Image Processing Algorithms for the Pipelined Image-Processing Engine

by

Gilbert Verghese
Shekhar Mehta
Charles R. Dyer

Computer Sciences Technical Report #668

September 1986

Image Processing Algorithms for the Pipelined Image-Processing Engine

Gilbert Verghese
Shekhar Mehta
Charles R. Dyer

Computer Sciences Department
University of Wisconsin
Madison, WI 53706

Abstract

In this paper we describe nine basic vision algorithms for the National Bureau of Standards' Pipelined Image-Processing Engine (PIPE). The algorithms presented are: local peak detection, median filtering, thinning, the Hough transform for line detection, photometric stereo, *n*-bit point operations, detecting edges at multiple resolutions, stereo vision, and multiresolution, model-based object recognition.

The support of the National Bureau of Standards under Order No. NANB510565 and the National Science Foundation under Grant No. DCR-8520870 is gratefully acknowledged.

1. Introduction

Recently a number of architectures have been designed for real-time, low-level image processing. In order to evaluate the relative advantages and disadvantages of these different architectures it is necessary to design and implement various standard operations on each of the machines and compare their performance. In this paper we describe algorithms for nine common image processing operations that can be implemented on the National Bureau of Standards' Pipelined Image-Processing Engine (PIPE) [1].

PIPE consists of a linearly connected array of processing stages. Each stage contains some local image buffer memory and specialized hardware functional units for performing several basic image processing operations such as point and neighborhood functions. Each operation is applied to an entire 256×256 image. The data communication paths between adjacent stages are used to transfer images produced in one stage to other stages where they are input to other operations.

In order to better assess the PIPE architecture for low-level image processing, we investigate in this paper the implementation of the following tasks:

- Local peak detection
- Median filtering
- Thinning
- Hough transform for line detection
- Photometric stereo
- n-bit point operations
- Computing zero-crossing pyramids
- Stereo vision
- Multiresolution, model-based object recognition

We have selected this set of tasks because it represents a wide range of types of operations which are frequently performed in image processing and computer vision applications. Local peak detection and median filtering are nonlinear, local operations. Thinning is an iterative, local process. The Hough transform is a global, many-to-one operation. Photometric stereo and n-bit point operations are examples of generalized point operations. The last three tasks all involve the

use of multiresolution representations. First we describe the detection of edges at multiple resolutions, constructing a zero-crossing pyramid using PIPE. Second, we describe an implementation of the Marr-Poggio stereo matching algorithm. Finally, we present a coarse-to-fine matching algorithm for model-based object recognition which uses the zero-crossing pyramid as its basic representation.

For each task we present a PIPE algorithm specified by a network of basic operations which can be directly executed by the appropriate functional units in one or more stages. Speeds of the algorithms are also given in terms of number of time units, where one time unit equals one PIPE field-time or 1/60 second. Several of the more complicated operations are not completely implemented in PIPE. For these algorithms, the operations associated with the host are also specified but their timings are not given.

Section 2 describes the details of the PIPE architecture. Sections 3-11 present PIPE algorithms for the tasks listed above (in that order).

2. PIPE Architecture

In this section we describe the architecture of PIPE. For more details, see [1,2]. PIPE is designed for parallel processing of two-dimensional image data only. All operations are applied to one or more images and all data paths communicate images. The organization of PIPE is a linear sequence of processors, called stages, plus two specialized stages for input and output. Each stage has its own local image memory, called image buffers, which is used to store intermediate image results. There is no shared global memory.

Image data communication between stages is accomplished via direct interconnections from each stage to its two adjacent neighbors and to itself as shown in Fig. 1. Because all operations performed in each stage take a fixed amount of time, communication between stages can be performed synchronously at the end of each time unit. Thus, at the end of each time unit each stage, i, can output up to three images independently. One output goes to stage $i+1$ via the forward path. A second goes to stage $i-1$ via the backward path. A third is output to stage i itself via the recursive path. In addition to direct communication with each stage's nearest neighbors, PIPE has two "wildcard" buses, called VBUS A and VBUS B, which may be used to output an image to one or more arbitrary stages. Only one image may occupy each of these wildcard paths during any time unit.

The input stage receives images from up to two input devices. It outputs an image either to stage 1 via its forward path, or to other stages via one of the wildcard buses. The output stage receives images from the last stage's forward path or from other stages via a wildcard bus. This stage outputs images to the host or to a special processor called ISMAP. ISMAP maps an image of values into an "inverted index" form which specifies for each possible value which coordinates in the image contain that value.

A pipeline of operations can be performed on a sequence of images passing through PIPE as follows. Each stage performs a sequence of operations on an image it receives from the previous

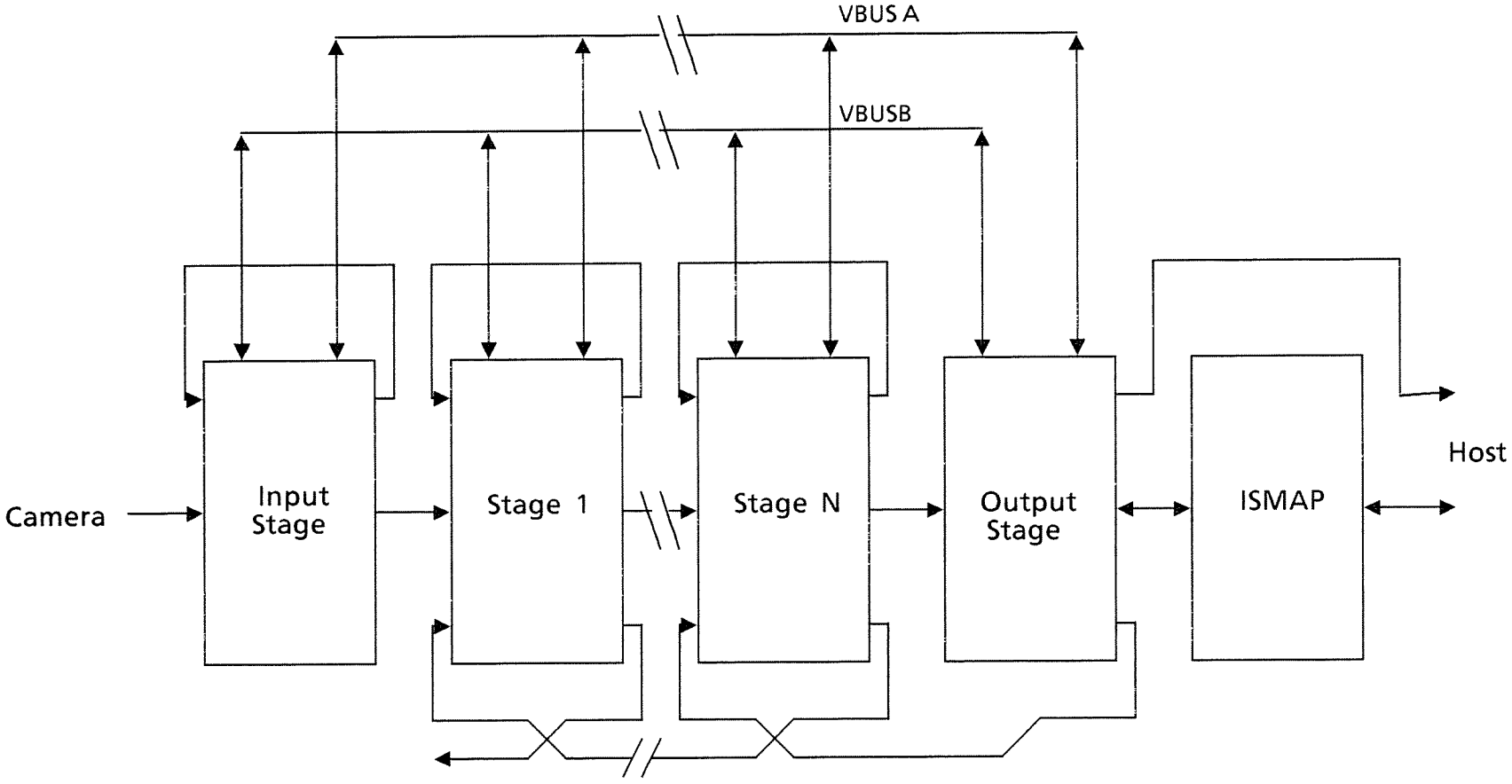


Fig. 1. Organization of stages and image communication paths in PIPE.

stage and then passes its resulting image to the next stage for further processing. The final image output by the last stage exits PIPE through the output stage to the host. In this scheme there is a continuous, one-way flow of images through PIPE using the forward paths to communicate images between stages. However, because of the presence of the additional communication pathways in PIPE (i.e., the backward path, recursive path and two wildcard buses), there is considerably more flexibility in defining sequences of image operations. That is, PIPE is in effect a linearly connected set of synchronous, MIMD parallel processors, where each processor is specially designed for performing image processing operations.

Fig. 2 shows an individual stage of PIPE. Within a stage three basic types of image operations can be performed. All operations in a stage take place within 1/60 second. The simplest is a point operation in which each pixel's value is transformed into a new pixel value by a specified function. This function depends only on the given pixel's value and not its spatial position in the image. These operations are implemented via lookup tables which are shown in Fig. 2 as LUT functional units. LUT's in this and subsequent figures are shown as rectangles. The second type of operation includes all arithmetic and Boolean functions of two operands which compute the value of an output pixel as a function of the corresponding pair of pixels in the two input images. A simple arithmetic logic unit, denoted ALU in Fig. 2, computes these results. ALU's are shown here and subsequently as parallelograms. Finally, 3×3 neighborhood operations can be specified which compute the value of an output pixel as an arithmetic or Boolean linear combination of the nine values in a 3 by 3 block of pixels centered at the corresponding point in the input image. Each PIPE stage contains two hardware units for performing these types of operations; in Fig. 2 these are shown as NBR functional units. NBR's are shown using circles in this and other figures below.

The hardware functional units in a stage can be grouped into two parts: those functional units that are before the stage's image buffers and those that follow them. Currently, each PIPE stage can contain up to thirty-two 256×256 image buffers.

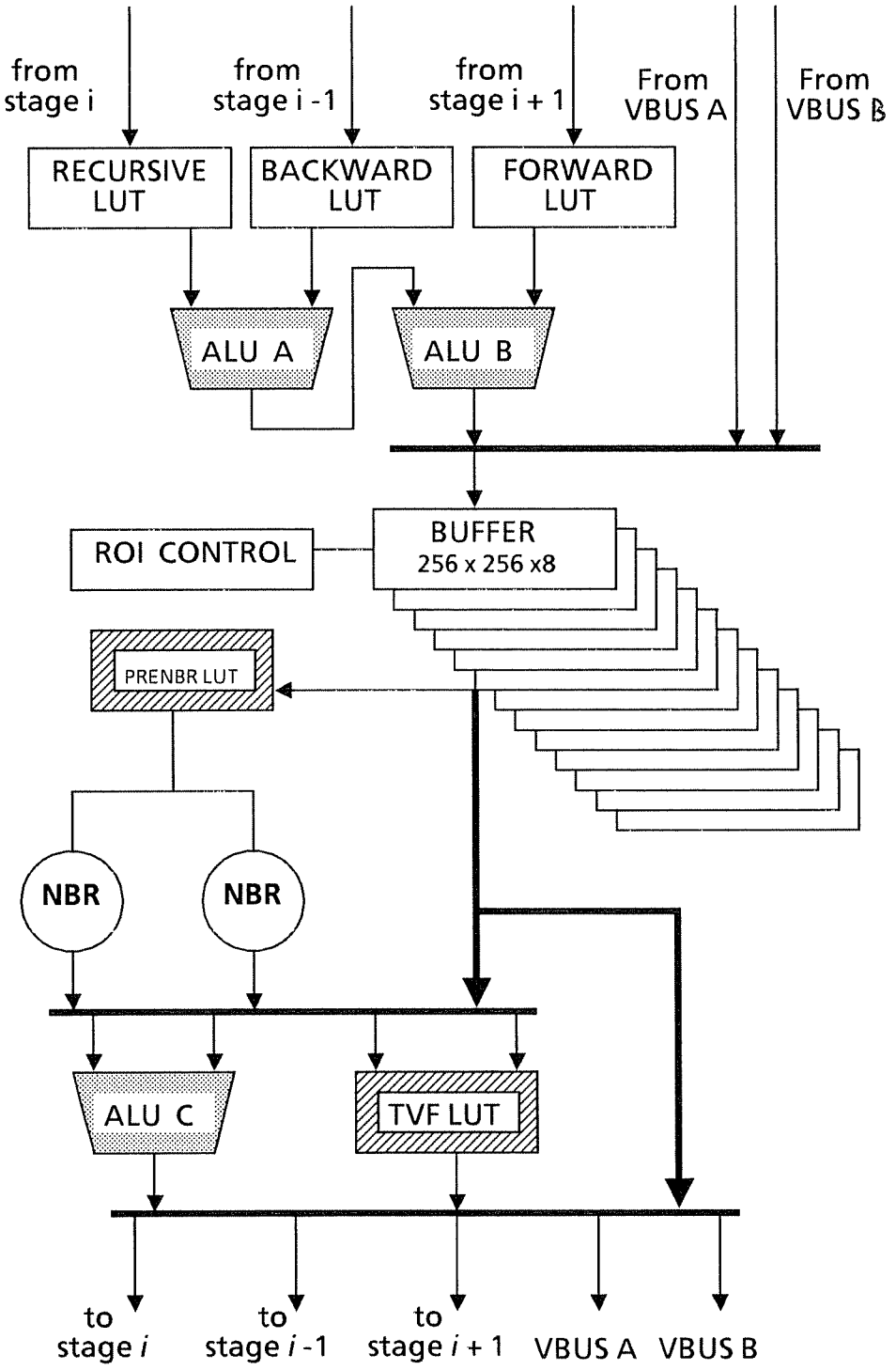


Fig. 2. Block diagram of a single stage of PIPE showing the image buffers, look up tables (LUT), neighborhood operators (NBR), region of interest controller (ROI) and ALU's. The shapes associated with these functional units will be used throughout the remaining figures.

The pre-buffer functional units are used to combine up to three input images into one using the LUT's and ALU's shown in Fig. 2. Input to the image buffers is from either ALU B or one of the wildcard buses. Stage buffers can store images for an arbitrary length of time. At each time unit some selected buffers may be written into. All others retain their previous contents. This is important in maintaining results from prior operations for future use.

Following the image buffers there are two sets of operations which can be performed. First, an image can be input to an LUT unit (PRE-NBR LUT). The output of this unit is then piped into *both* NBR's. The NBR's can compute separate functions. The second set of operations following the stage buffers allows images to be combined using an LUT and an ALU. The LUT (TVF LUT) receives one or two images as input. The ALU (ALU C) requires two images as input. The images input to the TVF LUT or ALU C may come from either of the stage buffers or from any of the NBR's.

Each PIPE stage has two additional operating modes which will be used in this paper. First, a "region of interest" (ROI) mode is used to specify on a pixel by pixel basis what particular operation should be performed in any of the functional units. More precisely, the complete operation of a stage at each time unit is controlled by a micro-instruction which specifies all functions performed and data paths used. Up to 256 micro-instructions can be stored and which of these instructions is used is selectable on a pixel by pixel basis using a designated second image to map each pixel to one of the micro-instructions. Thus the value of a pixel in the ROI map is an index into the proper micro-instruction for the corresponding pixel in the image being processed.

A second mode of processing allows an image's resolution to be changed. As an image is being stored in an image buffer, its resolution can be simultaneously reduced by one-half. In addition, the resolution of an image can be doubled as it is read from an image buffer.

Finally, images can be output from a stage in several ways. The first method of output allows the contents one of the image buffers to be output to the host via a DMA bus. The second method sends output to any subset of the forward, recursive and backward paths, and wildcard buses VBUS A and VBUS B. The output to each is determined independently and may come from any of the buffers, either NBR, the TVF LUT or ALU C.

3. Local Peak Detection

One common nonlinear local operation is the detection of peaks or local maxima with respect to some local property computed at each pixel. For example, when edges are detected using first derivative operators, a single edge will be detected in more than one position due to the size of the operator. In order to detect the single best position of the edge, nonmaximum responses of the operator can be suppressed in a neighborhood orthogonal to the direction of the edge. More generally, when a matching operator is applied to an image, there is frequently a region surrounding the best match position in which the operator gives some positive degree of match. In order to isolate the single best match position, a match at a given pixel is rejected if there is a better match in some neighborhood around the pixel. This procedure is frequently used to suppress nonmaxima following matching operators such as cross-correlation and the Hough transform.

In this section we describe the implementation of one particular algorithm for detecting peaks in an image [3]. Briefly, the method is a nonlinear operation which compares each pixel's value with its eight nearest neighbors' values and a predefined threshold to determine if the value at the point should be suppressed (set to 0). More specifically, consider a pixel e and its eight neighbors:

$$\begin{matrix} a\ b\ c \\ d\ e\ f \\ g\ h\ i \end{matrix}$$

e is considered a peak if there exists a one-dimensional cross-section in which e's value is strictly greater than both its two neighbors' values in the given orientation. In particular, four orientations are considered at each pixel corresponding to cross-sections oriented at multiples of 45°. For pixel e these four triples of pixels are $a\ e\ i$, $b\ e\ h$, $c\ e\ g$ and $d\ e\ f$. The value at pixel e is replaced by 0 if $nonmax(e)$ is true, as defined below using the user-specified threshold T:

$$nonmax(a,e,i) = (e<a)\ OR\ (e<i)\ OR\ ((e=a)\ AND\ (e=i))$$

$$nonmax(b,e,h) = (e<b)\ OR\ (e<h)\ OR\ ((e=b)\ AND\ (e=h))$$

$$nonmax(c,e,g) = (e<c)\ OR\ (e<g)\ OR\ ((e=c)\ AND\ (e=g))$$

$$nonmax(d,e,f) = (e<d)\ OR\ (e<f)\ OR\ ((e=d)\ AND\ (e=f))$$

$$w = nonmax(a,e,i)\ AND\ nonmax(b,e,h)\ AND\ nonmax(c,e,g)\ AND\ nonmax(d,e,f)$$

$$u = e < T$$

$$nonmax(e) = w\ OR\ u$$

3.1. Implementation

In order for PIPE to compare two pixels' values they must be stored in the same position in two different images. Creating a new image from a given original image by a uniform shift has the effect of putting two neighbors in register throughout the two images. In PIPE this can be accomplished for any pair of adjacent pixels by using the NBR unit to shift an image one pixel in any of eight directions as shown in Fig. 3. For example, the mask SE shifts an image southeast so that pixels e and a are aligned in the two images. These eight neighborhood operators will be used in later sections for the same purpose.

Assuming an eight stage PIPE, the steps required to compute the given peak detection algorithm are as follows. First the input image is broadcast to all eight stages. Next, the eight neighborhood operations, SE, NW, S, N, SW, NE, E, and W, are performed at the stages 1 through 8, respectively. Thus the neighbors brought in register with e are a, i, b, h, c, g, d, and f, at stages 1-8, respectively. Pairs of consecutive stages hold the images needed for the four triples. In each of these stages the image produced by the neighborhood operation is immediately subtracted from the original image e. Images $e-a$, $e-b$, $e-c$ and $e-d$ are output from their respective stages via the recursive paths, while simultaneously images $e-f$, $e-g$, $e-h$ and $e-i$ are output via the backward paths. All eight stages and the input stage also pass the original image e through the forward paths.

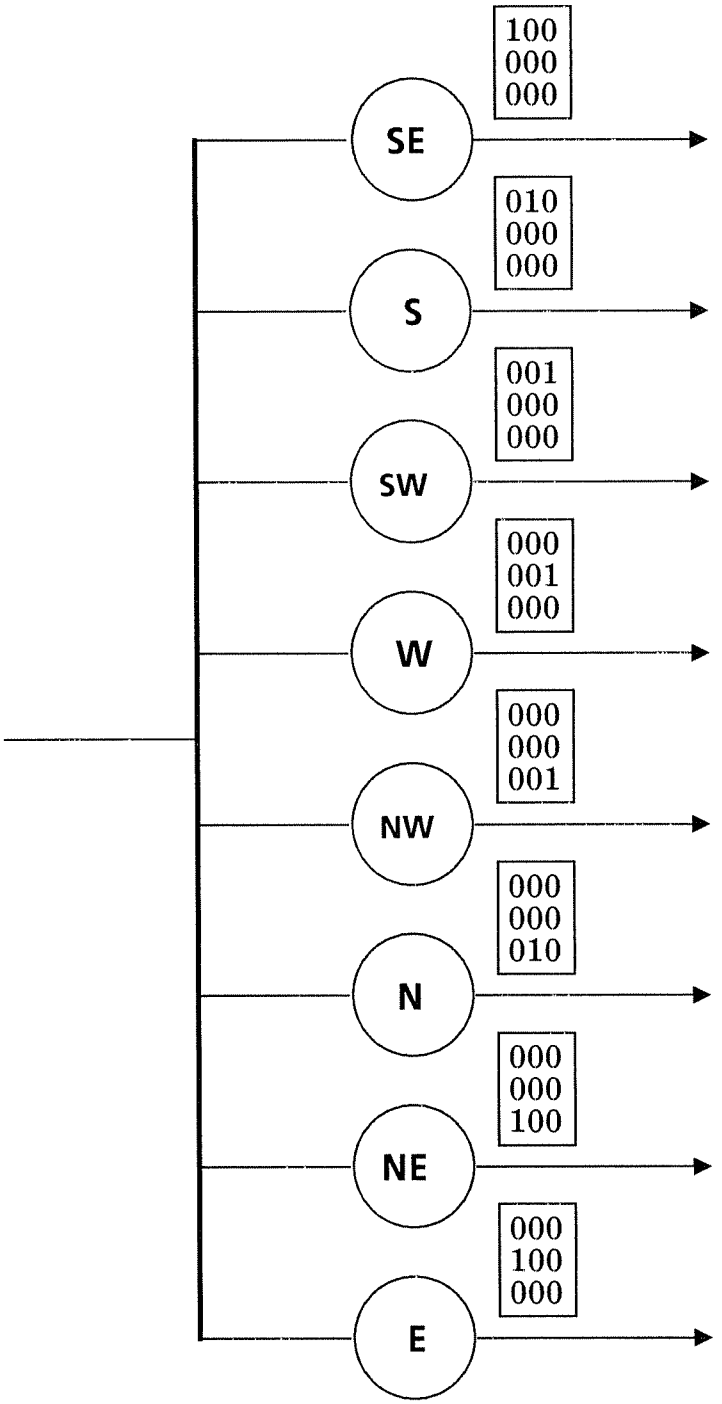


Fig. 3. Shifting an image one pixel in any one of eight directions using the NBR unit of PIPE.

Next, all the odd-numbered stages perform an OR operation on the individual bits (including the sign bit) of the pairs of images $e-a$ and $e-i$, $e-b$ and $e-h$, $e-c$ and $e-g$, and $e-d$ and $e-f$. Next in each odd stage the PRENBR LUT is used to determine if either the sign bit of the result is 1 or the rest of the bits are 0. Thus stage 1 produces $nonmax(a,e,i)$, stage 3 produces $nonmax(b,e,h)$, stage 5 produces $nonmax(c,e,g)$ and stage 7 produces $nonmax(d,e,f)$. Fig. 4 shows the network of operations required to compute $nonmax(a,e,i)$ from e. VBUS A is then used to send the result of stage 1 to stage 3 where an AND operation is done, and the result of stage 7 is sent to stage 4 using VBUS B where an AND operation is done with the output of stage 5. Meanwhile in stage 2, $e-T$ is performed to produce u. Finally, in the first half of stage 3 a final AND yields w which is ORed with u from stage 2 to produce v. In the second half of stage 3 v is inverted and ANDed with e to give the final result image. Fig. 5 shows the corresponding network for the complete algorithm.

3.2. Timing

The initial broadcast of the input image to all eight stages takes 1 time unit. The next time unit is used to shift this image appropriately and subtract each from the original. In the third time unit the nonmax results are computed in each direction. During the fourth time unit the partial conjunction of these results is performed and the test that e is above threshold is done. In the final time unit all partial results are combined and the original image is then used to produce the output image in which all nonmaxima have been suppressed. Hence a total of five time units are required to implement this peak detection algorithm on PIPE.

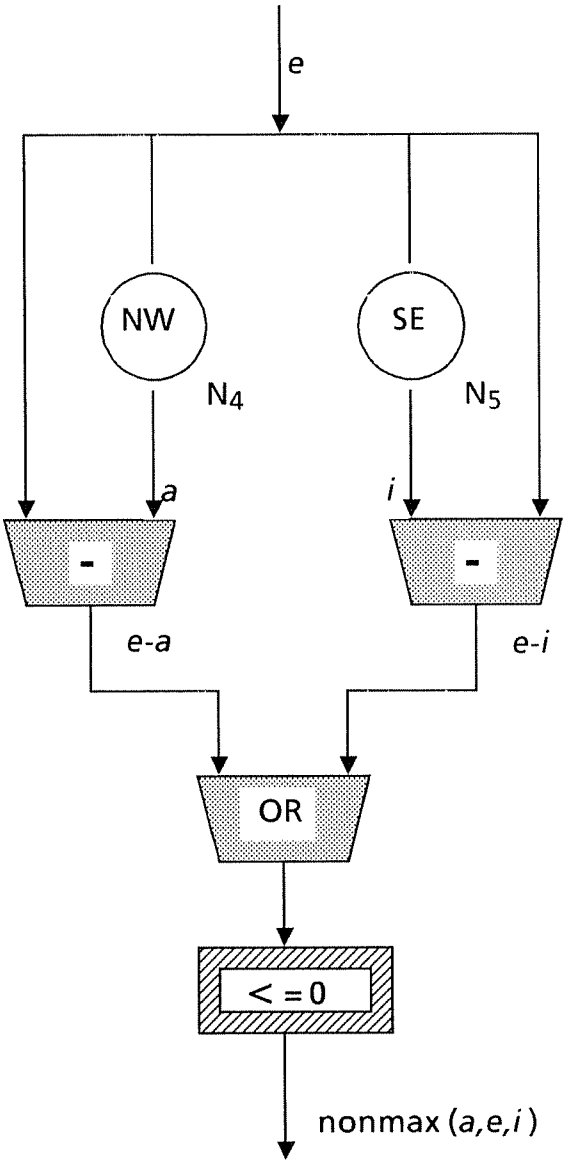


Fig. 4. Implementation of nonmax (*a*,*e*,*i*) in PIPE using 2 NBR's, 3 ALU's and 1 LUT. Arcs are labeled with their associated image names.

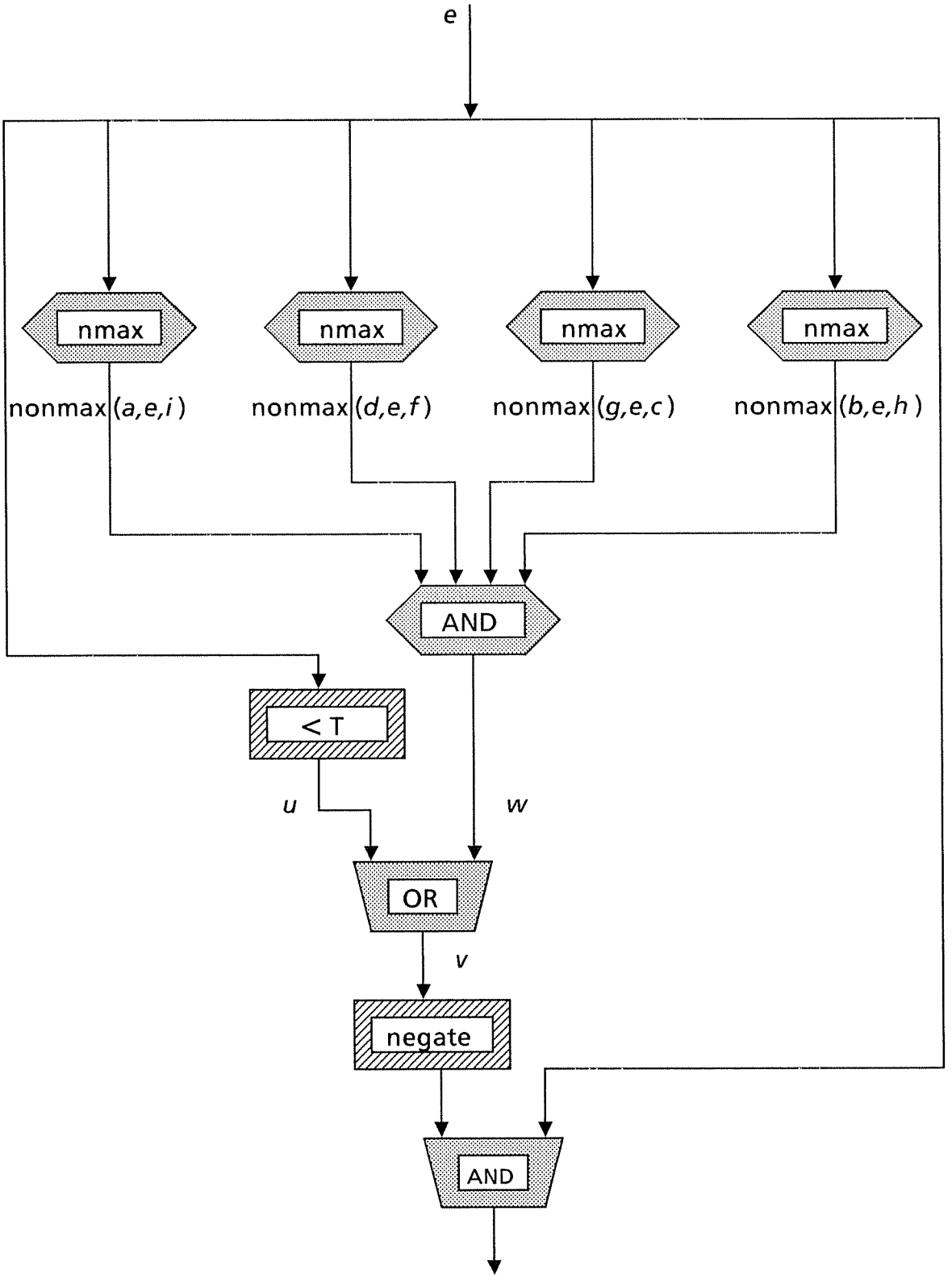


Fig. 5. Peak detection algorithm on PIPE. Six-sided nodes indicate non-primitive operations. The nmax nodes consist of operations shown in Fig. 4.

4. Median Filtering

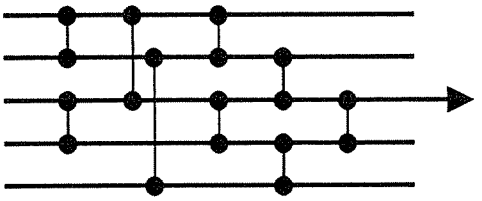
Median filtering is a powerful smoothing technique that does not blur edges. This nonlinear operator replaces the value at each pixel by the median value of its neighborhood. In this section we describe methods for performing median filtering using five point neighborhood:

```
  x
x x x
  x
```

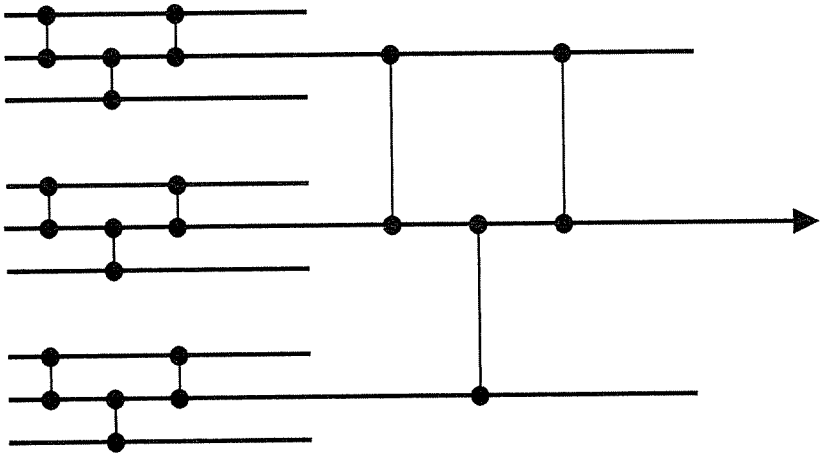
The basic operation required to perform median filtering is a comparator which takes two values and outputs the minimum and maximum. Collections of these comparators can be combined to implement sorting networks [4] which have a number of input lines equal to the number of points, n, in a given median filter. In our case, we need a network of size $n = 5$. Furthermore, since we are only interested in the median value, we can prune the sorting network so that only those comparisons which contribute to finding the median value are included. We call the resulting network a "median network." A median network for $n = 5$ is shown in Fig. 6(a). For large n the number of comparators required can be reduced by computing an approximate median value using a "median-of-medians" network [5]. For example, a median-of-medians network for $n = 9$ is shown in Fig. 6(b).

4.1. Implementation

A natural extension of median networks for n elements is median networks for n images. Each image is connected to an input of the median network. Corresponding pixels are synchronously pipelined through the network. The output of the network is an image in which each pixel is the median value of all the values at the given position in the input images. Thus the first step is to create a set of images in which the pixels in each median neighborhood are in register. This is achieved by shifting the original image to the required positions in different image planes as shown in Fig. 3. The resulting images are then input to a median network which



(a)



(b)

Fig. 6. (a) Median network for $n = 5$. Inputs are associated with the lines at the left end of the network. Vertical lines indicate comparators applied to the two values associated with the two input lines. The larger of the two values is output on the top line and the other value is output on the bottom line to the right out of each comparator.
(b) Median-of-medians network for nine input values.

performs the desired median operation at each pixel.

To implement a median network on PIPE we must describe how a comparator is implemented and how the comparator results are combined. Fig. 7 shows how the comparators are combined to produce a complete median network for $n = 5$ on three stages. Fig. 8 shows how the comparators are combined for a median-of-medians network for $n = 9$ using seven stages.

Comparators take two images as input and produce two images as output. Corresponding pixels in the two input images are compared and the larger pixel value is written to one output image while the smaller value is written to the other output image. To perform this operation, the two input images are first subtracted and the resulting image of sign bits forms a mask, where 1's indicate positions where the larger pixel is in the first image and 0's indicate positions where the larger pixel is in the second image. Assuming each stage has at least three image buffers to store the two input images and the mask, we can use the mask with PIPE's region of interest (ROI) mode to direct the greater pixels to one output image and the rest of the pixels to the other output image. Fig. 9 shows this comparator network for PIPE.

4.2. Timing

A comparator takes one time unit if we assume that the two input images are already stored in buffers in a stage (e.g. via the two wildcard buses) and are also input to the first half of the stage via two input paths (forward, recursive or backward). Alternatively, a comparator can be implemented in two time units using one of the wildcard buses to load one image and either the forward, recursive or backward path at the second time unit to load the other image. A three-time-unit solution is possible without using either wildcard bus.

In general, the timing of the median computation depends on the topology of the median network. The median network for the $n = 5$ case can be implemented with single-unit comparators since at most two comparisons need be done at a time (there are four wildcard paths; each comparator uses two of them). Thus five time units suffice for this case. The 9-input

median-of-medians network takes seven time units instead of six since three concurrent single-time-unit comparators cannot be implemented on PIPE. The net time delay of one time unit is achieved by careful placement of some two-time-unit comparators as shown in Fig. 8.

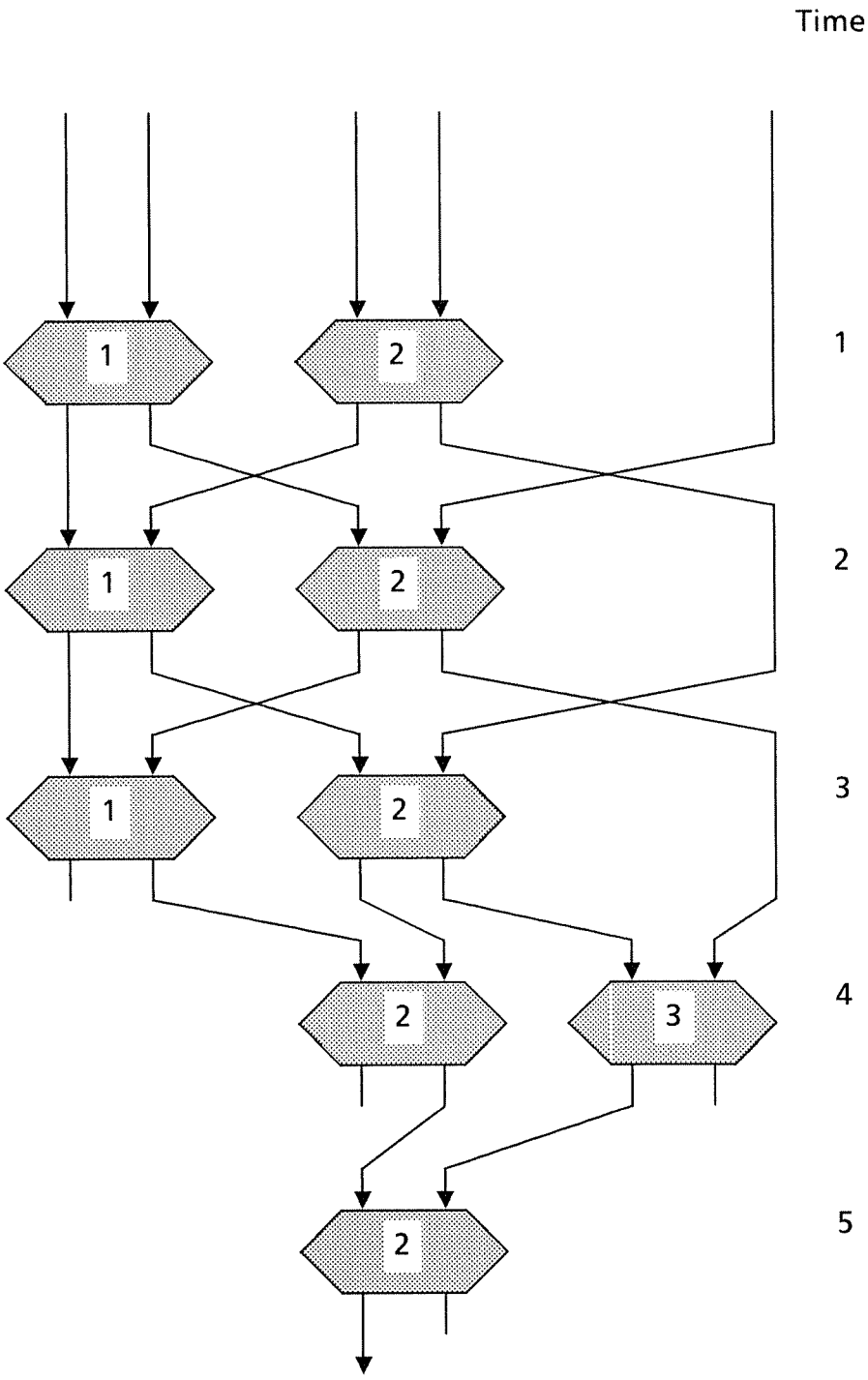


Fig. 7. Median network for 5 images using 3 stages. Each node is a comparator labeled with the stage that performs it.

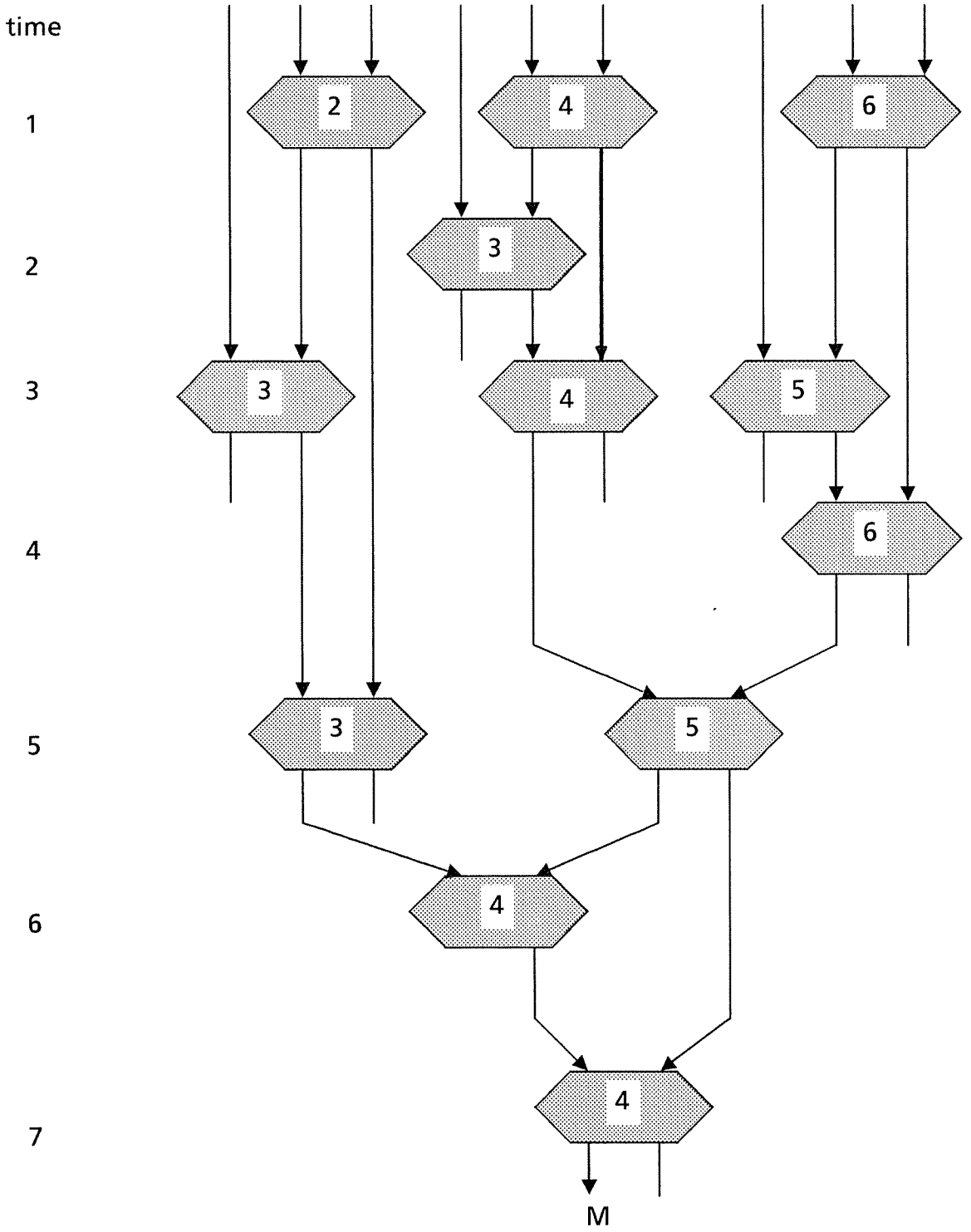


Fig. 8. Median of Medians network for 9 images mapped onto PIPE stages 2-6. Each node is a comparator. The image of medians occurs at M in stage 4 at the end time unit 7. Nodes are labeled with the stage number in which the comparison is performed.

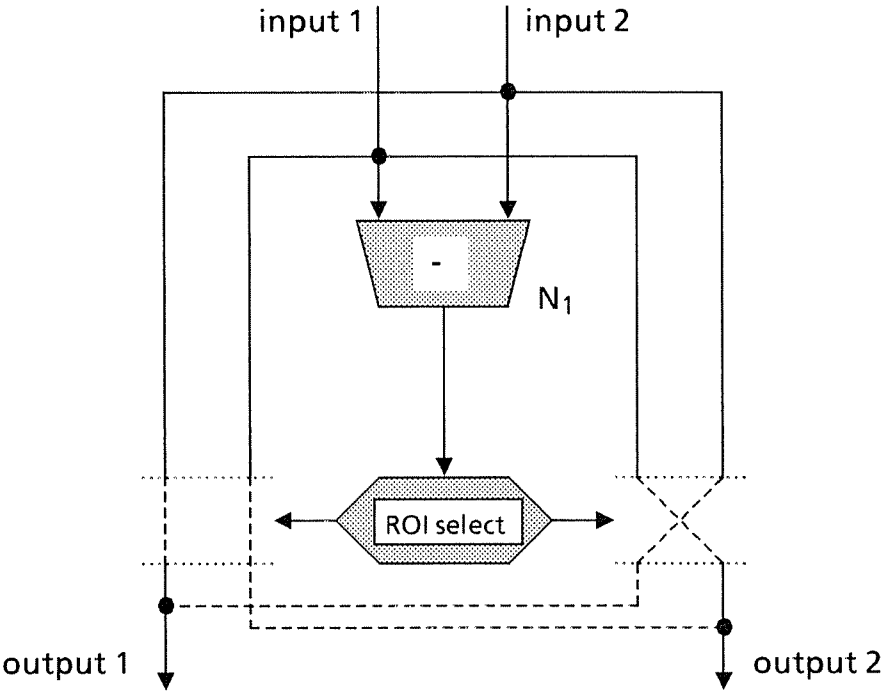


Fig. 9. A comparator. The ROI selector determines, on a pixel-by-pixel basis, the destinations of two corresponding pixels among a pair of output paths. The greater of the two pixel values is sent to output 1, and the lesser is sent to output 2.

5. Thinning

Thinning is an iterative, local process which successively shrinks an object by removing border points. The result is a skeleton-like representation of the object which is useful for shape analysis and description. In this section we describe the implementation of a version of the one-pass thinning algorithm developed by Chin *et al.* [6]. In this algorithm a set of masks is used to determine which border points can be safely removed without locally disconnecting an object. We assume the input image is binary where pixels which are part of an object have value 1 and background pixels have value 0. Objects are assumed to be 8-connected sets of 1's. Fig. 10 shows the eight masks corresponding to the cases in which the center object pixel can be removed.

The first step of the algorithm is to match all eight masks at each pixel and if any of them matches, the current (center) pixel is set to 0; otherwise it retains its previous value. Unfortunately, this step will delete objects that are only two pixels wide. Therefore, the second step "restores" two pixel wide object segments. In particular, we match two "restoring masks" at each pixel to determine if a given object pixel should not be removed despite the results of step 1. These two masks are:

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

These two steps are applied repeatedly until all objects in an image are shrunk to width less than or equal to 2. Notice that because the shrinking process is stable, i.e., iterating after an object has shrunk to a thin skeleton will not further change the object, we can most easily implement this algorithm by iterating for a fixed number of cycles corresponding to the maximum width of any object in the image.

0 0 0	0 1 x	x 1 x	x 1 0	x 0 0	0 0 x	x 1 x	x 1 x
1 1 1	0 1 1	1 1 1	1 1 0	1 1 0	0 1 1	0 1 1	1 1 0
x 1 x	0 1 x	0 0 0	x 1 0	x 1 x	x 1 x	0 0 x	x 0 0

Fig. 10. Eight thinning masks which are applied at each iteration of the algorithm. x indicates a "don't care" value.

5.1. Implementation

The input binary image is first broadcast to the first six stages of PIPE. In stages 1 through 4 the eight thinning templates are applied simultaneously, two per stage. As they exit from the NBR's, pairs of results are immediately ORed together in the same stage. At the next time unit the results of the first four stages are ORed together in stage 2. That is, the image from stage 1 is output via the forward path to stage 2, the image from stage 2 is output via the recursive path, the image from stage 3 is output via the backward path, and the image from stage 4 is output via wildcard bus VBUS A. At the next time unit the resulting binary image is subtracted from the original binary image in stage 2. The 4×1 and 1×4 restoring templates are simultaneously applied, requiring two time units, in stages 5 and 6. The results of these two matches are ORed together, and finally, at the fourth time unit, the result of stage 2 is ORed with the restored image to produce the output of one iteration of the thinning process. This image is then broadcast to the first six stages to begin the next iteration.

5.2. Timing

Each iteration takes 4 time units as described above. If the maximum width of any object in the image is w, then $w/2$ iterations are sufficient, requiring a total of $2w$ time units.

6. Hough Transform

The Hough transform is a technique for detecting global image features such as curves and lines by applying a coordinate transformation to an image such that all points which are part of a single instance of the feature, map into a point in the transformed space [7]. For example, lines at all possible positions and orientations can be specified using two parameters, ρ and θ, defined by

$$x \cos\theta + y \sin\theta = \rho$$

where θ is the angle the line makes with the y-axis and ρ is the distance from this line to the origin. Thus a point in (ρ, θ) space corresponds to a line in (x, y) space. Using this relation between image space and parameter space, the Hough algorithm for line detection consists of three steps. First, initialize an accumulator array, $A(\rho, \theta)$, to 0 for all possible values of ρ and θ. Second, for each edge point detected in an input image at coordinates (x, y), increment all points in A which satisfy the above equation. Finally, detect local maxima in A to determine the locations of lines in the image.

In this section we describe how PIPE can be used for line detection using the Hough transform. We assume that the input image is binary, a 1 indicating the presence of an edge point and 0 indicating no edge. Assume ρ is quantized to R possible values, and θ is quantized to T values.

6.1. Implementation

The Hough transform can be implemented on PIPE by iterating through all T possible values of θ. Thus, each iteration produces a row of the values in the accumulator array A. Given θ, ρ is computed for each possible value of x and y using the above equation, and the result is stored at the corresponding pixel. The number of times a particular ρ value occurs in this image is the final value of the accumulator cell $A(\rho, \theta)$. Hence the histogram of this image corresponds exactly to one row of A. The operation of actually storing the histogram of values into the

appropriate row of A will be performed by the host in this implementation.

To compute ρ at each (x,y) position, we first multiply (at node N_1 in Fig. 11) the input binary image by a constant image in which each pixel contains its x-address. The resulting masked x-address image is then multiplied (at node N_3) by the constant $\cos\theta$ (θ is fixed for each iteration). Similarly, and in parallel, the masked y-address image (produced at node N_2) is multiplied (at node N_4) by the constant $\sin\theta$. These two product images are then summed (at node N_5) to obtain ρ for each pixel. Finally, the resulting ρ image is input to the ISMAP processor which computes the histogram of ρ values and outputs this histogram to the host. The host stores these values in the appropriate row of array A. After T iterations the host performs the final step of detecting peaks in A.

6.2. Timing

It takes one time unit to mask both the x- and y-address images with the input image. This can be done in any two stages. At the second time unit the x-masked image is multiplied by $\cos\theta$ in one stage while the y-masked image is multiplied by $\sin\theta$ in the same stage. Next, in the same stage and time unit, these two results are summed. The resulting image is sent to ISMAP which produces a histogram of the ρ values at the third time unit and then outputs this structure to the host. Using three consecutive stages, $i-1$, i and $i+1$, the first time unit requires use of stages $i-1$ and $i+1$ and the second time unit uses only stage i. Only ISMAP is used at the third time unit. Consequently, the operations for each successive θ value can be pipelined so that ISMAP will output a row of the accumulator array at each time unit beginning at time unit 3. Hence the total number of time units required is $2+T$, where T is the number of θ values considered in Hough space.

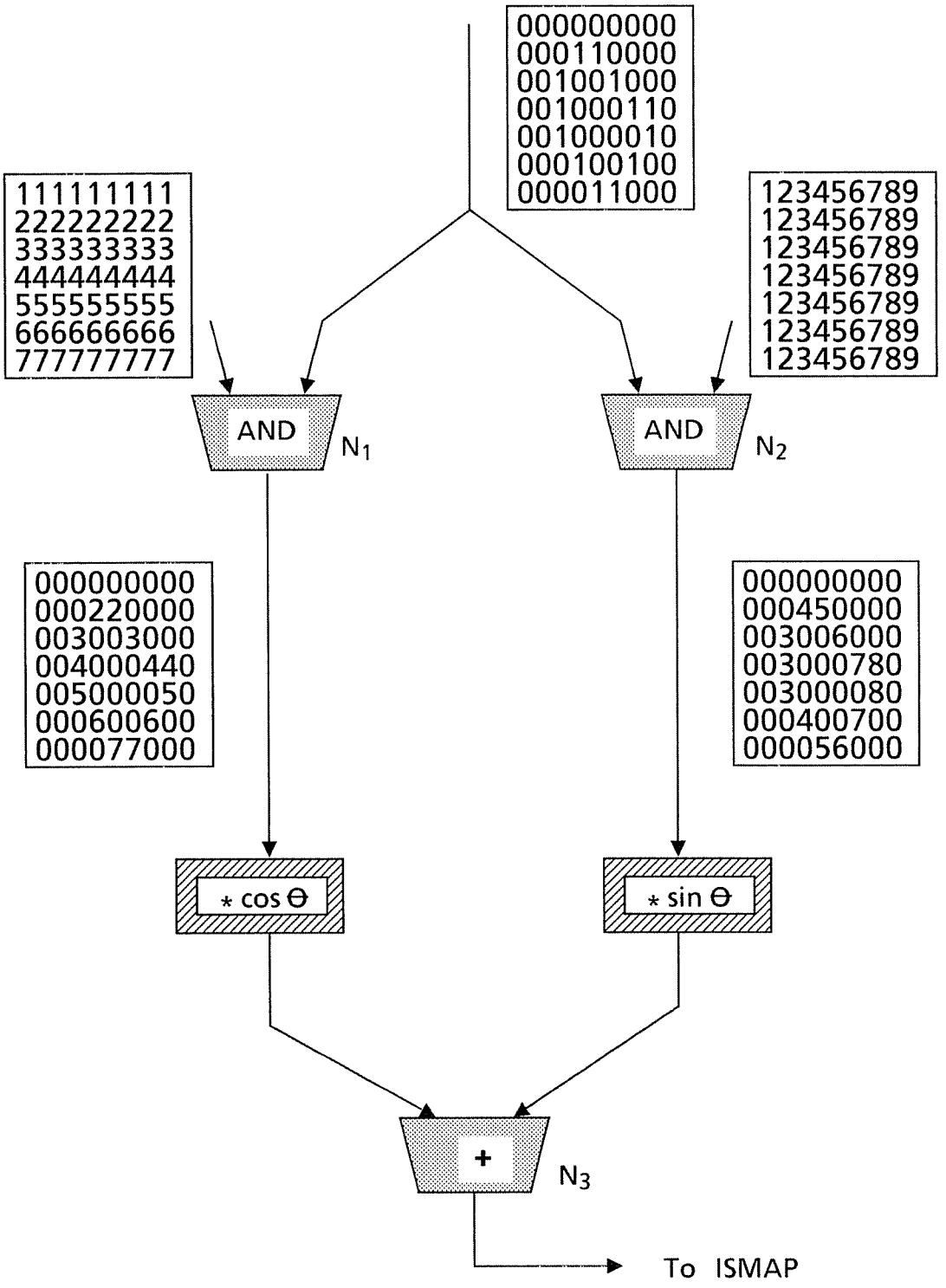


Fig. 11. Computation of one row of Hough space. The inputs to the LUT's remain fixed while the LUT's are successively reprogrammed to multiply the input image by cosΘ and sinΘ for each value of Θ.

7. Photometric Stereo

One method for recovering the local surface orientation at each point in an image is called photometric stereo [8]. It can be applied to scenes containing surfaces with known reflectance properties using multiple images of the same scene but under different lighting conditions. For each surface type and light source position we can precompute the reflectance map $R(p, q)$ which specifies the brightness at an arbitrary pixel as a function of surface orientation, specified by p and q. (p, q) is called the gradient of the surface, the two components representing the slopes of the surface in the $x-$ and $y-$directions. Although in general the mapping from brightness to surface orientation is not unique, if we consider three images, f_1, f_2 and f_3 of the same scene, but under different lighting conditions, we can uniquely solve for the two unknowns, p and q, at each pixel. Thus we have a set of three equations with two unknowns of the form:

$$R_i(p, q) = f_i(x, y), \quad i = 1, 2, 3$$

for each pixel at coordinates (x, y).

7.1. Implementation

Given three images, f_1, f_2 and f_3, and their associated reflectance maps, R_1, R_2 and R_3, the goal is to recover (p, q) at each pixel. Our implementation using PIPE will be to iterate through all possible pairs of (p, q) values and identify those pixels in the image that have this surface orientation. That is, given $p = a$, $q = b$, there is an associated *unique* triple of reflectance values, $(R_1(a, b), R_2(a, b), R_3(a, b))$. The three images will be searched for all occurrences of this particular triple of brightness values and each such pixel will be assigned the current orientation value (a, b).

For example, consider the implementation for one iteration, corresponding to the surface orientation $p = 0.6$, $q = 0.9$. The left half of Fig. 12 shows all the operations performed in one iteration. Assume the three reflectance maps imply the triple of brightness values

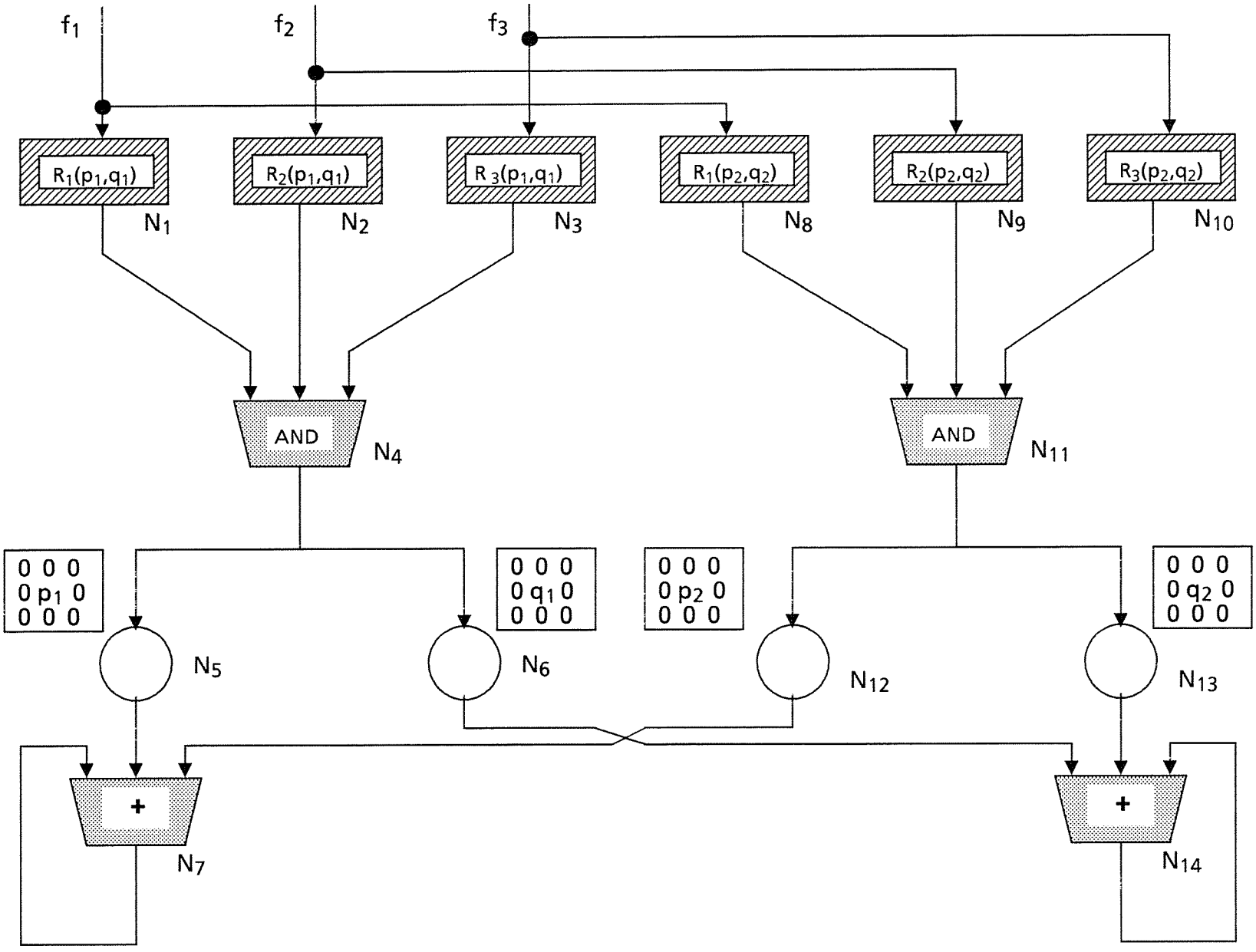


Fig. 12. Photometric Stereo algorithm given 3 images f_1, f_2, f_3 and their associated reflectance maps R_1, R_2, R_3. The left half of the figure finds all pixels having orientation (p_1, q_1). The right half of the figure finds the orientation (p_2, q_2). The results are accumulated in the images associated with the operations at N_7 and N_{14}.

$R_1(.6, .9) = 0.6$, $R_2(.6, .9) = 0.7$ and $R_3(.6, .9) = 0.3$. During the first time unit image f_1 is input to stage 1, f_2 is input to stage 2, and f_3 is input to stage 3. At the second time unit the three images are input to stage 2 using their respective stages' forward, recursive, and backward paths. The forward-path LUT is used to immediately mark those pixels in image f_1 which have brightness value 0.6 (node N_1 in Fig. 12), outputting a 1 at each such pixel and a 0 otherwise. The recursive-path LUT in stage 2 creates a similar mask image by identifying those pixels in f_2 having brightness value 0.7 (node N_2). The backward-path LUT also creates a mask by marking pixels in f_3 having value 0.3 (node N_3). The ALU's in the first half of stage 2 then compute the conjunction of these three binary mask images (node N_4). This image is multiplied by 0.6 (= p) and by 0.9 (= q) and the two resulting images are then added into their respective output accumulator images corresponding to p and q (at nodes N_7 and N_{14}, respectively). Multiplication by p and by q is achieved by using the two neighborhood operators (N_5 and N_6) in the second half of stage 2. The mask used simply weights the center pixel by the desired value and the rest by 0. These two images are sent to the p and q result images via the two wildcard buses.

With each iteration more and more pixels will have their (p, q) values determined and filled into the two result images. Since three stages suffice to perform the above operations, other stages can be used to simultaneously find points that have some other (p, q) value. In Fig. 12 we show a second complete set of operations being performed at stages 4-6 as shown by nodes N_8 through N_{13}. Notice that node N_7 adds the results of nodes N_5 and N_{12}, thereby simultaneously filling output values for two disjoint subsets of pixels corresponding to two different (p, q) pairs. Similarly, node N_{14} adds the results of nodes N_6 and N_{13}. Notice the recursive paths at the nodes N_7 and N_{14}. Each one causes the two new values to be added to the result image that was output by the node at the previous cycle. Therefore when all the (p, q) pairs have been tested, the final p and q result images will be stored in the image buffers associated with nodes N_7 and N_{14}.

7.2. Timing

Using the wildcard buses it takes three time units to initially input f_1 to stages 1 and 5, f_2 to stages 2 and 4, and f_3 to stage 3. Thus f_1, f_2, and f_3 are all permanently accessible at both stages 2 and 4. At each time unit, stage 2 produces a p image which is sent to stage 7 to be added to the p accumulator image, and stage 4 produces a q image which is sent to stage 8 to be added to the q accumulator image. Thus the entire process takes $3 + PQ/2$ time units, where (p,q) space is of size $P \times Q$.

7.3. Comments

The method could be improved by using a coarse-to-fine strategy for the values of (p, q). A first pass could be done using a coarse quantization of both p and q; the second pass would concentrate only on refining the "interesting" regions of (p, q) space as determined by the host using the output of ISMAP to find peaks or clusters of points.

8. n-bit Point Operations

A point operation is any operation on an image where the value of the resultant pixel depends only on the value of the corresponding operand pixel. Such operations are useful for performing selection functions, trigonometric functions, multiplication by a constant, bit pattern rotation, etc. More generally, when there is more than one operand, functions such as maximum, minimum, multiplication, division, and change of parameterization can be performed as point operations using sufficiently large word sizes.

If enough memory is available to store the result of an arbitrary point operation function for each possible input value (typically 2^8), then a lookup table can be used to implement the function directly. The advantage of this method is that the time needed to perform the operation on one pixel is the time required for one memory access. To implement an arbitrary function of two inputs, each containing an 8-bit value say, would require 2^{16} different input values. In this case, a lookup table of the desired size is probably not available. We describe in this section how to simulate a large lookup table by successive re-use of one or more strictly smaller lookup tables. In this way n-bit precision in point operations can be achieved for n larger than the data word size.

Assume, for example, that only one 8-bit lookup table is available and that there are 2^{13} different input values. By successively reprogramming the 8-bit lookup table 2^5 times all 2^{13} different input values are considered (see Fig. 13). Each table can look at only eight of the thirteen bits. Each lookup implements the result assuming a fixed value for the other five bits of the input. However the lookup table is applied to the 8-bit values regardless of the values of the other five bits. One way to correct these results is to first mask out (by setting to 0) those pixels which do not have the correct five-bit pattern for the current iteration. Finally, we must sum the resulting 2^5 images. Alternatively, after each lookup table operation we can immediately add the masked result to an accumulator image. If more lookup tables are simultaneously available, they

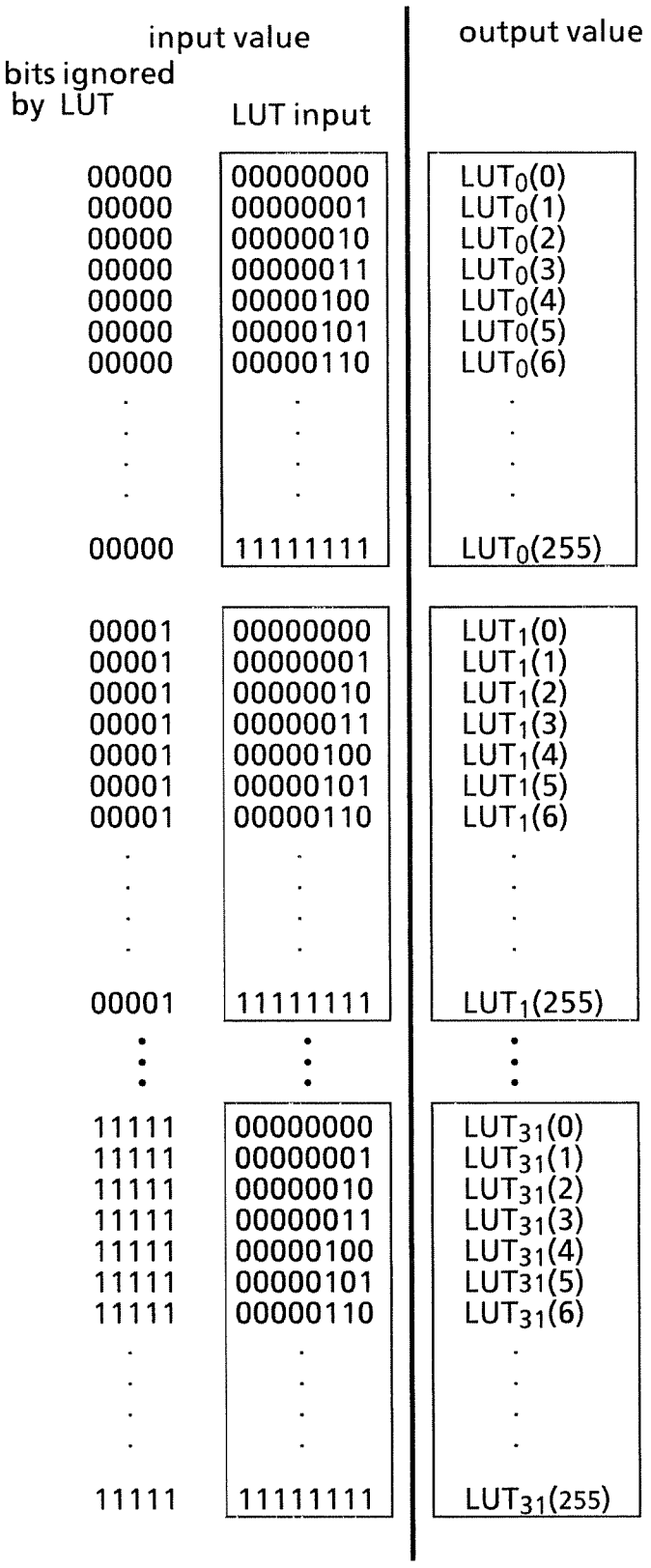


Fig. 13. 13-bit point operation with 8-bit output using an 8-bit LUT 32 times.

can be used to partition the work and save time. The corresponding accumulators will have to be added together finally as was done in the previous section.

8.1. Implementation

Let n be the number of input bits and m be the number of output bits. Viewing the output word as a sequence of $\lceil m/8 \rceil$ bytes, we compute $\lceil m/8 \rceil$ independent images in sequence, each encoding a byte of the result. Henceforth we will assume $m \leq 8$. If $n \leq 8$ as well, the operation can be implemented using a single LUT on PIPE. If $n \leq 12$, the operation can also be implemented directly using PIPE's TVF LUT. Since there are two tables available when the TVF LUT is programmed with 12 input bits, the ROI mode allows us to take advantage of one additional bit of the input to indicate which of the tables should be used. Thus if $n \leq 13$ the operation can still be done in one stage.

When $n > 13$, 2^{n-13} iterations are needed. The ROI mode can also be used to select positions based on the $n-13$ ignored bits so that the TVF LUT is applied only to positions of the input having a certain combination of the ignored bits. Pixels at the other positions are left at their previous (initially 0) values. Of course we require that $n-12 \leq 8$ in order for the ROI mask image to have 8-bit values. If $n > 20$, the scheme described in the example above should be used. That is, the TVF LUT is applied to all positions (the ROI mode being used only to decide which TVF LUT table to apply); 0's are stored at positions where the input failed to have the proper combination of the eight ignored bits. Each iteration still takes one time unit.

If there are between nine and sixteen bits ignored (i.e. $21 < n < 30$), then each iteration takes two time units. In this case an iteration involves computations that cannot all be performed in one time unit; i.e., selecting pixels from two images of ignored input values, ANDing the results together, ANDing this with the output of the previous TVF LUT, and finally adding the result to the accumulator. If $29 < n < 38$, there are three images of ignored input values. Nevertheless two time units still suffice for each iteration; there are enough resources for one

more selection and conjunction before the addition of the result to the accumulator image.

With s stages available the work is essentially divided by s since all the iterations are independently computable.

8.2. Timing

Assuming n is the number of input bits and m is the number of output bits, we compute $\lceil m/8 \rceil$ independent images in sequence, each encoding a byte of the result. If there are s stages available, each 8-bit result requires $2^{n-13}/s$ iterations plus the time to sum together the accumulators. Let $A(s)$ be the number of time units needed to sum s images located in s stages. $A(s)$ is bounded above by $\log_2 s$. If $13 < n \leq 21$, the entire point operation takes $\lceil m/8 \rceil \times (2^{n-13}/s + A(s))$ time units. If $21 < n < 38$, $\lceil m/8 \rceil \times \left\lceil 2^{n-12}/s \right\rceil + A(s)$ time units are required.

In some cases the $\lceil m/8 \rceil$ result images can be computed in parallel. For example if s is divisible by $\lceil m/8 \rceil$, the entire point operation takes $\left\lceil 2^{n-13} \times \lceil m/8 \rceil + A(s/\lceil m/8 \rceil) \right\rceil$ time units if $13 < n \leq 21$, and $\left\lceil 2^{n-12} \times \lceil m/8 \rceil + A(s/\lceil m/8 \rceil) \right\rceil$ time units if $21 < n < 38$.

8.3. Comments

One application of this method is the photometric stereo problem described in Section 7. In that case the 8-bit value of the resultant pixel depends only on the values of the corresponding pixels in each of three input images. Hence $m = 8$ and $n = 24$. If $s = 8$ stages are available, the entire operation takes 259 $(= 2^8 + 3)$ time units. However, if we first reduced the quantization of the input images to seven bits, then $n = 21$ and the entire operation takes 35 time units plus the time units to properly reorganize the 21 significant input bits in the three images.

9. Computing Multiresolution Representations

This section considers the problem of detecting edges at multiple resolutions of an image. The two ideas underlying their detection are that 1) brightness changes occur at different scales in an image, and 2) a sudden brightness change will give rise to a peak or trough in the first derivative or equivalently a zero-crossing in the second derivative. We will detect zero-crossings at each level of a pyramid of images. Each image is obtained from the original by approximating the result of a Laplacian operator applied to a Gaussian smoothed version of the original image. This is done at multiple resolutions resulting in a pyramid of images.

We describe the construction of three multiresolution representations of an image: the Gaussian pyramid, the Laplacian pyramid, and the zero-crossing pyramid. The algorithms we implement are similar to those defined by Burt [9]. The Gaussian pyramid is a sequence of low-pass filtered images, $G_0, G_1, ..., G_n$, where G_0 is the input image. G_{i+1} is constructed by sampling G_i at every other row and column, and then applying a low-pass Gaussian filter to the sampled image.

The Laplacian pyramid is a sequence of images $L_0, L_1, ..., L_{n-1}$. L_i is constructed by expanding G_{i+1} and subtracting the result from G_i. The expansion process is the reverse of the sampling process: We first move all pixels in G_{i+1} into positions with odd row and column numbers in a new higher resolution image and then this image is convolved with a low-pass Gaussian filter to produce the expansion of G_{i+1}. This expansion and differencing of Gaussians approximates the Laplacian operator.

The zero-crossing pyramid is computed from the Laplacian pyramid by detecting all pixels at each level where a zero-crossing occurs at the corresponding pixels in the Laplacian image. A zero-crossing occurs at a pixel if its sign differs from any of its eight nearest neighbors; thus detecting zero-crossings is a 3×3 neighborhood operation. Different scales of brightness changes are detected by zero-crossings at different resolutions. The zero-crossing pyramid will also be

used in the next two sections for performing coarse-to-fine operations in stereo vision and two-dimensional object recognition.

9.1. Implementation

A PIPE implementation is now briefly described which computes the levels of the zero-crossing pyramid in sequence from finest to coarsest resolution. Although each level must be explicitly programmed, the algorithm is illustrated in Fig. 14 as an iterative process, producing one level of the pyramid at a time starting at the base. A 5×5 low-pass filter will be used for computing the Gaussian pyramid; details are deferred until the end of this section. Suffice it to say that a 5×5 neighborhood operation can be implemented using two consecutive stages in 3 ½ time units.

Initially, image G_0 is stored in stages 1, 5 and 6. At the ith iteration, one level in each of the three pyramids is constructed, i.e., G_{i+1}, L_i and the ith level of the zero-crossing pyramid. In stages 5 and 6 G_i is sampled and the 5×5 low-pass filter is applied. At the first half of the fourth time unit G_{i+1} is produced. This image is then sent to four different stages for the following purposes: the start of the next iteration in stages 5 and 6, the expansion of G_{i+1} starting in stage 4, and storage for later use in stage 1. To compute L_i, G_{i+1} must be expanded and then subtracted from G_i. Stage 4 doubles the resolution of G_{i+1} using PIPE's resolution mode, and, using ALU C, this is ANDed with a Boolean mask image containing 1's at coordinates with two odd values. The result is sent via a wildcard bus to stages 2 and 3 where a 5×5 neighborhood operation is performed to produce the expansion of G_{i+1}. This image then goes to stage 1 to be subtracted from G_i, which was stored there 7 time units earlier. Thus image L_i is produced in the first half of stage 1. A 3×3 neighborhood operator is then applied to produce the ith level zero-crossing image. In the same manner, successive iterations begin at times 1, 4, 7, ..., and zero-crossing images are produced at times 8, 11, 14, ...

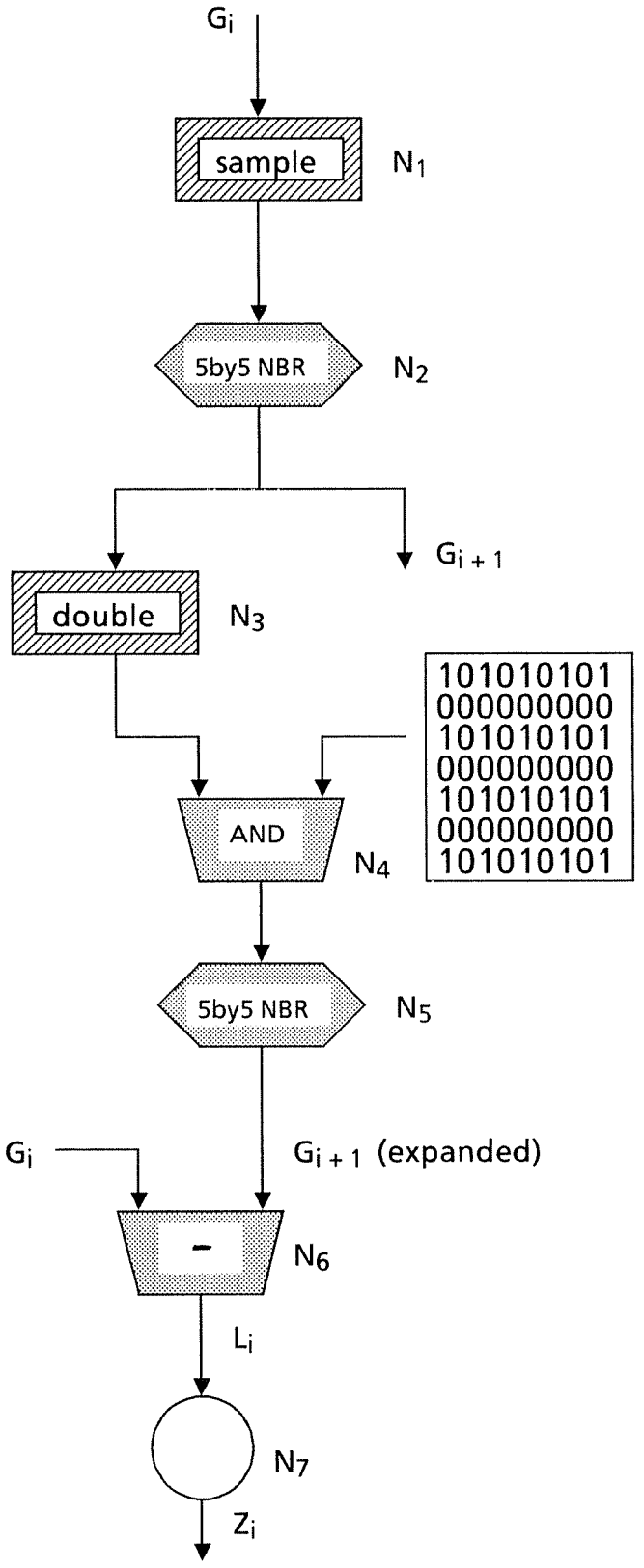


Fig. 14. Computing the i^{th} level of the Laplacian and Zero-crossing pyramids, and the $i + 1^{st}$ level of the Gaussian pyramid. The sample and double operations are primitive to PIPE.

With the exception of the 5×5 neighborhood operation, all the operations above are primitive to PIPE. The required Gaussian filter requires a 5×5 mask and therefore cannot be performed by a single functional unit of PIPE. As shown in Fig. 15, four 3×3 sub-mask convolutions are necessary (in general); the partial results are put in register by diagonal shifting and then added together to produce the result of the 5×5 convolution. Since these neighborhood operations represent natural bottlenecks in the algorithm above, we have emphasized speed in our implementation at the expense of extra stages and buses. Using two consecutive stages it takes 3 time units to perform all of the operations associated with nodes N_1 through N_{10} in Fig. 15: N_1 through N_4 are done in the first time unit, N_5 and N_6 in the second, and N_7 through N_{10} in the third. Wildcard buses must be used to pass the results of nodes N_1 and N_2 to nodes N_7 and N_8. At the beginning of the fourth time unit the results of nodes N_9 and N_{10} are sent to stages 1, 4, 5, and 6 where N_{11} is performed.

9.2. Timing

The number of time units required to compute a zero-crossing pyramid with n levels is $3n+5$ since iterations are pipelined, beginning every 3 time units and ending 8 units later.

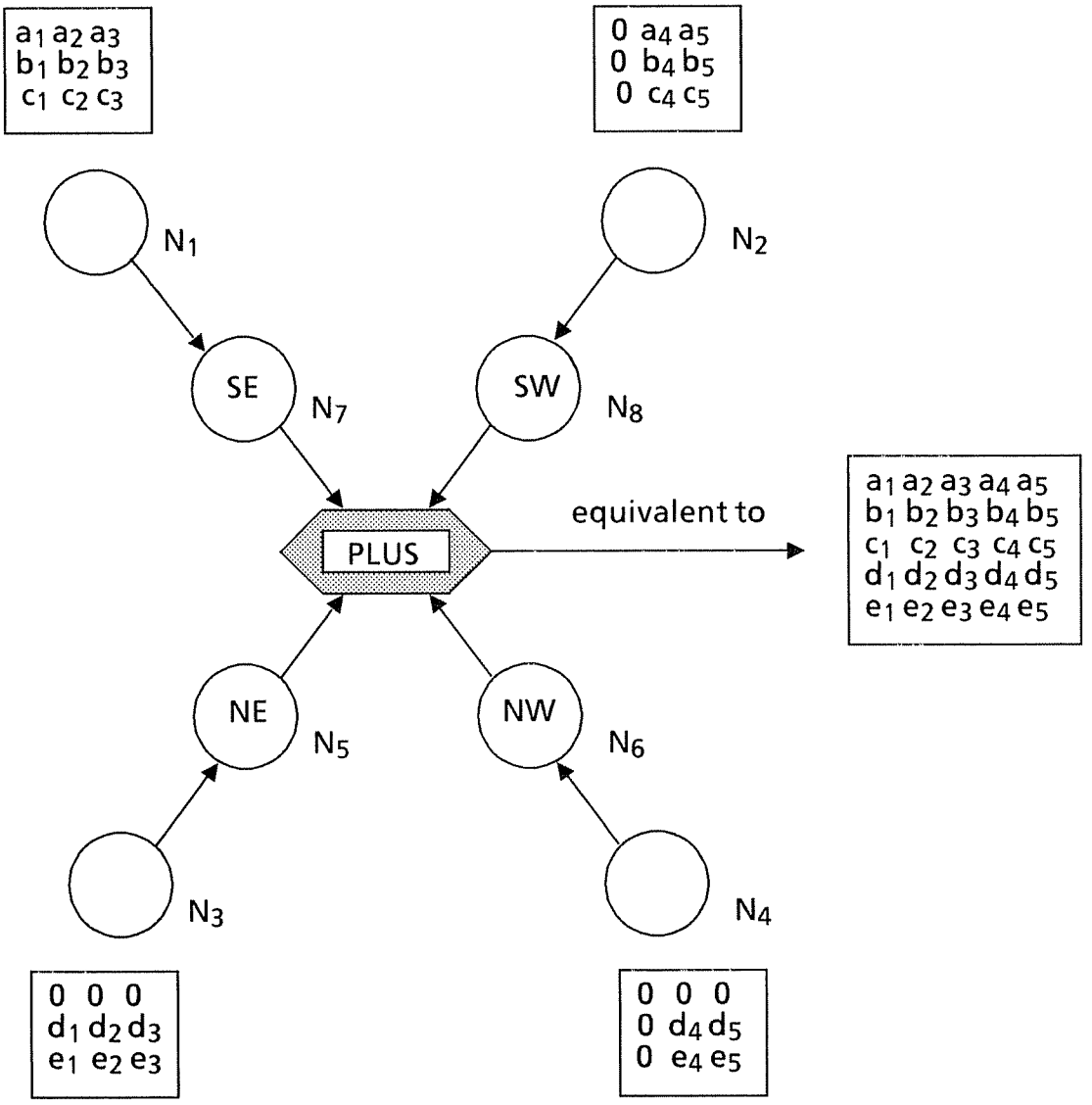


Fig. 15(a). Computation of a 5 x 5 convolution operation using four 3 x 3 convolutions.

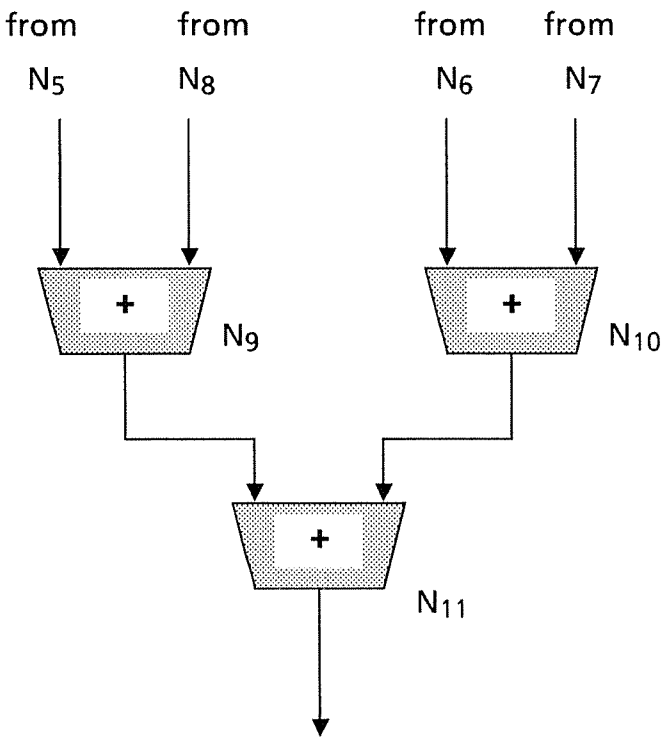


Fig. 15(b). Expansion of the PLUS node in (a).

10. Stereo Vision

Stereo vision requires three steps: (1) the point in one image corresponding to the projection of a point on a surface is located, (2) the point in the other image corresponding to the projection of the same surface point is located, and (3) the difference in the projection of the corresponding points is used, together with estimates of the parameters of the imaging geometry, to determine a measure of the distance to the surface point. In this section we describe a PIPE implementation of a version of the Marr-Poggio stereo algorithm [10]. The Marr-Poggio algorithm for solving the correspondence problem (step 2 above) can be best summarized as a feature-point-based matching procedure using a coarse-to-fine control strategy to limit the search space of possible matches. The main steps of the implementation are:

1. *Image filtering*: Laplacian pyramids of the left and right images of a stereo pair are constructed as described in Section 9.

2. *Symbolic Feature Detection*: Zero-crossings in the filtered images mark the locations of significant brightness changes in the original image at different resolutions and define the basic representations used for matching. An implementation of this step was also described in Section 9.

3. *Matching*: Assuming epipolar lines are horizontal, matching can be restricted to corresponding rows of the left and right zero-crossing images. For simplicity the orientations of the zero-crossings are not considered. Matching proceeds in a coarse-to-fine manner. The idea is to use a sparse representation of the features using a coarse level of the zero-crossing pyramid for the initial matching of points. The reduction in the number of feature points and the number of image sample points greatly reduces the search space required for matching. The initial coarse-level match can then be used to constrain the matching of finer detailed representations, again reducing the search space of the matching process while allowing finer detailed disparity information to be computed.

Feature point matching at each level of resolution (except the coarsest) assumes we are given an estimate $d_i(x,y)$ of the horizontal disparity at every position in the image, computed at the previous level of resolution. Given a zero-crossing in one image (say the left) at position (x,y), the search for a matching zero-crossing in the right image is constrained to the region $\left\{ (x',y) \mid x + d_i(x,y) - w \leq x' \leq x + d_i(x,y) + w \right\}$, where w is the width of the central region of the (approximate) Laplacian filter used to produce the zero-crossing images at the current resolution. Once this matching has been performed at a level, the new disparities are used to align the images and the same process is then begun at the next finer resolution.

10.1. Implementation

Implementation of steps 1 and 2 of the algorithm was described in Section 9. We will assume that the zero-crossing pyramids for the left and right images are available for the matching step described below.

Since the process involves the same operations at each resolution, we describe just one level. Let L_i and R_i be the left and right zero-crossing images at the current resolution, and let L_{i+1} and R_{i+1} be those at the next finer resolution. Each iteration consists mainly of computing the estimate $d_{i+1}(x,y)$ given $d_i(x,y)$, L_i and R_i. Due to the sparseness of the zero-crossing images and the need to compare pairs of pixels which are not in register, the host rather than PIPE is used to find matching zero-crossings (see comments below). ISMAP summarizes the sparse zero-crossing images in order to save the host from having to search for feature points. As the host does the matching it computes the new disparities and updates $d_i(x,y)$ where necessary. The result is sent back to PIPE, where $d_{i+1}(x,y)$ is then constructed.

ISMAP is designed to quickly compute the gray level histogram of an image and store with each "bin" of the histogram the addresses of the contributing pixels. The histogram of a binary zero-crossing image is not very useful, however. Therefore, each zero-crossing image is first

modified by ANDing it with the following image:

$$\begin{array}{cccccccc} 1 & 1 & 1 & 1 & 1 & 1 & 1 & \ldots \\ 2 & 2 & 2 & 2 & 2 & 2 & 2 & \ldots \\ 3 & 3 & 3 & 3 & 3 & 3 & 3 & \ldots \\ & & & \cdot & & & & \\ & & & \cdot & & & & \\ & & & \cdot & & & & \end{array}$$

As a result, bins of the histogram partition the zero-crossings by row. Furthermore, non-zero-crossings have value 0 and are all associated with the 0th bin. Thus each bin of the histogram except the 0th holds the addresses of the zero-crossings contained in the associated row of the image. Hence only corresponding bins from the right and left image histograms need to be matched.

Given the output of ISMAP for the modified (i.e., after ANDing) L_i and R_i images and the disparity image d_i, the host performs the actual matching step as follows. For each zero-crossing point (x,y) in the yth bin of L's histogram we index into the disparity image $d_i(x,y)$ to determine by approximately how much x should differ from the matching point's x-coordinate. Then we search for such a point in a restricted range of x values in the yth bin of R's histogram. Only a small number of the elements in the bin need to be searched since ISMAP orders the elements of the bin by x-coordinate.

Some additional calculations may be necessary to find the best matching pairs. The image $d_i(x,y)$ is updated at points where new disparities are implied as a result of the matching. $d_i(x,y)$ is then sent back to PIPE where it is multiplied by 2 and doubled in resolution to produce $d_{i+1}(x,y)$. Concurrently with this match-update-double phase in the host, PIPE ANDs L_{i+1} and R_{i+1} with the "row mask" and then sends them to ISMAP where the next pair of histograms is produced. Fortunately there is no need to actually "realign" images because the disparity map includes the shift information.

10.2. Comments

In general, implementing nonlinear functions of a pixel and its neighborhood requires bringing all the pixels involved into register at some point. This is due to PIPE's strict spatial indexing. The shift(s) required can only be done by the NBR units. Often no other operation can be performed in the same stage with this shift. This leads to bottlenecks in all but the simplest of neighborhood operations. The necessary redundancy also leads to space problems and data path contention. Finally, because of the restricted communication between stages, considerable time is spent bringing together the many images involved in the computation of a complex neighborhood operation. The problem in this section of matching two image rows is a case in point. Due to the sparsity of zero-crossings, the inability of PIPE to take advantage of this, and the complexity of the implementation on PIPE, we felt it preferable to set up an interface with the host which could more easily perform the matching.

11. Model-Based Object Recognition

In this section we describe a PIPE implementation of a multiresolution model-based matching technique for coarse-to-fine object recognition described by Neveu *et al.* [11]. Each two-dimensional object is modeled as a directed acyclic graph as follows. Each node in the graph stores a boundary segment of the object model at a selected level of resolution. The root node of the graph contains the coarsest resolution representation of the boundary of the object. Arcs are directed from boundary segments at one level of resolution to spatially related boundary segments at finer levels of resolution.

Given an input image's zero-crossing pyramid and the model graphs for several two-dimensional objects (created as described in [11]), matching can be carried out in a coarse-to-fine manner by following arcs of the graph representing a two-dimensional object to be recognized. That is, first the root node of the model graph is matched with the coarsest level of the input image pyramid, and an ordered list of hypothesized positions and orientations (poses) for the object is generated. These hypotheses limit the area in which the search for sub-objects (children nodes) must be conducted. If the sub-objects of a hypothesis are not found, the next best hypothesis for the pose of the object at the coarsest level is tried.

11.1. Implementation

Assume that each node stores a 15×15 binary template describing a segment of the object boundary at a given resolution. The node's parent describes a segment of the object boundary at the next coarser resolution. Hence the template at a node represents an area one quarter the size of that represented by its parent. Also stored at the node is the relative pose of the template (within the area represented by the parent's template), and a weight indicating the importance of the boundary segment in recognizing the object. Assume further that the coarsest level of the pyramid is 16×16, and the finest is 256×256.

Cross-correlation is used to match a model node's template with one level of the zero-crossing pyramid. The initial estimate for the best match pose of a template is determined from the match pose of its parent's template and the known relative pose of the given node with respect to its parent. The best match pose is then determined by trying a range of poses centered on this estimate. For each candidate orientation we construct a mask containing the appropriate rotation of the given template. For each candidate position we set the corresponding pixel in a ROI mask image which will be used to restrict the correlation operation to these marked (nonzero) candidate positions. PIPE will perform each of these correlations and send the results to ISMAP. The host will then determine the best match pose. The degree of match, weighted by the importance of the model template, indicates whether the process should be continued at each of the node's children. If this is the case the same process is applied at each of the node's children.

PIPE's ROI mode allows us to simultaneously perform the entire set of cross-correlations for a single node. Consider for example the 128×128 level of resolution. Four copies of the 128×128 zero-crossing image can be stored in one PIPE image, one in each quadrant. Each quadrant is then correlated with a different orientation of the current node's template as shown in Fig. 16. The ROI mode enables two restrictions in this process. First, it restricts the correlation operation for a single mask to one quadrant. Second, it further restricts the correlation operation to the candidate region of marked positions in this quadrant. Each such region is marked with a unique nonzero value. Hence four nonzero values are contained in the ROI mask image, one for each of four orientations of the template. Note, however, that only one of these four oriented-templates is cross-correlated at each marked position. The ROI mode allows the template to vary over the image by indicating which template is to be applied where. By having a copy of the image in each quadrant we effectively achieve four different correlations simultaneously. The results of the four correlations appear in the four quadrants of the image produced.

In [12] it was shown how convolution of an image with a 15×15 mask can be done on PIPE in 13 time units. The method is an extension of that described in Section 9. Each 15×15

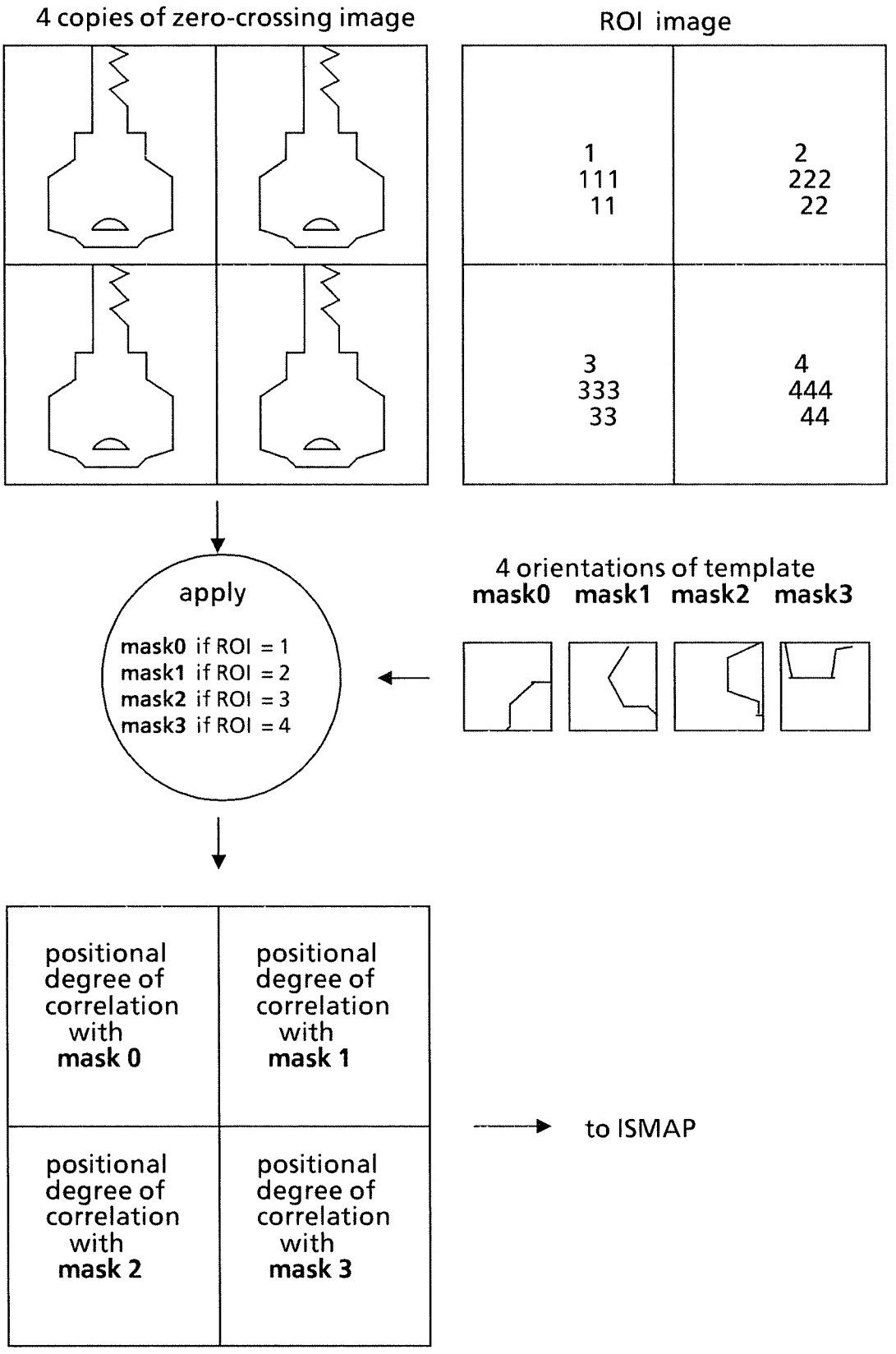


Fig. 16. Cross correlation of four different masks with a 128 x128 zero-crossing image at hypothesized positions and orientations.

correlation operation consists of a network of 3×3 neighborhood operations cascaded together. If there are four different templates being applied as described above, sets of four 3×3 masks are loaded into the mask tables of the NBR units. A ROI mask image is then used to select the appropriate 3×3 neighborhood operation of the four stored in each NBR unit. All of the 3×3 operations in the network are handled similarly. The final result image is then sent to ISMAP.

ISMAP and the host are then used to decide which match is best. Each correlation image is sent to ISMAP which computes a histogram and this is then sent to the host. Assuming all masks associated with a node are normalized, the host finds the coordinates of the pixel p with maximum value in the histogram. Four candidate orientations should be sufficient; however, the number can be doubled by using both NBR's in each stage in which the correlations are performed. Up to sixteen mask tables can be used by an NBR unit. Thus we may have up to thirty-two candidate orientations at each of the coarser levels. This should be sufficient particularly at the coarsest level when all orientations are possible.

The most time-consuming task of the host is to compute the candidate rotations of each node's template (assuming sufficient space is not available to precompute and store all possible rotations of all templates). The host must also store alternative pose hypotheses for each node in case backtracking is necessary.

In general the host and PIPE can perform few operations concurrently unless there are many nodes at each level of the model graph. When there are many nodes at each level, the completion of matching one node requires the generation of tasks for PIPE for each child node. The host can work constantly at preparing tasks and placing them in a task-queue for PIPE while PIPE performs them and places its results in a result-queue for the host. Since PIPE performs its work and reports its results in first-come-first-serve order, task ordering is preserved. Some additional synchronization may be required, however. That is, the host or PIPE may need to be suspended if either queue becomes empty or too large. To some extent this problem may be avoidable by

having the host select a node from the result-queue which generates more or less children, as needed.

11.2. Timing

Each node visited in the model graph represents one task to be generated by the host and to be performed by PIPE. In general each node may be visited several times due to backtracking. Visiting a node involves the time to prepare the task in the host, plus 13 time units in PIPE, plus two time units in ISMAP. However, as long as the two queues are neither empty nor full, these are done concurrently (for different nodes). The sizes of the two queues are a function of the width of the model graph and the relative speeds of PIPE and the host. Ideally, the lengths of the queues would stabilize and a new node would be visited every 13 time units.

11.3. Comments

It was noted that the number of possible orientations reduces greatly following the cross-correlation at the coarsest level. Hence it is not always necessary to perform correlations for thirty-two different orientations, although PIPE has the ability to do so with no additional time delay. Different templates at various orientations can be searched for just as easily as one template at many orientations. This corresponds to searching in the zero-crossing image for several children at once. However this involves a slightly more complex scheme for computing the best pose from the histogram.

References

1. E. W. Kent, M. O. Shneier, and R. Lumia, PIPE — Pipelined Image-Processing Engine, *J. Parallel and Distributed Computing* **2**, 1985, 50-78.
2. Digital/Analog Design Associates, Programming the PIPE, 1985.
3. T. Maruyama and T. Uchiyama, Real-time image processor with two convolution filter modules and a peak extraction module, *Proc. IEEE Conf. Computer Vision and Pattern Recognition,* 1983, 546-549.
4. D. E. Knuth, *The Art Of Computer Programming, Vol. 3: Sorting and Searching,* Addison-Wesley, Reading, MA, 1973.
5. M. I. Shamos, Robust picture processing operators and their implementation as circuits, *Proc. Image Understanding Workshop,* November 1978, 127-129.
6. R. T. Chin, H. K. Wan, D. L. Stover, and R. D. Iverson, A one pass thinning algorithm and its parallel implementation, submitted for publication.
7. D. H. Ballard, and C. M. Brown, *Computer Vision*, Prentice-Hall, Englewood Cliffs, NJ, 1982.
8. R. J. Woodham, Photometric stereo: a reflectance map technique for determining surface orientation from a single view, *Proc. SPIE,* Vol. 155, 1978, 136-143.
9. P. J. Burt, The pyramid as a structure for efficient computation, in *Multiresolution Image Processing and Analysis,* A. Rosenfeld, ed., Springer-Verlag, Berlin, West Germany, 1984, 109-120.
10. D. Marr and T. Poggio, A computational theory of human stereo vision, *Proc. Royal Society of London B* **204**, 1979, 301-328.
11. C. F. Neveu, C. R. Dyer, and R. T. Chin, Two-dimensional object recognition using multiresolution models, *Computer Vision, Graphics, and Image Processing* **34**, 1986, 52-65.
12. C. V. Stewart and C. R. Dyer, Convolution algorithms on the Pipelined Image-Processing Engine, Technical Report 643, Computer Science Department, University of Wisconsin—Madison, May 1986.