; pq.arg2=0;  /* R1=cache[0]=mask[i-1][j] */
prog[52]=pq;
pq.op_code=READ; pq.arg1=6; pq.arg2=2;  /* R2 = R6 */
prog[53]=pq;
pq.op_code=CONV; pq.arg1=0; pq.arg2=0;  /* R0+=R1*R2 = convolution */
prog[54]=pq;

  /* Convolution complete; output R0 as the convolution value */

pq.op_code=OUTP; pq.arg1=0; pq.arg2=0;
prog[55]=pq;
pq.op_code=WDRU; pq.arg1=71; pq.arg2=3; /* loop has produced 71x134 values */
prog[56]=pq;
pq.op_code=IMRD; pq.arg1=1; pq.arg2=3;  /* need one more image line for 72 */
prog[57]=pq;
pq.op_code=CLER; pq.arg1=0; pq.arg2=0;  /* clear R0 for next patch */
prog[58]=pq;
pq.op_code=LEND; pq.arg1=0; pq.arg2=0;  /* end of image loop */
prog[59]=pq;

  /* orderly exit */

pq.op_code=HALT; pq.arg1=0; pq.arg2=0;
prog[60]=pq;

/* Read input file name and open it */

printf("image file name: ");
scanf("%s",ifile);
if ((int)(image=fopen(ifile,"r")) == 0)
  {
  printf(" Error ! Could not open input file !\n");
```

```
    exit(1);
    }

  /* Read output file name and open it */

  printf("Output file name: ");
  scanf("%s",ofile);
  if ((int)(out=fopen(ofile,"wb")) == 0)
    {
    printf(" Error ! Could not open input file !\n");
    exit(1);
    }

  /* Read the image into the center SIZEDxSIZED section of the buffer  */

  for (i=SIZED-1; i>=0; i--)
    {
    fread(line,1,SIZED,image);
    for (j=0; j<=SIZED-1; j++)
      {
      word = line[j] & 0xff;
      if (word>255) word=255;
      else if (word<0) word=0;
      buffer[i][j] = word;
      }
    }

  fclose(image);

  controller(prog,buffer,output,256,256);

  printf(" number of machine cycles is %u\n",cycles);

    /* Output results */


  for(i=SIZED-1; i>=0; i--)
    {
    for(j=0; j<=SIZED-1; j++)
      {
      word = (output[i][j]/16) & 0xff;     /* application of weighting factor */
      if(word>255) word=255;
      else if (word<0) word=0;
      line[j]=(char)(word);
      }
    if ((fwrite(line,1,SIZED,out)) != SIZED)
      {
      printf(" Error ! line %3u written unsuccessfully\n",i);
      fclose(out);
      exit(1);
      }
    }

  fclose(out);
  }
```

```
/* This is a 3x3 mask algorithm for the systolic array. A horizontal Sobel mask
 * is used, as shown below:
 *
 *           /-1   -2   -1\
 *    mask = | 0    0    0|
 *           \ 1    2    1/
 */

#include <stdio.h>
#include <sys/file.h>
#include <sys/types.h>
#include <sys/uio.h>

#define SIZED 256
#define MASKD 3
#define EBUF (MASKD-1)/2
#define BSIZE SIZED+MASKD-1

#include "controller.c"

main()

{
char ifile[80];
char ofile[80];
int buffer[SIZED][SIZED];
int output[SIZED][SIZED];
char line[SIZED];
int i,j,word,temp;
INSTRUCTION pq;
FILE *image, *out;
static INSTRUCTION prog[SIZED];

  /* Initialization phase */

pq.op_code=CINI; pq.arg1=0; pq.arg2=1;  /* cache[0]=1 */
prog[0]=pq;
pq.op_code=STOR; pq.arg1=6; pq.arg2=2;  /* cache[2]=1 */
prog[1]=pq;
pq.op_code=READ; pq.arg1=6; pq.arg2=1;  /* R1 = R6(input) = 1 */
prog[2]=pq;
pq.op_code=READ; pq.arg1=1; pq.arg2=2;  /* R2 = R1 = 1        */
prog[3]=pq;
pq.op_code=READ; pq.arg1=1; pq.arg2=0;  /* R0 = R1 = 1        */
prog[4]=pq;
pq.op_code=CONV; pq.arg1=1; pq.arg2=8;  /* R0 = R0 + R1*R2 = 2 */
prog[5]=pq;
pq.op_code=STOR; pq.arg1=0; pq.arg2=1;  /* cache[1]=2 */
prog[6]=pq;
pq.op_code=CINI; pq.arg1=6; pq.arg2=-1; /* cache[6]=-1 */
prog[7]=pq;
pq.op_code=STOR; pq.arg1=6; pq.arg2=8;   * cache[8]=-1 */
prog[8]=pq;
pq.op_code=LOAD; pq.arg1=2; pq.arg2=6;  /* R2 = cache[6] = -1 */
prog[9]=pq;
pq.op_code=READ; pq.arg1=2; pq.arg2=0;  /* R0 = R2 = -1 */
prog[10]=pq;
pq.op_code=CONV; pq.arg1=2; pq.arg2=5;   * R0 = R0 + R1*R2 = -2 */
prog[11]=pq;
pq.op_code=STOR; pq.arg1=0; pq.arg2=7;  /* cache[7]=-2 */
```

```
prog[12]=pq;
pq.op_code=CLER; pq.arg1=0; pq.arg2=3;  /* R0 = 0 */
prog[13]=pq;
pq.op_code=STOR; pq.arg1=0; pq.arg2=3;  /* cache[3] = R0 = 0 */
prog[14]=pq;
pq.op_code=STOR; pq.arg1=0; pq.arg2=4;  /* cache[4] = R0 = 0 */
prog[15]=pq;
pq.op_code=STOR; pq.arg1=0; pq.arg2=5;  /* cache[5] = R0 = 0 */
prog[16]=pq;

  /* Initialize the chip array with an image patch, compensating */
  /* for the Sobel mask size.                                    */

pq.op_code=IMLD; pq.arg1=0; pq.arg2=0; pq.arg3=MASKD;
prog[17]=pq;

  /* Convolution phase */

pq.op_code=LOOP; pq.arg1=0; pq.arg2=0;  /* arg1=0 => loop thru entire image */
prog[18]=pq;
pq.op_code=MLRR; pq.arg1=1; pq.arg2=0;  /* R6=upper right pixel value */
prog[19]=pq;
pq.op_code=LOAD; pq.arg1=1; pq.arg2=8;  /* R1=cache[8]=mask[i-1][j+1] */
prog[20]=pq;
pq.op_code=READ; pq.arg1=6; pq.arg2=2;  /* R2 = R6 */
prog[21]=pq;
pq.op_code=CONV; pq.arg1=0; pq.arg2=0;  /* R0+=R1*R2 */
prog[22]=pq;
pq.op_code=MRRL; pq.arg1=1; pq.arg2=0;  /* R6=upper center pixel value */
prog[23]=pq;
pq.op_code=LOAD; pq.arg1=1; pq.arg2=7;  /* R1=cache[7]=mask[i-1][j] */
prog[24]=pq;
pq.op_code=READ; pq.arg1=6; pq.arg2=2;  /* R2 = R6 */
prog[25]=pq;
pq.op_code=CONV; pq.arg1=0; pq.arg2=0;  /* R0+=R1*R2 */
prog[26]=pq;
pq.op_code=MRRL; pq.arg1=1; pq.arg2=0;  /* R6=upper left pixel value */
prog[27]=pq;
pq.op_code=LOAD; pq.arg1=1; pq.arg2=6;  /* R1=cache[6]=mask[i-1][j-1] */
prog[28]=pq;
pq.op_code=READ; pq.arg1=6; pq.arg2=2;  /* R2 = R6 */
prog[29]=pq;
pq.op_code=CONV; pq.arg1=0; pq.arg2=0;  /* R0+=R1*R2 */
prog[30]=pq;
pq.op_code=MDRU; pq.arg1=1; pq.arg2=0;  /* R6=middle left pixel value */
prog[31]=pq;
pq.op_code=LOAD; pq.arg1=1; pq.arg2=3;  /* R1=cache[3]=mask[i][j-1] */
prog[32]=pq;
pq.op_code=READ; pq.arg1=6; pq.arg2=2;  /* R2 = R6 */
prog[33]=pq;
pq.op_code=CONV; pq.arg1=0; pq.arg2=0;  /* R0+=R1*R2 */
prog[34]=pq;
pq.op_code=MLRR; pq.arg1=1; pq.arg2=0;  /* R6=center pixel value */
prog[35]=pq;
pq.op_code=LOAD; pq.arg1=1; pq.arg2=4;  /* R1=cache[4]=mask[i][j] */
prog[36]=pq;
pq.op_code=READ; pq.arg1=6; pq.arg2=2;  /* R2 = R6 */
prog[37]=pq;
pq.op_code=CONV; pq.arg1=0; pq.arg2=0;  /* R0+=R1*R2 */
prog[38]=pq;
```

```
pq.op_code=MLRR; pq.arg1=1; pq.arg2=0;  /* R6=middle right pixel value */
prog[39]=pq;
pq.op_code=LOAD; pq.arg1=1; pq.arg2=5;  /* R1=cache[5]=mask[i][j+1] */
prog[40]=pq;
pq.op_code=READ; pq.arg1=6; pq.arg2=2;  /* R2 = R6 */
prog[41]=pq;
pq.op_code=CONV; pq.arg1=0; pq.arg2=0;  /* R0+=R1*R2 */
prog[42]=pq;
pq.op_code=MDRU; pq.arg1=1; pq.arg2=0;  /* R6=upper right pixel value */
prog[43]=pq;
pq.op_code=LOAD; pq.arg1=1; pq.arg2=2;  /* R1=cache[2]=mask[i+1][j+1] */
prog[44]=pq;
pq.op_code=READ; pq.arg1=6; pq.arg2=2;  /* R2 = R6 */
prog[45]=pq;
pq.op_code=CONV; pq.arg1=0; pq.arg2=0;  /* R0+=R1*R2 */
prog[46]=pq;
pq.op_code=MRRL; pq.arg1=1; pq.arg2=0;  /* R6=lower center pixel value */
prog[47]=pq;
pq.op_code=LOAD; pq.arg1=1; pq.arg2=1;  /* R1=cache[1]=mask[i+1][j] */
prog[48]=pq;
pq.op_code=READ; pq.arg1=6; pq.arg2=2;  /* R2 = R6 */
prog[49]=pq;
pq.op_code=CONV; pq.arg1=0; pq.arg2=0;  /* R0+=R1*R2 */
prog[50]=pq;
pq.op_code=MRRL; pq.arg1=1; pq.arg2=0;  /* R6=lower left pixel value */
prog[51]=pq;
pq.op_code=LOAD; pq.arg1=1; pq.arg2=0;  /* R1=cache[0]=mask[i-1][j] */
prog[52]=pq;
pq.op_code=READ; pq.arg1=6; pq.arg2=2;  /* R2 = R6 */
prog[53]=pq;
pq.op_code=CONV; pq.arg1=0; pq.arg2=0;  /* R0+=R1*R2 = Sh */
prog[54]=pq;
pq.op_code=READ; pq.arg1=0; pq.arg2=1;  /* R1 = Sh */
prog[55]=pq;
pq.op_code=LOAD; pq.arg1=2; pq.arg2=6;  /* R2 = -1 */
prog[56]=pq;
pq.op_code=CLER; pq.arg1=0; pq.arg2=2;  /* R0 = 0 */
prog[57]=pq;
pq.op_code=CONV; pq.arg1=0; pq.arg2=0;  /* R0 = -Sh */
prog[58]=pq;
pq.op_code=READ; pq.arg1=1; pq.arg2=3;  /* R3 = Sh */
prog[59]=pq;
pq.op_code=READ; pq.arg1=0; pq.arg2=4;  /* R4 = -Sh */
prog[60]=pq;
pq.op_code=COMP; pq.arg1=1; pq.arg2=3;  /* R5 = max(Sh,-Sh) */
prog[61]=pq;

  /* Convolution complete; output R5 as the convolution value */

pq.op_code=OUTP; pq.arg1=5; pq.arg2=0;
prog[62]=pq;
pq.op_code=WDRU; pq.arg1=71; pq.arg2=3; /* loop has produced 71x134 values */
prog[63]=pq;
pq.op_code=IMRD; pq.arg1=1; pq.arg2=3;  /* need one more image line for 72 */
prog[64]=pq;
pq.op_code=CLER; pq.arg1=0; pq.arg2=0;  /* clear R0 for next image patch */
prog[65]=pq;
pq.op_code=LEND; pq.arg1=0; pq.arg2=0;  /* end of image loop */
prog[66]=pq;
```

```
  /* orderly exit */

pq.op_code=HALT; pq.arg1=0; pq.arg2=0;
prog[67]=pq;

/* Read input file name and open it */

printf("image file name: ");
scanf("%s",ifile);
if ((int)(image=fopen(ifile,"r")) == 0)
  {
  printf(" Error ! Could not open input file !\n");
  exit(1);
  }

/* Read output file name and open it */

printf("Output file name: ");
scanf("%s",ofile);
if ((int)(out=fopen(ofile,"wb")) == 0)
  {
  printf(" Error ! Could not open input file !\n");
  exit(1);
  }

/* Read the image into the center SIZEDxSIZED section of the buffer  */

for (i=SIZED-1; i>=0; i--)
  {
  fread(line,1,SIZED,image);
  for (j=0; j<=SIZED-1; j++)
    {
    word = line[j] & 0xff;
    if (word>255) word=255;
    else if (word<0) word=0;
    buffer[i][j] = word;
    }
  }

fclose(image);

controller(prog,buffer,output,256,256);

printf(" number of machine cycles is %u\n",cycles);

  /* Output results */


for(i=SIZED-1; i>=0; i--)
  {
  for(j=0; j<=SIZED-1; j++)
    {
    word = output[i][j] & 0xff;
    if(word>255) word=255;
    else if (word<0) word=0;
    line[j]=(char)(word);
    }
  if ((fwrite(line,1,SIZED,out)) != SIZED)
    {
    printf(" Error ! line %3u written unsuccessfully\n",i);
```

```
        fclose(out);
        exit(1);
        }
    }

fclose(out);
}
```

```
/* This is a Sobel-based edge detection algorithm for the systolic array.
 * Horizontal and vertical Sobel masks are first used to detect horizontal
 * and vertical edges. The magnitude function is: mag=|Eh| + |Ev|. A user
 * supplied threshold value is also applied.
 *
 *           /-1   -2   -1\  (c[2],c[3],c[2])
 *      Eh = | 0    0    0|
 *           \ 1    2    1/  (c[0],c[1],c[0])
 *
 *           /-1    0    1\  (c[2],...,c[0])
 *      Ev = |-2    0    2|  (c[3],...,c[1])
 *           \-1    0    1/  (C[2],...,c[0])
 */

#include <stdio.h>
#include <sys/file.h>
#include <sys/types.h>
#include <sys/uio.h>

#define SIZED 256
#define MASKD 3
#define EBUF (MASKD-1)/2
#define BSIZE SIZED+MASKD-1

#include "controller.c"

main()

{
char ifile[80];
char ofile[80];
int buffer[SIZED][SIZED];
int output[SIZED][SIZED];
char line[SIZED];
int i,j,word,temp;
int *pt;
INSTRUCTION pq;
FILE *image, *out;
static INSTRUCTION prog[SIZED];

pt=&temp;
printf(" Enter the threshold value [0,255]: ");
scanf("%u",pt);

  /* Initialization phase */

pq.op_code=CINI; pq.arg1=0; pq.arg2=1;  /* cache[0]=1 */
prog[0]=pq;
pq.op_code=CINI; pq.arg1=10; pq.arg2=temp;  /* cache[10]=threshold */
prog[1]=pq;
pq.op_code=LOAD; pq.arg1=1; pq.arg2=0;  /* R1 = cache[0] = 1 */
prog[2]=pq;
pq.op_code=READ; pq.arg1=1; pq.arg2=2;  /* R2 = R1 = 1        */
prog[3]=pq;
pq.op_code=READ; pq.arg1=1; pq.arg2=0;  /* R0 = R1 = 1        */
prog[4]=pq;
pq.op_code=CONV; pq.arg1=1; pq.arg2=8;  /* R0 = R0 + R1*R2 = 2 */
prog[5]=pq;
pq.op_code=STOR; pq.arg1=0; pq.arg2=1;  /* cache[1]=2 */
prog[6]=pq;
```

```
pq.op_code=CINI; pq.arg1=2; pq.arg2=-1; /* cache[2]=-1 */
prog[7]=pq;
pq.op_code=LOAD; pq.arg1=2; pq.arg2=2;  /* R2 = cache[2] = -1 */
prog[8]=pq;
pq.op_code=READ; pq.arg1=2; pq.arg2=0;  /* R0 = R2 = -1 */
prog[9]=pq;
pq.op_code=CONV; pq.arg1=2; pq.arg2=5;  /* R0 = R0 + R1*R2 = -2 */
prog[10]=pq;
pq.op_code=STOR; pq.arg1=0; pq.arg2=3;  /* cache[3]=-2 */
prog[11]=pq;

  /* compute and store maximum gray scale value */

pq.op_code=CLER; pq.arg1=0; pq.arg2=3;  /* R0 = 0 */
prog[12]=pq;
pq.op_code=LOAD; pq.arg1=1; pq.arg2=1;  /* R1 = 2 */
prog[13]=pq;
pq.op_code=READ; pq.arg1=1; pq.arg2=2;  /* R2 = R1 = 2 */
prog[14]=pq;
pq.op_code=CONV; pq.arg1=0; pq.arg2=4;  /* R0 = R1*R2 = 4 */
prog[15]=pq;
pq.op_code=READ; pq.arg1=0; pq.arg2=1;  /* R1 = R0 = 4 */
prog[16]=pq;
pq.op_code=READ; pq.arg1=0; pq.arg2=2;  /* R2 = R0 = 4 */
prog[17]=pq;
pq.op_code=CLER; pq.arg1=0; pq.arg2=3;  /* R0 = 0 */
prog[18]=pq;
pq.op_code=CONV; pq.arg1=0; pq.arg2=4;  /* R0 = R1*R2 = 16 */
prog[19]=pq;
pq.op_code=READ; pq.arg1=0; pq.arg2=1;  /* R1 = R0 = 16 */
prog[20]=pq;
pq.op_code=READ; pq.arg1=0; pq.arg2=2;  /* R2 = R0 = 16 */
prog[21]=pq;
pq.op_code=LOAD; pq.arg1=0; pq.arg2=2;  /* R0 = -1 */
prog[22]=pq;
pq.op_code=CONV; pq.arg1=0; pq.arg2=4;  /* R0 = R0 + R1*R2 = 255 */
prog[23]=pq;
pq.op_code=STOR; pq.arg1=0; pq.arg2=11;  /* cache[11] = 255 */
prog[24]=pq;

  /* Initialize the chip array with an image patch, compensating */
  /* for the Sobel mask size.                                    */

pq.op_code=IMLD; pq.arg1=0; pq.arg2=0; pq.arg3=MASKD;
prog[25]=pq;

  /* Convolution phase. Both Sobel convolutions will be calculated.   */
  /* Sh will be stored in cache[9] and Sv will be stored in cache[8]. */

pq.op_code=LOOP; pq.arg1=0; pq.arg2=0;  /* arg1=0 => loop thru entire image */
prog[26]=pq;
pq.op_code=CLER; pq.arg1=0; pq.arg2=3;  /* R0 = 0 */
prog[27]=pq;
pq.op_code=MLRR; pq.arg1=1; pq.arg2=0;  /* R6=upper right pixel value */
prog[28]=pq;
pq.op_code=STOR; pq.arg1=6; pq.arg2=8;  /* cache[8]=image[i-1][j+1] */
prog[29]=pq;
pq.op_code=LOAD; pq.arg1=1; pq.arg2=2;  /* R1 = cache[2]=Eh[i-1][j+1]=-1 */
prog[30]=pq;
pq.op_code=READ; pq.arg1=6; pq.arg2=2;  /* R2 = R6 */
```

```
prog[31]=pq;
pq.op_code=CONV; pq.arg1=0; pq.arg2=0;  /* R0+=R1*R2 */
prog[32]=pq;
pq.op_code=MRRL; pq.arg1=1; pq.arg2=0;  /* R6=upper center pixel value */
prog[33]=pq;
pq.op_code=LOAD; pq.arg1=1; pq.arg2=3;  /* R1=cache[3]=Eh[i-1][j] */
prog[34]=pq;
pq.op_code=READ; pq.arg1=6; pq.arg2=2;  /* R2 = R6 */
prog[35]=pq;
pq.op_code=CONV; pq.arg1=0; pq.arg2=0;  /* R0+=R1*R2 */
prog[36]=pq;
pq.op_code=MRRL; pq.arg1=1; pq.arg2=0;  /* R6=upper left pixel value */
prog[37]=pq;
pq.op_code=LOAD; pq.arg1=1; pq.arg2=2;  /* R1=cache[2]=Eh,Ev[i-1][j-1] */
prog[38]=pq;
pq.op_code=READ; pq.arg1=6; pq.arg2=2;  /* R2 = R6 */
prog[39]=pq;
pq.op_code=CONV; pq.arg1=0; pq.arg2=0;  /* R0+=R1*R2 */
prog[40]=pq;
pq.op_code=STOR; pq.arg1=0; pq.arg2=9;  /* cache[9]=partial value of Sh */
prog[41]=pq;
pq.op_code=LOAD; pq.arg1=0; pq.arg2=8;  /* R0 = cache[8] = partial Sv */
prog[42]=pq;
pq.op_code=CONV; pq.arg1=0; pq.arg2=0;  /* R0+=R1*R2 */
prog[43]=pq;
pq.op_code=MDRU; pq.arg1=1; pq.arg2=0;  /* R6=middle left pixel value */
prog[44]=pq;
pq.op_code=LOAD; pq.arg1=1; pq.arg2=3;  /* R1=cache[3]=Ev[i][j-1] */
prog[45]=pq;
pq.op_code=READ; pq.arg1=6; pq.arg2=2;  /* R2 = R6 */
prog[46]=pq;
pq.op_code=CONV; pq.arg1=0; pq.arg2=0;  /* R0+=R1*R2 */
prog[47]=pq;
pq.op_code=MLRR; pq.arg1=2; pq.arg2=0;  /* R6=middle right pixel value */
prog[48]=pq;
pq.op_code=LOAD; pq.arg1=1; pq.arg2=1;  /* R1=cache[1]=Ev[i][j+1] */
prog[49]=pq;
pq.op_code=READ; pq.arg1=6; pq.arg2=2;  /* R2 = R6 */
prog[50]=pq;
pq.op_code=CONV; pq.arg1=0; pq.arg2=0;  /* R0+=R1*R2 */
prog[51]=pq;
pq.op_code=MDRU; pq.arg1=1; pq.arg2=0;  /* R6=lower right pixel value */
prog[52]=pq;
pq.op_code=LOAD; pq.arg1=1; pq.arg2=0;  /* R1=cache[0]=Ev[i+1][j+1] */
prog[53]=pq;
pq.op_code=READ; pq.arg1=6; pq.arg2=2;  /* R2 = R6 */
prog[54]=pq;
pq.op_code=CONV; pq.arg1=0; pq.arg2=0;  /* R0+=R1*R2 */
prog[55]=pq;
pq.op_code=MRRL; pq.arg1=2; pq.arg2=0;  /* R6=lower left pixel value */
prog[56]=pq;
pq.op_code=LOAD; pq.arg1=1; pq.arg2=2;  /* R1=cache[2]=Ev[i-1][j-1] */
prog[57]=pq;
pq.op_code=READ; pq.arg1=6; pq.arg2=2;  /* R2 = R6 */
prog[58]=pq;
pq.op_code=CONV; pq.arg1=0; pq.arg2=0;  /* R0+=R1*R2=Sv[i][j] */
prog[59]=pq;
pq.op_code=STOR; pq.arg1=0; pq.arg2=8;  /* cache[8]=Sv[i][j] */
prog[60]=pq;
pq.op_code=LOAD; pq.arg1=0; pq.arg2=9;  /* R0 = cache[9]=partial Sh */
```

```
prog[61]=pq;
pq.op_code=LOAD; pq.arg1=1; pq.arg2=0;  /* R1=cache[0]=Eh[i+1][j-1] */
prog[62]=pq;
pq.op_code=CONV; pq.arg1=0; pq.arg2=0;  /* R0+=R1*R2 */
prog[63]=pq;
pq.op_code=MLRR; pq.arg1=1; pq.arg2=0;  /* R6=lower center pixel value */
prog[64]=pq;
pq.op_code=LOAD; pq.arg1=1; pq.arg2=1;  /* R1=cache[1]=Eh[i+1][j] */
prog[65]=pq;
pq.op_code=READ; pq.arg1=6; pq.arg2=2;  /* R2 = R6 */
prog[66]=pq;
pq.op_code=CONV; pq.arg1=0; pq.arg2=0;  /* R0+=R1*R2 */
prog[67]=pq;
pq.op_code=MLRR; pq.arg1=1; pq.arg2=0;  /* R6=lower right pixel value */
prog[68]=pq;
pq.op_code=LOAD; pq.arg1=1; pq.arg2=0;  /* R1=cache[0]=Eh[i-1][j+1] */
prog[69]=pq;
pq.op_code=READ; pq.arg1=6; pq.arg2=2;  /* R2 = R6 */
prog[70]=pq;
pq.op_code=CONV; pq.arg1=0; pq.arg2=0;  /* R0+=R1*R2=Sh */
prog[71]=pq;
pq.op_code=STOR; pq.arg1=0; pq.arg2=9;  /* cache[9]=Sh */
prog[72]=pq;

  /* Both convolutions are complete. Next take the absolute value of Sh */
  /* and Sv in order to compute the magnitude.                          */

pq.op_code=LOAD; pq.arg1=1; pq.arg2=8;  /* R1 = Sv */
prog[73]=pq;
pq.op_code=LOAD; pq.arg1=2; pq.arg2=2;  /* R2 = -1 */
prog[74]=pq;
pq.op_code=CLER; pq.arg1=0; pq.arg2=2;  /* R0 = 0 */
prog[75]=pq;
pq.op_code=CONV; pq.arg1=0; pq.arg2=0;  /* R0 = -Sv */
prog[76]=pq;
pq.op_code=LOAD; pq.arg1=3; pq.arg2=8;  /* R3 = Sv */
prog[77]=pq;
pq.op_code=READ; pq.arg1=0; pq.arg2=4;  /* R4 = -Sv */
prog[78]=pq;
pq.op_code=COMP; pq.arg1=1; pq.arg2=3;  /* R5 = max(Sv,-Sv) */
prog[79]=pq;
pq.op_code=STOR; pq.arg1=5; pq.arg2=8;  /* cache[8] = abs(Sv) */
prog[80]=pq;
pq.op_code=LOAD; pq.arg1=1; pq.arg2=9;  /* R1 = Sh */
prog[81]=pq;
pq.op_code=LOAD; pq.arg1=2; pq.arg2=2;  /* R2 = -1 */
prog[82]=pq;
pq.op_code=CLER; pq.arg1=0; pq.arg2=2;  /* R0 = 0 */
prog[83]=pq;
pq.op_code=CONV; pq.arg1=0; pq.arg2=0;  /* R0 = -Sh */
prog[84]=pq;
pq.op_code=LOAD; pq.arg1=3; pq.arg2=8;  /* R3 = Sh */
prog[85]=pq;
pq.op_code=READ; pq.arg1=0; pq.arg2=4;  /* R4 = -Sh */
prog[86]=pq;
pq.op_code=COMP; pq.arg1=1; pq.arg2=3;  /* R5 = max(Sh,-Sh) */
prog[87]=pq;
pq.op_code=READ; pq.arg1=5; pq.arg2=0;  /* R0 = R5 = abs(Sh) */
prog[88]=pq;
pq.op_code=LOAD; pq.arg1=1; pq.arg2=0;  /* R1 = cache[0] = 1 */
```

```
prog[89]=pq;
pq.op_code=LOAD; pq.arg1=2; pq.arg2=9;  /* R2 = R5 = abs(Sv) */
prog[90]=pq;
pq.op_code=CONV; pq.arg1=5; pq.arg2=0;  /* R0 = abs(Sv) + abs(Sh) */
prog[91]=pq;
pq.op_code=READ; pq.arg1=0; pq.arg2=3;  /* R3 = R0 = mag */
prog[92]=pq;
pq.op_code=LOAD; pq.arg1=4; pq.arg2=11;  /* R4 = cache[11] = max grayscale */
prog[93]=pq;
pq.op_code=COMP; pq.arg1=5; pq.arg2=0;  /* R0 = min(255,abs(Sv)+abs(Sh)) */
prog[94]=pq;
pq.op_code=STOR; pq.arg1=0; pq.arg2=9;  /* cache[9] = limited Sobel mag */
prog[95]=pq;

  /* Compare the computed magnitude with the threshold value. Output 0 if */
  /* the threshold is larger, and the limited magnitude otherwise.        */

pq.op_code=LOAD; pq.arg1=3; pq.arg2=10;  /* R3 = threshold */
prog[96]=pq;
pq.op_code=LOAD; pq.arg1=4; pq.arg2=9;  /* R4 = cache[9] = limited Sobel mag */
prog[97]=pq;
pq.op_code=COMP; pq.arg1=5; pq.arg2=0;  /* R5 = max(threshold,limited mag) */
prog[98]=pq;
pq.op_code=READ; pq.arg1=5; pq.arg2=1;  /* R1 = max */
prog[99]=pq;
pq.op_code=READ; pq.arg1=0; pq.arg2=4;  /* R4 = min(threshold,limited mag) */
prog[100]=pq;
pq.op_code=CLER; pq.arg1=0; pq.arg2=9;  /* R0 = 0, default value */
prog[101]=pq;
pq.op_code=OUTP; pq.arg1=0; pq.arg2=0;  /* output register(7) = R0 = 0 */
prog[102]=pq;
pq.op_code=COMP; pq.arg1=0; pq.arg2=9;  /* eq=T if min = threshold */
prog[103]=pq;
pq.op_code=IFEQ; pq.arg1=5; pq.arg2=0;  /* if eq != T, output default */
prog[104]=pq;
pq.op_code=OUTP; pq.arg1=1; pq.arg2=9;  /* if eq == T, output max */
prog[105]=pq;

  /* output the final edge-image value and start on the next patch of the */
  /* input image.                                                         */

pq.op_code=WDRU; pq.arg1=71; pq.arg2=3; /* loop has produced 71x134 values */
prog[106]=pq;
pq.op_code=IMRD; pq.arg1=1; pq.arg2=3;  /* need one more image line for 72 */
prog[107]=pq;
pq.op_code=LEND; pq.arg1=0; pq.arg2=0;  /* end of image loop */
prog[108]=pq;

  /* orderly exit */

pq.op_code=HALT; pq.arg1=0; pq.arg2=0;
prog[109]=pq;

/* Read input file name and open it */

printf("image file name: ");
scanf("%s",ifile);
if ((int)(image=fopen(ifile,"r")) == 0)
  {
  printf(" Error ! Could not open input file !\n");
```

```
    exit(1);
    }

/* Read output file name and open it */

printf("Output file name: ");
scanf("%s",ofile);
if ((int)(out=fopen(ofile,'wb")) == 0)
  {
  printf(" Error ! Could not open input file !\n");
  exit(1);
  }

/* Read the image into the center SIZEDxSIZED section of the buffer  */

for (i=SIZED-1; i>=0; i--)
  {
  fread(line,1,SIZED,image);
  for (j=0; j<=SIZED-1; j++)
    {
    word = line[j] & 0xff;
    if (word>255) word=255;
    else if (word<0) word=0;
    buffer[i][j] = word;
    }
  }

fclose(image);

controller(prog,buffer,output,256,256);

printf(" number of machine cycles is %u\n",cycles);

  /* Output results */


for(i=SIZED-1; i>=0; i--)
  {
  for(j=0; j<=SIZED-1; j++)
    {
    word = output[i][j] & 0xff;
    if(word>255) word=255;
    else if (word<0) word=0;
    line[j]=(char)(word);
    }
  if ((fwrite(line,1,SIZED,out)) != SIZED)
    {
    printf(" Error ! line %3u written unsuccessfully\n",i);
    fclose(out);
    exit(1);
    }
  }

fclose(out);
}
```

APPENDIX 2:

SIMULATOR SOURCE CODE

```
/* controller(program,input,output,height,width)  */

/* This function controlls the operation of an image processing computer
 * composed of an array of mesh-connected systolic array processing chips.
 * The computer is called chip_array. A specified program is applied to
 * the input image and the result returned in the output image. The size of
 * the images (height and width) is specified. A HALT or any illegal command
 * terminates the program. The following list of defines are the legal
 * instructions. Those with opcodes>15 are controller directives only.
 * Required arguments are shown by an entry in the column. Optional
 * arguments are given in parentheses.
 */
/*     instr  opcode  arg1  arg2          description                    */
                  /*                                                     */
#define RESET    -1  /*                 Resets the compare unit in each PE to  */
                  /*                 an equal state.                     */
                  /*                                                     */
#define LOAD      0  /*  R      i       Loads register R from cache memory lo- */
                  /*                 cation i. R=0-4 are the convolution,  */
                  /*                 multiply, and compare input registers; */
                  /*                 R=5 is illegal; R=6 is the input      */
                  /*                 register; and R=7 is the output       */
                  /*                 register.                             */
                  /*                                                       */
#define STOR      1  /*  R      i       Stores the contents of register R in   */
                  /*                 cache memory location i. R=0-2,6,7 are */
                  /*                 defined as for LOAD. R=5 is the com-   */
                  /*                 pare output register. R=3 and R=4 are  */
                  /*                 illegal.                               */
                  /*                                                        */
#define OUTP      2  /*  R              Moves the contents of register R to    */
                  /*                 the output register. R is defined as   */
                  /*                 for STOR. OUTP 7 is a no operation.    */
                  /*                                                        */
#define READ      3  /*  R1     R2      Moves the contents of register R1 to   */
                  /*                 register R2. R1 may be any register    */
                  /*                 and R2 may be any register except 5.   */
                  /*                 R1 == R2 is a no operation.            */
                  /*                                                        */
#define CLER      4  /*  R              Sets the contents of register R to     */
                  /*                 zero. R is defined as for LOAD.        */
                  /*                                                        */
#define COMP      5  /*                 Compares the contents of the compare   */
                  /*                 input registers and moves the smaller  */
                  /*                 into the convclution register and the  */
                  /*                 larger into the compare output reg-    */
                  /*                 ister.                                 */
                  /*                                                        */
#define CONV      6  /*                 Multiplies the contents of the multi-  */
                  /*                 ply registers and adds the result to   */
                  /*                 the convolution register.              */
                  /*                                                        */
#define MDRU      8  /*  n              Updates the input register from the    */
                  /*                 output register of the upper neighbor  */
                  /*                 PE. If there is no upper neighbor, the */
                  /*                 input register is update from the      */
                  /*                 upper boundary register. If there is   */
                  /*                 no lower neighbor, the lower boundary  */
                  /*                 register is updated. Then the output   */
                  /*                 register is updated from the input     */
```

```
                    /*              register. The instruction is repeated  */
                    /*              n times.                               */
                    /*                                                     */
#define MURD     9  /*  n           Updates the input register from the    */
                    /*              output register of the lower neighbor  */
                    /*              PE. If there is no lower neighbor, the */
                    /*              input register is updated from the     */
                    /*              lower boundary register. If there is   */
                    /*              no upper neighbor, the upper boundary  */
                    /*              register is updated. Then the output   */
                    /*              register is updated from the input     */
                    /*              register. The instruction is repeated  */
                    /*              n times.                               */
                    /*                                                     */
#define MLRR    10  /*  n           Updates the input register from the    */
                    /*              output register of the right neighbor  */
                    /*              PE. If there is no right neighbor, the */
                    /*              input register is update from the      */
                    /*              right boundary register. If there is   */
                    /*              no left neighbor, the left boundary    */
                    /*              register is updated. Then the output   */
                    /*              register is updated from the input     */
                    /*              register. The instruction is repeated  */
                    /*              n times.                               */
                    /*                                                     */
#define MRRL    11  /*  n           Updates the input register from the    */
                    /*              output register of the left neighbor   */
                    /*              PE. If there is no left neighbor, the  */
                    /*              input register is update from the      */
                    /*              left boundary register. If there is    */
                    /*              no right neighbor, the right boundary  */
                    /*              register is updated. Then the output   */
                    /*              register is updated from the input     */
                    /*              register. The instruction is repeated  */
                    /*              n times.                               */
                    /*                                                     */
#define WDRU    16  /*  n1    n2    Controller directive only. The next    */
                    /*              line from the input image is read into */
                    /*              the upper boundary register. Then a    */
                    /*              MDRU instruction is executed. Finally, */
                    /*              the updated lower boundary register is */
                    /*              moved into the output image (at the    */
                    /*              location corresponding to the location */
                    /*              of the original input pixels). This    */
                    /*              whole process is repeated n1 times.    */
                    /*              The second parameter n2 indicates the  */
                    /*              mask size that should be compensated   */
                    /*              for when the end of a patch is reached.*/
                    /*                                                     */
#define LOOP    17  /*  n           Controller directive only. This marks  */
                    /*              the beginning of a loop that is to be  */
                    /*              repeated for every element of the in-  */
                    /*              put image, if n=0, or until the end of */
                    /*              the current swath, otherwise.          */
                    /*                                                     */
#define LEND    18  /*              Controller directive only. This marks  */
                    /*              the end of a loop.                     */
                    /*                                                     */
#define HALT    19  /*              Controller directive only. This marks  */
                    /*              the end of the program. A zero is      */
```

```
                    /*                  returned.                                  */
                    /*                                                             */
#define IFEQ    20  /*                  Controller directive only. Causes the      */
                    /*                  controller to set the if_control line      */
                    /*                  to each PE, which allows selective         */
                    /*                  execution of the next instruction          */
                    /*                  based on the results of the most           */
                    /*                  recent COMP instruction. (If the two       */
                    /*                  values compared at the PE were equal,      */
                    /*                  the next instruction is executed.          */
                    /*                  Otherwise, it isn't.)                      */
                    /*                                                             */
#define CINI    24  /*  i     n         Controller directive only. This init-      */
                    /*                  ializes the ith location of cache          */
                    /*                  memory on e ery PE in the chip array       */
                    /*                  to the value n.                            */
                    /*                                                             */
#define IMLD    25  /*  n1  n2  n3      Controller directive only. The entire      */
                    /*                  chip array is initialized via repeated     */
                    /*                  updates of the upper boundary regis-       */
                    /*                  ter followed by MDRU commands to the       */
                    /*                  subimage of the input image whose          */
                    /*                  lower left corner is located at            */
                    /*                  (n1,n2). The whole process is com-         */
                    /*                  pensated for a mask of size n3 x n3        */
                    /*                  such that the bottom row of PEs in the     */
                    /*                  chip array contains the image pixels       */
                    /*                  corresponding to the ones operated on      */
                    /*                  by the bottom row of the mask.             */
                    /*                                                             */
#define IMRD    26  /*  n               Controller directive only. This causes     */
                    /*                  n lines to be read from the input          */
                    /*                  image and moved from the upper reg-        */
                    /*                  isters into the chip array. If n=1,        */
                    /*                  only the upper registers are affected.     */

typedef int bool;           /* Defines a boolean variable.                        */

typedef struct              /* Defines the type INSTRUCTION. Each instruction     */
          {                 /* contains an opcode and two arguments as de-        */
          int op_code;      /* scribed above.                                     */
          int arg1;
          int arg2;
          int arg3;
          bool if_control;}
          INSTRUCTION;

#define TRUE -1
#define FALSE 0

#include "chip_array.c"

#define SWATH_WD CHIP_ARRAY_WD*PE_WD

#define PATCH_HT CHIP_ARRAY_HT*PE_HT

#define MAX_SIZE 1024

static int cycles = 0;
```

```
int controller(program,input,output,height,width)

INSTRUCTION program[SIZED];
int input[SIZED][SIZED];
int output[SIZED][SIZED];
int height,width;

{
int pc,i,j,loopc;
INSTRUCTION instr,instr1,instr2,instr3;
int loopb[10],count[10];
int inx,iny,outx,outy,tx,ty,tc,tp,tm,tmc,last_inx;
int errcode,stage;
bool full_image,if_control;

pc=0; stage=0;
loopc=-1;
inx=0; iny=0; outx=0; outy=0; last_inx=0;
full_image=TRUE; if_control=FALSE;
instr.op_code=RESET;
errcode=chip_array(instr);
while(errcode==0)
  {
  if(pc<MAX_SIZE)
    {
    instr=program[pc];
    instr.if_control=if_control;
    switch(instr.op_code)
      {
      case    LOAD:  errcode=chip_array(instr);
                     cycles+=1;
                     if_control=FALSE;
                     break;

      case    STOR:  errcode=chip_array(instr);
                     cycles+=1;
                     if_control=FALSE;
                     break;

      case    OUTP:  errcode=chip_array(instr);
                     cycles+=1;
                     if_control=FALSE;
                     break;

      case    READ:  errcode=chip_array(instr);
                     cycles+=1;
                     if_control=FALSE;
                     break;

      case    CLER:  errcode=chip_array(instr);
                     cycles+=1;
                     if_control=FALSE;
                     break;

      case    COMP:  errcode=chip_array(instr);
                     cycles+=2;
                     if_control=FALSE;
                     break;

      case    CONV:  errcode=chip_array(instr);
```

```
                cycles+=3;
                if_control=FALSE;
                break;

case    MDRU:   for(i=instr.arg1; i>0 && errcode ==0; i--)
                  {
                  if((errcode=chip_array(instr))==0)
                    {
                    instr1.op_code=OUTP; /* inp reg -> out reg */
                    instr1.arg1=6;
                    instr1.if_control=instr.if_control;
                    if((errcode=chip_array(instr1))!=0)
                      errcode+=MDRU-OUTP;
                    }
                  cycles+=2;
                  }
                if_control=FALSE;
                break;

case    MURD:   for(i=instr.arg1; i>0 && errcode ==0; i--)
                  {
                  if((errcode=chip_array(instr))==0)
                    {
                    instr1.op_code=OUTP; /* inp reg -> out reg */
                    instr1.arg1=6;
                    instr1.if_control=instr.if_control;
                    if((errcode=chip_array(instr1))!=0)
                      errcode+=MURD-OUTP;
                    }
                  cycles+=2;
                  }
                if_control=FALSE;
                break;

case    MLRR:   for(i=instr.arg1; i>0 && errcode ==0; i--)
                  {
                  if((errcode=chip_array(instr))==0)
                    {
                    instr1.op_code=OUTP; /* inp reg -> out reg */
                    instr1.arg1=6;
                    instr1.if_control=instr.if_control;
                    if((errcode=chip_array(instr1))!=0)
                      errcode+=MLRR-OUTP;
                    }
                  cycles+=2;
                  }
                if_control=FALSE;
                break;

case    MRRL:   for(i=instr.arg1; i>0 && errcode ==0; i--)
                  {
                  if((errcode=chip_array(instr))==0)
                    {
                    instr1.op_code=OUTP; /* inp reg -> out reg */
                    instr1.arg1=6;
                    instr1.if_control=instr.if_control;
                    if((errcode=chip_array(instr1))!=0)
                      errcode+=MRRL-OUTP;
                    }
                  cycles+=2;
```

```
                    }
                  if_control=FALSE;
                  break;

case    WDRU:     tm=(int)((instr.arg2-1)/2);
                  for(i=0; i<instr.arg1 && errcode==0 &&
                          ((full_image && outy<width) ||
                           (full_image==FALSE && outx<height)); i++)
                    {
                    if(inx>=height && full_image)
                      {
                      if(tm==0)
                        {
                        iny+=SWATH_WD;
                        errcode=image_line(0,iny-tm,input,
                                           height,width,if_control,WDRU);
                        inx=1;
                        }
                      else
                        {
                        if(stage==0)
                          {
                          stage+=1;
                          tmc=tm;
                          last_inx=height-1;
                          }
                        else if(stage==1 && tmc<=0)
                          {
                          stage+=1;
                          tmc=tm;
                          last_inx=0;
                          iny+=(SWATH_WD)-2*tm;
                          }
                        else if(stage==2 && tmc<=0)
                          {
                          stage=0;
                          errcode=image_line(last_inx,iny-tm,input,
                                             height,width,if_control,WDRU);
                          inx=1;
                          }
                        if(stage>0)
                          {
                          errcode=image_line(last_inx,iny-tm,input,
                                             height,width,if_control,WDRU);
                          tmc--;
                          }
                        }
                      }
                    else if(inx<height)
                      {
                      errcode=image_line(inx,iny-tm,input,
                                         height,width,if_control,WDRU);
                      inx++;
                      }
                    else
                      errcode=image_line(inx,iny-tm,input,
                                         height,width,if_control,WDRU);
                    cycles+=2;
                    if((full_image && outy<width && outx>=0) ||
                       (full_image==FALSE && outx<height))
```

```
                    {
                    for(j=tm; j<SWATH_WD-tm && errcode==0; j++)
                      {
                      ty=j+outy-tm;
                      if(ty<0)
                        errcode=WDRU;
                      else if(ty<width)
                        output[outx][ty]=
                           lower_registers[(int)(j/(PE_WD))][j%(PE_WD)];
                      }
                    cycles+=1;
                    }
                  outx++;
                  if(outx>=height && full_image)
                    {
                    outx=-2*tm;
                    outy+=(SWATH_WD)-2*tm;
                    }
                  if(outy<width) cycles+=1;
                  }
                if_control=FALSE;
                break;

case    LOOP:   loopc++;
                loopb[loopc]=pc;
                count[loopc]=0;
                if(instr.arg1==0) full_image=TRUE;
                else full_image=FALSE;
                if_control=FALSE;
                break;

case    LEND:   if((count[loopc]++)<height*(width/(SWATH_WD)+1))
                  if((full_image && outy<width) ||
                     (full_image==FALSE && outx<height))
                    pc=loopb[loopc];
                  else loopc--;
                else loopc--;
                if_control=FALSE;
                break;

case    HALT:   errcode=-1; /* normal exit */
                break;

case    IFEQ:   if_control=TRUE;
                break;

case    CINI:   for(i=0; i<CHIP_ARRAY_WD; i++)
                  for(j=0; j<PE_WD; j++)
                    upper_registers[i][j]=
                          instr.arg2;
                instr1.op_code=MDRU;
                instr1.arg1=1;
                instr1.if_control=instr.if_control;
                instr3.op_code=OUTP; /* inp reg -> out reg */
                instr3.arg1=5;
                instr3.if_control=instr.if_control;
                for(i=0; i<PATCH_HT && errcode==0; i++)
                  {
                  if((errcode=chip_array(instr1))!=0)
                    errcode+=CINI-MDRU;
```

```
                        else if((errcode=chip_array(instr3))!=0)
                          errcode+=CINI-OUTP;
                        cycles+=2;
                        }
                      if(errcode==0)
                        {
                        instr2.op_code=STOR;
                        instr2.arg1=6;
                        instr2.arg2=instr.arg1;
                        instr2.if_control=instr.if_control;
                        if((errcode=chip_array(instr2))!=0)
                          errcode+=CINI-STOR;
                        cycles+=1;
                        }
                      if_control=FALSE;
                      break;

        case    IMLD: tm=(int)((instr.arg3-1)/2);
                      for(i=0; i<PATCH_HT && errcode==0; i++)
                        {
                        errcode=image_line(i+instr.arg1-tm,instr.arg2-tm,
                                           input,height,width,if_control,IMLD);
                        cycles+=2;
                        }
                      if(errcode==0)
                        {
                        errcode=read_line(i+instr.arg1-tm,instr.arg2-tm,
                                          input,height,width,IMLD);
                        cycles+=1;
                        }
                      outx=instr.arg1;
                      outy=instr.arg2;
                      inx=outx+PATCH_HT-2*tm;
                      iny=outy;
                      if_control=FALSE;
                      break;

        case    IMRD: tm=(int)((instr.arg2-1)/2);
                      for(i=0; i<instr.arg1; i++)
                        {
                        errcode=image_line(inx,iny-tm,input,
                                           height,width,if_control,IMRD);
                        cycles+=2;
                        }
                      if(errcode==0)
                        {
                        errcode=read_line(inx+1,iny-tm,input,height,width,IMRD);
                        cycles+=1;
                        }
                      if_control=FALSE;
                      break;

        DEFAULT:      errcode=2;  /* illegal instruction error */
                      break;
        };
      pc++;
      }
    else errcode=1;
    }
  if (errcode==-1 || errcode==0)
```

```
  errcode=0;    /* normal exit; no run-time errors */
else if(errcode==1)
  printf(" Error ! Program exceeds maximum size !");
else if(errcode==2)
  printf (" Error ! Illegal instruction %u at pc=%u, execution halted !",
           instr.op_code,(pc-1));
else
  printf (" Error ! Instruction %u at pc=%u not executed properly !",
           errcode,pc);
return(errcode);
}


int image_line(x,y,buf_il,h,w,ifc,err)

int x,y;
int buf_il[SIZED][SIZED];
int h,w;
bool ifc;
int err;

{
INSTRUCTION instr1,instr2,instr3;
int tx,ty,i,j,t1,t2;
int errcode;

if(y<w) errcode=read_line(x,y,buf_il,h,w,err);
else errcode=0;
if(errcode==0)
  {
  instr1.op_code=MDRU;
  instr1.arg1=1;
  instr1.if_control=ifc;
  instr3.op_code=OUTP; /* inp reg -> out reg */
  instr3.arg1=6;
  instr3.if_control=ifc;
  if((errcode=chip_array(instr1))!=0)
    errcode+=err-MDRU;
  else if((errcode=chip_array(instr3))!=0)
    errcode+=err-OUTP;
  }
return(errcode);
}



int read_line(x,y,buf_il,h,w,err)

int x,y;
int buf_il[SIZED][SIZED];
int h,w,err;

{
int tx,ty,i,j,t1,t2;
int errcode;

if(x>=h) errcode=err;
else
  {
  errcode=0;
```

```
  if(x<0) tx=0;
  else tx=x;
  for(j=0; j<SWATH_WD; j++)
    {
    if((ty=j+y)<0) ty=0;
    else if(ty>=w) ty=w-1;
    t1=(int)(j/(PE_WD));
    t2=(int)(j%(PE_WD));
    upper_registers[t1][t2]=buf_il[tx][ty];
    }
  }
return(errcode);
}
```

```
/* int chip_array(instr) */

/* This procedure executes an instruction from the controller on the chip
 * array. Each chip is a systolic mesh-connected array of processing elements
 * (PE). The array is 4 chips wide by 3 chips high, for a total of 12*24*34
 * PEs. Each PE executes one instruction from the controller at the same time.
 * The chip array is shown below:
 *
 *                              input image memory
 *                                      |
 *                 .--------------.--------------.--------------.
 *                 |              |              |              |
 *                 v              v              v              v
 *                reg            reg            reg            reg
 *                 ^              ^              ^              ^
 *                 |              |              |              |
 *                 v              v              v              v
 *           .----------.   .----------.   .----------.   .----------.
 *           |  chip    |   |  chip    |   |  chip    |   |  chip    |
 *      .--> | [0][0]   | <-> | [0][1]   | <-> | [0][2]   | <-> | [0][3]   |<--.
 *      |    '----------'   '----------'   '----------'   '----------'   |
 *      |         ^              ^              ^              ^         |
 *      '-------- | ------------ | ------------ | ------------ | --------'
 *                v              v              v              v
 *           .----------.   .----------.   .----------.   .----------.
 *           |  chip    |   |  chip    |   |  chip    |   |  chip    |
 *      .--> | [1][0]   | <-> | [1][1]   | <-> | [1][2]   | <-> | [1][3]   |<--.
 *      |    '----------'   '----------'   '----------'   '----------'   |
 *      |         ^              ^              ^              ^         |
 *      '-------- | ------------ | ------------ | ------------ | --------'
 *                v              v              v              v
 *           .----------.   .----------.   .----------.   .----------.
 *           |  chip    |   |  chip    |   |  chip    |   |  chip    |
 *      .--> | [2][0]   | <-> | [2][1]   | <-> | [2][2]   | <-> | [2][3]   |<--.
 *      |    '----------'   '----------'   '----------'   '----------'   |
 *      |         ^              ^              ^              ^         |
 *      '-------- | ------------ | ------------ | ------------ | --------'
 *                v              v              v              v
 *               reg            reg            reg            reg
 *                |              |              |              |
 *                '--------------'--------------'--------------'
 *                                      |
 *                                      v
 *                              output image memory
 *
 * The top and bottom rows of interface buffers are used by the controller
 * to read from and write to image memory. These edge registers can be affected
 * by a "move" instruction issued to the chip array. The controller will
 * decide when the data is useful and when it is not. This procedure returns
 * the error code from the last chip successfully executed.
 */

#define CHIP_ARRAY_HT 3
#define CHIP_ARRAY_WD 4

#include "chip.c"

static int upper_registers[CHIP_ARRAY_WD][PE_WD];
static int lower_registers[CHIP_ARRAY_WD][PE_WD];
```

```
int chip_array(instr)

INSTRUCTION instr;

{
int i,j,k,temp;
int *upper[PE_WD],*lower[PE_WD];
int *pur[CHIP_ARRAY_WD][PE_WD],*pdr[CHIP_ARRAY_WD][PE_WD];
int errcode;


errcode=0;
for(i=0; i<CHIP_ARRAY_HT && errcode==0; i++)
  for(j=0; j<CHIP_ARRAY_WD && errcode==0; j++)
    errcode= chip(i,j,upper_registers[j],lower_registers[j],instr);
return(errcode);
}
```

```
/* int chip(chipx, chipy, up, down, instr) */

/* This program simulates a 24x34 systolic processing array chip existing in
 * an image processing computer using multiple chips. Each processing element
 * (PE) within the chip is mesh-connected. The PEs on the edge of the chip
 * are connected to an output pin such that if attached to the corresponding
 * pin of the adjacent chip, the two PEs connected to the pins are also mesh
 * connected. The arrangements are illustrated below for a hypothetical
 * connection between four chips (numbered [1][1], [1][2], [2][1], and [2][2]).
 * Chip boundaries are shown by +.
 *
 *         chip[1][1]                                    chip[1][2]
 *          ^        ^        ^                  ^       ^       ^
 *      <--PE22,32--PE22,33--PE22,34+ --- +PE22,1--PE22,2--PE22,3-->
 *          |        |        |                  |       |       |
 *      <--PE22,32--PE22,33--PE22,34+ --- +PE22,1--PE22,2--PE22,3-->
 *          |        |        |                  |       |       |
 *      <--PE22,32--PE22,33--PE22,34+ --- +PE22,1--PE22,2--PE22,3-->
 *          +        +        +                  +       +       +
 *          !        |        |                  |       |       !
 *          +        +        +                  +       +       +
 *      <--PE22,32--PE22,33--PE22,34+ --- +PE22,1--PE22,2--PE22,3-->
 *          |        |        |                  |       |       |
 *      <--PE22,32--PE22,33--PE22,34+ --- +PE22,1--PE22,2--PE22,3-->
 *          |        |        |                  |       |       |
 *      <--PE22,32--PE22,33--PE22,34+ --- +PE22,1--PE22,2--PE22,3-->
 *          v        v        v                  v       v       v
 *          chip[2][1]                                    chip[2][2]
 *
 * The function returns the error code received from the last PE.
 */

#define PE_HT 24
#define PE_WD 34

#include "pe.c"

int chip(chipx, chipy, up, down, instr)

int chipx, chipy;
int up[PE_WD], down[PE_WD];
INSTRUCTION instr;

{
int i,j;
int *pe_u, *pe_d;
int errcode;

errcode=0;
for(i=0; i<PE_HT && errcode==0; i++)
  for(j=0; j<PE_WD && errcode==0; j++)
    errcode= pe(chipx, chipy, i, j, up, down, instr);
return(errcode);
}
```

```
/* pe(cx, cy, px, py, pu, pd, instr) */

/* Each PE within the chip contains the elements diagrammed below. Registers
 * are marked with the number by which they are commanded. Xs represent two-
 * phase clock nodes at which points the controller commands instruction
 * execution. The "equal" output of the compare register is ORed with the
 * inverse of if_control from the controller; the result is ANDed with every
 * other control line. The effect is that if if_control is set and equal is
 * false, the instruction is inhibited; otherwise it proceeds. Due to lack of
 * space, this feature is not shown in full.
 *
 *                                     (from upper neighbor)
 *                                             ^
 *                                             |
 *                                             X1
 *                                             |
 *                   (to left neighbor)<--X2--|--X3--> (to right neighbor)
 *    .----X4----------------------------------|
 *    |                                        |
 *    v                                        v
 * (from lower                           register_inp
 *  neighbor)                                  |
 *                                             X6
 *                                             |
 *                                             v
 * ===========================================================================
 *    |          |          |          |       |          |         |     |
 *    X5         X9         X10        X8      X11        X12       X13   X7
 *    |          |          |          |       |          |         |     |
 *    |          |          |          |       v          v         |     |
 *    |      register 1  register 2    |   register 3  register 4   |     |
 *    |          |          |          |         |        |         |     |
 *  c m          |          |          |         v        v         |     |
 *  a e          |          |          |     compare unit<--X14     |     |
 *  c m          |          |          |         |  |     |         |     |
 *  h o          v          v          |         |  |     |         |     |
 *  e r       12_bit_multiplier     register 0<--<  |     v         |     |
 *    y             |                  |         |  | register 5    |     |
 *                  v                  |         |  |     |         |     |
 *               register              |         |  |     '---------'     |
 *                  |                  |         |  |eq                   |
 *                  X15                X16       |  '---|\                |
 *                  |                  |         |       > >---> (to      |
 *                  '-------.       .-'          |  .---|/       all      |
 *                          |       |            |  |            clocks)  |
 *                          v       v            |  - (not)               |
 *                        12_bit_adder           | if-control             |
 *                             |                 |  (from                 |
 *                             v                 |   controller)          |
 *                        register--X17----------'                        |
 *  (to upper                                                             |
 *   neighbor)                                                            |
 *    ^                   register_out<-----------------------------------'
 *    |                        |
 *    '------------------------|
 *         (to left neighbor)<---|--->(to right neighbor)
 *                             v
 *                     (to bottom neighbor)
 *
 * The PE returns an error code equal to the opcode of the instruction -3.
```

```
 */

#define CACHESIZE 32
#define REG_NUM 6

int pe(cx, cy, px, py, pu, pd, instr)

int cx, cy, px, py;
int pu[PE_WD], pd[PE_WD];
INSTRUCTION instr;

{
static int reg_inp[CHIP_ARRAY_HT]
                  [CHIP_ARRAY_WD]
                  [PE_HT]
                  [PE_WD];

static int reg[CHIP_ARRAY_HT]
              [CHIP_ARRAY_WD]
              [PE_HT]
              [PE_WD]
              [REG_NUM];

static int reg_out[CHIP_ARRAY_HT]
                  [CHIP_ARRAY_WD]
                  [PE_HT]
                  [PE_WD];

static int cache  [CHIP_ARRAY_HT]
                  [CHIP_ARRAY_WD]
                  [PE_HT]
                  [PE_WD]
                  [CACHESIZE];

static bool eq[CHIP_ARRAY_HT]
              [CHIP_ARRAY_WD]
              [PE_HT]
              [PE_WD];

int errcode,temp;
bool open_clock;

errcode=0;
open_clock=(instr.if_control && eq[cx][cy][px][py]) ||
           (instr.if_control == FALSE);

/* Execute the finite state machine */

switch(instr.op_code)
  {
    /* RESET initializes the eq signal from the compare unit to TRUE. */


  case RESET: eq[cx][cy][px][py]=TRUE;
              break;

    /* LOAD moves data from cache memory into the specified register.
     * For this operation, 5 is an illegal argument.
     */
  case LOAD:  if(open_clock)
```

```
                {
                if(instr.arg2>=0 &&
                   instr.arg2<CACHESIZE)
                  {
                  if(instr.arg1==6)
                    reg_inp[cx][cy][px][py]=
                          cache[cx][cy][px][py][instr.arg2];
                  else if(instr.arg1==7)
                    reg_out[cx][cy][px][py]=
                          cache[cx][cy][px][py][instr.arg2];
                  else if(instr.arg1>=0 &&
                          instr.arg1<=4)
                    reg[cx][cy][px][py][instr.arg1]=
                          cache[cx][cy][px][py][instr.arg2];
                  else errcode=instr.op_code+3;
                  }
                else errcode=instr.op_code+3;
                }
              break;

  /* A STOR instruction causes a register value to be placed in cache
   * memory. For this operation, 3 and 4 are illegal arguments.
   */
case STOR:  if(open_clock)
              {
              if(instr.arg2>=0 &&
                 instr.arg2<CACHESIZE)
                {
                if(instr.arg1==6)
                  cache[cx][cy][px][py][instr.arg2]=
                        reg_inp[cx][cy][px][py];
                else if(instr.arg1==7)
                  cache[cx][cy][px][py][instr.arg2]=
                        reg_out[cx][cy][px][py];
                else if((instr.arg1>=0 &&
                         instr.arg1<=2) ||
                         instr.arg1==5)
                  cache[cx][cy][px][py][instr.arg2]=
                        reg[cx][cy][px][py][instr.arg1];
                else errcode=instr.op_code+3;
                }
              else errcode=instr.op_code+3;
              }
            break;

  /* An OUTP instruction moves data from a holding or result register to the
   * output register. Output from the inputs to the compare unit is illegal.
   * Output from the output register is a no operation.
   */
case OUTP:  if(open_clock)
              {
              if(instr.arg1==6)
                reg_out[cx][cy][px][py]=
                      reg_inp[cx][cy][px][py];
              else if(instr.arg1==7);
              else if((instr.arg1>=0 &&
                       instr.arg1<=2) ||
                       instr.arg1==5)
                reg_out[cx][cy][px][py]=
                      reg[cx][cy][px][py][instr.arg1];
```

```
            else errcode=instr.op_code+3;
            }
          break;

  /* A READ instruction moves data from the register specified in arg 1 to
   * the register specified in arg 2. Arg 1 may be any register. Arg 2 may
   * not be the output of the compare unit. If arg 1 == arg2, the result is
   * no operation.
   */
case READ:  if(open_clock)
            {
            if(instr.arg1==6)
              temp=reg_inp[cx][cy][px][py];
            else if(instr.arg1==7)
              temp=reg_out[cx][cy][px][py];
            else if(instr.arg1>=0 &&
                    instr.arg1<=5)
              temp=reg[cx][cy][px][py][instr.arg1];
            else errcode=instr.op_code+3;
            if(errcode==0)
              {
              if(instr.arg2==6)
                reg_inp[cx][cy][px][py]=temp;
              else if(instr.arg2==7)
                reg_out[cx][cy][px][py]=temp;
              else if(instr.arg2>=0 &&
                      instr.arg2<=4)
                reg[cx][cy][px][py][instr.arg2]=temp;
              else errcode=instr.op_code+3;
              }
            }
          break;

  /* A CLER instruction clears the cited register to zero. */

case CLER:  if(open_clock)
            {
            if(instr.arg1==6)
              reg_inp[cx][cy][px][py]=0;
            else if(instr.arg1==7)
              reg_out[cx][cy][px][py]=0;
            else if(instr.arg1>=0 &&
                    instr.arg1<=4)
              reg[cx][cy][px][py][instr.arg1]=0;
            else errcode=instr.op_code+3;
            }
          break;

  /* A COMP instruction compares the contents of the two compare input
   * registers (3 and 4). The larger value is placed in the compare out-
   * put register (5) and the smaller in the convolution register (0).
   */
case COMP:  if(open_clock)
            {
            if(reg[cx][cy][px][py][3]<          /* COMP =>           */
                  reg[cx][cy][px][py][4])       /*   R5 <- max(R3,R4) */
              {                                 /*   R0 <- min(R3,R4) */
              reg[cx][cy][px][py][5]=
                    reg[cx][cy][px][py][4];
              reg[cx][cy][px][py][0]=
```

```
                    reg[cx][cy][px][py][3];
                  eq[cx][cy][px][py]=FALSE;
                  }
                else
                  {
                  reg[cx][cy][px][py][5]=
                        reg[cx][cy][px][py][3];
                  reg[cx][cy][px][py][0]=                /* COMP =>            */
                        reg[cx][cy][px][py][4];          /*   if R3 == R4      */
                  if(reg[cx][cy][px][py][3]==            /*     eq=TRUE        */
                     reg[cx][cy][px][py][4])             /*   else             */
                    eq[cx][cy][px][py]=TRUE;             /*     eq=FALSE       */
                  else eq[cx][cy][px][py]=FALSE;
                  }
                }
              break;

    /* A CONV instruction convolves the image, or at least executes the
     * current commanded step in a convolution operation. Registers 1 and
     * 2 are multiplied together, and the result is added to the current
     * contents of the convolution register (0).
     */
  case CONV:  if(open_clock)
                {
                reg[cx][cy][px][py][0]+=             /* CONV =>            */
                      reg[cx][cy][px][py][1]*        /*   R0 <- R1 * R2    */
                      reg[cx][cy][px][py][2];        /*          + R0      */
                }
              break;

    /* MDRU (move down, receive up) causes the input register of this PE
     * to become equal to the output register of the PE's upper neighbor.
     * If there is no upper neighbor, the new input value is read from the
     * specified upper register location from the chip array. If there is
     * no lower neighbor to grab the PE's output, the PE writes its output
     * register into the specified lower register location from the chip array.
     */
  case MDRU:  if(open_clock)
                {
                if(px==0)
                  if(cx==0)      /* pe is on the upper edge of an upper chip */
                    reg_inp[cx][cy][px][py]=pu[py];
                  else           /* pe is on the upper edge of a lower chip */
                    reg_inp[cx][cy][px][py]=
                          reg_out[cx-1][cy][PE_HT-1][py];
                else             /* pe is not on the upper edge of the chip */
                  reg_inp[cx][cy][px][py]=
                        reg_out[cx][cy][px-1][py];
                if(px==PE_HT-1 && /* pe is on the lower edge of a lower chip */
                   cx==CHIP_ARRAY_HT-1)
                  pd[py]=reg_out[cx][cy][px][py];
                }
              break;

    /* MURD (move up, receive down) causes the input register of this PE
     * to become equal to the output register of the PE's lower neighbor.
     * If there is no lower neighbor, the new input value is read from the
     * specified lower register location from the chip array. If there is
     * no upper neighbor to grab the PE's output, the PE writes its output
     * register into the specified upper register location from the chip array.
```

```
     */
    case MURD:  if(open_clock)
                  {
                  if(px==PE_HT-1)
                    if(cx==CHIP_ARRAY_HT-1)          /* lower edge of lower chip */
                      reg_inp[cx][cy][px][py]=pd[py];
                    else                             /* lower edge of other chip */
                      reg_inp[cx][cy][px][py]=
                           reg_out[cx+1][cy][0][py];
                  else                               /* not lower edge of a chip */
                    reg_inp[cx][cy][px][py]=
                         reg_out[cx][cy][px+1][py];
                  if(px==0 && cx==0)                 /* upper edge of upper chip */
                    pu[py]=reg_out[cx][cy][px][py];
                  }
                break;

      /* MLRR (move left, receive right) causes the input register of this PE
       * to become equal to the output register of the PE's right neighbor.
       * If there is no right neighbor, the new input value is read from the
       * specified right register location from the chip array. If there is
       * no left neighbor to grab the PE's output, the PE writes its output
       * register into the specified left register location from the chip array.
       */
    case MLRR:  if(open_clock)
                  {
                  if(py==PE_WD-1)
                    if(cy==CHIP_ARRAY_WD-1)          /* right edge of right chip */
                      reg_inp[cx][cy][px][py]=
                           reg_out[cx][0][px][0];
                    else                             /* right edge of other chip */
                      reg_inp[cx][cy][px][py]=
                           reg_out[cx][cy+1][px][0];
                  else                               /* not right edge of a chip */
                    reg_inp[cx][cy][px][py]=
                         reg_out[cx][cy][px][py+1];
                  }
                break;

      /* MRRL (move right, receive left) causes the input register of this PE
       * to become equal to the output register of the PE's left neighbor.
       * If there is no left neighbor, the new input value is read from the
       * specified left register location from the chip array. If there is
       * no right neighbor to grab the PE's output, the PE writes its output
       * register into the specified right register location from the chip array.
       */
    case MRRL:  if(open_clock)
                  {
                  if(py==0)
                    if(cy==0)                        /* left edge of left chip   */
                      reg_inp[cx][cy][px][py]=
                           reg_out[cx][CHIP_ARRAY_WD-1][px][PE_WD-1];
                    else                             /* left edge of other chip  */
                      reg_inp[cx][cy][px][py]=
                           reg_out[cx][cy-1][px][PE_WD-1];
                  else                               /* not left edge of a chip  */
                    reg_inp[cx][cy][px][py]=
                         reg_out[cx][cy][px][py-1];
                  }
                break;

    DEFAULT:    errcode=2;                           /* illegal instruction */
                break;
    }
  return(errcode);
  }
```

IMAGE MOTION INTERPOLATION

TERM REPORT of EE599, SUMMER, 1991

CHIA-SEN PENG

888-03-4655

9938 Jovita Ave.

Chatsworth, CA 91311

Tel: (818)885-6317

Abstract

In image processing, interpolation can be used on many applications, especially in motion compensation and motion estimation. In this report, the topic will focus on the interpolation algorithms and their software simulations. The result of simulations can attribute some idea on saving transmission bandwidth by sending small images and few image frames. In order to explain all clearly, I divided the report into two parts: the first part is to process one image; the second part is to deal with two image frames.

Interpolation on one image

The Algorithm and the Simulation

Assume $f_c(x,y)$ is evaluated by a linear combination of $f(n_1, n_2)$ at the four closest pixels, and $f_c(x,y)$ exists for $n1T1 \le x \le (n1+1)T1$, $n2T2 \le y \le (n2+1)T2$,shown below:

The interpolated pixel in the bilinear interpolation method [1] is:

$f_c(x,y)= (1-\Delta x)(1-\Delta y) f(n_1,n_2)+ (1-\Delta x)\Delta y f(n_1,n_2+1)+\Delta x(1-\Delta y) f(n_1+1,n_2)+\Delta x\Delta y f(n_1+1,n_2+1)$

where $\Delta x = \dfrac{(x-n1T1)}{T1}$, $\Delta y = \dfrac{(y-n2T2)}{T2}$

And the above formula can be analyzed by combining them together as follows:

$n1 \le \dfrac{x}{T1} \le n1+1, n2 \le \dfrac{y}{T2} \le n2+1$ and $\Delta x = \dfrac{x}{T1} - n1$, $\Delta y = \dfrac{y}{T2} - n2$

That means $0 \le \Delta x \le 1, 0 \le \Delta y \le 1$, so it is not difficult to obtain the appropriate interpolated pixel by adjusting the Δx, Δy values. In my simulation, the values($\Delta x = 0.5, \Delta y = 0.5$) is adopted and can reach the better result than the other test values. Here one place should be noticed that besides the interpolated value there are four pixels left between the every two boundary pixels, shown in Appendix.

And those undecided pixels also have to be gotten by the bilinear interpolation method like a special case application. So in my software simulation, in total, there are 9 steps to complete the whole operation. And those 9 steps have been denoted in my source code (refers to the Appendix).

Interpolation on two image frames

The Algorithm and the Simulation

Compare with one image processing, it seems more complex in processing two image frames like two consecutive TV scenes. In order to avoid jerkiness in those interpolated scenes, it is necessary to consider one reliable method to do this job. There are many conditions which limit the performance of applied algorithm and the quality in those processed frames:

1. which kind of motion the object belongs to? Translation, Rotation or Random motion?

2. which direction the object moves along? with the X, Y or two-dimension axis, or even not orthogonal to the camera axis?

3.Whether the luminance on each pixels varies or not during the motion occurs?

4. In one frame, how many objects move? all in the same direction? or in the different directions?

5. Whether the whole object appears in the frame or not? or just part of it?

So in the above different conditions, there are several applications to solve them, such as motion compensation, motion estimation and motion interpolation. Here, because this report only concern on the interpolation, the assumption is built as one object moves translationally in two-dimension space and the luminance on the same pixel does not change, even after moving and it is the only one moving in the frame under the condition which the background on the image does not change. However, in my simulation, the "shuttle" image can not meet all of the above assumption. For example, it has but also some displacement along the camera axis, not only along the X or Y axis.

Moreover, its original shape exceeds the range of the image boundary, so that will cause distortion after interpolation, even if I have do some simple compensation on it.

In fact, before doing interpolation, it is essential to estimate the displacement of the moving object.

And there are some papers mentioning this topic and offering various algorithms to solve this problem[2][3]. Here I just apply the same method (try to find the same luminance for two pixels between the given frames within a small region) by three iterations, average those displacements and take an integer value of that averaging. Once the approximate value is taken, then the interpolation can be applied. For the "shuttle" images, I took the first, the third and fifth one as my reference images. Later I interpolated two frames between the 1st and the 3rd, the 3rd and the 5th images by the following method:

assume the criterion is $|f(x, y, t1) - f(x, y, t0)| > 0, d(x, y) = (3, 12)$

$$otherwise, d(x, y) = (0, 0)$$

where $f(x, y, t1)$ denotes the luminance level on the time $t1$ at a certain pixel

$f(x, y, t0)$ denotes the luminance level on the time $t0$ at the same pixel

$d(x, y)$ denotes the displacement from time $t0$ to time$t1$ for the pixel

As I have said that before, the pixels on the boundary of the images should be compensated by prediction, here I assume the continuity of those pixels and then repeat the values to the boundary.

So it is inevitable that this decision will introduce distortion on the boundary. On the other hand, it has no affect that if the reference images have been processed by a low pass filter or not before interpolating. The conclusion above can be proven by the results of my simulations, shown below.

However, it will make no sense that if I interpolated two frames between the first and fifth frames, it will cause worse result which is due to little information got from the boundary of the first one, or dominances by the fifth one which will result in abrupt variation among the first and the interpolated frames. Therefore, in my simulation, if the pixel meets the first statement of the criterion, the first interpolated pixel takes the luminance value of the current pixel on the "previous" image, on the contrary, the second interpolated pixel takes the value from the "next" image. This operation will decrease the bias from the boundary of either of those two images.

Conclusion

Through several simulations on software, the better result can be obtained and that result will derive one appropriate algorithm. But any algorithms derived here is based on the most fundamental assumptions. Therefore there are some interesting research topics extended from this report:

1. if the luminance values of the moving object are changed, how to decide the threshold value (such as 10, 20)?

2. how to interpolate the motion of the object moving along the camera axis, how to estimate the displacement of that kind of motion?

3. For more complicated background rather than white or black, how to compensate the background after interpolation?

To sum up, it is significant that we can achieve the target on saving the three -fourth of the transmission bandwidth by sending every two images, and recovering them from self-interpolation method and adding other two interpolated ones later.

Supplement (TA request)

Motion compensation is also important in the simulation after we have done the interpolation well. As we predicted before, there are some pixels which are assigned to the wrong luminance values and moved to the wrong positions while interpolating, and some others which became undecided after interpolating. The former can be compensated by checking the last one neighboring to it along with the X or Y axis, the latter can be compensated by referencing the associated pixel on the 2nd image frame. Both of above have been described detailed in the source code.

After compensation, the interpolated frames look like the consecutive ones among the whole serials of image frames.

References

[1] Jae S.Kim, "Two-Dimensional Signal and Image Processing", 1991, pp495-511.

[2] Ciro Cafforio, Fabio Rocca, and Stefano Tubaro, "Motion Compensdated Image Inperpo lation", IEEE Trans. on Comm. Vol.38, NO.2, Feb, 1990, pp215-222.

[3] C. Cafforio and F. Rocca, "The Differiential Method for Image Motion Estimation", in Image Sequence Processing and Dynamic Scene Analysis, T.S. Huang, Ed. New York: Springer-Verlag, 1983, pp104-124.

Appendix

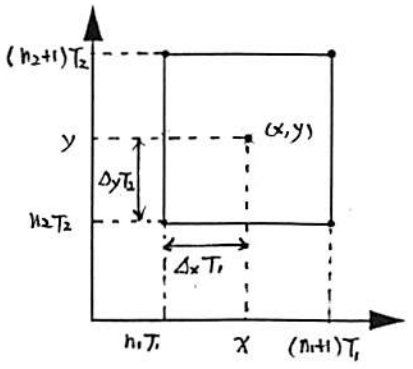


Fig. 1. Region where $f_c(x,y)$ is interpolated from the four neighboring pixels.
$f_c(n_1T_1, n_2T_2)$, $f_c((n_1+1)T_1, n_2T_2)$
$f_c(n_1T_1, (n_2+1)T_2)$,
$f_c((n_1+1)T_1, (n_2+1)T_2)$.

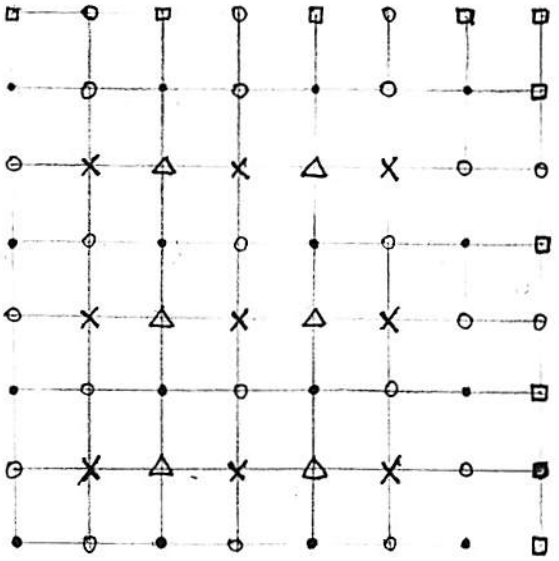


Fig. 2. Bilinear Interpolation. (8×8)
"•" = pixels from original images
"X" = interpolated pixels from neighboring four pixels.
"o" = interpolated pixels from neighboring two pixels.
"Δ" = interpolated pixels from neighboring two interpolated pixels.
"□", copy pixels from those original pixels.

Fig. 3.
Original "lenna" image
(256 × 256)

Fig.4. Interpolated Images (512×512) according to the formula.

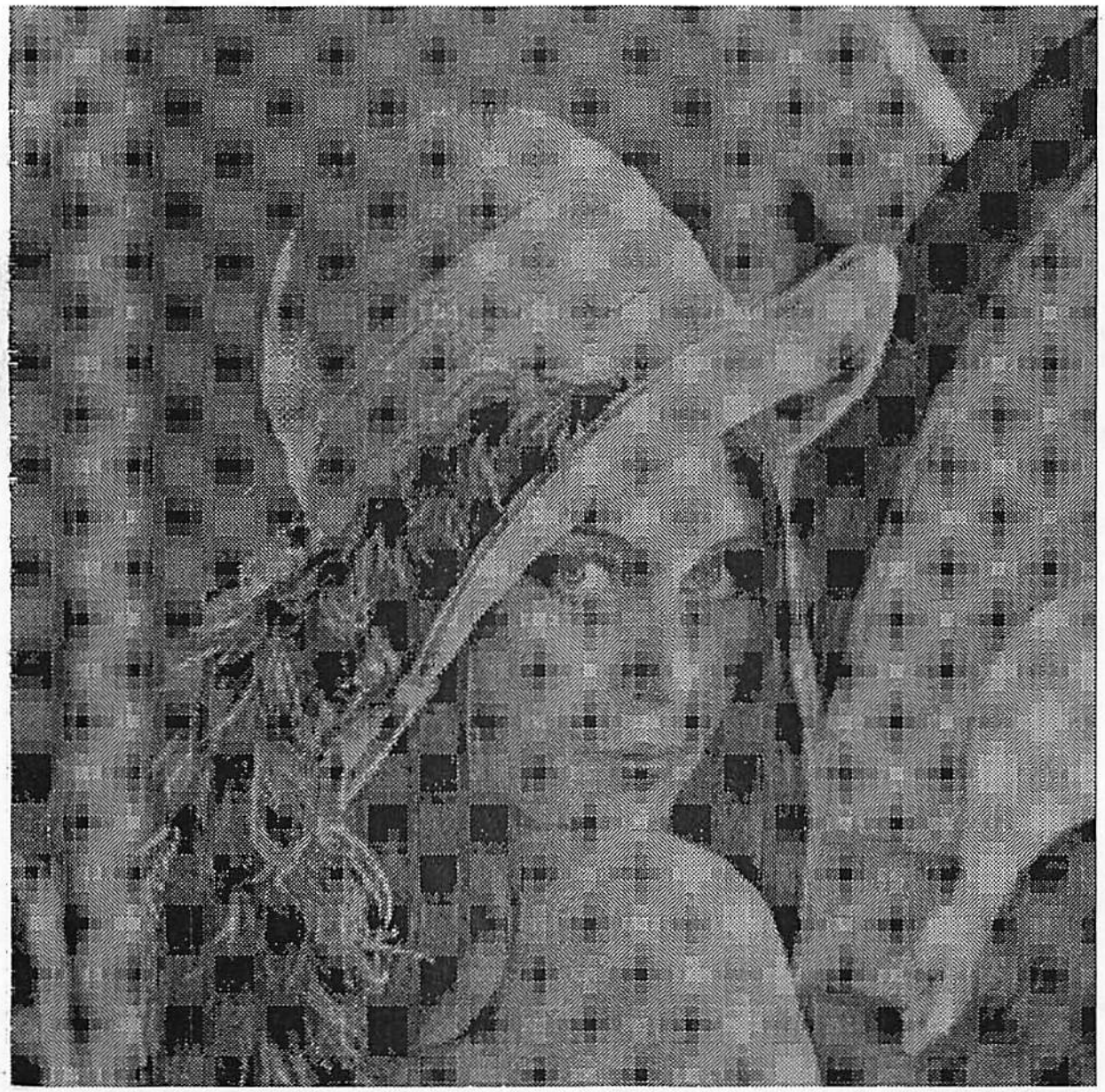


Fig. 6. Interpolated Images (512×512) with some improvement.

"Interpolation on One Image" only by formula.

t2.c


```
#include <sys/fcntl.h>
#include <stdio.h>
static int image[256][256],image1[512][512];
main(argc,argv)
int argc;
char **argv;

{
  int a,b,p,p1,i,j;
  char bline[256],bline1[512];

  if (argc!=3){
  write(2,"usage; ",7);
  write(2,*argv,strlen(*argv));
  write(2," from-file to-file",19);
  exit(1);
  }
/* */
  if((p=open(argv[1], O_RDONLY))<0){
   perror(argv[1]);
   exit(1);
   }
/*  */
  if((p1=open(argv[2], O_WRONLY|O_CREAT|O_TRUNC,0644))<0){
   perror(argv[2]);
   exit(1);
  }
/* */
  for(i=0;i<256;i++){
   read(p,bline,256);
   for(j=0;j<256;j++){
    image[i][j]=bline[j]&0xff;
     }
}
 /* */
for(i=0;i<512;i+=2){
   for(j=0;j<512;j+=2){
     a=i/2;b=j/2;
     image1[i][j]=image[a][b];
    }
 }
for(j=0;j<510;j+=2){
  for(i=0;i<510;i+=2){
  image1[i+1][j+1]=(int)(image1[i][j]*0.25+image1[i+2][j]*0.25+image1[i][j+2]*0.25+image
1[i+2][j+2]*0.25);
   }
}
/* */
 for(i=0;i<512;i++){
     for(j=0;j<512;j++){
      bline1[j]=(char) image1[i][j];
      }
    write(p1,bline1,512);
   }

 close(p);
 close(p1);

 exit(0);
}
```

Input operation

→ interpolation operation

output operation.

"Interpolation on One Image" with improvement in formula.

t11.c


```
#include <sys/fcntl.h>
#include <stdio.h>
static int image[256][256],image1[512][512];
main(argc,argv)
int argc;
char **argv;

{
  int a,b,p,p1,i,j;
  char bline[256],bline1[512];

  if (argc!=3){
  write(2,"usage; ",7);
  write(2,*argv,strlen(*argv));
  write(2," from-file to-file",19);
  exit(1);
  }
/* */
  if((p=open(argv[1], O_RDONLY))<0){
   perror(argv[1]);
   exit(1);
   }
/*  */
  if((p1=open(argv[2], O_WRONLY|O_CREAT|O_TRUNC,0644))<0){
   perror(argv[2]);
   exit(1);
  }
/* */
  for(i=0;i<256;i++){
   read(p,bline,256);
   for(j=0;j<256;j++){
    image[i][j]=bline[j]&0xff;
     }
}
 /* */
for(i=0;i<512;i+=2){
   for(j=0;j<512;j+=2){
     a=i/2;b=j/2;
     image1[i][j]=image[a][b];
    }
 }
for(j=0;j<510;j+=2){
  for(i=0;i<510;i+=2){
  image1[i+1][j+1]=(int)(image1[i][j]*0.25+image1[i+2][j]*0.25+image1[i][j+2]*0.25+image
1[i+2][j+2]*0.25); ①
   }
}
for(j=0;j<512;j+=2)image1[511][j]=image1[510][j];②
for(i=0;i<512;i+=2)image1[i][511]=image1[i][510];③
for(j=1;j<511;j+=2)image1[511][j]=(int)((image1[511][j-1]+image1[511][j+1])/2);④
for(i=1;i<511;i+=2)image1[i][511]=(int)((image1[i-1][511]+image1[i+1][511])/2);⑤
image1[511][511]=image1[510][510]; ⑥
for(j=1;j<511;j+=2){
 for(i=1;i<511;i+=2)image1[i+1][j]=(int)((image1[i][j]+image1[i+2][j])/2);} ⑦
for(j=0;j<512;j+=2){
 for(i=0;i<510;i+=2)image1[i+1][j]=(int)((image1[i][j]+image1[i+2][j])/2);} ⑧
for(j=0;j<510;j+=2)image1[0][j+1]=(int)((image1[0][j]+image1[0][j+2])/2); ④
/* */
 for(i=0;i<512;i++){
     for(j=0;j<512;j++){
      bline1[j]=(char) image1[i][j];
      }
    write(p1,bline1,512);
   }

 close(p);
 close(p1);

 exit(0);
}
```

Input operations.

Interpolation Procedure (9 steps)

Output operation.

Fig. 6. original "shuttle. 1" (through a low-pass filter)

Fig. 7. Interpolated "shuttle. 2"

Fig. 8. Interpolated "shuttle. 25"

Fig. 9. original "shuttle. 3" (through a low-pass filter)

6	8
7	9

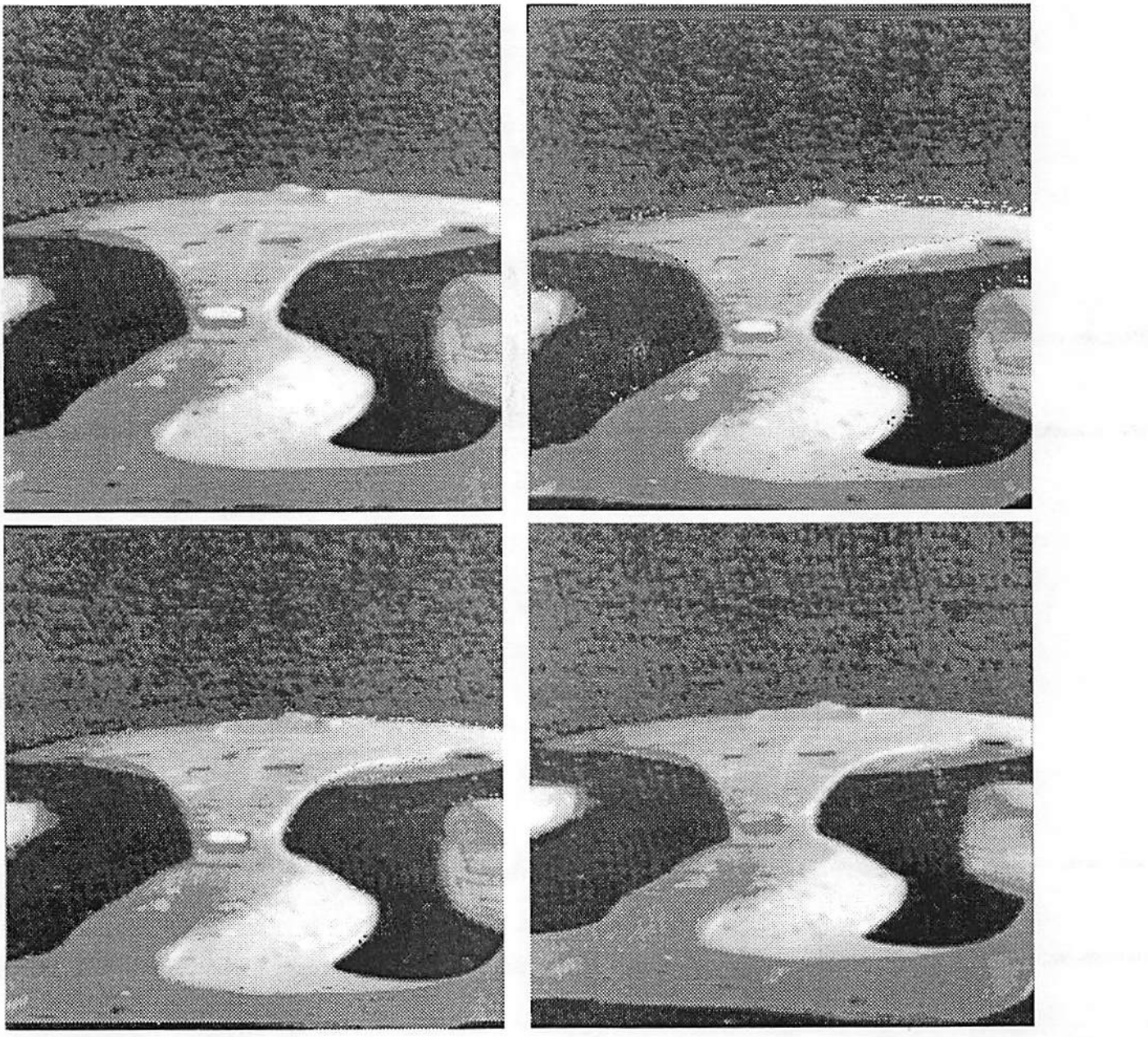


Fig. 10. original "shuttle.1" (not through a Lowpass filter)

Fig. 11. Interpolated "shuttle.2"

Fig. 12. Interpolated "shuttle.25"

Fig. 13. original "shuttle.3" (not through a Lowpass filter)

10	12
11	13

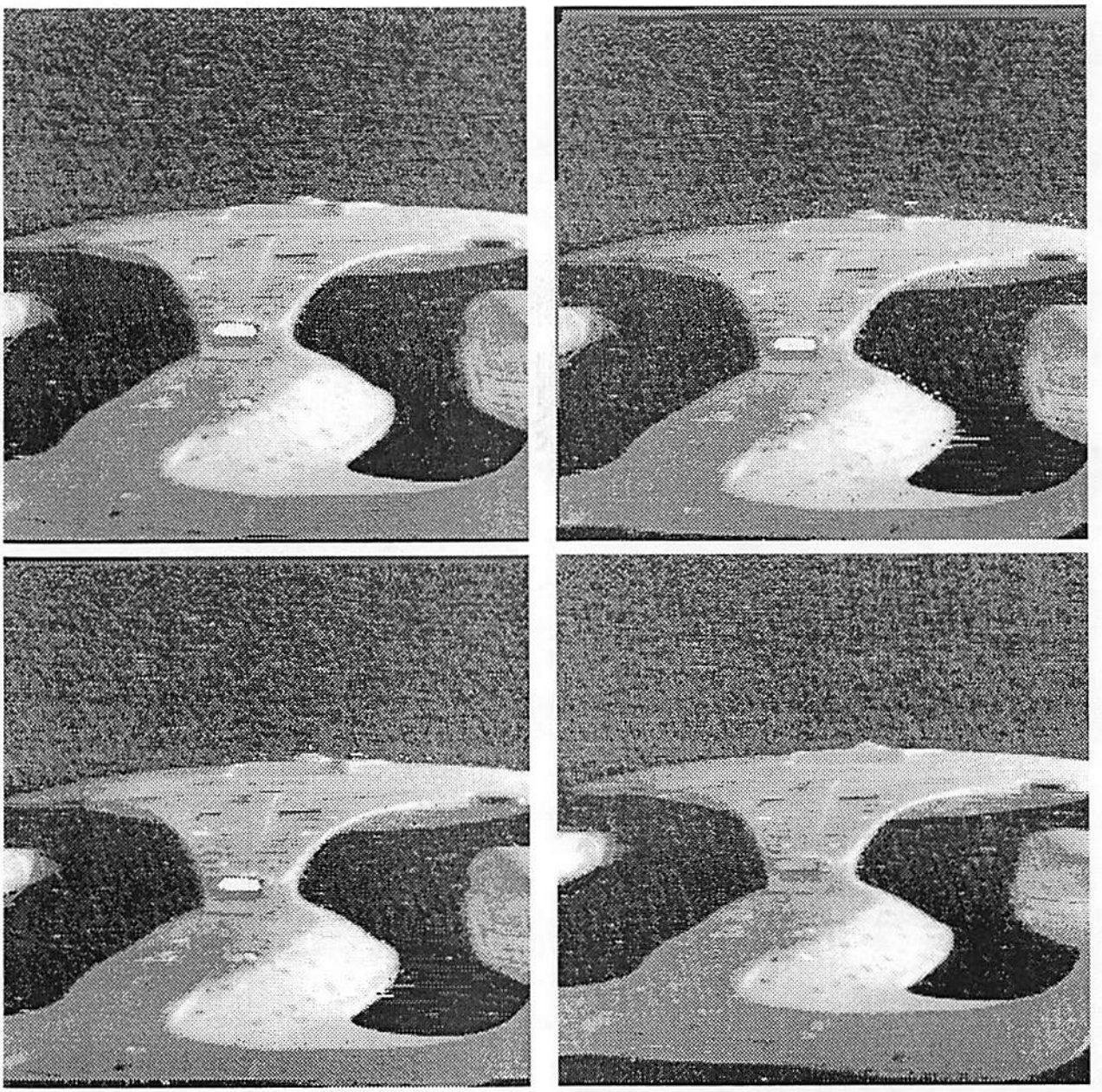


Fig. 14. original "shuttle.3" (through a low pass filter)

Fig. 15. Interpolated "shuttle.4"

Fig. 16. Interpolated "shuttle.45"

Fig. 17. original "shuttle.5" (through a low pass filter)

14	16
15	17

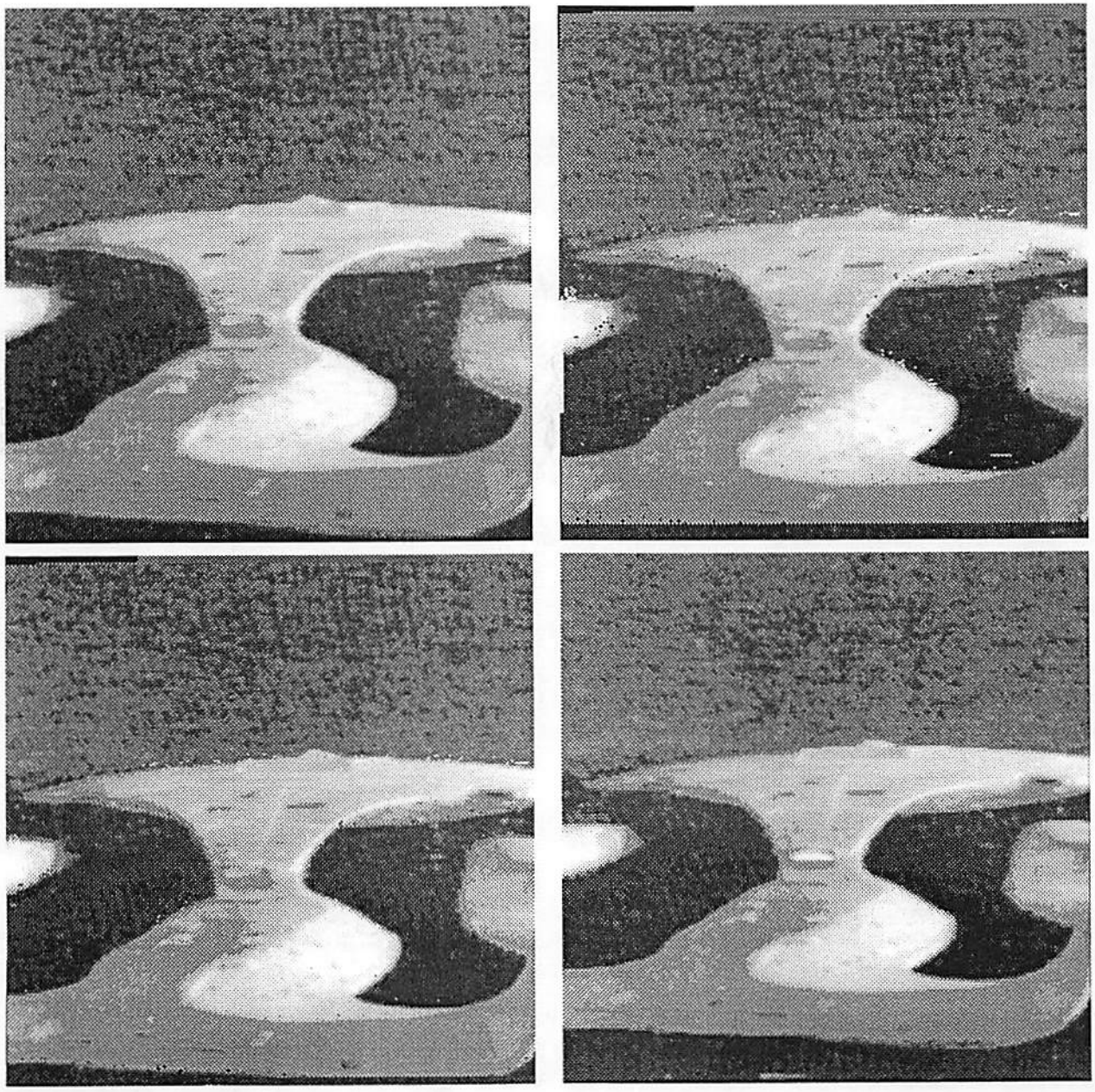


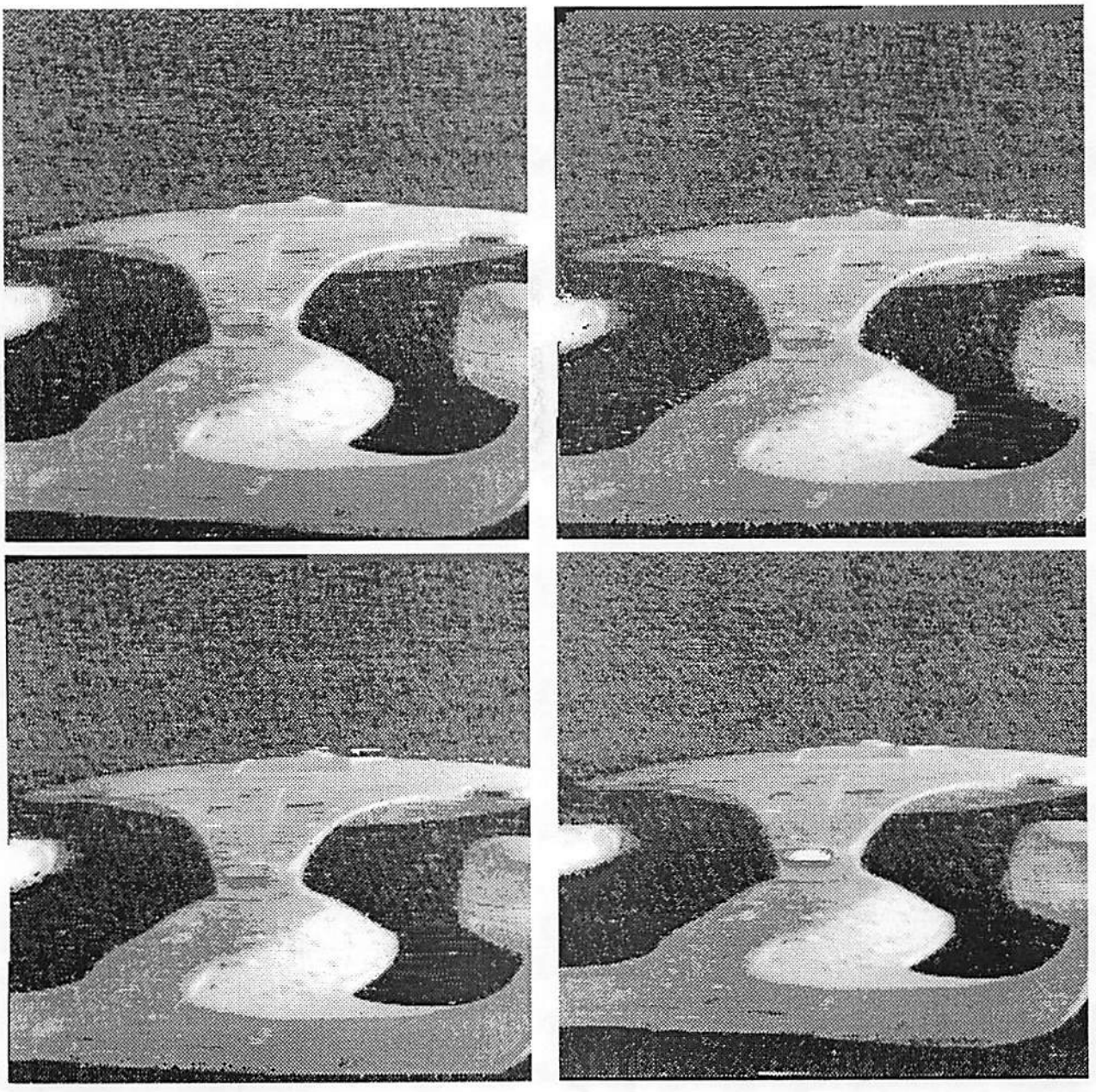
Fig. 18. original "shuttle.3" (not through a Low pass filter)

Fig. 19. Interpolated "shuttle.4"

Fig. 20. Interpolated "shuttle.45"

Fig. 21. original "shuttle.5" (not through a Low pass filter)

18	20
19	21.



"Low-Pass Filter."

p1a.c

```
#include <sys/fcntl.h>
#include <math.h>
static int r[258][258];
main(argc,argv)
int argc;
char **argv;

{
  int a,b,p,p1,i,j;
  int image[256][256],image1[256][256];
  char bline[256];
  double ex,db;

  if (argc!=3){
  write(2,"usage; ",7);
  write(2,*argv,strlen(*argv));
  write(2," from-file to-file\n",19);
  exit(1);
  }
/* */
  if((p=open(argv[1], O_RDONLY))<0){
   perror(argv[1]);
   exit(1);
   }
/*  */
  if((p1=open(argv[2], O_WRONLY|O_CREAT|O_TRUNC,0644))<0){
   perror(argv[2]);
   exit(1);
  }
/* */
  for(i=0;i<256;i++){
   read(p,bline,256);
   for(j=0;j<256;j++){
    image[i][j]=bline[j]&0xff;
    r[i+1][j+1]=image[i][j];
    image1[i][j]=image[i][j];
    while((i==0&&j==0)||(i==0&&j==255)||(i==255&&j==0)||(i==255&&j==255)){
     r[i][j]=r[i+1][j+1];
     r[i][j+1]=r[i+1][j+1];
     r[i+1][j]=r[i+1][j+1];break;
    }
    while((i==0||i==255)&&(j!=0&&j!=255)){
     r[i][j+1]=r[i+1][j+1];break;
    }
    while((j==0||j==255)&&(i!=0&&i!=255)){
     r[i+1][j]=r[i+1][j+1];break;
    }
  }
}
/* */
  a=0;b=0;
  for(i=0;i<256;i++){
    for(j=0;j<256;j++){
      image[i][j]=window(i,j,0.0625,0.125,0.0625,0.125,0.25,0.125,0.0625,0.125,0.0625);
      a+=(image1[i][j]-image[i][j])*(image1[i][j]-image[i][j]);
      b+=image1[i][j]*image1[i][j];


    }
   }
     ex=(double) (b/a);
     db=10.0*log10(ex);
     printf("The SNR is %f dB\n",db);
/* */ for(i=0;i<256;i++){
     for(j=0;j<256;j++){
      bline[j]=(char) image[i][j];
      }
    write(p1,bline,256);
   }

 close(p);
 close(p1);
 exit(0);
}

window(i,j,h1,h2,h3,h4,h5,h6,h7,h8,h9)
int i,j;
double h1,h2,h3,h4,h5,h6,h7,h8,h9;
{
   int v;
      v= h1*r[i][j]+h2*r[i+1][j]+h3*r[i+2][j]+
         h4*r[i][j+1]+h5*r[i+1][j+1]+h6*r[i+2][j+1]+
         h7*r[i][j+2]+h8*r[i+1][j+2]+h9*r[i+2][j+2];
   return(v);
}
```

Input operation.

output operation

window function

"original ones processed by this prog"

Filter. Window operation

"Interpolation on two image frames"


```
/* The program is for the interpolation on two images. */
#include <sys/fcntl.h>
#include <stdio.h>
#include <math.h>
static int image[256][256],image1[256][256],ip1[256][256],ip2[256][256];
main(argc,argv)
int argc;
char **argv;

{
  int a1,b1,a2,b2,p1,p2,p3,p4,i,j,alpha,beta;
  char bline[256],bline1[256];

  if (argc!=5){
  write(2,"usage; ",7);
  write(2,*argv,strlen(*argv));
  write(2," f1,f2,f3,f4\n",12);
  exit(1);
  }
/* */
  if((p1=open(argv[1], O_RDONLY))<0){
   perror(argv[1]);
   exit(1);
   }
  if((p2=open(argv[2], O_RDONLY))<0){
   perror(argv[2]);
   exit(1);
   }

/*  */
  if((p3=open(argv[3], O_WRONLY|O_CREAT|O_TRUNC,0644))<0){
   perror(argv[3]);
   exit(1);
  }
 if((p4=open(argv[4], O_WRONLY|O_CREAT|O_TRUNC,0644))<0){
   perror(argv[4]);
   exit(1);
  }

/* */
  for(j=0;j<256;j++){
   read(p1,bline,256);read(p2,bline1,256);
   for(i=0;i<256;i++){
    image[i][j]=bline[i]&0xff;image1[i][j]=bline1[i]&0xff;
     }
}
 /* The estimated displacement is (3,12) */
alpha=3;beta=12;a1=alpha/3;b1=beta/3;a2=alpha*2/3;b2=beta*2/3;
 /* Interploation to get image #1 */
for(j=0;j<(256-b1);j++){
   for(i=0;i<(256-a1);i++){
if(abs(image1[i][j]-image[i][j])==0){
ip1[i][j]=image[i][j];/* Threshold value is zero */
}
else {ip1[i+a1][j+b1]=image[i][j];}
 }
 }
/* Interpolation on image #2 */
for(j=0;j<(256-b2);j++){
   for(i=0;i<(256-a2);i++){
if(abs(image1[i][j]-image[i][j])==0){ip2[i][j]=image[i][j];}
else {ip2[i+a2][j+b2]=image[i][j];}
}
}
/*Smoothing the images #1 & #2 */
for(i=1;i<256;i++){
for(j=0;j<256;j++){
if(ip1[i][j]==0){if(ip1[i-1][j]!=0)ip1[i][j]=ip1[i-1][j];}
if(ip2[i][j]==0){if(ip2[i-1][j]!=0)ip2[i][j]=ip2[i-1][j];}
}
}


/*Compensation on the pixels which the first image can't offer information to. */
for(i=0;i<256;i++){
for(j=(256-b1);j<256;j++){
ip1[i][j]=image1[i][j-b1];}}

/*Compensation on the pixels which the first image can't offer information to. */
for(i=0;i<256;i++){
for(j=(256-b2);j<256;j++){
ip2[i][j]=image1[i][j-b2];}}

/*Smoothing the images #1 & #2 */
for(i=0;i<256;i++){
for(j=1;j<256;j++){
if(ip1[i][j]==0){if(ip1[i][j-1]!=0)ip1[i][j]=ip1[i][j-1];}
if(ip2[i][j]==0){if(ip2[i][j-1]!=0)ip2[i][j]=ip2[i][j-1];}
}
}
/*Smoothing the images #1 & #2 */
for(i=1;i<256;i++){
for(j=0;j<256;j++){
if(ip1[i][j]==0){if(ip1[i-1][j]!=0)ip1[i][j]=ip1[i-1][j];}
if(ip2[i][j]==0){if(ip2[i-1][j]!=0)ip2[i][j]=ip2[i-1][j];}
}
}


 for(j=0;j<256;j++){
     for(i=0;i<256;i++){
      bline1[i]=(char) ip2[i][j]; bline[i]=(char) ip1[i][j];
      }
    write(p4,bline1,256);write(p3,bline,256);
   }

 close(p1);
 close(p2);
 close(p3);
 close(p4);
 exit(0);
}
```

The Two Dimensional Motion Estimation Based On Hough Transform

Hsiu Shaw
Department of Electrical Engineering

Summary

The special implementation of the Hough Transform in motion detection depends on its robust. Because the HT can solve the problem of noise and camera shift. Its principal algorithm in estimating translation and rotation of moving object depends on the accurate translation parameters that produced by detect the moving way of object first. Due to the HT technique needs two consecutive images to estimate the displacement and the rotation degree of objects. It is obviously that the system of motion detection is a real-time system. One another feature of the HT algorithm is that it translation parameter of a point can be determined independently and will not be influenced by data of other points. Base on this character, the multiprocessor system is more suitable than any other system architecture. Especially, the SIMD system is recommended. The HT algorithm need more time to perform complex computation and more storage space to store data and parameters. The usage of uniprocessor in motion detection by the HT method is not a good choice. Because the large uniprocessor system will spend a lot of time on data fetching and it must provide large memory space to store data. For example, when we use SUN-4 machine to run the Hough Transform in motion estimation using 128 x 128 image frame, we need CPU time more than 20 minutes. It will require more than 30 minutes if we run a 256 x 256 image picture by Hough Transform. Because the SUN-4 belongs to time-sharing system, 30 minutes are the maximum CPU accessing time for each user. Therefore, we only show the results of the Hough Transform by 28 x128 images. This is why we present our design by multiprocessor system. In addition, its price-performance ratio is also not satisfied. According to our designs, 2 x 80286 computer systems and 4 x PEs are used in our architecture of motion detection system. One of the two 80286 computers is used as main CPU to control the four PEs, I/O CPU and bus. The other 80286 computer system is to handle the data coming from outside world through camera. If the data come from camera is analog signal, the I/O CPU can convert analog signal to digital signal by controlling the A/D converter and store the input image data in I/O memory temporarily for waiting the access of bus. The first image data will transfer from I/O memory to the main memory and the input part of cache memory when I/O CPU receive the control signal of main CPU. The second image data will be loaded in main memory and local memories of each PE. The main CPU will send eight points information to all four PEs in one clock period, The PE partitioned into two parts, one for Hough Transform algorithm and the other for segmentation. After performing the Hough Transform algorithm, we can get many corresponding points to each input data and the coordinates and degree of each corresponding point will be stored in temporary part of cache memory. The main CPU will use these data to increase the value of each corresponding counter and sort out the coordinates and degree of the biggest counter. After sorting out the coordinate (displacement) and degree, these data will be transferred into local memory of each PE to perform segmentation. Through the process of segmentation, the results will be stored in main memory and wait for output. After these procedures, the third image picture will be stored in local memory

of each PE and the second one will be transferred into input part of cache memory. This process go again and again to from a real-time multiprocessor system. By this system, we save our time, cost and memory capacity. With this algorithm and real-time system, we can get more accurate parameters and it can be implemented in real world.

Part I.

Abstract-It is a traditional way that use the difference between two consecutive images to estimate the change in motion. But it will fail for shape and motion analysis in image containing noisy, missing, extraneous data because of its computational and storage complexity, and especially in the situation of camera moves or shifts or vibrates. However, in recent years much progress has been made in this area. Here we develop a new algorithm based on Hough Transform, HT, and use this new method to solve the segmentation problem when the camera shift. We present results on the analytic and empirical performance. We also report the major advantages of the HT. It seems likely that the HT is the most reliable and an increasingly used technique in this area.

1. INTRODUCTION

Motion detection plays an important role in image understanding and has been an main research region in last ten years. Many other algorithm of motion detection such as Spatial Technique, its method is to form the different image between two consecutive pictures. This approach can be achieved only if the two images are registered and the illumination is relatively constant within the bonds. It also need a reference image to compare each of the subsequent image in sequence. But the difference between two images in dynamic imaging problem will cancel all stationary components and leave only image points that correspond to noise and the moving objects. The noise can be handled by filtering techniques. However, in practice, it is not always possible that obtain a reference image with only stationary elements. It is also not possible to detect more than one objects moving. The most important factor is that it can not detect the rotation of object and the vibration of camera. Therefore, we use Hough Transform, HT, to improve the quality of the image and solve the shifting problem of camera. The HT also can detect more than one object moving and can obtain higher quality segmentation of the moving objects so that feature points such as straight line, corner points, and high curvature contours can be extracted without large distortions. Here, we consider the general case that the moving objects have significant size, background is cluttered, and also the camera may shift or vibrate. The HT technique is considered in two dimensions case in our paper. First, we try to find the two dimensional motion parameters to describe the way of motion of any potential moving objects. Second, we can know the relationships and solve the registration problem from the motion parameters between the moving objects in the 1st image and those of the moving objects in the 2nd image. Third, we implement some method to filter out the redundant and meaningless moving objects and their motion parameters produced by noisy or vibration of camera. Finally, we obtain the images of the moving objects and the ways they move can be calculated too.

2. THE ALGORITHM

The Hough Transform, HT, is a method of detecting complex patterns of points in binary image data. It can detect complex pattern by determining specific values of parameters which characterize these pattern. The HT converts the difficult global image problem in image space into a more easily solved local peak detection problem in a parameter space.When we use the HT technique to detect objects moving, we need to make some assumptions to implement our algorithm. First, the difference of gray level between the background and the moving objects must differ to some degree, otherwise it is almost impossible to extract the moving objects from the image because the background and the moving objects look almost the same. So, one of the necessary conditions is that for a pixel to be a part of an object, its gray level must be different from that of its background to a certain degree. Second, the noise of the camera or other input devices is unavoidable. Thus, the gray level of background and objects may vary slightly in consecutive image pictures. So, the gray levels are the same if the difference between two images is within some tolerated value. Last, the moving objects be segmented usually are required to have significant size because a very small objects extracted may produced by noise. Therefore we only want the moving objects which have significant size and discard the parts produced by noise. Our algorithm can be divided into two cases: (A) The detection and estimation of the moving objects with pure translation. (B) The detection and estimation of the moving objects with translation and rotation.

A. The Detection and Estimation of the Moving Objects with Pure Translation

The key ideas of the method can be illustrated by considering sets of colinear points in an image. A set of image points (X , Y) which lie on a straight line can be describe by a relation, F,such that

$$F((\bar{m},\bar{c}),(x,y)) = y - \bar{m}x - \bar{c} = 0 \qquad (1)$$

where m and c are two parameters, the slope and intercept, which characterize the line. Equation (1) maps each value of the parameter combination ($\bar{m}$, $\bar{c}$) to a set of image points. The mapping is one to many from the space of possible parameter values to the space of image points. Equation (1) can be viewed as a mutual constraint between image points and parameter points and therefore it can be interpreted as defining a one to many mapping from an image point to a set of possible parameter values. This corresponds to calculating the parameters of all straight lines which belong to the set that pass through a given image point ($\bar{x}$, $\bar{y}$). This operation can be called backprojection and the relation, G, can be defined as

$$| G_1 (X_{1j} , Y_{1j}) - G_2 (X_{1j} , Y_{1j}) | \geq T_b$$

$$| G_1 (X_{2j} , Y_{2j}) - G_2 (X_{2j} , Y_{2j}) | \geq T_b \qquad , \qquad 1 \leq j \leq m$$

$$| G_1 (X_{1j} , Y_{1j}) - G_2 (X_{2j} , Y_{2j}) | \leq T_n$$

From these three equations, we can calculate the displacements of the moving objects which are not rotate.

B. The Detection and Estimation of the Moving Objects with Translation and Rotation

In the situation of objects rotation, we can not know the rotation centers of the potential unknown moving objects; thus we can not estimate how the objects rotate if we try to do it by way of their rotation centers. We discard the process for finding the rotation centers in our algorithm and try to find the rotation and translation directly.

For some point (X , Y) in I_1, the coordinate after rotating some degree D and translating T_x and T_y in X and Y axes, respectively, would be (X' , Y'), where

$$X' = X * \cos (D) - Y * \sin (D) + T_x \qquad (5)$$

$$Y' = X * \cos (D) - Y * \sin (D) + T_y \qquad (6)$$

Why the formulas (5) and (6) can simulate all kind of the motions is explained in Fig. 4, let a point be the center of an object O_1 and $\overline{aa'}$ be the principal axis of O_1. If O_1 moves and rotates around point P_r for Ddegree counterclockwise and then results in O_3 , $\overline{cc'}$ is the principal axis of O_3. It is evident that no matter where P_r is, the angle between $\overline{aa'}$ and $\overline{cc'}$ is always D degree. Let object O_2 is O_1 after rotating D degree around the origin(0 , 0) and $\overline{bb'}$ be the principal axis of O_2. We can see easily that $\overline{bb'}$ and $\overline{cc'}$ are parallel; that is, O_2 and O_3 are parallel. If we translate O_2 for some T_x and T_y along X and Y axes, respectively, O_2 can overlap O_3 completely. Hence we conclude that any kind of motion can be simulated by rotating for some degree D around the origin (0 , 0) and then translating for some (T_x , T_y).

3. THE ADVANTAGES OF THE HOUGH TRANSFORM

The HT method has many desirable features. First, each image point is treated independently and therefore the method can be implemented using more than one processing unit; i.e.,parallel processing of all points is possible. This makes it an algorithm suitable for real-time applications and a possible module for shape detection in biological systems. Second, its independent combination of evidence means that it can recognize partial or slightly deformed

$$G((\bar{x}, \bar{y}), (m, c)) = \bar{y} - \bar{x}m - c = 0 \qquad (2)$$

In the case of a straight line each image point ($\bar{x}$, $\bar{y}$) backprojects or defines a straight line in (m , c) parameter space. Figure 1 is a typical point image and Figure 2 shows the parameter lines produced by backprojecting image points into parameter space using Eq. (2). Points which are colinear in image space all intersect at a common point in parameter space and the coordinates of this parameter point characterizes the straight line connecting the image points. The HT determine the point of intersection in parameter space is a local operation and should be considerably easier than detecting extended point patterns in image space.

The extension of the method to detect parametrically defined image curves other than straight line is straightforward. Image points on a curve characterized by n parameters, A_1,.....A_n, can be defined by equation of the form

$$F((\bar{A}_1, \ldots\ldots, \bar{A}_n), (x, y)) = 0 \qquad (3)$$

By changing the roles of parameters and variables, Eq. (3) can be used to give the defining relation for the backprojection mapping of image points to parameter space, i.e.

$$G((\bar{x}, \bar{y}), (A_1, \ldots\ldots, A_n)) = 0 \qquad (4)$$

This backprojection equation maps out a hypersurface in the n-dimensional parameter space. The most probable parameters for image curves are indicated by the intersection of several of these hypersurfaces.

Let the sizes of two images I_1 and I_2 are the same Lx L, where L represent the length of width and height of images. The moving object O_1 in I_1 is consisted of m points P_1 , ... , P_m. Let the set (X_{1j} , Y_{1j}), $1 \le j \le m$, be the coordinates of these points in I_1 and (X_{2j} , Y_{2j}), $1 \le j \le m$, be their coordinates in I_2. The following relationships between (X_{1j} , Y_{1j}) and (X_{2j} , Y_{2j}) must exist:

$$T_x = X_{2j} - X_{1j} \text{ and } T_y = Y_{2j} - Y_{1j}, \; 1 \le j \le m$$

for some T_x and T_y. Let $G_1(X, Y)$ and $G_2(X, Y)$ are the gray level of I_1 and I_2. According to the three assumptions, we can obtain three equations:

shapes. Occlusion is a severe problem for most other shape detection techniques but the HT degrades gracefully because to first order the size of a parameter peak is directly proportional to the number of matching boundary and template points. The size and spatial localization of the peak provides a measure of similarity of shape and model. Third, the HT method is very robust to the addition of random data produced by poor image segmentation. Random image points are very unlikely to contribute coherently to a single bin of the accumulator and therefore produce only a very low level background of counts in the array. A more serious problem than random data is data from the boundaries of shapes other than those searched for. These can produce structured backgrounds and some care must be taken to eliminate or identify such situations. Finally, the HT can simultaneously accumulate evidence for several examples of a particular shape class occurring in the same image. Generally each instance of the shape simply produces a distinct peak or cluster in the accumulate array.

4. THE CONCLUSION

We have shown the basic HT algorithm to eliminate the noise produced by input devices. We also describe the principal ideas of the HT technique to calculate the displacement and the rotation of one or more objects. Although the HT method need more computational time and storage elements, it is still the most reliable algorithm to detect the translation and the rotation simultaneously. The HT algorithm adopt the inverse order of the traditional methods that segment moving objects first and then estimate the ways that they move.We can get more accurate transformation parameters by estimating the way of motion first. Besides, the disadvantage of time consuming of the HT algorithm can be improved by using real- time multiprocessor system. Finally, this algorithm is very simple and robust.

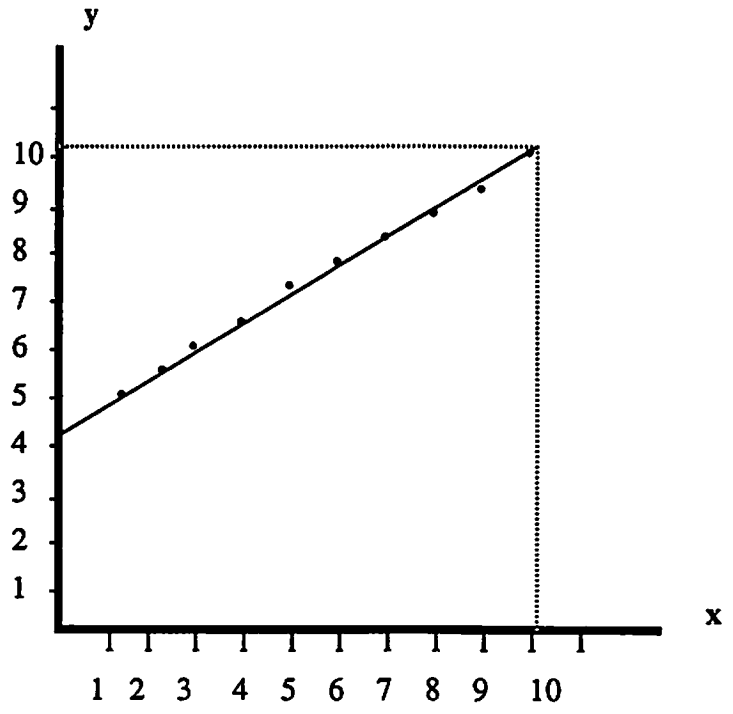


Fig. 1. (x , y) point image space

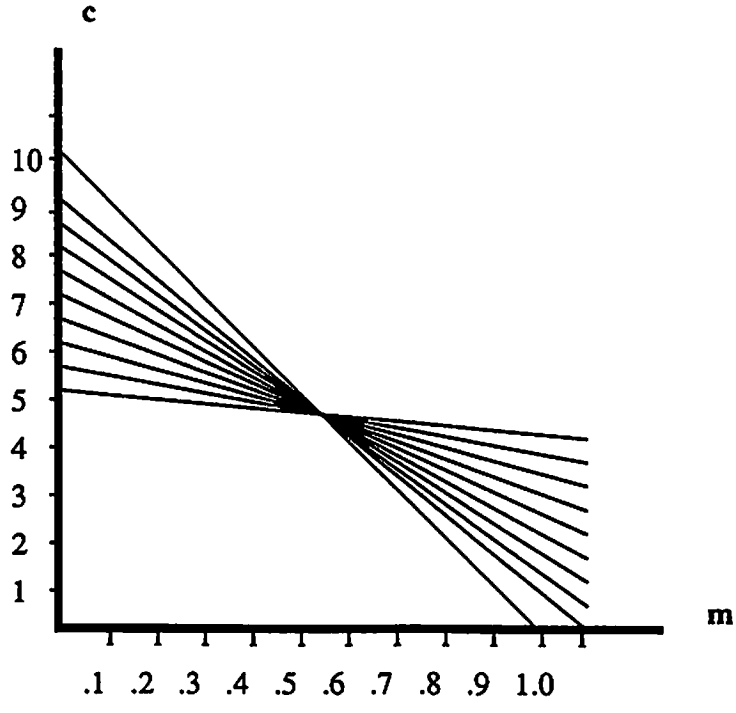


Fig. 2. (m , c) parameter space

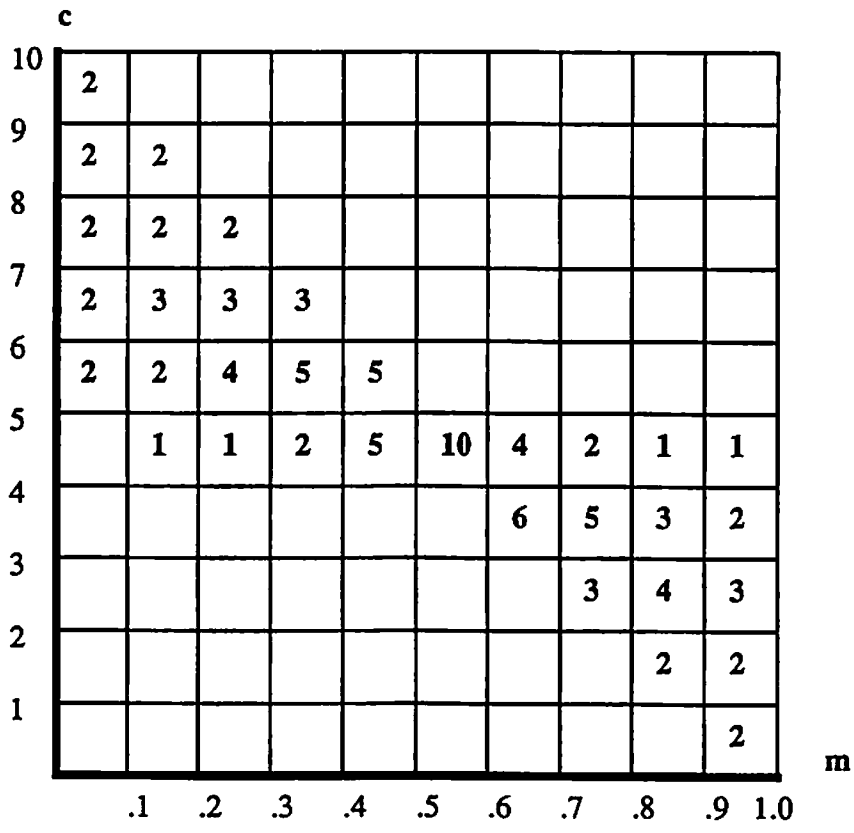


Fig. 3. accumulator space corresponding to Fig. 2.

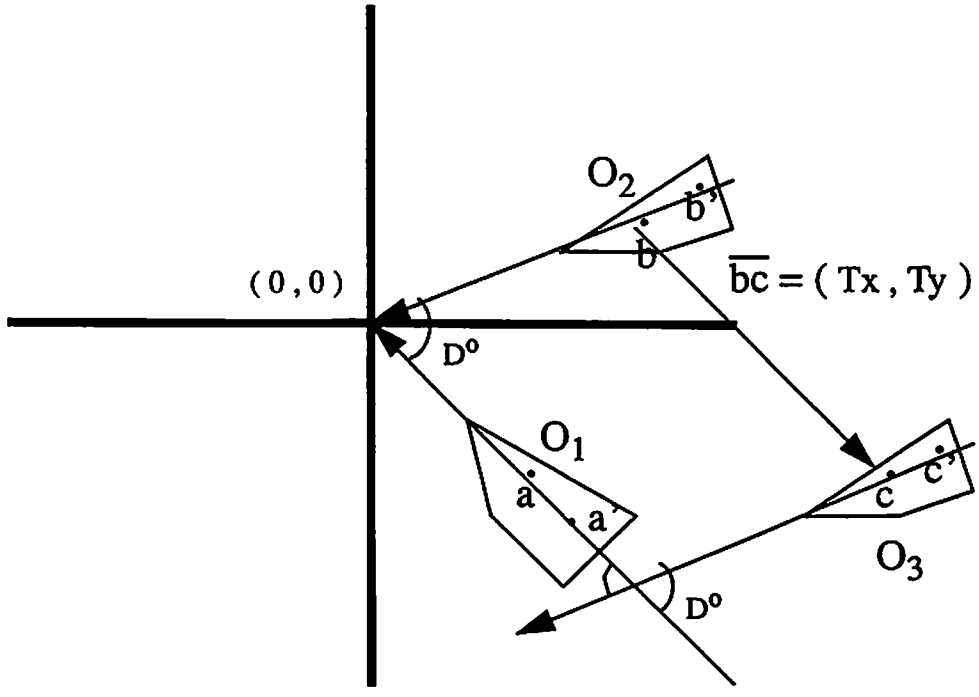


Fig. 4. moving object with rotation and translation

REFFERENCE

[1] J. Weng, T. S. Huang, and N. Ahuja, " Motion and Structure from Two Perspective Views : Algorithm, Error Analysis, and Error Estimation," IEEE Trans. Pattern Anal. Machine Intell., Vol. 11, No.5, 1989, 451-476.

[2] O. D. Faugeras, F. Lustman, and G. Toscani, " Motion and Structure from Point and Line Matches, " Proceedings First Int. Conf. Comput. Vision, London, England, 1987.

[3] J. K. Kearney, W. B. Thompson, and D. L. Boley, " Optical Flow Estimation : An Error Analysis of gradient - Based Methods with Local Optimization," IEEE Trans. Pattern Anal. Machine Intell., Vol. 9, No.2, 1987, 229-244.

[4] B. G. Schunck, " The Image Flow Constraint Equation, " Computer Vision, Graphics, and Image Processing, Vol. 35, 1986, 20-46.

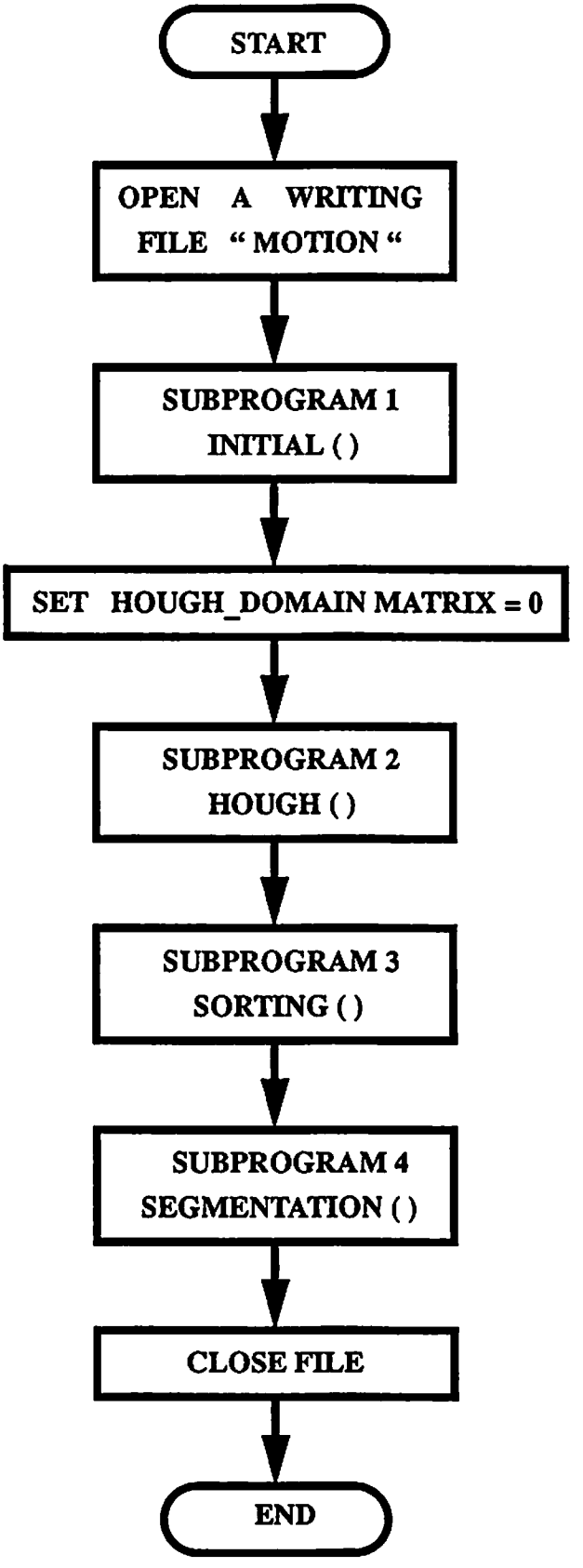
[5] R. M. Haralick and L. G. Shapiro, " Survey : Image Segmentation Techniques," Computer Vision, Graphics, and Image Processing, 29, 1985, 100-132.

[6] J. Illingworth and J. Kittler, " A Survey of Hough Transform," Computer Vision, Graphics, and Image Processing, 44, 1988, 87-116.

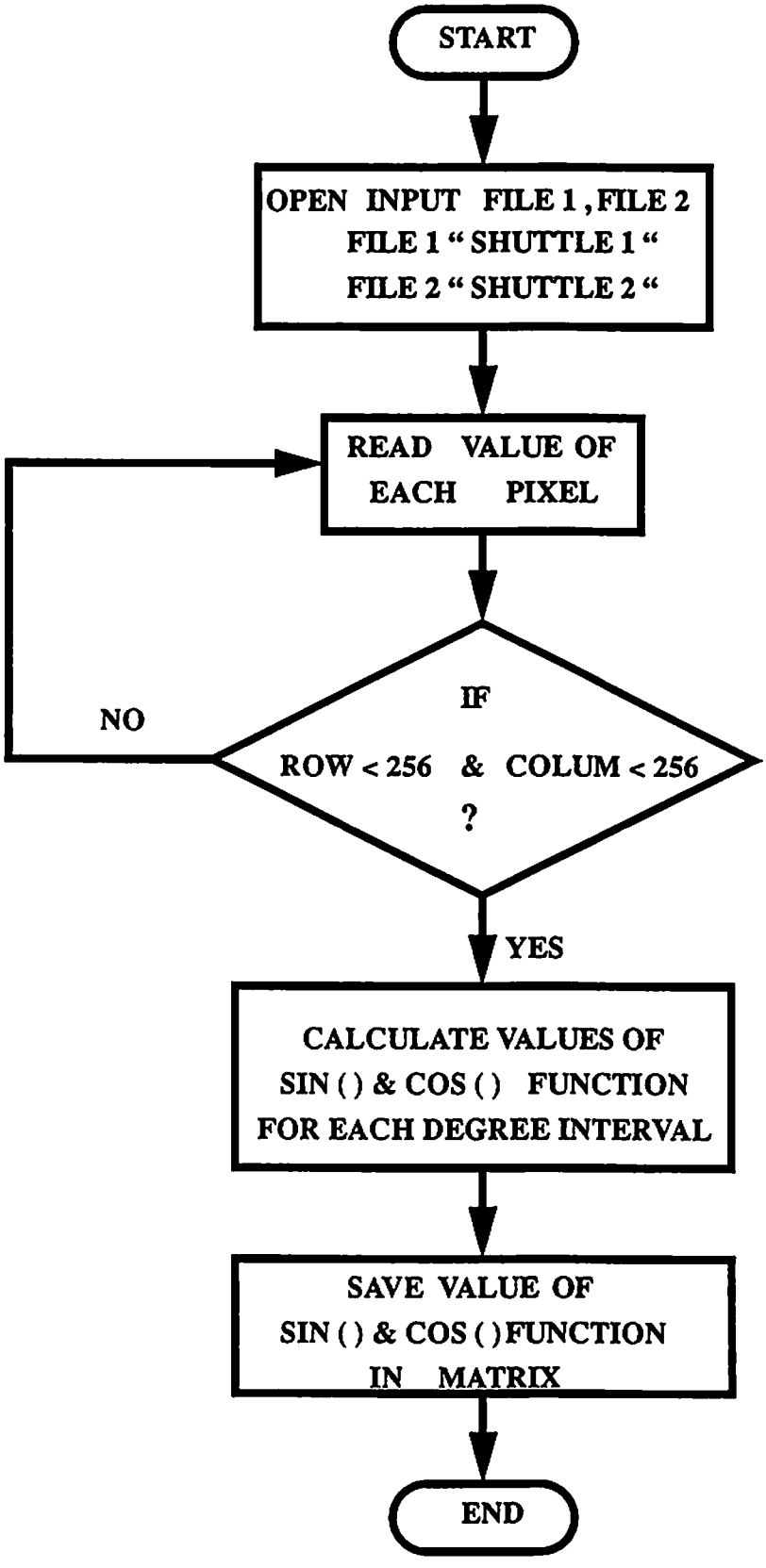
[7] A. E. Cowart and W. E. Snyder, " The Detection of Unresolved Targets Using the Hough Transform," Computer Vision, Graphics, and Image Processing, 21, 1983, 222-238.

[8] A. N. Choudhary and J. H. Patel, " Parallel Architectures and Parallel Algorithms for Integrated Vision Systems," Kluwer Academic Pubishers, 3300 AH Dordrecht, THE NETHERLANDS, 1990, 109-116.

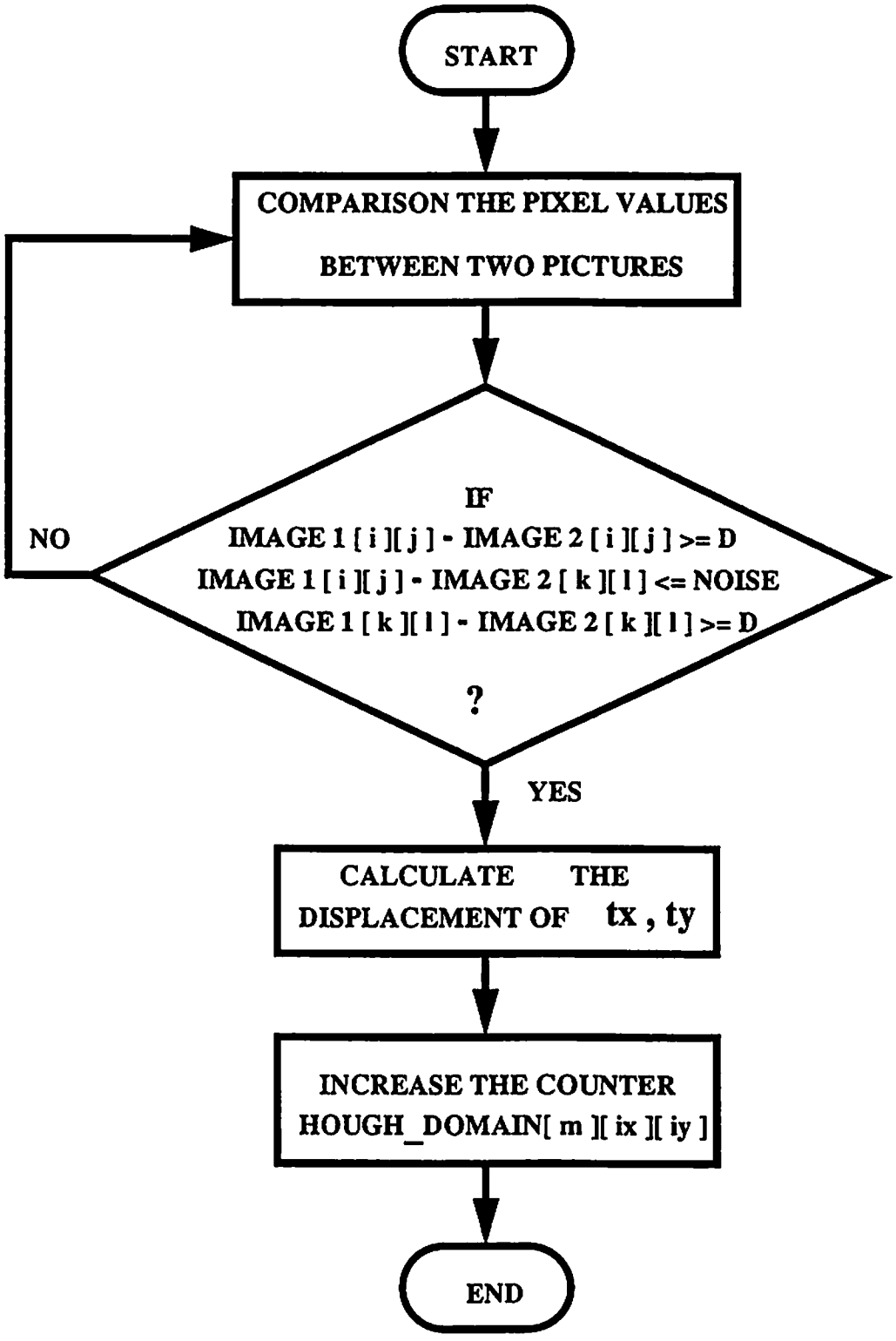
[9] R. C. Gonzalez and P. Wintz, " Digital Image Processing," Addison-Wesley Publishing Company, Calif. 1987.



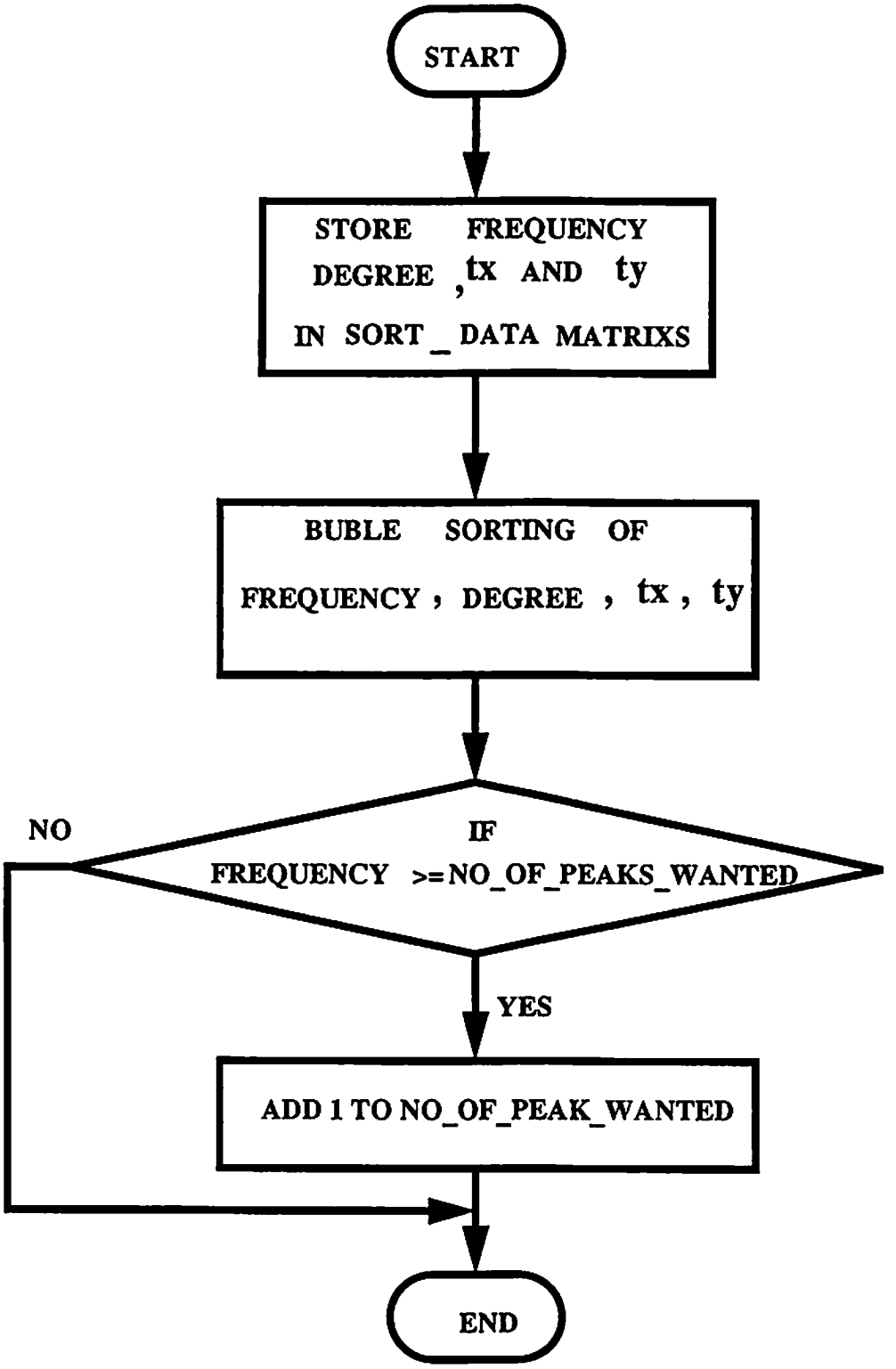
FLOW CHART OF MAIN PROGRAM



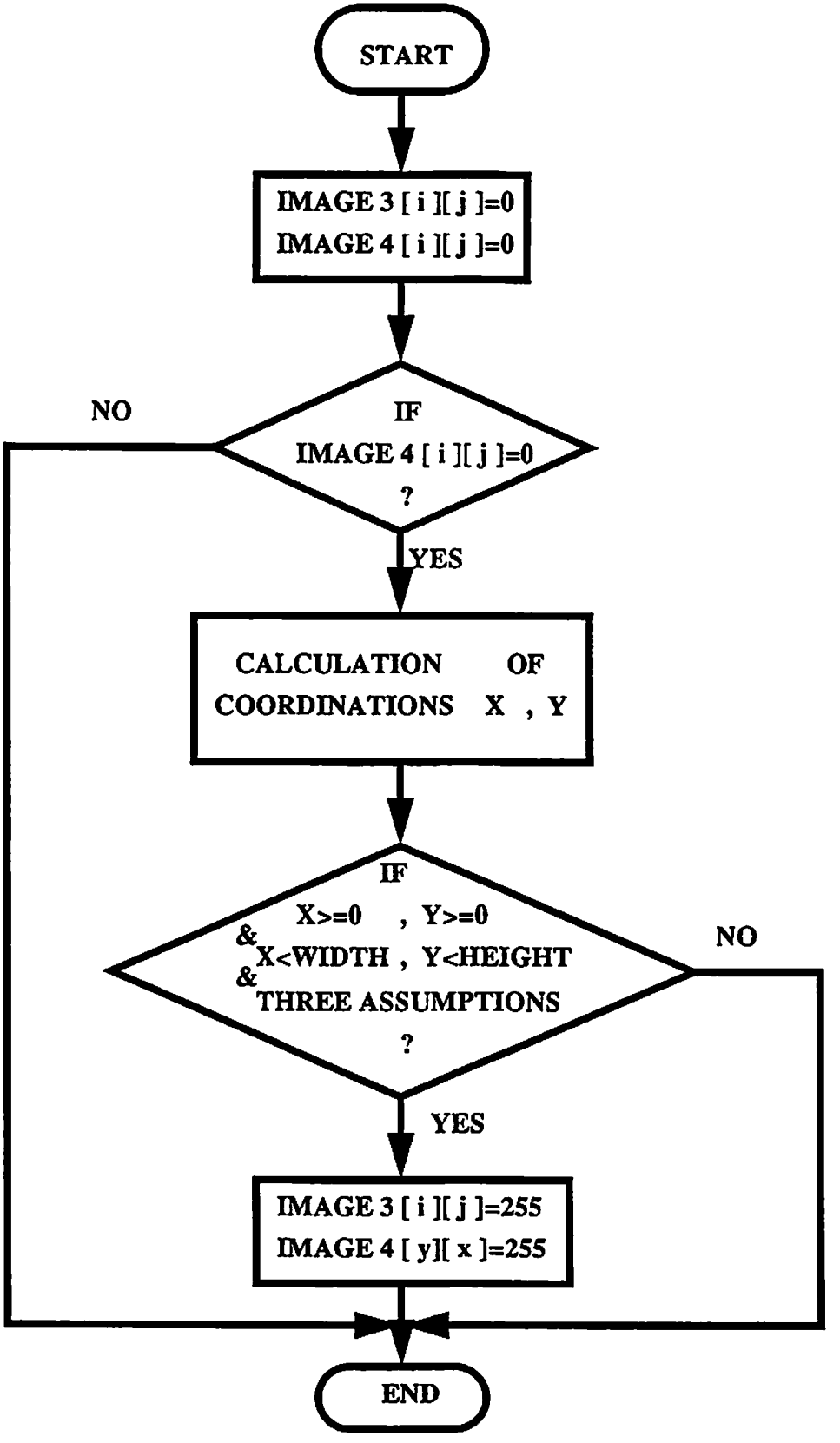
FLOW CHART OF SUBPROGRAM 1



FLOW CHART OF SUBPROGRAM 2 " HOUGH TRANSFORM "



FLOW CHART OF SUBPROGRAM 3 " SORTING "



FLOW CHART OF SUBPROGRAM 4 " SEGMENT "

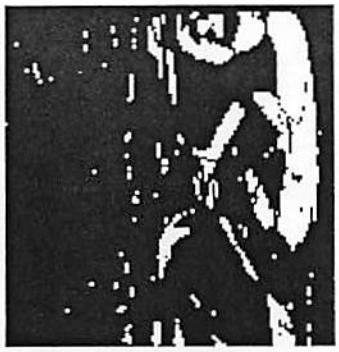
```
#include <math.h>
#include <stdio.h>

#define WIDTH 128
#define HEIGHT WIDTH
#define NO_OF_DEGREE_INTERVAL 1L
#define NO_OF_TX_INTERVAL 128L
#define NO_OF_TY_INTERVAL 128L
#define TOTAL_ENTRY NO_OF_DEGREE_INTERVAL*NO_OF_TX_INTERVAL*NO_OF_TY_INTERVAL
#define PI 3.14159
#define FILE_HEADER 0

#define PWIDTH 128
#define PHEIGHT 128

#define MINFREQUENCY 0
#define MAX_NO_OF_OBJECTS 3
#define SAME 15
#define DIFFERENCE 35
#define NEIGHBOR_SQUARE 6
#define MAX_NO_OF_PEAKS 1

struct sort_data_type
        {
        unsigned long frequency;
        unsigned short degree;
        int tx;
        int ty;
        };
struct sort_data_type SORT_DATA[TOTAL_ENTRY];

unsigned char IMAGE1[HEIGHT][WIDTH], IMAGE2[HEIGHT][WIDTH];
unsigned char IMAGE3[HEIGHT][WIDTH], IMAGE4[HEIGHT][WIDTH];
unsigned long HOUGH_DOMAIN[NO_OF_DEGREE_INTERVAL][NO_OF_TX_INTERVAL]

/* DEGREE_INTERVAL= 180/NO_OF_DEGREE_INTERVAL */
/* TX_INTERVAL= (the real translation x)*(NO_OF_TX_INTERVAL/2)/WIDTH
/* TY_INTERVAL= (the real translation x)*(NO_OF_TY_INTERVAL/2)/HEIGHT

double SIN_ARRAY[NO_OF_DEGREE_INTERVAL],COS_ARRAY[NO_OF_DEGREE_INTERVAL];
int NO_OF_PEAK_WANTED;


void main()
        {
        void sorting();
        void initial();
        void hough();
        void segment();
        void connected_component();
        FILE *fp1,*fopen();
        unsigned long i,i0,j,k,counter;
        unsigned short cluster[MAX_NO_OF_PEAKS][MAX_NO_OF_PEAKS];
        unsigned short cluster_counter[MAX_NO_OF_PEAKS];
        unsigned short total_cluster;

        fp1=fopen("result","wb");
        initial();
        counter=0;
        for(i=0;i<NO_OF_DEGREE_INTERVAL;i++)
          for(j=0;j<NO_OF_TX_INTERVAL;j++)
            for (k=0;k<NO_OF_TY_INTERVAL;k++)
              HOUGH_DOMAIN[i][j][k]=0;

        hough();
        sorting();
```

```
        i0=TOTAL_ENTRY - 1 - NO_OF_PEAK_WANTED;
        for(i=0;i<NO_OF_PEAK_WANTED;i++)
          fprintf(fp1,"degree= %u, tx= %d, ty= %d, frequency= %lu.\n",SORT_DATA[
              i].degree,SORT_DATA[i].tx,SORT_DATA[i].ty,SORT_DATA[i].frequency);
        fprintf(fp1,"\n");
        segment((int)(SORT_DATA[0].tx),(int)(SORT_DATA[0].ty),(int)(SORT_DATA[0]
        fclose(fp1);
        }


void initial()
        {
        void input_image();
        unsigned long i,j;
        double sin(),cos(),t0;

        input_image("ms1.128",IMAGE1);
        input_image("ms5.128",IMAGE2);

        t0=PI/NO_OF_DEGREE_INTERVAL;
        for(i=0;i<NO_OF_DEGREE_INTERVAL;i++)
          {
          SIN_ARRAY[i]=sin(((double)(i))*t0);
          COS_ARRAY[i]=cos(((double)(i))*t0);
          }
        }


void input_image(file_name,image)
unsigned char *file_name,image[HEIGHT][WIDTH];
        {
        FILE *fp,*fp1,*fopen();
        unsigned long i,j,k,height_offset,width_offset;

        width_offset=(PWIDTH-WIDTH)/2;
        height_offset=(PHEIGHT-HEIGHT)/2;

        fp=fopen(file_name,"rb");
        for(i=0;i<FILE_HEADER;i++)
              getc(fp);
        for(i=0;i<height_offset;i++)
          for(j=0;j<PWIDTH;j++)
            getc(fp);
        for(i=0;i<HEIGHT;i++)
          {
          k=0;
          for(j=0;j<PWIDTH;j++)
            {
            if (j<width_offset)
              getc(fp);
            else if (k<WIDTH)
                  {
                  image[i][k]=getc(fp);
                  k++;
                  }/* else if (k<WIDTH) */
                else
                  getc(fp);
            }/* for j */
          }/*for i*/
        fclose(fp);
        }


void hough()
        {
        unsigned long i,j,k,l,m;
```

```
        int ix,iy,offsetx,offsety,tx,ty;
        double fabs(),max_tx,max_ty,sqrt();

        offsetx=NO_OF_TX_INTERVAL/2;
        offsety=NO_OF_TY_INTERVAL/2;
        max_tx=WIDTH*(1+sqrt(2.0));
        max_ty=HEIGHT*(1+sqrt(2.0));

        for(i=0;i<HEIGHT;i++)
          for(j=0;j<WIDTH;j++)
            {
            for(k=0;k<HEIGHT;k++)
              for(l=0;l<WIDTH;l++)
                {
                if ((fabs((double)(IMAGE1[i][j])-(double)(IMAGE2[i][j]))>=
                  {
                  for(m=0;m<NO_OF_DEGREE_INTERVAL;m++)
                    {
                    tx=(int)(l+i*SIN_ARRAY[m]-j*COS_ARRAY[m]);
                    ty=(int)(k-i*COS_ARRAY[m]-j*SIN_ARRAY[m]);
                    ix=(int)((tx*offsetx)/max_tx+offsetx);
                    iy=(int)((ty*offsety)/max_ty+offsety);
                    if (ix<0) ix=0;
                    if (ix>=NO_OF_TX_INTERVAL) ix=NO_OF_TX_INTERVAL-1;
                    if (iy<0) iy=0;
                    if (iy>=NO_OF_TY_INTERVAL) iy=NO_OF_TY_INTERVAL-1;
                    (HOUGH_DOMAIN[m][ix][iy])++;
                    }/* for m */
                  }/*if (abs) */
                }/*for k,l */
            }/* for i,j */
        }


void sorting()
        {
        unsigned long i,j,k,l,i0;
        unsigned short degree0;
        int offsetx,offsety;
        double max_tx,max_ty,sqrt();

        offsetx=NO_OF_TX_INTERVAL/2;
        offsety=NO_OF_TY_INTERVAL/2;
        max_tx=WIDTH*(1+sqrt(2.0));
        max_ty=HEIGHT*(1+sqrt(2.0));

        i0=NO_OF_TX_INTERVAL*NO_OF_TY_INTERVAL;
        for(i=0;i<NO_OF_DEGREE_INTERVAL;i++)
          {
          degree0=(unsigned short)(i*180.0/NO_OF_DEGREE_INTERVAL);
          for(j=0;j<NO_OF_TX_INTERVAL;j++)
            for(k=0;k<NO_OF_TY_INTERVAL;k++)
              {
              l=i*i0+j*NO_OF_TY_INTERVAL+k;
              SORT_DATA[l].frequency=HOUGH_DOMAIN[i][j][k];
              SORT_DATA[l].degree=degree0;
              SORT_DATA[l].tx=(int)(((double)(j)/offsetx-1)*max_tx);
              SORT_DATA[l].ty=(int)(((double)(k)/offsety-1)*max_ty);
              }/*for j,k */
          }/*for i*/
        for(NO_OF_PEAK_WANTED=0,i=TOTAL_ENTRY-1;(i>0) && (NO_OF_PEAK_WANTED<
          {
          for(j=TOTAL_ENTRY-1;j>(TOTAL_ENTRY-i-1);j--)
            {
              k=j-1;
              if (SORT_DATA[j].frequency>SORT_DATA[k].frequency)
```

```
                {
                l=SORT_DATA[j].frequency;
                SORT_DATA[j].frequency=SORT_DATA[k].frequency;
                SORT_DATA[k].frequency=l;

                degree0=SORT_DATA[j].degree;
                SORT_DATA[j].degree=SORT_DATA[k].degree;
                SORT_DATA[k].degree=degree0;

                offsetx=SORT_DATA[j].tx;
                SORT_DATA[j].tx=SORT_DATA[k].tx;
                SORT_DATA[k].tx=offsetx;

                offsety=SORT_DATA[j].ty;
                SORT_DATA[j].ty=SORT_DATA[k].ty;
                SORT_DATA[k].ty=offsety;
                }
            }/*for j*/
          if (SORT_DATA[TOTAL_ENTRY-i-1].frequency>=MINFREQUENCY)
            NO_OF_PEAK_WANTED++;
          else
            break;
          }/*for i*/
        }
/* looking for peak */


void segment(tx,ty,thita,range)
int tx,ty,thita,range;
        {
        int i,j,k,l,m,a,b,c;
        int x,y;
        unsigned long o;
        double fabs();
        FILE *fp,*fopen();

        for(i=0;i<HEIGHT;i++)
          for(j=0;j<WIDTH;j++)
            {
            IMAGE4[i][j]=0;
            IMAGE3[i][j]=0;
            }
        for(i=0;i<HEIGHT;i++)
          for(j=0;j<WIDTH;j++)
            for(k=ty-range;k<=ty+range;k++)
              for(l=tx-range;l<=tx+range;l++)
                {
                x=(int)(j*COS_ARRAY[thita]-i*SIN_ARRAY[thita]+l);
                y=(int)(j*SIN_ARRAY[thita]+i*COS_ARRAY[thita]+k);
                if ((x>=0) && (y>=0) && (x<WIDTH) && (y<HEIGHT) && (fabs(
                  {
                  IMAGE3[i][j]=255;
                  IMAGE4[y][x]=255;
                  }
                }
        fp=fopen("p1","wb");
        for(i=0;i<HEIGHT;i++)
          for(j=0;j<WIDTH;j++)
            putc(IMAGE1[i][j],fp);
        fclose(fp);

        fp=fopen("p2","wb");
        for(i=0;i<HEIGHT;i++)
          for(j=0;j<WIDTH;j++)
            putc(IMAGE2[i][j],fp);
        fclose(fp);
```

```
fp=fopen("p3","wb");
a=0;
b=0;
for(i=0;i<HEIGHT;i++)
  for(j=0;j<WIDTH;j++){
    if (IMAGE3[i][j]==0){
       a++;
       }
    else if (IMAGE3[i][j]>0){
             IMAGE3[i][j]=255;
             b++;
             }
             putc(IMAGE3[i][j],fp);
    }
fclose(fp);

c=0;
fp=fopen("p4","wb");
for(i=0;i<HEIGHT;i++)
  for(j=0;j<WIDTH;j++){
    if (IMAGE4[i][j]>0){
        IMAGE4[i][j]=255;
        c++;
        }
    putc(IMAGE4[i][j],fp);
    }
fclose(fp);

for(i=0;i<HEIGHT;i++)
  for(j=0;j<WIDTH;j++)
    {
    if (IMAGE3[i][j]==255) IMAGE1[i][j]=IMAGE2[i][j];
    if (IMAGE4[i][j]==255) IMAGE2[i][j]=IMAGE1[i][j];
    }
}
```



The Two Dimensional Motion Estimation Based On Hough Transform

Part II:

Shih-Chia Yang

Department of Electrical Engineering

Abstract : The traditional way of motion segmentation, which uses the difference between two consecutive frames, is not valid when the camera shifts or vibrates. This results in some people to think of other algorithms to solve these drawbacks. Here we introduce a new algorithm, based on Hough _Transform, which has been proven to be a very reliable method for the motion segmentation. But, due to the fact that Hough Transform is time and space consuming, we will focused on hardware implementation by introducing real time multiprocessor system in this paper and try to evaluate the performance hopefully.

1. Introduction

Practically since the first working computer, architects have been striving for composing a powerful computer by simply connecting many existing smaller ones. The user orders as many CPUs as he can afford and gets a commensurate amount of performance. In addition, for decades, computer designer have been looking for the missing piece of the puzzle that allows this speedup to happen, as if by image. People are heard making statements that begin "Now that computers have dropped to such a low price...." or " This new interconnection scheme will overcome the scaling problem, so .... " and end with "MINDs(stands for multiprocessor system)will (finally) dominate computing."

Research and development of multiprocessor systems are aimed at improving throughput-flexibility and availability. A basic multiprocessor system contains two or more processors of approximately comparable capabilities. All processors share access to common sets of memory modules, I/O channels, and peripheral devices. Most importantly, the entire system must be controlled by a single integrated operating system providing interactions between processors and their programs at various level. Besides, each processor has its own local memory and private devices. Interprocessor communications can be done through the shared memories or though an interrupt network,.

In this paper, we will introduce real time multiprocessor for the hardware design of the motion detection based on Hough Transform. Because of the data independency during the arithmetic processing, the demand of the real time situation and the dicision, between the efficiency and hardware cost ,we can predict that the real time multiprocessor systembased on-Hough_transform-based algorithm will have the highest absolute performance and highest reliability compared with the largest uniprocessor.

2. The 2D Motion Estimation Based On Hough Transform

In this section, we make three fundamental assumptions. And the algorithm for motion detection and estimation is based on them. These assumptions are:

(1) The difference of the gray levels between the background and the moving object(s) is greater than T_b.
(2) The difference of the gray levels, due to the noise, between all the corresponding pixels in two consecutive images is less than T_n.
(3) The area of the moving object should be greater than T_a.

Now we try to develop a Hough Transform algorithm for motion detection and estimation on 2D images. The estimation of the motion parameters is restricted to 2D case since 3D case must be based on the "good result" of the 2D motion analysis.
Our algorithm can be divided into 2 cases for discussion:
Case (A). To detect the moving object(S) without rotation; that is, there are only translations for the moving object(S);
Case (B). To detect the object(S) with rotation as well as translation.

Case (A) is not complicated for analysis and implementation while case (B) is rather complicated and has some nontrivial problems to be encountered with. We first discuss case (A) in the following paragraph.

(A) Detection and Estimation of the Moving Objects with Pure Translation

Let I_1 be the first image of size hxw, and I_2 be the second one with the same size, Suppose there is a moving object O_1 which consists of m points $P_1, P_2,....,P_m$. Let (X_{1j},Y_{1j}), $1\leq j\leq m$, be the coordinates of these points in I_1 and (X_{2j},Y_{2j}), $1\leq j\leq m$, be their coordinates in I_2. Then the following relationships between (X_{1j},Y_{1j}) and (X_{2j},Y_{2j}) must exist:

$$T_x=X_{2j}-Y_{1j} \text{ and } T_y=Y_{2j}-Y_{1j},\ 1\leq j\leq m$$

for some Tx and T_y.

Now we want to estimate $T_x$and T_y based on the fundamental assumptions and Hough_ Transform. Let $G_1(X,Y)$ and $G_2(X,Y)$ be the gray levels at point (X,Y) of the images I_1 and I_2 respectively. According to the assumption 1 in the previous paragraph, the difference of the gray levels between a moving object and its background should be greater than T_b; that is,

$$|G_1(X_{1j},Y_{ij})-G_2(X_{1j},Y_{1j})| \geq T_b \text{ and}$$
$$|G_1(X_{2j},Y_{2j})-G_2(X_{2j},Y_{2j})| \geq T_b,\ 1\leq j\leq m \text{..... (1)}$$

Any pixel which is not a part of the moving object(s) will have the same gray level, not satisfying equation (1), in the images I_1 and I_2 except when the cameras shifts. Also, from the assumption 2, the noise variation of the camera is less than T_n, so

$$|G_1(X_{1j},Y_{1j})-G_2(X_{2j},Y_{2j})| \leq T_n,\ 1\leq j\leq m \text{..... (2)}$$

Please note that a moving object of size m in images I_1 and I_2 should satisfy constraints (1) and

(2).

Now we show how T_x and T_y are estimated by Hough _Transform. Let P be a point in I_1, and we check all the points in I_2 to get the points $P_1,P_2,P_3,.....,P_k$which satisfy constraints (1) and (2) when compared with P; that is, P_j's in $I_2$are the potential corresponding points of P in I_1. Then we figure out the vectors $\overline{P\,P_1}$, $\overline{P\,P_2}$, $\overline{P\,P_3}$,.....,$\overline{P\,P_k}$, and round the X and Y components of each vector. Let (T_{x1},T_{y1}), (T_{x2},T_{y2}), (T_{x3},T_{y3}),....., (T_{xk},T_{yk}) be the result vectors (T_{xi}'s and T_{yi}'s are all integers). Because $I_1$and I_2 are of the size hxw, $-w \le T_{xi} \le w$, $-h \le T_{yj} \le h$, we initialize a frequency counter array frequency[-w..w,-h..h] to zero, and frequency[-w..w,-h..h] is used to record the frequencies of all the possible vectors.

Whenever a vector (T_x, T_y) is figured out, we increment the corresponding entry of frequency$[T_x, T_y]$ by 1; that is, frequency$[T_x,T_y]$ = frequency$[T_x,T_y]$ + 1. After we have selected all the points in I_1, found all the potential corresponding points in I_2 and finally incremented all the corresponding entries of frequency[-w..w,-h..h], we get all the frequencies of all possible motions.

Finally we want to segment the moving objects by these frequency vectors. Let (X_k, Y_k) be one of the f highest frequency vectors, and I_3 and I_4 be two images with uniform gray level 0. For all the points (X,Y) in I_1, if ($|G_1(X,Y) - G_2(X+X_k,Y+Y_k)| \le T_n$) and ($G_1(X,Y) - G_2(X,Y)| \ge T_b$) and ($G_1(X+X_k,Y+Y_k)-G_2(X+X_k,Y+Y_k)| \ge T_b$), then set $I_3[X,Y]=255$ and $I_4[X+X_k,Y+Y_k]=255$ (G_1 and G_2 are defined in constraints (1) and (2).). Thus I_3 will show the moving object(s) in I_1, and I_4 will show those in I_2. The objects shown in I_3 and I_4 will satisfy the translation (X_h, Y_h). By this method, we can get all the possible moving objects and all the possible ways of their motions.

(B). Detecting the Moving Objects with Translation and Rotation.

An important problem incurred, when rotation is considered, is that we can not know for sure the rotation center(s) of the potential unknown moving object(s); thus we can not estimate how the object(s) rotate if we try to do it by way of their rotation center(s). In our algorithm we skip the process for finding the rotation center(s) of the object(s) and try to estimate the rotation and translation directly.

For some point (X,Y) in I_1, the coordinate after rotating θ and translating T_x and T_y in X and Y axes, respectively, would be (X', Y'), where

$$X' = X * \cos(\theta) - Y * \sin(\theta) + T_x \quad \text{..... (3)}$$
$$Y' = X * \sin(\theta) + Y * \cos(\theta) + T_y \quad \text{.....(4)}$$

These two simple formulas, (3) and (4), are used to describe any kind of motion (translation and rotation) for all the potential moving objects. In Fig. 2.1., let A be the center of an object$_a$ and $\overline{AA'}$ be the principal axis of object$_a$, If object$_a$ moves and rotates around some point P_r(not necessarily the center point A) for α^0 counterclockwise and then results in object$_c$. C and $\overline{CC'}$ are the center and principal axis of object$_c$, respectively. It is evident that no matter

where P_r is, the angle between $\overline{C'C}$ and $\overline{A'A}$ is always α^0. Let $object_b$ be $object_a$ after rotating D^0 degrees around the origin (0,0) and $\overline{B'B}$ be the principal axis of $object_b$. We can see easily that $\overline{B'B}$ and $\overline{C'C}$ are parallel; that is, $object_b$ and $object_c$ are parallel. If we translate $object_b$ for some T_x and T_y along X and Y axes, respectively, $object_b$ can overlap $object_c$ completely. Hence we conclude that any kind of motion (including translation and rotation) can be simulated by rotating for some degree α^0 around the origin (0,0) and then translating for some (T_x,T_y).

Now we want to extract moving objects and estimate how they move. Firstly, we initialize all the entries of the value of the frequency_counter array to be zero. We quantize $[0,2\pi]$equally into n intervals, $\theta_0 (= 0)$, $\theta_1,\theta_2,\theta_3,.....\theta_n(= 2\pi)$. If (X_1,Y_1) in I_1 and (X'_2,Y'_2) in I_2 satisfy the fundamental assumptions 1 and 2, then for all the θ_i, $0 \leq i \leq n$, figure out all the T_{xi} and T_{yi} according to the equations (3)and(4) and then increment the corresponding entry frequency $[\theta_i,T_{xi},T_{yi}]$ by 1; that is,

$$\text{frequency}[\theta_i,T_{xi},T_{yi}] = \text{frequency}[\theta_i,T_{xi},T_{yi}] + 1$$

After all the corresponding points, which are in images I_1 and I_2 and satisfy the relations shown in constraints (1) and (2), have been processed, we can get all the frequencies, stored in the frequency counter array , of all the possible transformations, including rotation and translation. Compared with the previous translational motion analysis in case (A), there is an extra component θ in the frequency counter array now. But we can treat θ just like the other two components T_x and T_y when considering the connected relation in case (A). In the same way shown in case (A), all the possible moving objects can be extracted from the images I_1 and I_2 by the information from the value of the frequency_counter array .

3. Architecture of the Real Time Multiprocessor

As mentioned in the earlier section, the computer architecture of Hough_Transform-based system is designed to have the feature of multiprocessors (4 processors for real-time subtasks, 1 X 80286 microprocessor for real-time I/O tasks, and 1 X 80286 microprocessor for main tasks).

The basic configuration of this computer system is shown in the figure 3.1. It can be observed that the whole system is composed of three subsystems. The first one is called main microprocessor system, which consists of a unibus and various devices addressed by this bus system (main microprocessor or master, exclusive memory, shared memory, local memory, and I/O interface with outside world). This subsystem is designed to work as a coordinator between outside world (human beings) and the center of task execution. The second subsystem is namely a local processor system, which is designated to execute real-time tasks. The relation between the main MPU and its local PE's is shown in Fig.3.2.1and Fig.3.2.2 ... The last one is the I/O system. This subsystem has its I/O microprocessor, its data and operating system memory, and I/O channels to the cameras.We will not discuss this topic in this paper.One more

thing has to mention here is that the internal communication among these local processors is independent, due to the data independency for our special algorithm based on Hough _Transform.

3.1 Main - processor system

As mentioned earlier, the main-microprocessor system is just like a 80286 personal computer. The basic configuration is shown in the figure 3.1. The omnibus system can access all of resources in the whole system, such as, main memory, local memory, I/O memory, local processors, an I/O microprocessor, and other peripheral devices.

In order to reduce the turn-around time of the system, the communication between the master and its slaves is implemented through the hardware--interrupting interface circuit and I/O interface circuit, which is much faster than through the software absolutely, The interrupting interface circuit has five parallel 8259A interrupt controllers. Each of them is designed to connect with one of five slaves. In the real-time situation, only three of interrupt inputs (IR0, IR1, IR2) are defined in each 8259A chip, and the rest of them can be used for the initialization and the future expansion.

In the main-microprocessor system, IBM-80286 system, is to cooperate with local-processor system and I/O processor unit. We can group the signals of IBM-80286 into four groups : the memory /IO interface, interrupt interface, DMA interface and processor extension interface. The memory /IO interface include data bus, address bus, bus high enable $\overline{BHE}$, code or interrupt acknowledge, memory or IO select, status lines $\overline{S}_1$ and $\overline{S}_0$, $\overline{READY}$ and $\overline{LOCK}$ totally seven signal and two kind of buses. The 80286 microcomputer has an independent I/O address space which is 64K bytes in lenth. Therefore, just address lines A_0 through A_{15} are used when addressing I/O devices. Its data bus is formed by 16 data lines D_0 through D_{15}.

3.2 local - processor system

The basic operation and relation between each PE and MPU are showned in Fig.3.2.1.Besides the Local Memory, there are two key parts in the local PE. One is ALU, another is control unit. In our special design, there are 2 independent arithmetic units controlled by 2 independent control units. Fig.3.3 shows a functional block of Hough_Transform processing unit. The final results, T_x and T_y ,are the horizontal displacement and vertical displacement respectly. One thing to notice is that Tx and T_y is not the coordinate corresponding to the maximum frequency we want, therefore, we need to put their values into the temporary memory. Through the control signal in Fig.3.4,signal C9 will interrupt MPU to increase the value of the frequency array corresponding certain coordinate with the value of T_x and T_y. The controller unit 1 in Fig.3.4 which is responsible for all the procedure to get the all possible T_x and T_y is completely controlled by MPU. The detailed control signal specification is depicted in Table 1.

	Operation Controlled
C0	Transfer i, j on INBUS to REG 1
C1	Transfer i, j on REG 1 to REF 5
C2	Transfer SIN_ARRAY[thita] , COS_ARRAY[thita] and REG 5 to the input of Multi 1
C3	Transfer Multi 1 and Multi 2 to Add/Sub
C4	Perform Subtraction
C5	Transfer Add/Sub to Add_1 and Add_2
C6	Transfer Add_1 and Add_2 to REG 8 and REG 9
C7	Transfer T_x, T_y to Temporary Memory and Control unit
C8	Clear Counter
C9	Increase frequency array [m] [T_x] [T_y]

Table 1 Control Signals for the Hough_Transform Processing Unit

Fig.3.5shows the functional block diagram of image segmentation processing unit. After finding out all possible T_x, T_y, MPU will do the sorting job and send the final result of the coordinate with the maximum frequency to the local memory of each PE and then sends the begin signal to trigger the control unit 2 in Fig.3.6. At this moment, the image segmentation processing unit will start the job followed by the control signal and finally get the coordinate value that the moving object in image 1 move toward. The detailed control signal specification is showed in Table 2.

Control Signal	Operation Controlled
C0	Transfer i, j on INBUS to REG 1
C1	Transfer REG 1, REG 2 to REG 5, REG 6
C2	Transfer SIN_ARRAY[thita], COS_ARRAY[thita], REG 5,....to Multi 1,2,....
C3	Transfer Multi 1, 2 to Add_1
C4	Perform Subtraction
C5	Transfer Add_1, Add_2 to REG 7, REG 8
C6	Transfer X,Y to control unit 2 and Temporary Memory
C7	set IMAGE3[i][j]=255;IMAGE4 [X][Y]=255

Table 2 Contjrol Signals for the Image Segmentation Processifng Unit

After all the input images are compared and executed through Hough_Transform and segmentation processing unit, the corresponding coordinates in image 1 and image 2 are put in the temporary memory. As the control signal C7 goes up, this interrupts MPU to store the result from the temporary memory into the main memory. After the counter inside the control unit 2 reach the set value, all the coordinates will be stored back to the main memory finally.

4. Conclusion

Hough_Transform based algorithm applied to the motion dection and estimation involves in many repeated operations with no data dependency hazard. This observation gives us the idea that MIMD or SIMD which distributes the repeated job to the individual PE controlled by a host computer will substantially increase the performance if we can conquer the complexity of the operating system. Then, the consideration we need to make left is between the efficiency and the price it costs. In this paper, we use 2 x 80286 microprocessor as the MPU and fthe I/O processor. If there is no restriction processing time on the numbers of the PE we can implement, it is sure that the processing time we achieve vs different numbers of PE is shown in Fig.4.1. From this diagram, we positively believe that implementing this special algorithm with real time multiprocessor system is the right choice absolutely.

REFIRENCE

[1] J. Illingworth and J. Kittler, "A Surver of Hough Transform," Computer Vision, Graphics, and Image Processing, 44, 1988, 87-116

[2] A.E. Cowart and W.E Snyder, "The detection of unresolved targets using the Hough Transform", Computer Vision, Graphics, and Image Processing, 21, 1983, 222-238

[3] A.N. Chaudhary and J.H. Patel, "Parallel Architectures and Parallel Algorithms for Integrated Vision Systems," Kluwer Academic Publishers, 3300 AH Dordrecht, THE NETHERLANDS, 1990, 109-116.

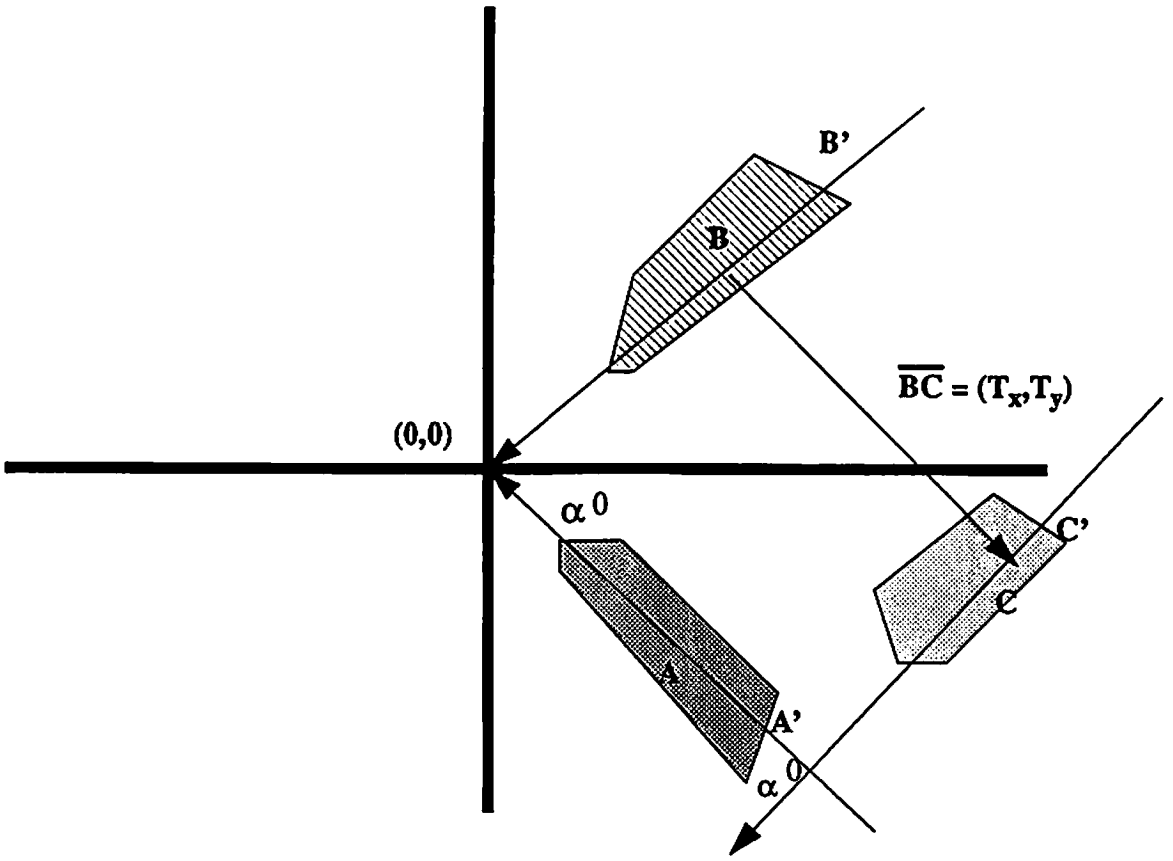
[4] John P. Hayes, "Computer Architecture and Organization", Mcgraw-Hill International Editions, 285-315.

[5] Kai Hwang and Faye'A. Briggs, "Computer Architecture and Parallel processing", Mcgraw-Hill International Editions, 459-502.

[6] John L. Hennessy and David A. Patterson, "Computer Architecture a quantitative Approach", Morgan Kanfmann Publishers INC, 545-575.

[7] The 80286 Microprocessor Hardware, software and Interface. (Walter A. Triebel and Avtar Singh) 1990.

[8] Intel Corporation, Microprocessor and Peripheral Hand book, 1983.



: initial location of an object 0.

: final location of the object 0 after motion.

:location of the object 0 after rotating α^0 degrees around the origin (0,0).

Fig. 2.1

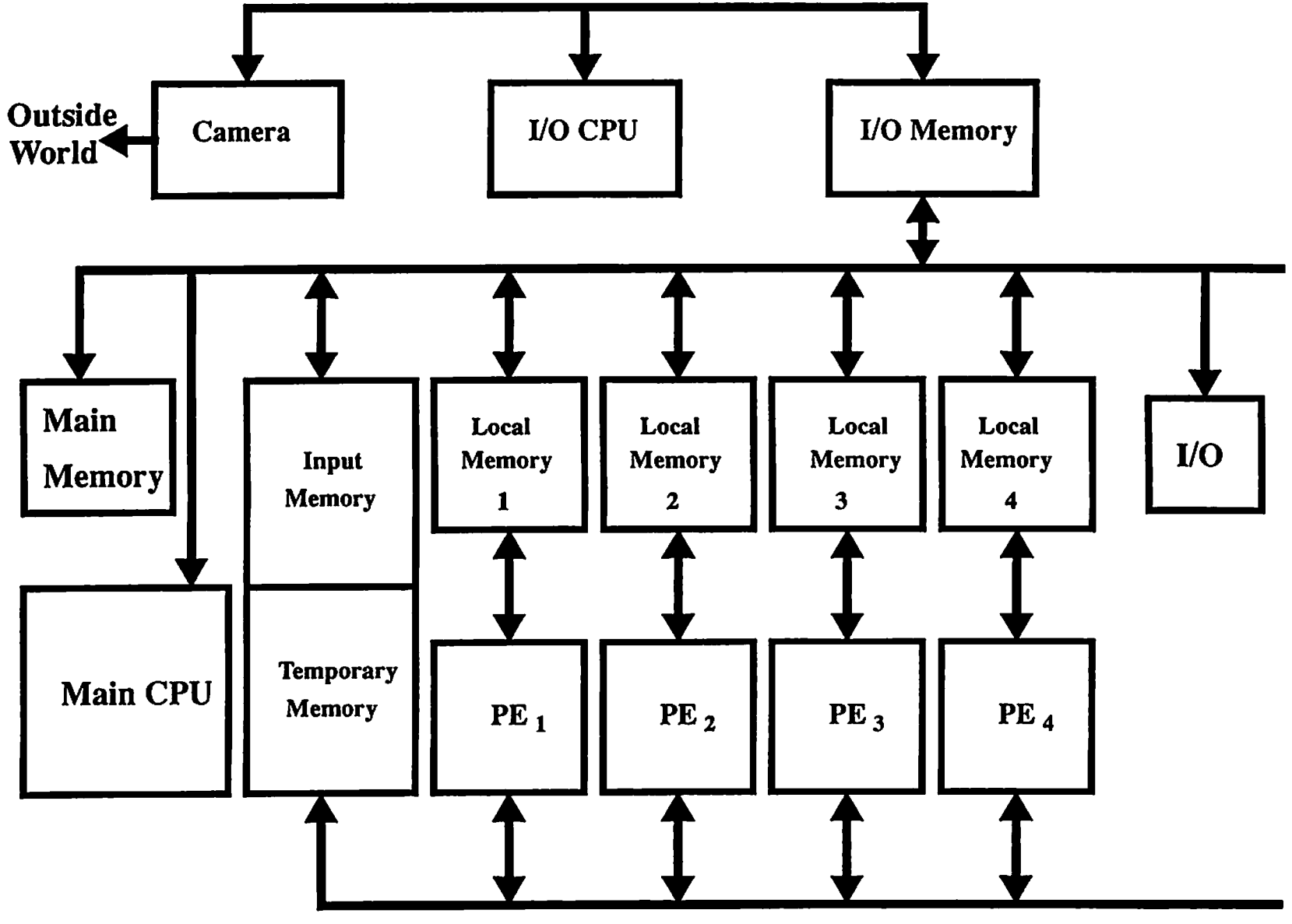


Fig.3.1 Basic Configuration of Computer System

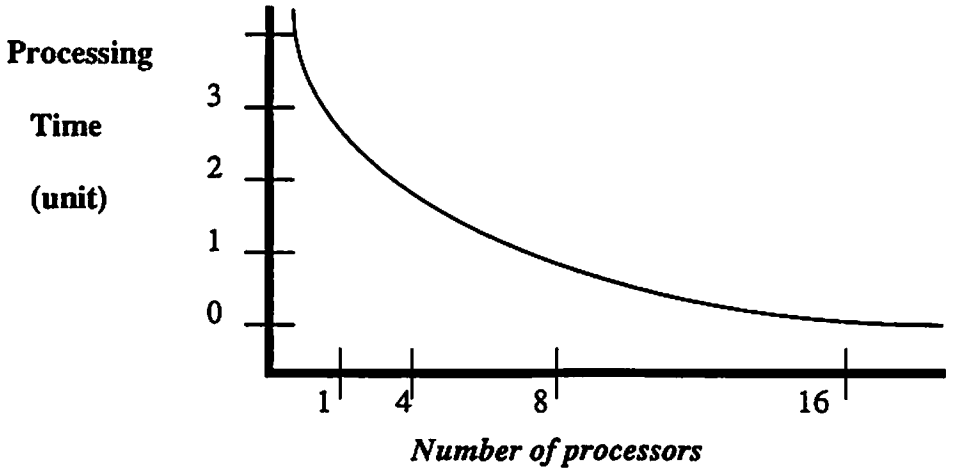


Fig. 4.1 Computation Time for Hough_Transfer

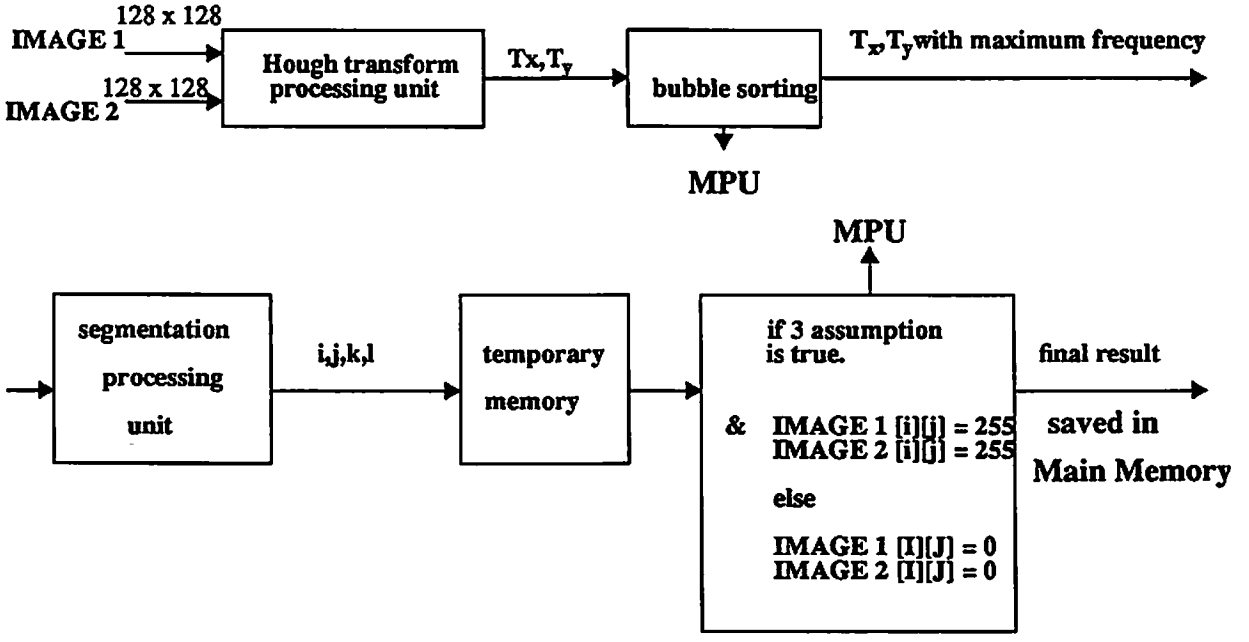


Fig.3.2.1 Block Diagram of the Basic operation relationship between the local PE and MPU

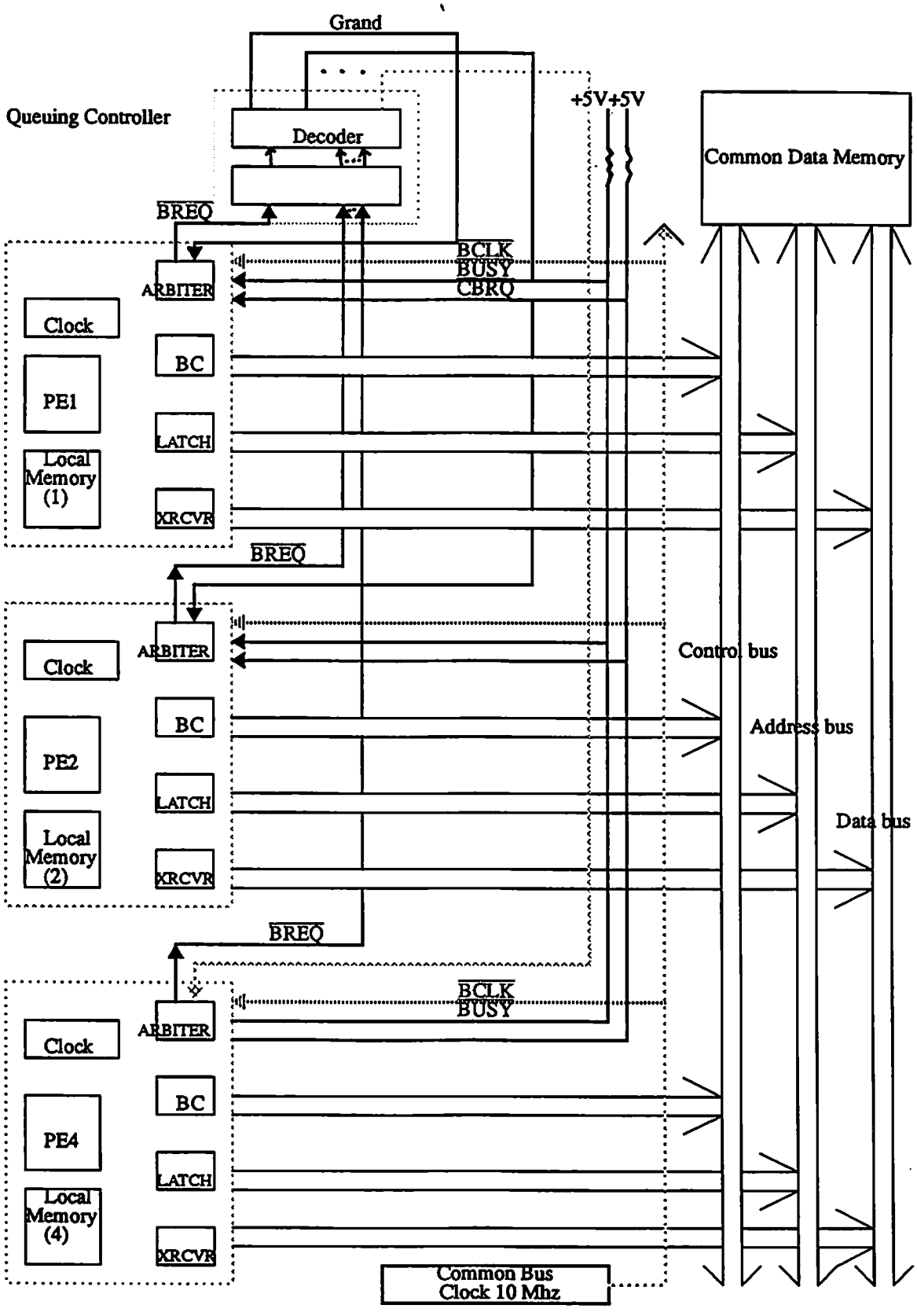


Fig.3.2.2 Configuration of Local MPU Subsystem

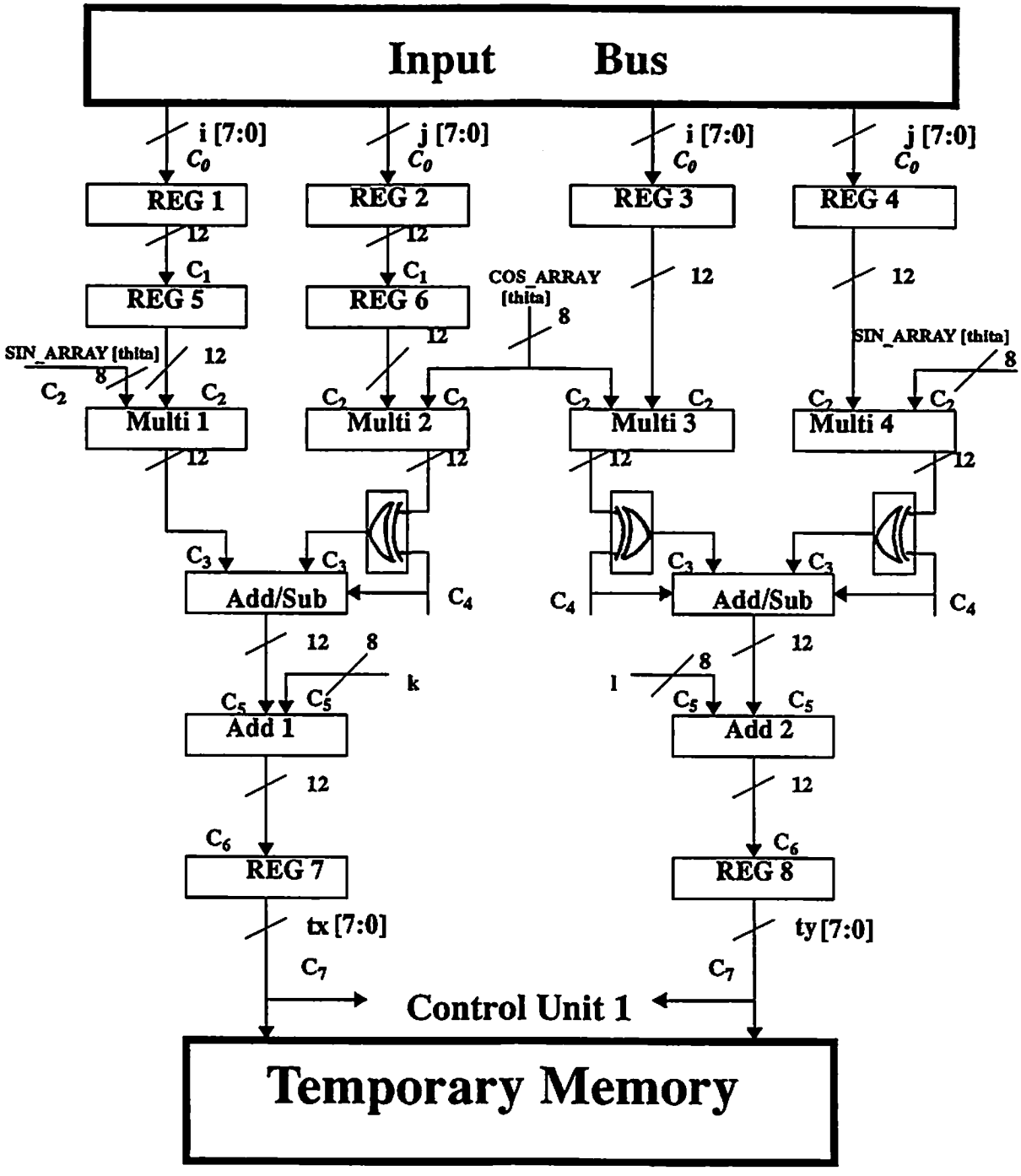


Fig.3.3 Functional block dragram of the hough_ transform processing unit

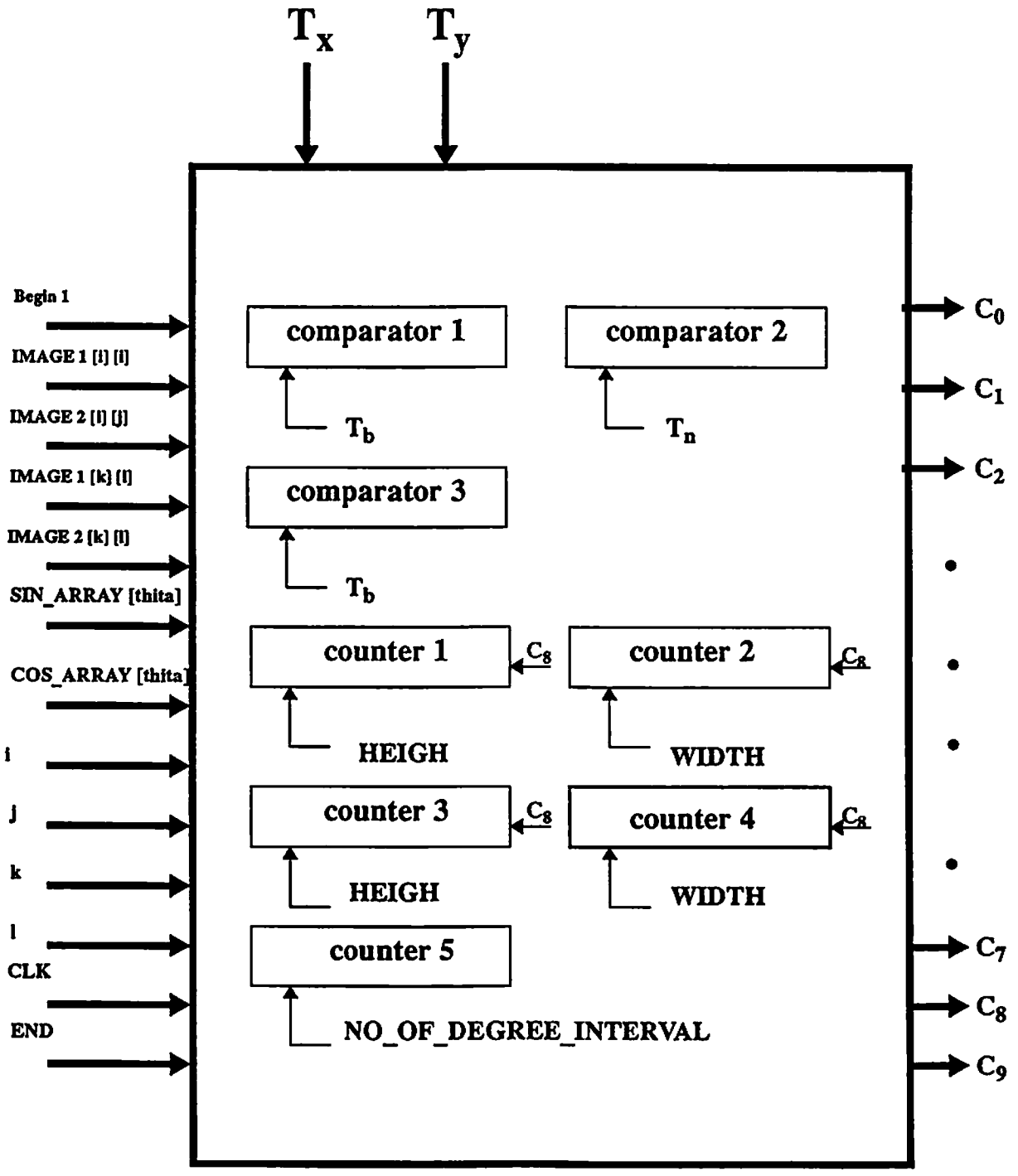


Fig. 3.4 Control Unit 1

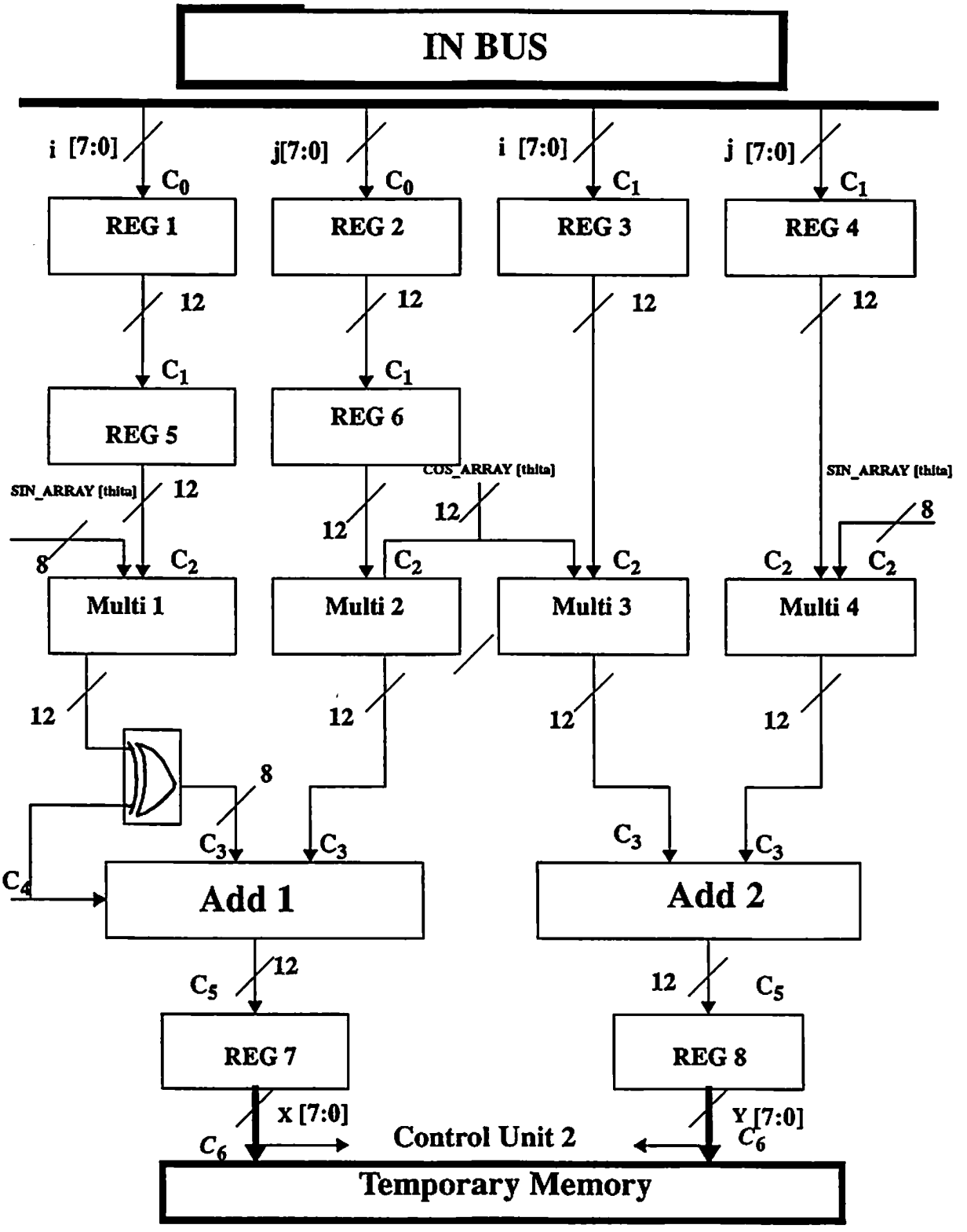


Fig.3.5 Functional block diagram of the image segmentation processing unit

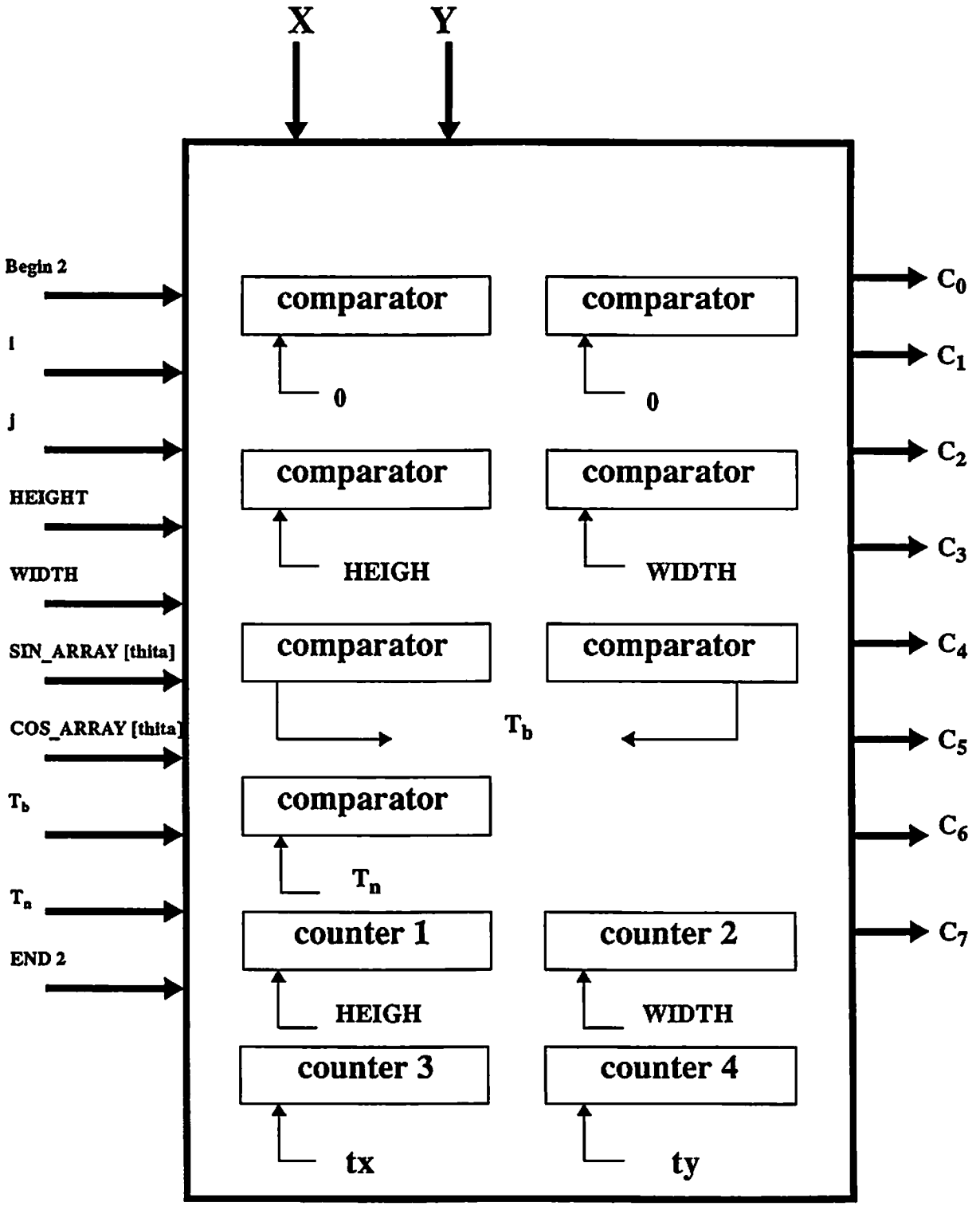


Fig. 3.6 Control Unit 2

EE 599

Special Topics

Digital Image Processing on VLSI

Summer 1991

Term Project

Transform Image Coding by the DCT, the Lloyd-Max Quantization, the Huffman coding, and the IDCT

Abstract -The motivation to do the transform image coding is to reduce the number of bits used to transmit the image. To get this goal, we can use the DCT, the Lloyd-Max Quantization, the Huffman Coding, and the IDCT. In this project, I use software to simulate the algorithms mentioned above. Then, I analyze the result image and compare the relationship between the compression ratio and the image quality. And, my project partner (Chi-Fang Ma) does the hardware mapping to the DCT by using the systolic array.

Instructor : Dr. Bing Sheu
Student : Chun-Kun Chen
(888-04-2893)

August 12, 1991

I. INTRODUCTION :

In transform image coding, an image is transformed to a frequency domain, and the transform coefficients are then coded. We can find that a large amount of the energy is concentrated in a small part of the coefficients(low frequency parts). This phenomenon is called energy compaction property. With this property, we can discard many coefficients that the energy is not concentrated in. Therefore, we can code images in fewer bits(could be below 1 bit / pixel), and still get a clear image.

In this project, I simulate the transform image coding by using the DCT, the Lloyd-Max Quantization, the Huffman Coding, and the IDCT. And, I calculate the SNR(signal to noise ratio), NMSE(nornalized mean square error), the number of bits used to represent the gray level in an image, and the compression ratio. Moreover, I analyze the result image and compare the relationship between the compression ratio and the image quality.

My project partner(Chi-Fang Ma) does the hardware mapping to DCT by using the systolic array. In this report, I will only discuss the software part of this project

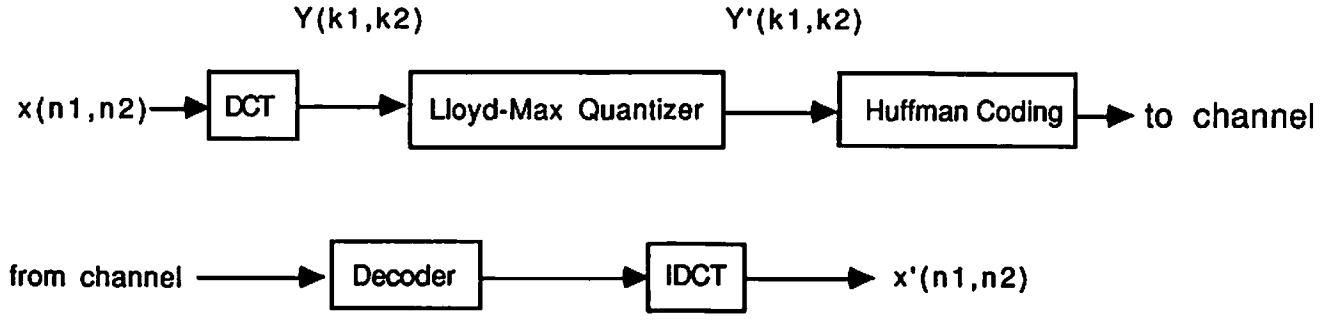


Figure 1. The block diagram in transform image coding.

II. ALGORITHM :

1. Discrete cosine transform (DCT) & Inverse Discrete consine transform (IDCT) :

DCT is one kind of transforms that have energy compaction property. A 8X8 (N=8) window is used in this project, and the DCT and IDCT are defined by

$$DCT(k1,k2) = \sum_{n1=0}^{N-1} \sum_{n2=0}^{N-1} x(n1,n2)\, a(k1)\cos\frac{\pi(2*n1+1)k1}{2*N}\; a(k2)\cos\frac{\pi(2*n2+1)k2}{2*N}$$

$$\text{where} \quad a(0) = (1/N)^{0.5}, \quad a(ki) = (2/N)^{0.5} \quad \text{for } 1 \le ki \le N-1 \;, i=1,\ 2$$

$$\text{I-DCT}(n1,n2) = \sum_{n1=0}^{N-1} \sum_{n2=0}^{N-1} \text{DCT}(k1,k2)\, a(k1)\cos\frac{\pi(2*n1+1)k1}{2*N}\; a(k2)\cos\frac{\pi(2*n2+1)k2}{2*N}$$

where $a(0) = (1/N)^{0.5}$, $a(k_i) = (2/N)^{0.5}$ for $1 \le n_i \le N-1$, $i=1, 2$

2. Lloyd-Max Quantization :

d_k : decision boundary , r_k : reconstruction level

$$d_k = \frac{r_k + r_{k-1}}{2}$$

$$r_k = \frac{\int_{d_k}^{d_{k+1}} f_0 * P_f(f_0)\, df_0}{\int_{d_k}^{d_{k+1}} P_f(f_0)\, df_0}$$

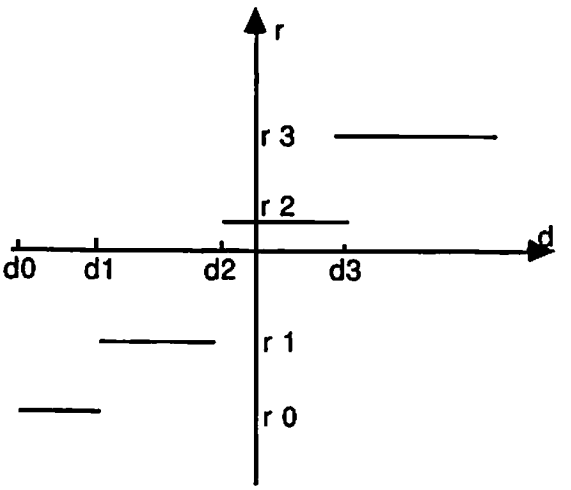


Figure 2: Example of Lloyd-Max quantization.
The number of reconstruction level is 4, and
the probability distribution is random.

The reason I choose Lloyd-Max Qnantization instead of uniform quantization is that the uniform quantization may not be optimal even though it is simple and easy to implement. The Lloyd-Max Quantization is better than the uniform quantization due to the consideration of the probability of the distribution. Therefore, the Lloyd-Max Quantization can have higher resolution by using the same number of the quantization levels.

3. Huffman Coding :

Procedure for coding n level messages :

1. Arrange the message probability in descending order (each one is a node).
2. Combine the two lowest probability, and set the combination to a new node(# n+1).

Assign a "0" to the upper member, and assign a "1" to the lower member.

Repeat 2. until the combination is equal to 1.0.

3. Trace the path from each message node to the largest numbered node(probability combination = 1.0).

For each message, write the "0"-"1" sequence thus obtained from right to left.

The reason I choose Huffman coding instead of uniform length coding is that we can get the lower average bit rate by using the Huffman Coding. And, I will compare the bit rate in the output analysis.

III. OUTPUT ANALYSIS :

1. Definition :

a. $$\text{SNR in dB} = 10 \log_{10} \frac{\text{Var } |f(n1,n2)|}{\text{Var } |f(n1,n2) - p(n1,n2)|}$$

b. $$\text{MSE / pixel} = \frac{1}{NxN} \sum_{n1=0}^{N-1} \sum_{n2=0}^{N-1} (f(n1,n2) - p(n1,n2))^2$$

f(n1,n2) : original image MSE : mean square error
p(n1,n2) : processed image

c. $$\text{Compression Ratio} = \frac{256x256x8 \quad \text{(bit)}}{\text{total bits used in transform image coding}}$$

2. output information :

Table 1. Comparison of the girl (lenna) images coded by different bits / pixel :

	(without overhead) bits/pixel	(without overhead) comp. ratio	(with overhead) bits/pixel	(with overhead) comp. ratio	SNR	MSE / pixel	coeff.
image 1	0.33	24.06	0.36	22.22	25.42	220.04	23%
image 2	0.40	19.99	0.43	18.65	27.02	187.50	28%
image 3	0.53	15.21	0.56	14.35	29.49	146.61	36%
image 4	0.80	10.02	0.84	9.56	32.11	112.89	59%
image 5	1.51	5.30	1.57	5.09	41.27	45.27	100%
image 6	2.83	2.83	3.05	2.62	52.02	17.41	100%

Table 2. Comparison of the pepper images coded by different bits / pixel :

	(without overhead) bits/pixel	(without overhead) comp. ratio	(with overhead) bits/pixel	(with overhead) comp. ratio	SNR	MSE / pixel	coeff.
image 1	0.38	20.91	0.41	19.45	31.081	253.07	26%
image 2	0.46	17.31	0.49	16.23	33.25	203.63	31%
image 3	0.56	14.40	0.59	13.49	35.67	159.96	36%
image 4	0.77	10.33	0.82	9.81	38.499	120.61	50%
image 5	1.55	5.16	1.62	4.94	47.203	50.44	100%
image 6	2.89	2.76	3.13	2.55	61.95	11.77	100%

3. Compression ratio :

For the same image, the quality of the image definitely decreases as the compression ratio increases. According to my simulation, we can still get a good image when the compression ratio is up to 10 (or the bit rate is about 0.8 bits / pixel) . On the other hand, the image becomes quite poor due to the obvious blocking effect while the compression ratio is over 15 (or the bit rate is lower than 0.5 bits / pixel). Generally , we can get a good image when the SNR is about 38 dB.

4. Degradation of the image :

a. Loss of spatial resolution : In the DCT, we discard the transform coefficients that are typically high frequency components. Therefore, the detail of the image is not quite clear.

b. Quantization induced degradation : Due to the quantization, we can see the increase of the graininess in the image.

c. Blocking effect : Due to the subimage-by subimage coding, the pixels at the subimage boundaries may have artificial intensity discontinuities. This problem is more serious when the bit rate decreases. For example, this phenomenon is quite obvious in the images whose bit rates are about 0.5 bit / pixel.

5. Huffman coding vs uniform length coding :

From the result, we can find that the Huffman Coding can reduce about 15% to 30% codes compared with the uniform length coding. Moreover, we can observe that for either the Huffman Coding or the uniform length coding, more percentage of the codes can be reduced as the bit rate increases.

6. Bit overhead for coding :

In any kind of coding, we have to send the extra bits to transmit the mean value (8 bits / per mean)

and the variance value(5 bits / per variance). Furthermore, we have to specify the different levels using 8 bits / per level. Therefore, the overhead has to be considered.

IV. DISCUSSION :

1. The degradation of the image due to the loss of the spatial resolution and the quantization is not easy to overcome. This is a tradeoff between bit reduction and the image quality. On the other hand, the blocking effect can be reduced by using overlap method or using the low-pass filter at the subimage boundary even though these result in the computation and processing overhead.

2. Window size problem : Using subimage-by-subimage coding can reduce the storage and computation requirement. Usually, if the subimage is smaller, the computation requirement will be more reduced. But we cannot choose too small subimage due to that the correlation among neighboring subimage will increase while the subimage size decreases. Therefore, we usually use 8x8 or 16x16 subimage in the transform image coding by the tradeoff.

V. CONCLUSION :

By transform image coding, we can reduce the number of bits used to transmit the image. And, if we do not care the quality of the image very much, we can use 0.3 bit / pixel to transmit the image. Then, the compression ratio could be up to 24. Even though we emphasize the image quality, we can still use about 0.8 bits / pixel (compression ratio is about 10) to transmit the image. Therefore, the transform image coding is quite important and useful especially in transmitting a large amout of image data.

VI. REFERENCE :

[1] J. S. Lim, "Two-Dimensional Signal and Image Processing," Prentice-Hall, 1990.

[2] Anil K. Jain, "Fundamentals of Digital Image Processing," Prentice-Hall, 1989.

[3] Rafael C. Gonzalez, Paul Wintz, "Digital Image Processing," 2nd Ed., Addison-Wesley, 1987.

Original girl (lenna) image of 256 x 256

image 1. Reconstructed image with
0.33 bits/pixel, SNR = 25.42 dB
MSE/pixel = 220.04

image 2. Reconstructed image with
0.40 bits/pixel, SNR = 27.02
MSE/pixel = 187.50

image 3. Reconstructed image with
0.53 bits/pixel, SNR = 29.49 dB
MSE/pixel = 146.61

image 4. Reconstructed image with
0.80 bits/pixel, SNR = 32.11 dB
MSE/pixel = 112.89

image 5. Reconstructed image with
1.51 bits/pixel, SNR = 41.27 dB
MSE/pixel = 45.27

image 6. Reconstructed image with
2.83 bits/pixel, SNR = 50.02 dB
MSE/pixel = 17.41

original image of 256×256

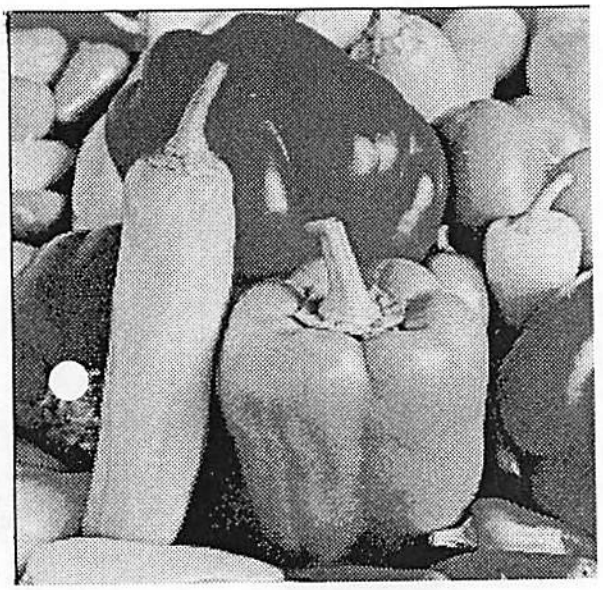


image 7. Reconstructed image with
0.38 bits/pixel, SNR = 31.05 dB
MSE/pixel = 253.07

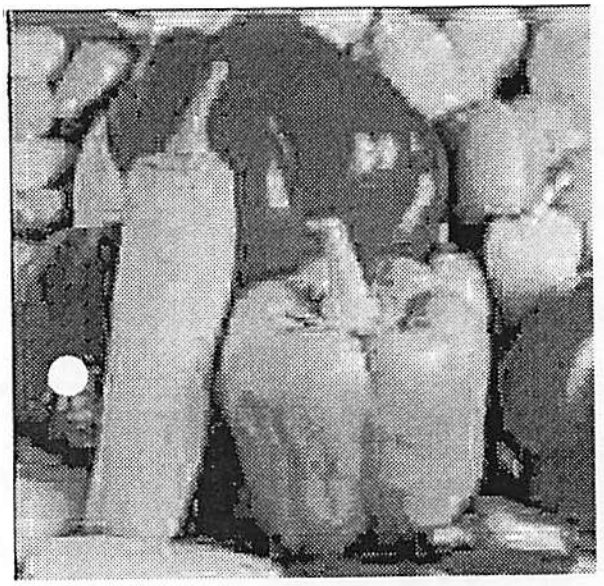


image 8. Reconstructed image with
0.46 bits/pixel, SNR = 33.25 dB
MSE/pixel = 203.63

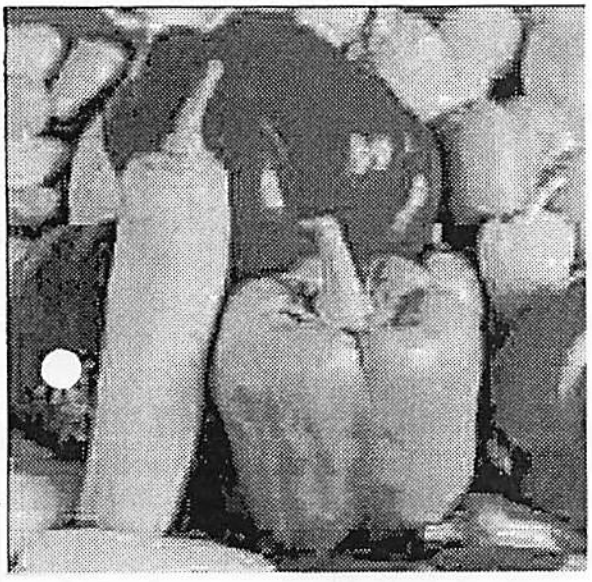


image 9. Reconstructed image with
0.56 bits/pixel, SNR = 35.67 dB
MSE/pixel = 159.96

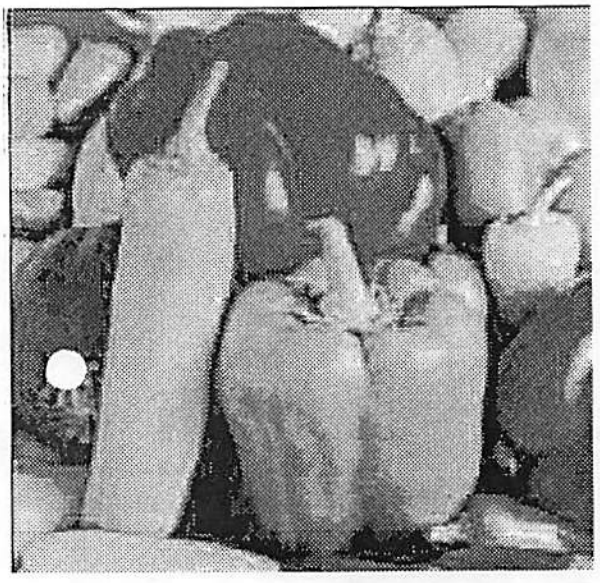


image 10. Reconstructed image with
0.77 bits/pixel, SNR = 38.49 dB
MSE/pixel = 120.6

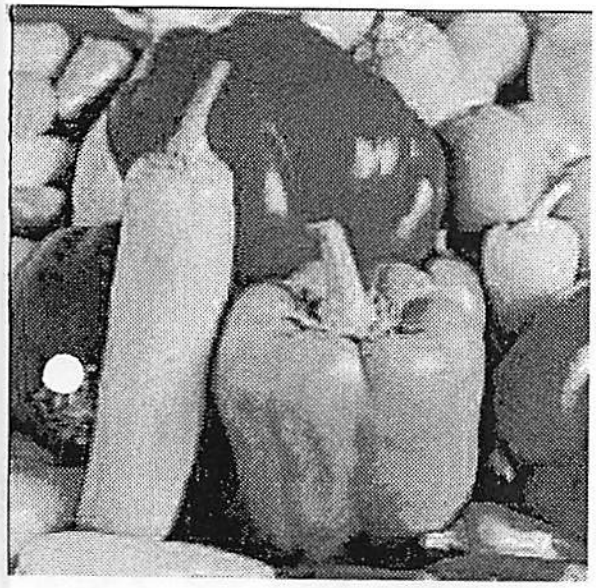


image 11. Reconstructed image with
1.55 bits/pixel, SNR = 47.23 dB
MSE/pixel = 50.44

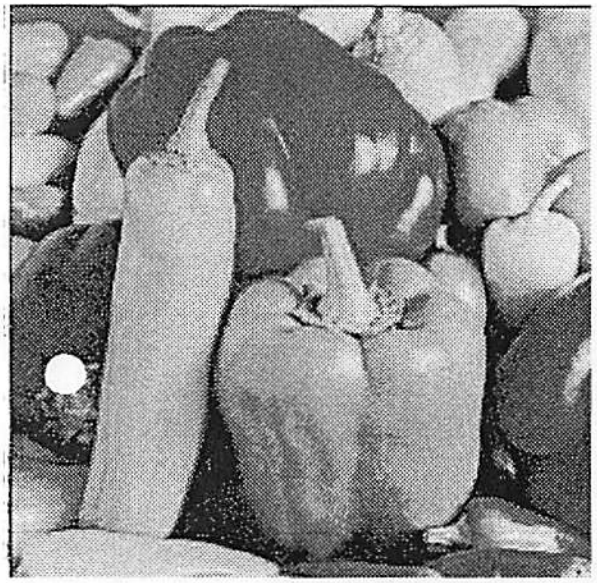
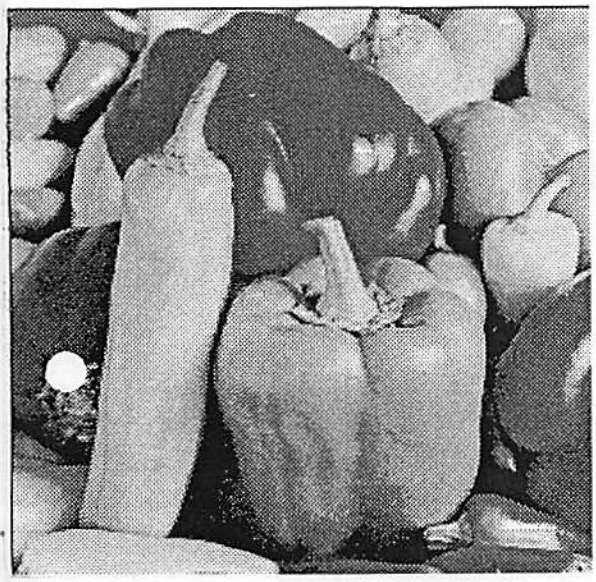


image 12. reconstructed image with
2.89 bits/pixel, SNR = 61.95 dB
MSE/pixel = 11.77



```
input image:pepper.256
output image:pepper.1

 6.1  4.3  3.3  2.6  2.1  1.6  1.2  0.7
 4.0  3.0  2.4  1.7  1.2  0.8  0.6  0.4
 2.9  2.2  1.8  1.3  0.9  0.5  0.3  0.2
 2.4  1.7  1.2  0.9  0.5  0.3  0.2  0.1
 2.1  1.3  0.9  0.7  0.4  0.1  0.0  0.0
 1.8  0.8  0.7  0.5  0.1  0.0  0.0  0.0
 1.3  0.4  0.4  0.2  0.0  0.0  0.0  0.0
 0.7  0.3  0.4  0.2  0.0  0.0  0.0  0.0


***** bit allocation *****
   6    4    3    3    2    2    1    1
   4    3    2    2    1    1    1    0
   3    2    2    1    1    1    0    0
   2    2    1    1    0    0    0    0
   2    1    1    1    0    0    0    0
   2    1    1    0    0    0    0    0
   1    0    0    0    0    0    0    0
   1    0    0    0    0    0    0    0

 MSE/pixel= 120.611404
 SNR = 38.49 dB

 HUFFMAN CODING (without overhead) :
 bit rate(huffman coding): 0.77 bit/pixel
 compression ratio : 10.33

 HUFFMAN CODING (with overhead) :
 bit rate(huffman coding): 0.82 bit/pixel
 compression ratio : 9.81


 UNIFORM LENGTH CODING(without overhead) :
 bit rate(uniform length): 0.98 bit/pixel
 compression ratio : 8.13

 UNIFORN LENGTH CODING(with overhead):
 bit rate(uniform length): 1.03 bit/pixel
 compression ratio : 7.80
```

```
Input image:pepper.256
output image:pepper.2

 7.1  5.3  4.3  3.6  3.1  2.6  2.2  1.7
 5.0  4.0  3.4  2.7  2.2  1.8  1.6  1.4
 3.9  3.2  2.8  2.3  1.9  1.5  1.3  1.2
 3.4  2.7  2.2  1.9  1.5  1.3  1.2  1.1
 3.1  2.3  1.9  1.7  1.4  1.1  0.9  0.8
 2.8  1.8  1.7  1.5  1.1  0.9  0.9  0.7
 2.3  1.4  1.4  1.2  1.0  0.9  0.7  0.6
 1.7  1.3  1.4  1.2  1.0  0.8  0.7  0.8


***** bit allocation *****
   7    5    4    4    3    3    2    2
   5    4    3    3    2    2    2    1
   4    3    3    2    2    2    1    1
   3    3    2    2    1    1    1    1
   3    2    2    2    1    1    1    1
   3    2    2    1    1    1    1    1
   2    1    1    1    1    1    1    1
   2    1    1    1    1    1    1    1


 MSE/pixel= 50.442856
 SNR = 47.23 dB

 HUFFMAN CODING (without overhead) :
 bit rate(huffman coding): 1.55 bit/pixel
 compression ratio : 5.16

 HUFFMAN CODING (with overhead) :
 bit rate(huffman coding): 1.62 bit/pixel
 compression ratio : 4.94


 UNIFORM LENGTH CODING(without overhead) :
 bit rate(uniform length): 1.98 bit/pixel
 compression ratio : 4.03

 UNIFORN LENGTH CODING(with overhead):
 bit rate(uniform length): 2.05 bit/pixel
 compression ratio : 3.90
```

```
Input image:pepper.256
output image:pepper.4

 9.1  7.3  6.3  5.6  5.1  4.6  4.2  3.7
 7.0  6.0  5.4  4.7  4.2  3.8  3.6  3.4
 5.9  5.2  4.8  4.3  3.9  3.5  3.3  3.2
 5.4  4.7  4.2  3.9  3.5  3.3  3.2  3.1
 5.1  4.3  3.9  3.7  3.4  3.1  2.9  2.8
 4.8  3.8  3.7  3.5  3.1  2.9  2.9  2.7
 4.3  3.4  3.4  3.2  3.0  2.9  2.7  2.6
 3.7  3.3  3.4  3.2  3.0  2.8  2.7  2.8


***** bit allocation *****
   9    7    6    6    5    5    4    4
   7    6    5    5    4    4    4    3
   6    5    5    4    4    4    3    3
   5    5    4    4    3    3    3    3
   5    4    4    4    3    3    3    3
   5    4    4    3    3    3    3    3
   4    3    3    3    3    3    3    3
   4    3    3    3    3    3    3    3


 MSE/pixel= 11.771545
 SNR = 61.95 dB


 HUFFMAN CODING (without overhead) :
 bit rate(huffman coding): 2.89 bit/pixel
 compression ratio : 2.76

 HUFFMAN CODING (with overhead) :
 bit rate(huffman coding): 3.13 bit/pixel
 compression ratio : 2.55



 UNIFORM LENGTH CODING(without overhead) :
 bit rate(uniform length): 3.98 bit/pixel
 compression ratio : 2.01

 UNIFORN LENGTH CODING(with overhead):
 bit rate(uniform length): 4.22 bit/pixel
 compression ratio : 1.89
```

```
#include <math.h>
#include <stdio.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>

main()
{
    int ifd, ofd;
    float bit_ave;
    char in_image[80],out_image[80];
    char bline[65536];
    int ip[65536], iq[65536];
    float coeff[8][8],dct[32][32][8][8],mean[8][8],var[8][8];
    int i,j,k,l,m,n,bit,p[32][32][8][8];
    float avar,b[8][8],amean,nor[32][32][8][8],idct[32][32][8][8];
    int test,v[32][32],level,index,s,mm,nn;
    float nums[1025],value2[1025],min,sort[32][32],value1[1025],t[1025],r1[1025];
    float number,density,goal,stp,d,x,y;
    int qz_pseudo[1025],jj,nr[2050],rpt[2050],rbit[2050],nb[2050];
    float sum,r2[1025],qntz[32][32][8][8];
    long code[1025],mse;
    float ngoal,tvar1,tvar2,yy,zz,tmean,total,snr;
    int totlbt1,totlbt2,mean2,tv[65536],tot;

    printf("Input image:");scanf("%s",in_image);
    printf("output image:");scanf("%s",out_image);
    printf("DC term(bit ave):");scanf("%f",&bit_ave);

    if ((ifd=open(in_image,O_RDONLY)) == -1){
       printf("error! could not open input file.\n");
       exit(1);
     }
     if ((ofd=open(out_image, O_CREAT|O_TRUNC|O_WRONLY)) == -1) {
       printf("Error ! Could not open output file.\n");
       exit(1);
     }
    read(ifd,bline,65536);
    for (i=0;i<65536;i++)
      ip[i]=bline[i] & 0xff;

    for(m=0;m<32;m++)
      for(n=0;n<32;n++)
        for(k=0;k<8;k++)
          for(l=0;l<8;l++)
            p[m][n][k][l]=ip[256*(8*m+k)+8*n+l];

ngoal=0;totlbt1=0; totlbt2=0;

/***** set the coefficient matrix ****/
    for(n=0;n<8;n++)
      coeff[0][n]=0.3535533906;
    for(k=1;k<8;k++)
      for(n=0;n<8;n++)
        coeff[k][n]=0.5*cos(k*(2*n+1)*3.14159/16);

/******** DCT transform *********/
    for(m=0;m<32;m++)
      for(n=0;n<32;n++)
        for(k=0;k<8;k++)
          for(l=0;l<8;l++)
            {
             dct[m][n][k][l]=0;
             for(i=0;i<8;i++)
               for(j=0;j<8;j++)
                 dct[m][n][k][l]=dct[m][n][k][l]+coeff[k][i]*(float)(p[m][n][i][j])*coeff[
l][j];
            }

/***** calculate the mean and the variance *****/
    for(k=0;k<8;k++)
      for(l=0;l<8;l++)
        {
          mean[k][l]=0;
          for(m=0;m<32;m++)
            for(n=0;n<32;n++)
              mean[k][l]=mean[k][l]+dct[m][n][k][l];
          mean[k][l]=mean[k][l]/1024;
        }

    for(k=0;k<8;k++)
      for(l=0;l<8;l++)
       {
         var[k][l]=0;
         for(m=0;m<32;m++)
           for(n=0;n<32;n++)
             {
               avar=dct[m][n][k][l]-mean[k][l];
               var[k][l]=var[k][l]+avar*avar;
             }
         var[k][l]=var[k][l]/1024.;
       }
/******** bit allocation ********/
    avar=0.;
    for(k=0;k<8;k++)
      for(l=0;l<8;l++)
        avar=avar+0.5*(log(var[k][l])/log(2.));

    avar=avar/64.;
    for(k=0;k<8;k++)
      for(l=0;l<8;l++)
       {
         b[k][l]=bit_ave+0.5*(log(var[k][l])/log(2.))-avar;
         if(b[k][l]<0) b[k][l]=0;
         if(b[k][l]>13) b[k][l]=13;
       }

    for(k=0;k<8;k++)
      {
        for(l=0;l<8;l++)
          printf("%4.1f ",b[k][l]);
        printf("\n");
      }
    printf("\n\n\n");
    printf("***** bit allocation *****\n");
    for(k=0;k<8;k++)
      {
        for(l=0;l<8;l++)
          printf("%4d ",(int)(b[k][l]+0.5));
        printf("\n");
      }

    printf("\n\n");

/*********** normalization **********/

    for(k=0;k<8;k++)
      for(l=0;l<8;l++)
```

```
        {
         avar=var[k][l];
         amean=mean[k][l];
         for(m=0;m<32;m++)
           for(n=0;n<32;n++)
             nor[m][n][k][l]=(dct[m][n][k][l]-amean)/avar;
        }

    for(k=0;k<8;k++)
      for(l=0;l<8;l++)
  {
    printf("(k,l)=(%ld,%ld)",k,l);

/************ sorting **************/

        for(m=0;m<32;m++)
           for(n=0;n<32;n++)
               sort[m][n]=nor[m][n][k][l];
        for (i=1;i<1025;i++)
           nums[i]=0;
        i=0;test=1;
        while(test!=0)
          {
           min=65536;test=0;
           for(m=0;m<32;m++)
             for(n=0;n<32;n++)
               if(sort[m][n]<min)
                 {test=1; mm=m; nn=n; min=sort[m][n];}
           if(test==1)
             {
              i++;
              value2[i]=sort[mm][nn];
              if(value2[i]==value2[i-1])
                i=i-1;
              nums[i]=nums[i]+1;
              v[mm][nn]=i;
              sort[mm][nn]=65537;
             }
          }

/****************** Lloyd-Max quantization *************/

        value2[0]=value2[1];value2[i+1]=value2[i];
        goal=1;
        for(j=1;j<=(int)(b[k][l]+.5);j++)
           goal=goal*2.;
        stp=(value2[i]-value2[1])/goal;
        for(j=0;j<=(int)goal;j++)
           t[j+1]=value2[1]+stp*(float)j;
        for(j=1;j<=i+1;j++)
           value1[j]=(value2[j]+value2[j-1])/2.;
        j=i;t[(int)goal+1]=value1[i+1];
        ngoal=ngoal+goal;
  /**** quantizing ****/
        test=1;
        while(test!=0)
          {
           test=0;y=t[1];i=s=1;
           density=0.;number=0;
           while(s!=(int)(goal)+1)
             {
              x=y;i++;
              if(value1[i]<t[s+1])
                {
                 y=value1[i];
                 d=nums[i-1]/(value1[i]-value1[i-1]);
                 number=number+d*(y*y-x*x)/2.;
                 density=density+d*(y-x);
                }
              else
                {
                 y=t[s+1];
                 d=nums[i-1]/(value1[i]-value1[i-1]);
                 number=number+d*(y*y-x*x)/2.;
                 density=density+d*(y-x);
                 i=i-1;
                 rl[s]=number/density;
                 number=0;density=0.;
                 if((int)(100000*r2[s])!=(int)(100000*rl[s]))
                   {
                     test=1;
                     r2[s]=rl[s];
                   }
                 s=s+1;
                }
             }

            for(s=2;s<=(int)goal;s++)t[s]=(rl[s]+rl[s-1])/2;
          }
        s=1;
        for(i=1;i<=j;i++)
          {
           if(value2[i]>t[s+1])
             s=s+1;
           qz_pseudo[i]=s;
          }

        for(m=0;m<2050;m++) nr[m]=0;
        for(m=0;m<32;m++)
          for(n=0;n<32;n++)
            {
             index=v[m][n];
             level=qz_pseudo[index];
             nr[level]++;
             qntz[m][n][k][l]=rl[level];
            }

        avar=var[k][l];amean=mean[k][l];
        for(m=0;m<32;m++)
          for(n=0;n<32;n++)
            dct[m][n][k][l]=qntz[m][n][k][l]*avar+amean;
/***uniform length ***/
  totlbt2=totlbt2+1024*(int)(b[k][l]+0.5);
/************** Huffman encoding ********************/
        for(m=1;m<2050;m++) rpt[m]=0;
        jj=(int)goal;
        while(nr[jj]<1024)
          {
            min=1024;
            for(m=1;m<=jj;m++)
              if((nr[m]<min)&&(rpt[m]==0)) {i=m;min=nr[m];}
               rpt[i]=jj+1;min=1024;
            for(m=1;m<=jj;m++)
              if((nr[m]<min)&&(rpt[m]==0)) {j=m;min=nr[m];}
            jj=jj+1;rpt[i]=rpt[j]=jj;

            if((i<=(int)goal) && (j<=(int)goal))
               {rbit[i]=1;rbit[j]=0;}
```

```
          else
            {
              if(i<j)
                 {rbit[i]=0;rbit[j]=1;}
               else
                  {rbit[i]=1;rbit[j]=0;}
            }
          nr[jj]=nr[i]+nr[j];

        }

      printf("\n");
      for(jj=1;jj<=(int)goal;jj++)
        {j=jj;d=0.125;nb[jj]=0;code[jj]=0;
         while(rpt[j]!=0)
           {d=d*8;nb[jj]=nb[jj]+1;
            code[jj]=code[jj]+rbit[j]*(int)d;
            i=rpt[j];j=i;
           }
         totlbt1=totlbt1+nr[jj]*nb[jj];

         printf("%11d %10.6f %4d/1024 %o\n",jj,r1[jj],nr[jj],code[jj]);
        }

    }
/********** I D C T ******************/

   for(m=0;m<32;m++)
     for(n=0;n<32;n++)
       for(i=0;i<8;i++)
         for(j=0;j<8;j++)
           {
            idct[m][n][i][j]=0;
            for(k=0;k<8;k++)
              for(l=0;l<8;l++)
                idct[m][n][i][j]=idct[m][n][i][j]+coeff[k][i]*dct[m][n][k][l]*coeff[l][
j];
           }

/***************** output display ***************/
   k=-1;mse=0;mean2=0;
   for(m=0;m<32;m++)
     for(i=0;i<8;i++)
       for(n=0;n<32;n++)
         for(j=0;j<8;j++)
           {
            k++;
            iq[k]=(int)idct[m][n][i][j];
            if(iq[k]<0) iq[k]=0;
            if(iq[k]>255) iq[k]=255;
            jj=ip[k]-iq[k];
            mse=mse+jj*jj;
            tv[k]=jj;
            mean2=mean2+ip[k];
           }
      tvar1=0. ;tot=0;
      tmean=(float)(mean2)/65536.;
      for(k=0;k<65536;k++)
       {
         yy=(float)(ip[k])-tmean;
         tvar1=tvar1+yy*yy;
         tot=tot+tv[k];
       }
      total=(float)(tot)/65536.;tvar2=0;
      for(k=0;k<65536;k++)
        {
          zz=(float)(tv[k])-total;
          tvar2=tvar2+zz*zz;
        }
      snr=10*log(tvar1/tvar2);


   printf("\n MSE/pixel= %f\n",(float)mse/65536.);
   printf("\n SNR =%4.2f dB",snr);

   printf("\n\n\n HUFFMAN CODING (without overhead) : ");
   printf("\n bit rate(huffman coding): %4.2f bit/pixel ",(float)totlbt1/65536.0);
   printf("\n compression ratio : %3.2f",65536.0*8.0/(float)totlbt1);
   totlbt1=totlbt1+((int)ngoal+64)*8+64*5;
   printf("\n\n HUFFMAN CODING (with overhead) :");
   printf("\n bit rate(huffman coding): %4.2f bit/pixel ",(float)totlbt1/65536.0);
   printf("\n compression ratio : %3.2f",65536.0*8.0/(float)totlbt1);
   printf("\n\n\n\n UNIFORM LENGTH CODING(without overhead) :");

   printf("\n bit rate(uniform length): %4.2f bit/pixel ",(float)totlbt2/65536.0);
   printf("\n compression ratio : %3.2f",65536.0*8.0/(float)totlbt2);
   totlbt2=totlbt2+((int)ngoal+64)*8+64*5;
   printf("\n\n UNIFORN LENGTH CODING(with overhead):");
   printf("\n bit rate(uniform length): %4.2f bit/pixel ",(float)totlbt2/65536.0);
   printf("\n compression ratio : %3.2f \n",65536.0*8.0/(float)totlbt2);

   for (i=0; i<65536; i++)
      bline[i] = (char) iq[i];
   write(ofd, bline,65536);

   close(ifd);
   close(ofd);

}
```

```
Input image:lenna.256
output image:lenna.0.25

 4.9  3.4  2.3  1.6  1.0  0.7  0.3  0.0
 2.5  2.3  1.7  1.1  0.7  0.4  0.1  0.0
 1.4  1.3  1.2  0.9  0.4  0.2  0.0  0.0
 0.4  0.6  0.8  0.5  0.4  0.1  0.0  0.0
 0.0  0.1  0.2  0.2  0.1  0.1  0.0  0.0
 0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0
 0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0
 0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0


***** bit allocation *****
   5    3    2    2    1    1    0    0
   3    2    2    1    1    0    0    0
   1    1    1    1    0    0    0    0
   0    1    1    1    0    0    0    0
   0    0    0    0    0    0    0    0
   0    0    0    0    0    0    0    0
   0    0    0    0    0    0    0    0
   0    0    0    0    0    0    0    0
```

```
                          prob.    Huffman code
(k,l)=(0,0)
          1  -0.004415   25/1024 110
          2  -0.004136   36/1024 10111
          3  -0.003863   37/1024 11001
          4  -0.003490   38/1024 11011
          5  -0.003062   31/1024 10000
          6  -0.002747   39/1024 11010
          7  -0.002378   28/1024 1010
          8  -0.002029   58/1024 100
          9  -0.001700   50/1024 0
         10  -0.001401   37/1024 11000
         11  -0.001093   33/1024 10010
         12  -0.000733   44/1024 11100
         13  -0.000335   60/1024 110
         14   0.000009   45/1024 11111
         15   0.000326   32/1024 10011
         16   0.000677   36/1024 10110
         17   0.001003   34/1024 10100
         18   0.001339   46/1024 11110
         19   0.001743   52/1024 10
         20   0.002080   58/1024 111
         21   0.002384   47/1024 1
```

```
          22   0.002668   21/1024 111010
          23   0.002963   12/1024 1111
          24   0.003342   14/1024 1110
          25   0.003819   17/1024 101010
          26   0.004189   16/1024 100010
          27   0.004550   10/1024 1110110
          28   0.004900   16/1024 101011
          29   0.005241    9/1024 1110111
          30   0.005481   14/1024 10111
          31   0.005844   14/1024 10110
          32   0.006170   15/1024 100011
(k,l)=(0,1)
           1  -0.022219   26/1024 11010
           2  -0.012894   57/1024 1100
           3  -0.005017  195/1024 10
           4  -0.000164  524/1024 0
           5   0.004715   84/1024 1111
           6   0.010755   99/1024 1110
           7   0.019837   25/1024 110110
           8   0.029683   14/1024 110111
(k,l)=(0,2)
           1  -0.057365   26/1024 111
           2  -0.016484  165/1024 10
           3   0.000621  689/1024 0
           4   0.026200  144/1024 110
(k,l)=(0,3)
           1  -0.057589   84/1024 110
           2  -0.002300  773/1024 0
           3   0.030945  149/1024 10
           4   0.114333   18/1024 111
(k,l)=(0,4)
           1  -0.054957  167/1024 1
           2   0.010730  857/1024 0
(k,l)=(0,5)
           1  -0.129984   68/1024 1
           2   0.009333  956/1024 0
(k,l)=(0,6)
           1   0.000000 1024/1024 0
(k,l)=(0,7)
           1   0.000000 1024/1024 0
(k,l)=(1,0)
           1  -0.083821    2/1024 1111111
           2  -0.061866    6/1024 1111110
           3  -0.033228   36/1024 11110
           4  -0.014421  122/1024 110
           5  -0.001005  559/1024 0
           6   0.006606  207/1024 10
           7   0.022154   63/1024 1110
           8   0.044563   29/1024 111110
```

```
(k,l)=(1,1)
          1  -0.051986   39/1024 111
          2  -0.013806  165/1024 10
          3   0.002057  750/1024 0
          4   0.039539   70/1024 110
(k,l)=(1,2)
          1  -0.084314   25/1024 111
          2  -0.028831  148/1024 10
          3   0.001655  745/1024 0
          4   0.048830  106/1024 110
(k,l)=(1,3)
          1  -0.007545  932/1024 0
          2   0.076363   92/1024 1
(k,l)=(1,4)
          1  -0.007109  975/1024 0
          2   0.140806   49/1024 1
(k,l)=(1,5)
          1   0.000000 1024/1024 0
(k,l)=(1,6)
          1  -0.000000 1024/1024 0
(k,l)=(1,7)
          1  -0.000000 1024/1024 0
(k,l)=(2,0)
          1  -0.005174  962/1024 0
          2   0.080341   62/1024 1
(k,l)=(2,1)
          1  -0.066171   91/1024 1
          2   0.006487  933/1024 0
(k,l)=(2,2)
          1  -0.009284  867/1024 0
          2   0.051316  157/1024 1
(k,l)=(2,3)
          1  -0.007086  963/1024 0
          2   0.112119   61/1024 1
(k,l)=(2,4)
          1  -0.000000 1024/1024 0
(k,l)=(2,5)
          1   0.000000 1024/1024 0
(k,l)=(2,6)
          1  -0.000000 1024/1024 0
(k,l)=(2,7)
          1  -0.000000 1024/1024 0
(k,l)=(3,0)
          1  -0.000000 1024/1024 0
(k,l)=(3,1)
          1  -0.010728  931/1024 0
          2   0.107006   93/1024 1
(k,l)=(3,2)
          1  -0.008475  953/1024 0
```

```
          2   0.113010   71/1024 1
(k,l)=(3,3)
          1  -0.109887   99/1024 1
          2   0.011658  925/1024 0
(k,l)=(3,4)
          1   0.000000 1024/1024 0
(k,l)=(3,5)
          1  -0.000000 1024/1024 0
(k,l)=(3,6)
          1   0.000000 1024/1024 0
(k,l)=(3,7)
          1   0.000000 1024/1024 0
(k,l)=(4,0)
          1  -0.000000 1024/1024 0
(k,l)=(4,1)
          1  -0.000000 1024/1024 0
(k,l)=(4,2)
          1   0.000000 1024/1024 0
(k,l)=(4,3)
          1  -0.000000 1024/1024 0
(k,l)=(4,4)
          1   0.000000 1024/1024 0
(k,l)=(4,5)
          1  -0.000000 1024/1024 0
(k,l)=(4,6)
          1  -0.000000 1024/1024 0
(k,l)=(4,7)
          1  -0.000000 1024/1024 0
(k,l)=(5,0)
          1  -0.000000 1024/1024 0
(k,l)=(5,1)
          1  -0.000000 1024/1024 0
(k,l)=(5,2)
          1  -0.000000 1024/1024 0
(k,l)=(5,3)
          1  -0.000000 1024/1024 0
(k,l)=(5,4)
          1  -0.000000 1024/1024 0
(k,l)=(5,5)
          1   0.000000 1024/1024 0
(k,l)=(5,6)
          1  -0.000000 1024/1024 0
(k,l)=(5,7)
          1  -0.000000 1024/1024 0
(k,l)=(6,0)
          1   0.000000 1024/1024 0
(k,l)=(6,1)
          1   0.000000 1024/1024 0
(k,l)=(6,2)
```

```
          1   0.000000 1024/1024 0
(k,l)=(6,3)
          1   0.000000 1024/1024 0
(k,l)=(6,4)
          1   0.000000 1024/1024 0
(k,l)=(6,5)
          1  -0.000000 1024/1024 0
(k,l)=(6,6)
          1   0.000000 1024/1024 0
(k,l)=(6,7)
          1   0.000000 1024/1024 0
(k,l)=(7,0)
          1  -0.000000 1024/1024 0
(k,l)=(7,1)
          1   0.000000 1024/1024 0
(k,l)=(7,2)
          1   0.000000 1024/1024 0
(k,l)=(7,3)
          1   0.000000 1024/1024 0
(k,l)=(7,4)
          1  -0.000000 1024/1024 0
(k,l)=(7,5)
          1   0.000000 1024/1024 0
(k,l)=(7,6)
          1   0.000000 1024/1024 0
(k,l)=(7,7)
          1   0.000000 1024/1024 0

MSE/pixel = 187.498825
SNR = 27.02 dB


HUFFMAN CODING (without overhead) :
bit rate(huffman coding): 0.40 bit/pixel
compression ratio : 19.99

HUFFMAN CODING (with overhead) :
bit rate(huffman coding): 0.43 bit/pixel
compression ratio : 18.65



UNIFORM LENGTH CODING(without overhead) :
bit rate(uniform length): 0.47 bit/pixel
compression ratio : 17.07

UNIFORN LENGTH CODING(with overhead):
bit rate(uniform length): 0.50 bit/pixel
compression ratio : 16.08
```

```
Input image:lenna.256
output image:lenna.0.05

 4.7  3.2  2.1  1.4  0.8  0.5  0.1  0.0
 2.3  2.1  1.5  0.9  0.5  0.2  0.0  0.0
 1.2  1.1  1.0  0.7  0.2  0.0  0.0  0.0
 0.2  0.4  0.6  0.3  0.2  0.0  0.0  0.0
 0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0
 0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0
 0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0
 0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0


***** bit allocation *****
   5    3    2    1    1    0    0    0
   2    2    1    1    1    0    0    0
   1    1    1    1    0    0    0    0
   0    0    1    0    0    0    0    0
   0    0    0    0    0    0    0    0
   0    0    0    0    0    0    0    0
   0    0    0    0    0    0    0    0
   0    0    0    0    0    0    0    0


 MSE/pixel= 220.035645
 SNR = 25.42 dB


 HUFFMAN CODING (without overhead) :
 bit rate(huffman coding): 0.33 bit/pixel
 compression ratio : 24.06

 HUFFMAN CODING (with overhead) :
 bit rate(huffman coding): 0.36 bit/pixel
 compression ratio : 22.22


 UNIFORM LENGTH CODING(without overhead) :
 bit rate(uniform length): 0.38 bit/pixel
 compression ratio : 21.33

 UNIFORN LENGTH CODING(with overhead):
 bit rate(uniform length): 0.40 bit/pixel
 compression ratio : 19.88
```

```
Input image:lenna.256
output image:lenna.0.5

 5.2  3.6  2.6  1.8  1.2  0.9  0.5  0.2
 2.8  2.5  1.9  1.3  1.0  0.7  0.3  0.2
 1.6  1.6  1.4  1.2  0.7  0.5  0.2  0.0
 0.6  0.9  1.0  0.8  0.7  0.3  0.3  0.0
 0.2  0.4  0.4  0.4  0.3  0.3  0.0  0.0
 0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0
 0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0
 0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0


***** bit allocation *****
   5    4    3    2    1    1    1    0
   3    3    2    1    1    1    0    0
   2    2    1    1    1    0    0    0
   1    1    1    1    1    0    0    0
   0    0    0    0    0    0    0    0
   0    0    0    0    0    0    0    0
   0    0    0    0    0    0    0    0
   0    0    0    0    0    0    0    0


 MSE/pixel= 146.614746
 SNR = 29.49 dB

 HUFFMAN CODING (without overhead) :
 bit rate(huffman coding): 0.53 bit/pixel
 compression ratio : 15.21

 HUFFMAN CODING (with overhead) :
 bit rate(huffman coding): 0.56 bit/pixel
 compression ratio : 14.35


 UNIFORM LENGTH CODING(without overhead) :
 bit rate(uniform length): 0.62 bit/pixel
 compression ratio : 12.80

 UNIFORN LENGTH CODING(with overhead):
 bit rate(uniform length): 0.66 bit/pixel
 compression ratio : 12.18
```

```
Input image:lenna.256
output image:lenna.1
.
 5.7  4.1  3.1  2.3  1.7  1.4  1.0  0.7
 3.3  3.0  2.4  1.8  1.5  1.2  0.8  0.7
 2.1  2.1  1.9  1.7  1.2  1.0  0.7  0.4
 1.1  1.4  1.5  1.3  1.2  0.8  0.8  0.5
 0.7  0.9  0.9  0.9  0.8  0.8  0.5  0.5
 0.2  0.3  0.4  0.5  0.4  0.4  0.3  0.1
 0.0  0.1  0.2  0.1  0.3  0.2  0.1  0.2
 0.0  0.0  0.0  0.0  0.0  0.1  0.1  0.1


***** bit allocation *****
   6    4    3    2    2    1    1    1
   3    3    2    2    1    1    1    1
   2    2    2    2    1    1    1    0
   1    1    2    1    1    1    1    0
   1    1    1    1    1    1    1    0
   0    0    0    1    0    0    0    0
   0    0    0    0    0    0    0    0
   0    0    0    0    0    0    0    0


 MSE/pixel = 112.886780
 SNR = 32.11 dB


 HUFFMAN CODING (without overhead) :
 bit rate(huffman coding): 0.80 bit/pixel
 compression ratio : 10.02

 HUFFMAN CODING (with overhead) :
 bit rate(huffman coding): 0.84 bit/pixel
 compression ratio : 9.56


 UNIFORM LENGTH CODING(without overhead) :
 bit rate(uniform length): 0.95 bit/pixel
 compression ratio : 8.39

 UNIFORN LENGTH CODING(with overhead):
 bit rate(uniform length): 0.99 bit/pixel
 compression ratio : 8.06
```

```
Input image:lenna.256
output image:lenna.2

 6.7  5.1  4.1  3.3  2.7  2.4  2.0  1.7
 4.3  4.0  3.4  2.8  2.5  2.2  1.8  1.7
 3.1  3.1  2.9  2.7  2.2  2.0  1.7  1.4
 2.1  2.4  2.5  2.3  2.2  1.8  1.8  1.5
 1.7  1.9  1.9  1.9  1.8  1.8  1.5  1.5
 1.2  1.3  1.4  1.5  1.4  1.4  1.3  1.1
 1.0  1.1  1.2  1.1  1.3  1.2  1.1  1.2
 0.8  0.9  0.9  0.9  1.0  1.1  1.1  1.1


***** bit allocation *****
   7    5    4    3    3    2    2    2
   4    4    3    3    2    2    2    2
   3    3    3    3    2    2    2    1
   2    2    3    2    2    2    2    1
   2    2    2    2    2    2    2    1
   1    1    1    2    1    1    1    1
   1    1    1    1    1    1    1    1
   1    1    1    1    1    1    1    1

 MSE/pixel= 45.272736
 SNR = 41.27 dB


 HUFFMAN CODING (without overhead) :
 bit rate(huffman coding): 1.51 bit/pixel
 compression ratio : 5.30

 HUFFMAN CODING (with overhead) :
 bit rate(huffman coding): 1.57 bit/pixel
 compression ratio : 5.09


 UNIFORM LENGTH CODING(without overhead) :
 bit rate(uniform length): 1.95 bit/pixel
 compression ratio : 4.10

 UNIFORN LENGTH CODING(with overhead):
 bit rate(uniform length): 2.02 bit/pixel
 compression ratio : 3.96
```

```
Input image:lenna.256
output image:lenna.4

 8.7  7.1  6.1  5.3  4.7  4.4  4.0  3.7
 6.3  6.0  5.4  4.8  4.5  4.2  3.8  3.7
 5.1  5.1  4.9  4.7  4.2  4.0  3.7  3.4
 4.1  4.4  4.5  4.3  4.2  3.8  3.8  3.5
 3.7  3.9  3.9  3.9  3.8  3.8  3.5  3.5
 3.2  3.3  3.4  3.5  3.4  3.4  3.3  3.1
 3.0  3.1  3.2  3.1  3.3  3.2  3.1  3.2
 2.8  2.9  2.9  2.9  3.0  3.1  3.1  3.1


***** bit allocation *****
   9    7    6    5    5    4    4    4
   6    6    5    5    4    4    4    4
   5    5    5    5    4    4    4    3
   4    4    5    4    4    4    4    3
   4    4    4    4    4    4    4    3
   3    3    3    4    3    3    3    3
   3    3    3    3    3    3    3    3
   3    3    3    3    3    3    3    3

 MSE/pixel= 17.414352
 SNR = 52.02 dB


 HUFFMAN CODING (without overhead) :
 bit rate(huffman coding): 2.83 bit/pixel
 compression ratio : 2.83

 HUFFMAN CODING (with overhead) :
 bit rate(huffman coding): 3.05 bit/pixel
 compression ratio : 2.62


 UNIFORM LENGTH CODING(without overhead) :
 bit rate(uniform length): 3.95 bit/pixel
 compression ratio : 2.02

 UNIFORN LENGTH CODING(with overhead):
 bit rate(uniform length): 4.17 bit/pixel
 compression ratio : 1.92
```

```
Input image:pepper.256
output image:pepper.0.05

 5.1  3.3  2.3  1.6  1.1  0.7  0.2  0.0
 3.1  2.1  1.4  0.7  0.3  0.0  0.0  0.0
 2.0  1.3  0.9  0.3  0.0  0.0  0.0  0.0
 1.5  0.7  0.3  0.0  0.0  0.0  0.0  0.0
 1.2  0.3  0.0  0.0  0.0  0.0  0.0  0.0
 0.9  0.0  0.0  0.0  0.0  0.0  0.0  0.0
 0.4  0.0  0.0  0.0  0.0  0.0  0.0  0.0
 0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0


***** bit allocation *****
   5    3    2    2    1    1    0    0
   3    2    1    1    0    0    0    0
   2    1    1    0    0    0    0    0
   1    1    0    0    0    0    0    0
   1    0    0    0    0    0    0    0
   1    0    0    0    0    0    0    0
   0    0    0    0    0    0    0    0
   0    0    0    0    0    0    0    0


 MSE/pixel= 253.065247
 SNR = 31.08 dB


 HUFFMAN CODING (without overhead) :
 bit rate(huffman coding): 0.38 bit/pixel
 compression ratio : 20.91

 HUFFMAN CODING (with overhead) :
 bit rate(huffman coding): 0.41 bit/pixel
 compression ratio : 19.45



 UNIFORM LENGTH CODING(without overhead) :
 bit rate(uniform length): 0.45 bit/pixel
 compression ratio : 17.66

 UNIFORN LENGTH CODING(with overhead):
 bit rate(uniform length): 0.48 bit/pixel
 compression ratio : 16.60
```

```
Input image:pepper.256
output image:pepper.0.25

 5.3  3.5  2.5  1.8  1.3  0.9  0.4  0.0
 3.3  2.3  1.6  0.9  0.5  0.0  0.0  0.0
 2.2  1.5  1.1  0.5  0.1  0.0  0.0  0.0
 1.7  0.9  0.5  0.2  0.0  0.0  0.0  0.0
 1.4  0.5  0.2  0.0  0.0  0.0  0.0  0.0
 1.1  0.1  0.0  0.0  0.0  0.0  0.0  0.0
 0.6  0.0  0.0  0.0  0.0  0.0  0.0  0.0
 0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0


***** bit allocation *****
   5    4    3    2    1    1    0    0
   3    2    2    1    0    0    0    0
   2    1    1    1    0    0    0    0
   2    1    0    0    0    0    0    0
   1    1    0    0    0    0    0    0
   1    0    0    0    0    0    0    0
   1    0    0    0    0    0    0    0
   0    0    0    0    0    0    0    0


 MSE/pixel= 203.629990
 SNR = 33.25 dB

 HUFFMAN CODING (without overhead) :
 bit rate(huffman coding): 0.46 bit/pixel
 compression ratio : 17.31

 HUFFMAN CODING (with overhead) :
 bit rate(huffman coding): 0.49 bit/pixel
 compression ratio : 16.23


 UNIFORM LENGTH CODING(without overhead) :
 bit rate(uniform length): 0.56 bit/pixel
 compression ratio : 14.22

 UNIFORN LENGTH CODING(with overhead):
 bit rate(uniform length): 0.59 bit/pixel
 compression ratio : 13.48
```

```
Input image:pepper.256
output image:pepper.0.5

 5.6  3.8  2.8  2.1  1.6  1.1  0.7  0.2
 3.5  2.5  1.9  1.2  0.7  0.3  0.1  0.0
 2.4  1.7  1.3  0.8  0.4  0.0  0.0  0.0
 1.9  1.2  0.7  0.4  0.0  0.0  0.0  0.0
 1.6  0.8  0.4  0.2  0.0  0.0  0.0  0.0
 1.3  0.3  0.2  0.0  0.0  0.0  0.0  0.0
 0.8  0.0  0.0  0.0  0.0  0.0  0.0  0.0
 0.2  0.0  0.0  0.0  0.0  0.0  0.0  0.0


***** bit allocation *****
   6    4    3    2    2    1    1    0
   4    3    2    1    1    0    0    0
   2    2    1    1    0    0    0    0
   2    1    1    0    0    0    0    0
   2    1    0    0    0    0    0    0
   1    0    0    0    0    0    0    0
   1    0    0    0    0    0    0    0
   0    0    0    0    0    0    0    0


MSE/pixel= 159.956619
SNR = 35.67 dB

HUFFMAN CODING (without overhead) :
bit rate(huffman coding): 0.56 bit/pixel
compression ratio : 14.40

HUFFMAN CODING (with overhead) :
bit rate(huffman coding): 0.59 bit/pixel
compression ratio : 13.49


UNIFORM LENGTH CODING(without overhead) :
bit rate(uniform length): 0.70 bit/pixel
compression ratio : 11.38

UNIFORN LENGTH CODING(with overhead):
bit rate(uniform length): 0.74 bit/pixel
compression ratio : 10.80
```

EE 599
Special Topics
Digital Image Processing on VLSI

Summer 1991

Term Project

Hardware Implementation of Discrete Cosine Transform in Systolic Array for Real-Time Image Processing

Abstract -This report is one part of the project concerning anout the image compression. Following the presentation of "Transform Image Coding " by Chun-Kun Chen, this paper considers the hardware implementation of discrete cosine transform (DCT) and Inverse-DCT. We introduce the systolic array architecture using 64 processing elements (PEs) to achieve the real-time image processing.

Instructor : Dr. Bing Sheu
Student : Chi-Fang Ma
(888-02-3985)

August 14, 1991

I. Introduction

Image compression is an important technique in image processing. In order to implement this processing by hardware, we use the systolic array architecture in which 64 PEs are connected in mesh-like architecture mentioned in the class[1]. Actually, we only implement the DCT part and Inver-DCT part of all four parts presented by the project partner Chun Kun Chen. By pipelining instructions and computations, we improve the performance to 30 ns per clock which results real-time processing. Besides, this design makes it possible to modify the coefficients in order to apply different algorithms.

II. Transformation

Two-dimension Discrete Cosine Transformation from original image x(n1,n2) is shown as the following :

$$DCT(k1,k2) = \sum_{n1=0}^{N-1} \sum_{n2=0}^{N-1} x(n1,n2)\ a(k1)\cos\frac{\pi(2*n1+1)k1}{2*N}\ a(k2)\cos\frac{\pi(2*n2+1)k2}{2*N} \qquad -(1)$$

$$\text{where} \quad a(0) = (1/N)^{0.5}, \quad a(ki) = (2/N)^{0.5} \quad \text{for } 1 \le ki \le N-1$$

$$I\text{-}DCT(n1,n2) = \sum_{n1=0}^{N-1} \sum_{n2=0}^{N-1} DCT(k1,k2)\ a(k1)\cos\frac{\pi(2*n1+1)k1}{2*N}\ a(k2)\cos\frac{\pi(2*n2+1)k2}{2*N} \qquad -(2)$$

$$\text{where} \quad a(0) = (1/N)^{0.5}, \quad a(ni) = (2/N)^{0.5} \quad \text{for } 1 \le ni \le N-1$$

By observing equation (1) and (2), we find that they are the same operation in hardware level. Therefore, we only consider DCT. Now it is defined by

$$DCT(k1,k2) = \sum_{n1=0}^{N-1} \sum_{n2=0}^{N-1} x(n1,n2)\ \ c(k1,k2,n1,n2)$$

According to the the specification of this project, we assume that the input image contains 256x256 pixels and the window size of DCT operator is 8x8.

III.Architecture

3.1. The global system architecture

Sixty-four PEs are connected as a mesh-like architecture shown in Figure 1. Horizontally, eight PEs in each row are connected as a ring array. The 'Mux' in front of the ring array selects the input data either from system memory or from the ring itself. Vertically, each PE sends out the output data to the PE at next level. The 'Buf' latches the vertical output data and then sends them to coprocessor bus. The system controller sends instructions to handle all PEs according to the requirement from host.

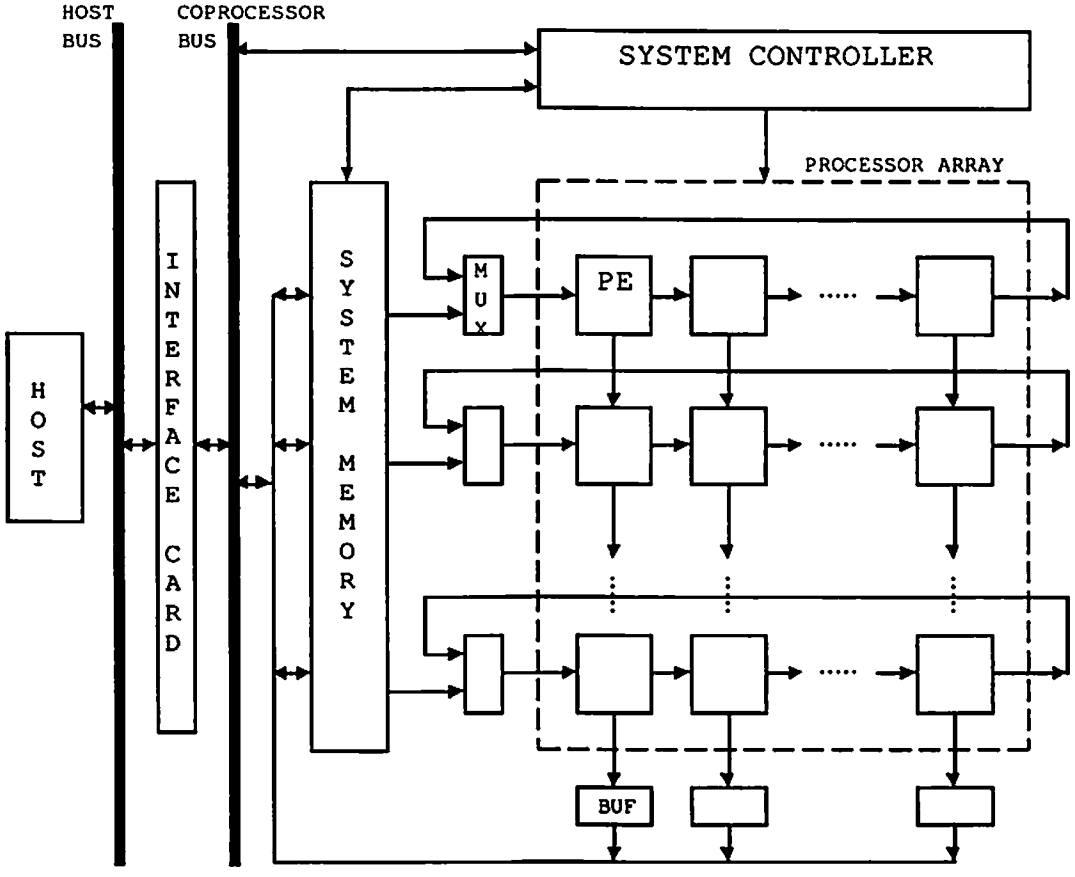


Figure 1 The block diagram in global system architecture

2.2. The function block in each PE

There are four registers, one latch, one multiplier, two adders, 64-byte cache and the decoder which is the controller of the PE in this architecture shown in Figure 2. The Instruction decoder

decodes the instructions from system controller and then starts the micro-instruction to send the control signals to each function block. The cache acts like a FIFO. The data in cache can be programmed and are loaded from system memory through R1 and R4 at the beginning of the processing. The input data size is 8 bits while the output data size is 16 bits.

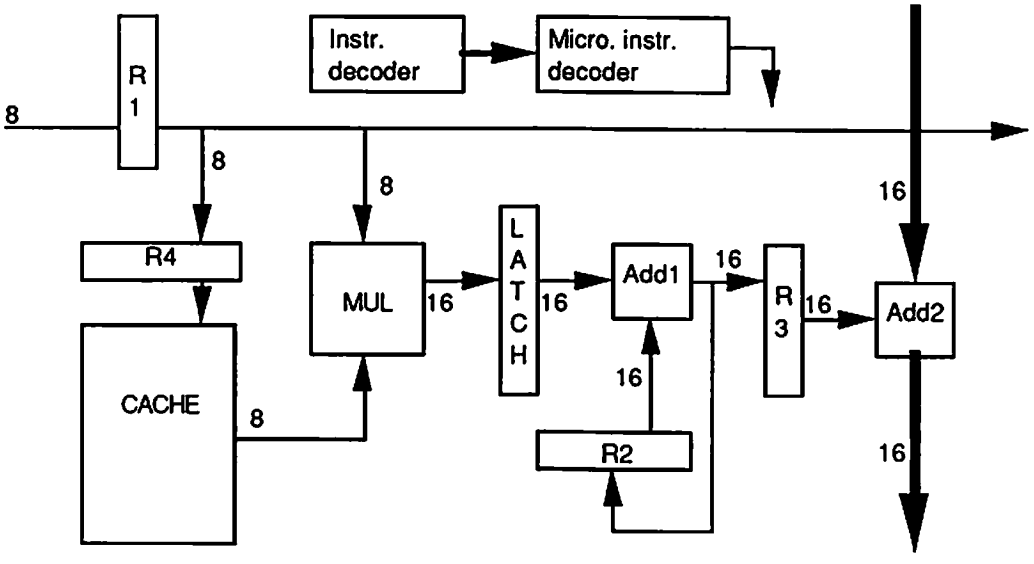


Figure 2 The function block in a PE

2.3. Memory mapping

The size of each coefficient (c(k,l,m,n)) is one byte. There are sixty-four bytes in a PE. Totally there are at least 4k bytes cache size required in this system. The mapping is shown in Figure 3.

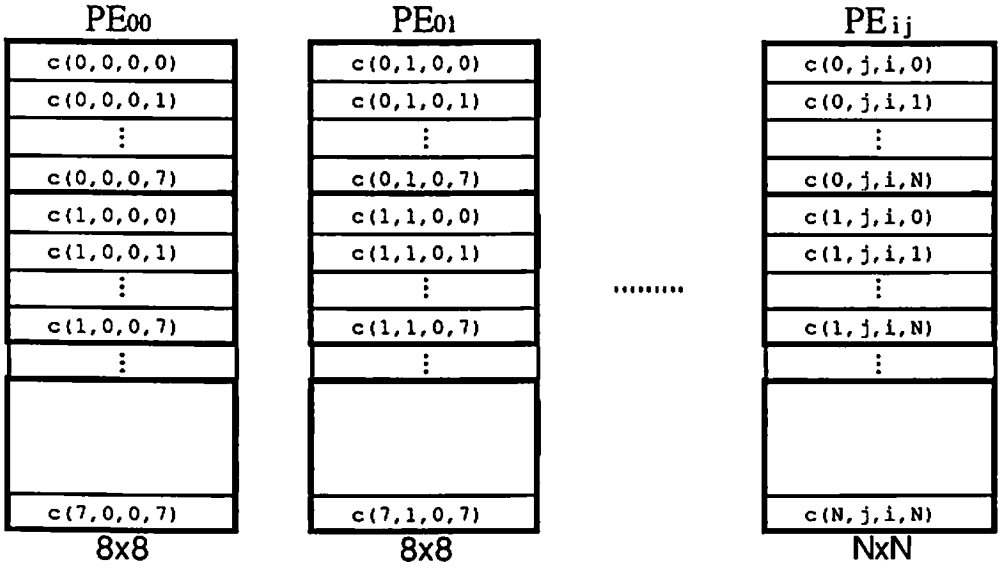


Figure 3 Memory mapping of each PE

IV. Description of Operation and Computation

4.1. Operation at system level

Host sends instructions to system controller through Interface card. The coefficients of DCT which are needed in each PE are stored at the beginning of the processing. When operation starts, the eight data from input image file are sent into processor array, and continue the following data for eight clocks (That means there are sixty-four output data from system memory within eight clock cycles). Then set the multiplexer to connect the feedback to form a ring and continue for another seven ring operation cycles. Then feed the next sixty-four data to do the whole operation above again. The rough timing is shown in Figure 4. By pipelining the operations, we can get an output data per clock cycle.

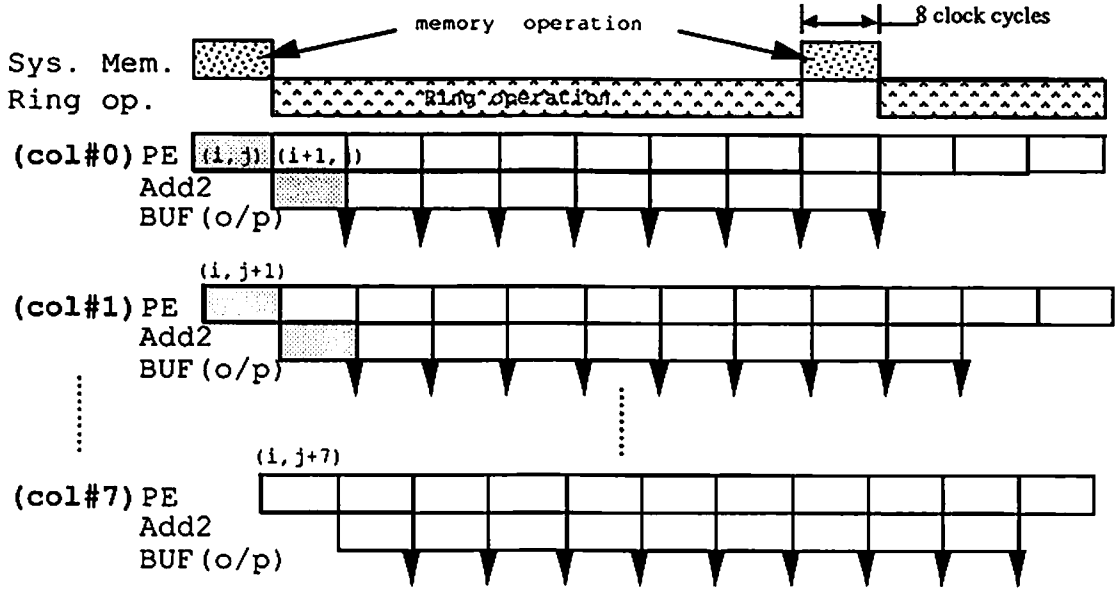


Figure 4 Pipeline timing of system level

4.2. The mapping of operation to each PE

Step 1 : Load x(n1,n2) to R1 and R1 will put this data at the leading edge of clock. At the same time, start the cache to fetch one coefficient according to the FIFO counter.

Step 2 : Do multiplication operation of two data from R1 and cache.

Step 3 : Latch the output data(16 bit) of multiplier and add it with the output data(16 bit) of R2. These operations need to be finished in one half cycle. Then, either R2 latches the output of Add1 or clear R2 and enable R3 in the following half cycle.

Step 4 : This step is executed every eight clock cycles. Enable Add2 and send result to the PE in the next level. Because this operation includes eight addition operations, it lasts eight clock cycles.

Step 5 : Each Buf. outside the processor array gets the result of Add2 and sends it to coprocessor bus per eight clock cycles.

The timing chart is shown as Figure 5.

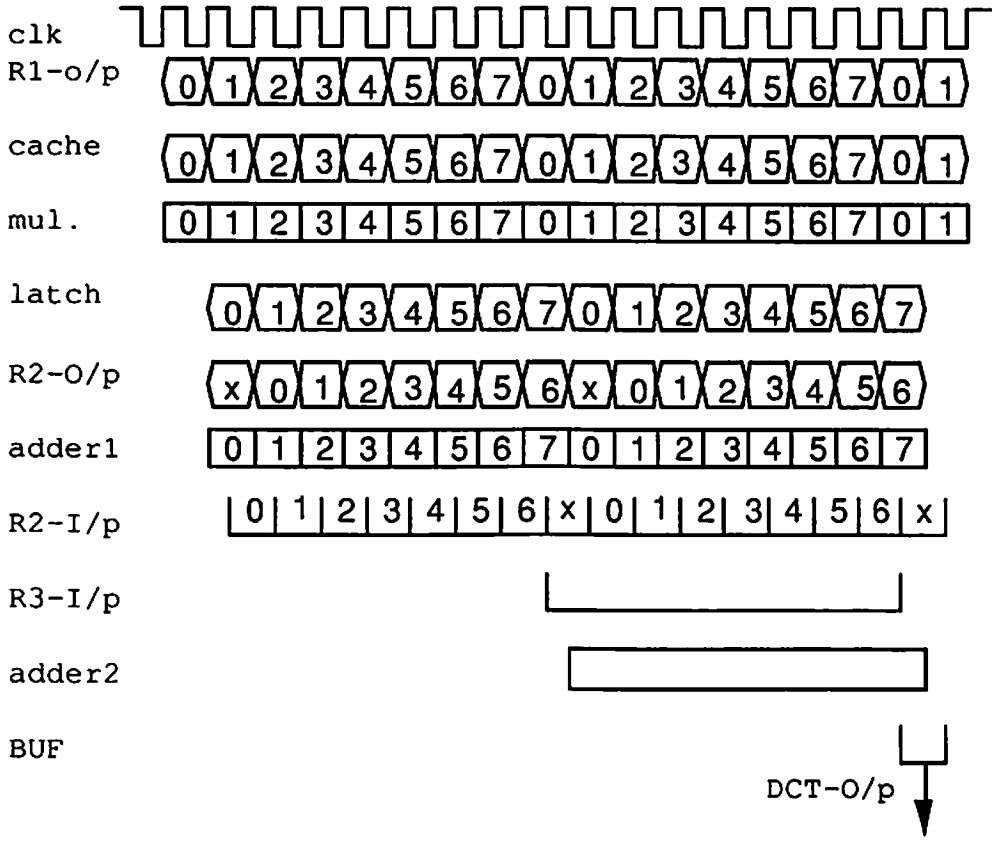


Figure 5 The pipeline timing of operation in each PE

V. Performance Evaluation

If we implement this design by using 0.5 um CMOS technology. We can make sure that every step can be finished in 30ns which are estimated as the following :

Multiplier(8x8)	25n, 200x200 um^2, 880 transistors
Adder(16)	15ns, 150x180 nm^2, 400 transistors
Register(1)	Master-Slaver JK-flipflop, 40x30 nm^2, 26 transistors
Cache	8 x 8bytes FIFO in each PE(8x8window size)
	Shift register(4),70x110 um^2, 140 transistors
Latch(1), Buffer(1)	leading edge trigger flipflop, 20x10 um^2, 4 transistors

On these assumption, the performance of processing can be caculated as the following : Clock rate could reach to 33Mhz (30 ns per cycle) due to the critical operation - multiplication. The area size of each PE is at least $3x10^{-3}$ cm^2 which does not include decoder and controller parts. Almost 5,500 transistors are in each PE not including the decoder part.

VI. Extensibility

This design can be suitable for the DCT algorithm which operator window size is larger than 8x8 by programming the input data sequence of image file. Of course, the cache size in each PE need to increase. But the speed is the same. Besides, several processor arrays can be cascaded to form a larger processor array to perform more powerful computation. Of course, the programable comtroller is very important.

VII. Conclusion

From the above design, we find the following features :

Programmable : First, the coefficients could be programmed from host in order to fit different algorithms. Second, the whole operations could be programmed to fit different size of operator window.

Real-time operation : According to the requirement of 30 frames per second, this design achieves it by pipelining each operation in 30 ns per clock , when we consider using 0.5 um VLSI technique to

process 1024x1024 image.

This hardware architecture is designed specifically to perform the operation of discrete cosine transform (DCT). Therefore, it needs a lot of consideration to be modified to apply other operations of image processings. Moreover, the detailed control function still needs a lot of time to design.

VIII. References

[1] Chia-Fen Chang and Bing J. Sheu, "Design of Digital VLSI Neuroprocessor for Signal and Image Processing.

[2] Tony Denater, Etienne Vanzieleghem and Paul G. A. Jespers, "A class of Multiprocessors for Real_Time Image and Mulitidimensional Signal Processing", Vol 23, No 3, June 1988.

[3] Anil K. Jain, "Fundamentals of Digital Image Processing", Prentice-Hall, 1989.

VLSI Implementations of the Discrete Cosine Transform in Real Time

by

Dean Liu #412-19-1722

August 14, 1991

Abstract: A case study involving three methods of implementing the discrete cosine transform is presented. Only the mesh-connected systolic array provides both real-time capability and easy implementability. It is shown that the systolic array can process each 1024x1024 pixel frame in 32ms.

INTRODUCTION:

Real time transmission of sound and picture over a physical channel can only be accomplished if the amount of raw data that has to be sent can be reduced. This necessary reduction can be achieved using several coding techniques one of which is the discrete cosine transform. In this paper, possible implementations of the DCT are examined. If picture frames are 1024x1024 pixels, then each system will need to perform 16,384 8x8 cosine transforms per frame.

The forward kernel of the two-dimensional DCT is defined as

$$w(n_1, n_2, k_1, k_2) = 4\,[\cos(2n_1 + 1)\,k_1\,(\pi/(2N))]\,[\cos(2n_2 + 1)\,k_2\,(\pi/(2N))]$$

for n_1, n_2, k_1, k_2 = 0, 1, ..., N-1. Thus it follows that the two-dimensional DCT is given by the expression

$$C(k_1,k_2) = \sum_{n_1=0}^{N-1}\sum_{n_2=0}^{N-1} x(n_1,n_2)w(n_1, n_2, k_1, k_2)$$

Since the kernel is separable, the DCT can be expressed in terms of the multiplication of three matrices. The DCT can now be expressed as

$$C = W^T \bullet X \bullet W$$

where W is a matrix containing the transform coefficients (the evaluated cosine terms) and X and C are matrices containing the input data and the transformed results respectively. If the W matrix is defined as follows

$$W = \begin{bmatrix} w_{00} & w_{01} & w_{02} \cdots & w_{07} \\ w_{10} & w_{11} & w_{12} \cdots & w_{17} \\ w_{20} & w_{21} & w_{22} \cdots & w_{27} \\ \vdots & \vdots & \vdots & \vdots \\ w_{70} & w_{71} & w_{72} \cdots & w_{77} \end{bmatrix}$$

then the elements w_{nk} are

$$w_{nk} = 2\cos\left((2n+1)\ (\pi k) / (2N) \right)$$

Since the coefficients for the C and X matrices are defined similarly, the DCT equation can now be expressed as

$$C = \begin{bmatrix} w_{00} \cdots w_{70} \\ \vdots \quad \vdots \\ w_{07} \cdots w_{77} \end{bmatrix} \begin{bmatrix} x_{00} \cdots x_{07} \\ \vdots \quad \vdots \\ x_{70} \cdots x_{77} \end{bmatrix} \begin{bmatrix} w_{00} \cdots w_{07} \\ \vdots \quad \vdots \\ w_{70} \cdots w_{77} \end{bmatrix}$$

Therefore, the transform of the 8x8 pixel block is still an 8x8 matrix with each element c_{ij} being

$$c_{ij} = \sum_k \sum_l x_{kl} \cdot w_{ki} \cdot w_{lj}$$

DESIGNS INVESTIGATED:

The 64 cycle matrix method: Each of the transformed points is a weighted sum of all the data points. Hence, one possible design would be to have all the data values presented serially to an array of 64 multiplier/accumulators (MAC). A MAC is a device with an internal structure optimized for performing operations of multiplication and addition. These devices are relatively common and a block diagram of a MAC is shown below in figure 1.

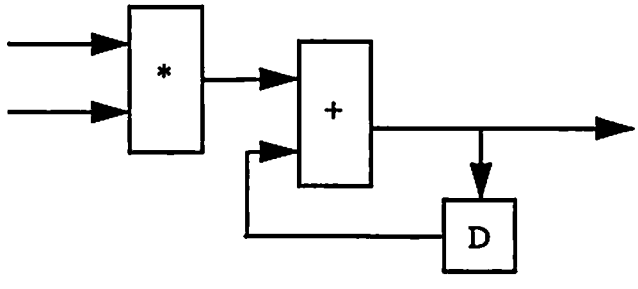


Figure 1: Block diagram of multiplier/accumulator

Each MAC calculates one transform point and has an associated ROM containing a look-up table of relevant weights. Therefore, the data point and its associated weight are read in, multiplied together, and the result is added to the sum of the previous values. After 64 such cycles, the

result becomes the appropriate transform value. At the output of the MAC the 64 transformed points can be combined back into a serial stream by a demultiplexing system. A 2x2 equivalent system is illustrated in figure 2.

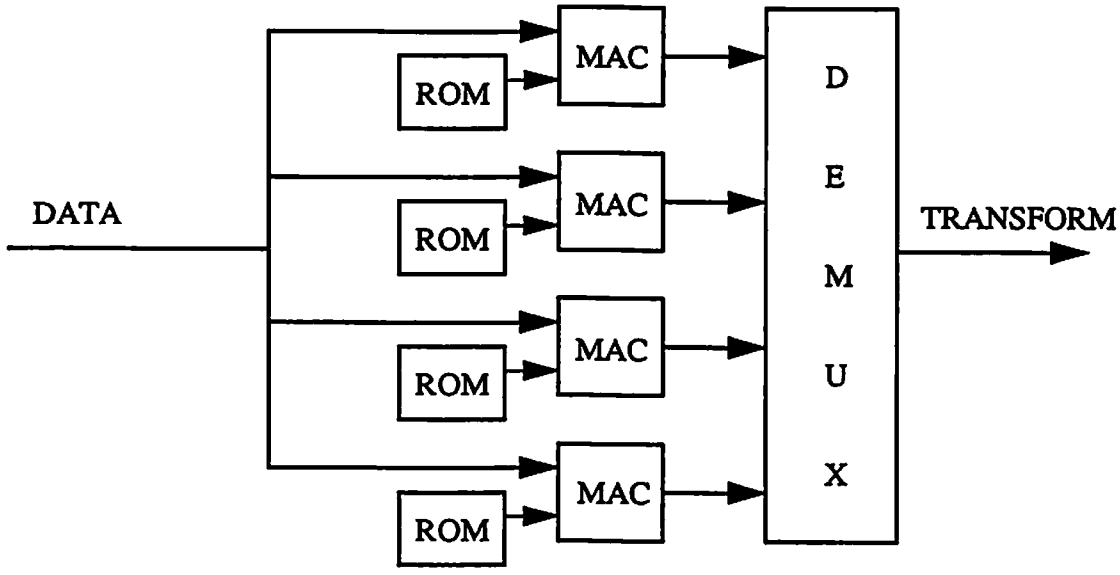


Figure 2: 2x2 equivalent transform system for 64 cycle method

If 0.5μm CMOS technology is implemented, each MAC will have a delay time of ~45ns. Thus, assuming a 40ns ROM access time, clock periods of ~100ns are acceptable in this design. Using these values, 128x128x64x100ns = 105ms are needed to process each frame. Therefore this implementation is not capable of handling the required 30 frames/sec and other designs must be considered.

The 16 cycle matrix method: If the DCT is implemented as 2 8x8 matrix multiplications, 8 multiply/add operations are needed for each of the 64 points per matrix multiplication. Since each point can be calculated concurrently, 16 cycles are required for the entire calculation. Therefore, 64 devices are required for each matrix multiplication and thus 128 MACs are required overall. In order to reduce the device count to 64 MACs, a feedback system suggests itself. However, this method requires somewhat complicated interconnections due to the feedback routing.

A 2x2 equivalent system is shown in figure 3. The design is similar to the 64 cycle method described earlier except an additional synchronous multiplexing system in which the lower half of the input addresses must be clocked between matrix multiplications is required. Also, since each

MAC no longer requires all 64 input data points, the data input line is separated.

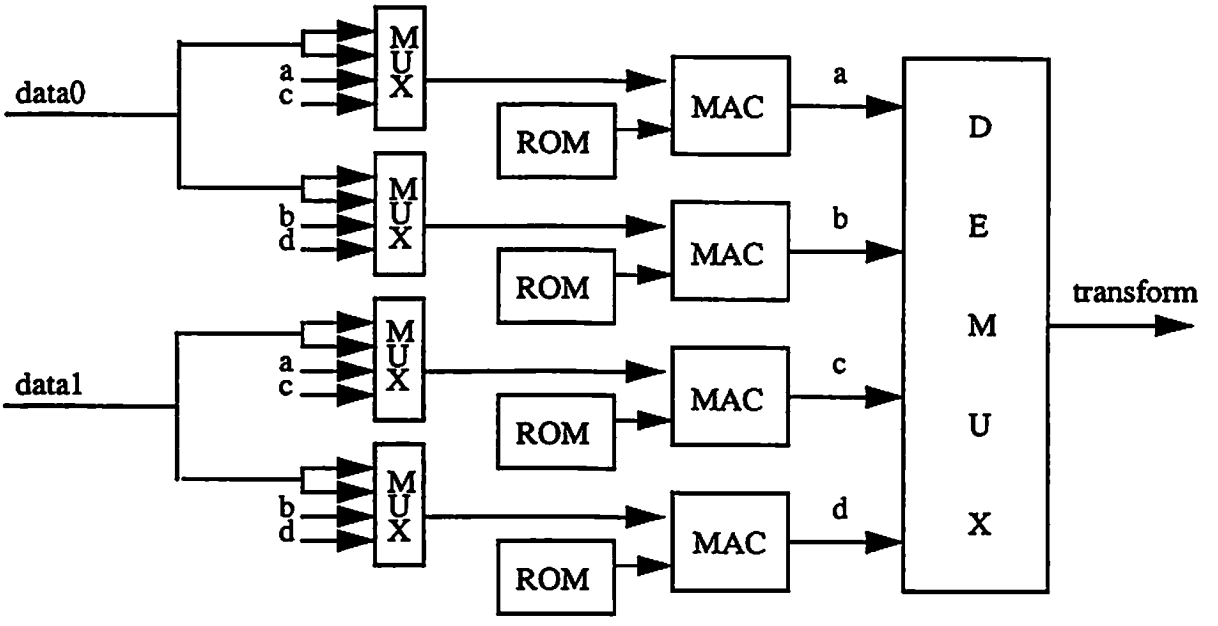


Figure 3: 2x2 equivalent transform system for the 16 cycle method

If a clock period of 125ns is incorporated to account for the extra delay of the multipliers, then a complete transform requires 32ms. Thus, the 16 cycle method is capable of real time operation but is difficult to implement due to complex wiring and logic.

The systolic array method: As a final implementation method, the 2 matrix multiplication algorithm is mapped to the two-dimensional mesh connected systolic array. This method achieves the best results because 1) each transform can be completed in 16 clock cycles and 2) the regularity of the array provides greater implementability.

Because the architecture of the mesh-connected systolic array has already been discussed in class, only the contents of each PE and a general algorithm of operation will be presented. An 8x8 array of PE's are required to perform the transform and a 16 address cache memory is required in each PE. At initialization, each memory is written with the necessary weights w_{ij}. This initialization does not need to be re-performed for subsequent transform calculations. The activation value, a_{ij}, of each PE will be the input data points x_{ij}. During the first eight clock cycles, the activation values will be passed downward along each column of PE's. The accumulated sum in each PE is then moved to that PE's new activation value and then the sum must be cleared. During the next eight clock cycles, activation values will be passed horizontally along each row of PE's. The

value of each transform point, c_{ij}, is now the accumulated total at the corresponding PE.

A 2x2 equivalent transform is performed in figure 4. Initialization is assumed to have been already performed. The values inside each PE represent the contents of the cache memory. Each PE's current activation value and accumulated sum are displayed above and below the PE respectively.

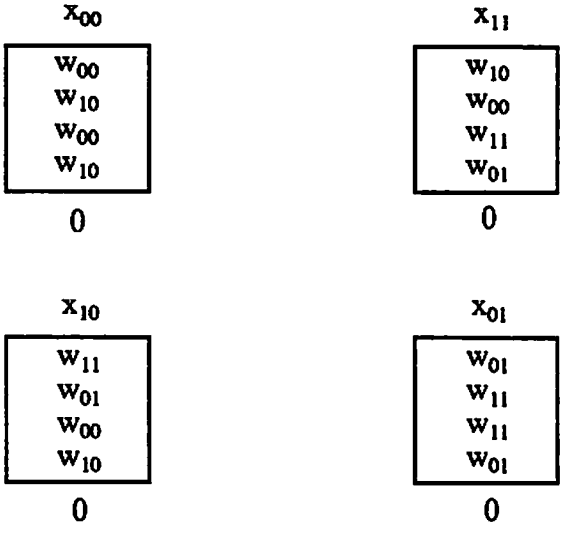


Figure 4a: Systolic array after initialization

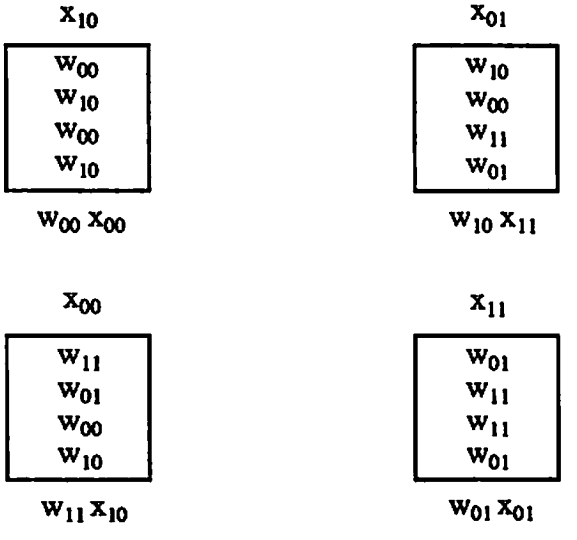


Figure 4b: Systolic array after 1 clock pulse

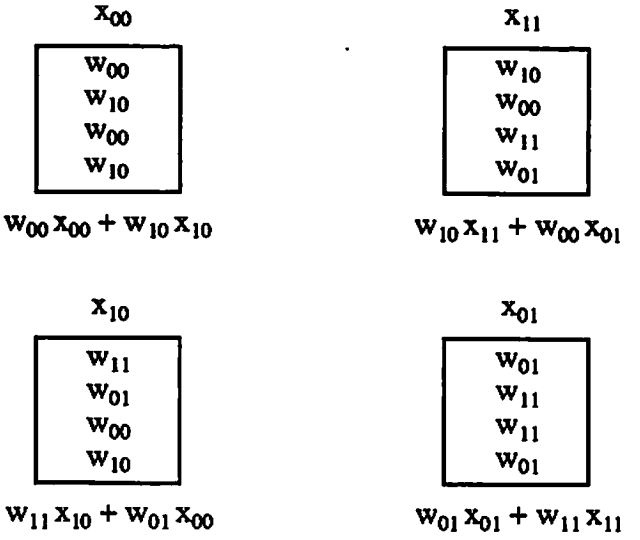


Figure 4c: Systolic array after 2 clock pulses

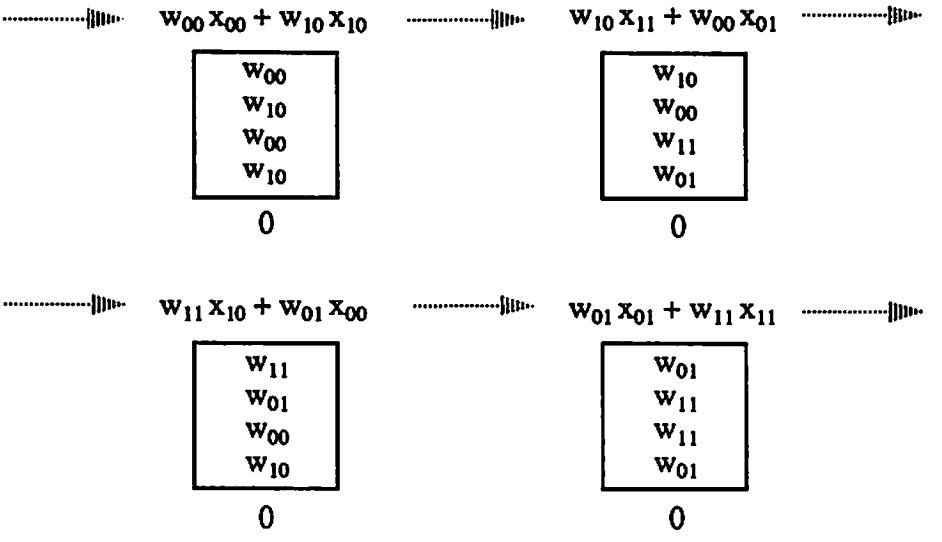


Figure 4d: Systolic array after activation value re-initialization. Arrows indicate new direction of activation value transfer.

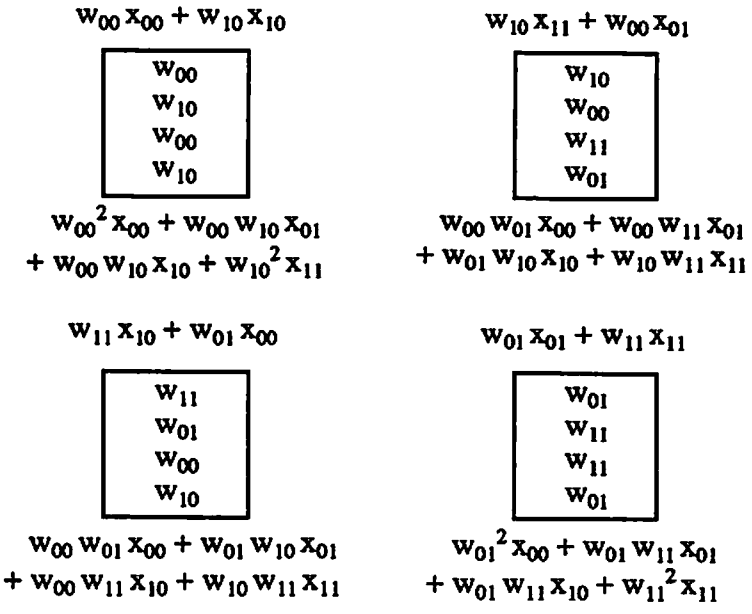


Figure 4e: Final results

Considering delay times for the multiplication, addition, memory access, and PE interconnections, a clock period of ~80ns seems sufficient. Each 8x8 transform requires 16 clock cycles for the calculation and 8 cycles for the loading and unloading of data. Therefore, 128x128x24x80ns = 32ms are needed to perform each transform and thus real-time operation is possible.

CONCLUSION:

Three methods of implementing the DCT were investigated. Although it was found that two of the methods could be implemented in real-time, one proved very difficult to implement due to complex wiring. The remaining possible solution was the mesh-connected systolic array. It was calculated that 33ms were needed per frame and that this was sufficient for real-time operation of 30 frames per second. Also, because of its regularity, implementation would be considerably less costly. One drawback of the array is the necessity of a controller for the PE's, but because the controller would lie on-chip, its added complexity would almost appear transparent to the system designer.

REFERENCES:

1. LIM, J.S., 'Two-Dimensional Signal and Image Processing', Prentice Hall, 1990.
2. GONZALEZ, R.C., and WINTZ, P., 'Digital Image Processing', Addison-Wesley, 1977.
3. BREBNER, G.E., and RITCHINGS, R.T., 'Image Transform Coding', IEE Proceedings Part E, January 1988, p. 41-8.
4. CHANG, C.F., and SHEU, B.J., 'Design of a Digital VLSI Neuroprocessor for Signal and Image Processing', September 1991.