List of Figures

Hardware Design of a 2-D Motion Estimation System Based on The Hough Transform [1]

Hsiang-Ling Li and Chaitali Chakrabarti

Department of Electrical Engineering
Telecommunications Research Center
Arizona State University
Tempe, AZ 85287-5706
Ph: (602) 965-9516
Fax: (602) 965-8325
li@dspsun.eas.asu.edu
chaitali@asu.edu

Abstract

A novel feature-domain 2D motion estimation system based on the straight-line Hough transform (SLHT) is presented. This system implements the motion estimation technique proposed in an earlier work that represents objects uniquely by straight-line approximations of the boundary $\{(\theta, p)\}$, and estimates the motion parameters from shifts in the $\theta - p$ space. It operates on 256×256-pixel binary images and consists of two main blocks. The first block does preprocessing work including smoothening the boundary, tracing and integrating the contours, and detecting dominant points. The second block computes the Hough transform on contour segments as well as the rotation and translation parameters. Each of the modules has been implemented (gate level) and simulated using Mentor Graphics tools. Finally, the experimental results have been presented and compared with the results of the software implementation.

[1] This research was supported in part by a grant from NSF MIP-9309504.

1 Introduction

The motion estimation unit is a key component in systems for object tracking, robotic vision, dynamic scene understanding, and autonomous navigation. Most of the VLSI implementations for 2-D motion estimation are based on the block matching algorithms (BMAs), since BMA has been adopted in the MPEG standards [1]-[7]. A large number of application-specific architectures have been proposed for full search BMA (FBMA) [1]-[5], hierarchical search BMA (with fixed block size) [3], [6] and hierarchical BMA (with variable block size) [7]. All the block matching algorithms are pixel-domain based approaches and require a large amount of computations. If instead, the object is represented by a set of features, then the number of computations can be significantly reduced. In such a case, only one translation vector and rotation angle per object has to be transmitted, instead of displacement vectors for every group of pixels. The computations can also be reduced by using Fourier transform descriptors, differential descriptors etc.

This work presents a 2-D motion estimation system based on a novel feature-domain algorithm. The motivation here is to develop a better alternative (that is computationally less intensive) to current BMA based VLSI implementations. The motion estimation system implements the Hough transform based motion estimation technique that we proposed in an earlier work [14]. In the proposed technique, a 2-D object is uniquely represented by a small set of straight lines (i.e. $\{(\theta, p)\}$[2]) that approximates its boundary, and the motion parameters are computed from shifts in the $\theta - p$ space. The corresponding system consists of two main blocks. The first block (the preprocessor) smoothens the boundary, traces the contours, and detects dominant points (dominant points are high curvatured points in the contour) which are then used to break the contour into several segments. Next, the second block (the Hough Transform processor) computes the straight line Hough Transform on the boundary pixels of each contour segment. The motion parameters are computed in the second block using modules such as the divider, the cordic unit, and the vote accumulator. The system operates on 256×256-pixel binary images and has been implemented (gate level) using Mentor Graphics tools. The architecture is well suited for standard cell or FPGA

[2] θ is the angle between the x-axis and the normal, and p is the length of the normal.

implementation.

The Hough transform (HT) has also been used for 2-D motion estimation by some other researchers in [8] and [9]. The work proposed in [8] avoids the problems of high dimensionality and extensive preprocessing of the generalized HT but is limited to analytic curves such as circles, ellipses, and parabolas. A more recent work [9] has generalized and extended this method for arbitary shapes by utilizing the perpendicular distances between pairs of parallel tangent lines as the signature of a shape. The rotation and translation is then determined by utilizing 1-D correlations of the reference signature with the test signature. The main drawback of this approach is that it is computationally intensive since the object is represented by a large number of straight lines. Recently, a few Hough transform (HT) based hardware systems have also been developed. VLSI hardware implementations of HT have been reported in the literature [10]-[12]. These HT processors can be integrated into a HT based 2-D motion estimation system. Schonfeld and Pirsch [13] have developed an image processing board which extracts straight lines using HT in real time from video sequences for the street boundaries. The proposed compact image processing unit is utilized in a vision based automatic vehicle guidance system.

The rest of this paper is organized as follows. Section 2 gives the summary of the motion estimation algorithm proposed in [14]. The overview of the proposed 2-D motion estimation system is described in Section 3. The hardware design of several key modules in the preprocessor is described in Section 4. The hardware design of several key modules in the HT processor is described in Section 5. Section 6 gives the simulation results of the 2D motion estimation system. This paper is concluded in Section 7.

2 Algorithm Summary

In order to reduce the number of computations in motion estimation, the boundary of an object needs to be represented compactly. In the HT based procedures of [8] and [9], the boundary of an object is approximated by tangent lines through almost every boundary pixel. Such a procedure requires an enormous amount of computation. Moreover, a threshold value in the $\theta - p$ space is required to reduce the number of spurious peaks. In [14], we describe a method to represent an object compactly using a *small* set of straight-lines, where each straight-line approximates the contour between two consecutive dominant points. We use dominant points in the representation of an object since it is well known that information on a 2-D shape is concentrated at the dominant points [16]. Figure 1 shows the block diagram of the motion estimation algorithm described in [14]. This algorithm consists of two steps. The first step is to represent the 2-D object by a set of straight lines of $\{(\theta, p)\}$. The second step is to compute the motion parameters from the two representations of the object at times t_a (i.e. $\{(\theta, p)\}_{t_a}$) and t_b (i.e. $\{(\hat{\theta}, \hat{p})\}_{t_b}$). We describe the algorithm in [14] here for the sake of completeness.

2.1 Object Representation

As mentioned earlier, the object boundary is approximatd by a set of straight lines. This is achieved by breaking the contour into groups of edge pixels with dominant points serving as the end points of each group, and then approximating each group of pixels by a straight line. The reason why we do not rely solely on dominant points to represent the object is that discretization errors can cause displacements in the positions of the dominant points. This is especially true when objects are rotated. By computing the Hough transform of pixels between two adjacent (along the contour) dominant points, an object is represented more robustly. In the end, the 2-D object at time t_a is represented by $\{(\theta, p)\}_{t_a}$ and at time t_b by $\{(\hat{\theta}, \hat{p})\}_{t_b}$. No thresholding is required here, since each group of pixels gives rise to a single peak for one straight line. It should be noted that the computations of (θ, p) can be reduced by first computing $\tilde{\theta}$, the θ for the line connecting adjacent dominant points. Hence, the votes for (θ, p) are computed only in the range around $\tilde{\theta}$ (instead of in the range 0^o to 180^o).

Figure 2 gives a pictorial description of the formulation of straight line Hough Transform (SLHT) applied in the algorithm. Let the line segment $l = (\theta, p)$ be a straight line approximating part of the boundary of an object C centered at (c_x, c_y). The value of θ (between the $x-$axis and the normal) is confined to the range 0^o to 180^o, and p (length of the normal) is either positive or negative.

$$p(\theta) = p_0(\theta) + t_0 cos(\theta - \alpha_0), \tag{1}$$

where $p_0(\theta)$ is the intrinsic term (SLHT of that line segment when C is located at the origin), $t_0 cos(\theta - \alpha_0)$ is the translation term, $t_0 = \sqrt{c_x^2 + c_y^2}$, and $\alpha_0 = tan^{-1}(c_y/c_x)$. If the shape C is rotated by an angle ϕ ($< 180^o$ to avoid confusion) in the counterclockwise direction and the center of the shape C is translated to $(\hat{c_x}, \hat{c_y})$, then the point (θ, p) is mapped to $(\hat{\theta}, \hat{p})$, where

$$\hat{\theta} = (\theta + \phi) \bmod 180^o, \tag{2}$$

$$\hat{p}(\hat{\theta}) = p_0(\theta) + t_0 cos(\theta + \phi - \alpha_0) + t_1 cos(\theta + \phi - \alpha_1), \tag{3}$$

where $t_1 = \sqrt{(\hat{c_x} - c_x)^2 + (\hat{c_y} - c_y)^2}$ and $\alpha_1 = tan^{-1}(\frac{\hat{c_y} - c_y}{\hat{c_x} - c_x})$.

2.2 Computation of the Motion Parameters

Once an object is represented uniquely by straight-line approximation of the boundary or by a set of (θ, p)s, we estimate the motion parameters of the object by computing the shifts in the $\theta - p$ space. We first determine the rotation angle ϕ by computing the votes on $\Delta\theta$, the difference in the θ values of the object at times t_a and t_b and assigning $\tilde{\phi}$ to be the angular difference with the highest vote. Since the differences between any two corresponding angles θ and $\hat{\theta}$ are not exactly equal to $\tilde{\phi}$ ($\hat{\theta} \cong (\theta + \phi) \bmod 180^o$) due to errors caused by the discretization of the image and quantization, the detected peak does not always correspond to the correct rotation angle. By matching $\hat{\theta}_j$ with $\theta_i + \tilde{\phi} \pm \Delta\tilde{\phi}$, a one to one correspondence can be obtained between matched $\{(\theta, p)\}_{t_a}$ (denoted as $\{(\theta, p)\}'_{t_a}$) and $\{(\hat{\theta}, \hat{p})\}_{t_b}$ (denoted as $\{(\hat{\theta}, \hat{p})\}'_{t_b}$). Then, the differences between the corresponding angles θ (in $\{(\theta, p)\}'_{t_a}$) and $\hat{\theta}$ (in $\{(\hat{\theta}, \hat{p})\}'_{t_b}$) are averaged. A better estimate of the rotation angle, ϕ is thus obtained.

The correspondence between the elements in $\{(\theta, p)\}'_{t_a}$ and $\{(\hat{\theta}, \hat{p})\}'_{t_b}$ is required for the calculation of the translation vector. First, the intrinsic term, $p_0(\theta)$, is calculated for each θ by subtracting the translation term $t_0 cos(\theta - \alpha_0)$, from $p(\theta)$ of the reference object. Next, $\Delta p(\theta) = \hat{p}(\hat{\theta}) - p_0(\theta)$ is calculated for each θ. If $\theta + \phi > \pi$, the corresponding $\hat{p}(\hat{\theta})$ is made negative during the computation of $\Delta p(\theta)$. Since Δp can be rearranged as $\Delta p = \hat{c}_x cos(\theta + \phi) + \hat{c}_y sin(\theta + \phi)$, where $(\hat{c}_x, \hat{c}_y)$

is the center point of the test object, we can compute $\hat{c}_x$ and $\hat{c}_y$ by performing an inverse Hough Transform on $\{\theta, \Delta p\}$. The time to locate the peak is reduced by first approximating $\hat{c}_x$ and $\hat{c}_y$ and then refining the search in a smaller area. The approximate values of $\hat{c}_x$ and $\hat{c}_y$ are the centroid of the polygonal approximation of the test object[3], (c'_x, c'_y) whose vertices are the dominant points. The translation vector (t_x, t_y) is equal to $(\hat{c}_x - c_x, \hat{c}_y - c_y)$, where (c_x, c_y) are the coordinates of the centroid of the polygonal approximation of the reference object. The above procedure has been implemented in software. The results have been accurate and have been presented in [14].

2.3 Complexity Comparison

Figure 3 shows the comparison of the computational complexities of the proposed method and Pao et al's method [9]. Assume that input image is of size $N \times N$, the length of the object contour is P pixels, and the number of dominant points is n. $\hat{m_\theta}$ (usually less than 10) denotes the number of quantization levels for the θ values during the computation of SLHT. b is the time to locate the peak in the $x - y$ plane, and $b << N$. m_θ denotes the number of quantization levels in [9] and is equal to 180 for $\Delta\theta = 1^o$ (quantization interval). K_D is the average number of points in each column of the reference signature [9]. The comparison shows that our method involves much less computation than Pao et al's method. This matches the observation that a compact representation results in fewer computations provided that the preprocessing is not excessive. Please refer to [14] for details of the comparison.

3 System Overview

The 2-D motion estimation system operates on 256×256-pixel binary images and consists of two main blocks, the preprocessor and the HT processor. The preprocessor does preprocessing work including smoothening the boundary, tracing and integrating the contours, and detecting dominant points. The HT processor computes the Hough transform on contour segments. It is also responsible for computing the rotation and translation parameters.

[3]The centroid of the polygonal approximation of an object is determined by adding the coordinates of vertices and dividing the resulting sum by the number of vertices.

The block diagram of the preprocessor is shown in Figure 4. The noisy boundary pixels of the input image are removed by the "boundary smoothening module". Then, the "contour tracer" detects the boundary pixels, and the "contour integrator" links the broken contours. The dominant points are then detected along the final contour by the "dominant point detector", and sent to the HT processor.

The contour segments (bound by dominant points) are sent to the HT-processor one after the other. The HT-processor then outputs a (θ, p) value for each contour segment. The block diagram of the HT-processor is shown in Figure 5. Consider a contour segment: (x_0, y_0), (x_1, y_1), ..., (x_n, y_n), where (x_0, y_0) and (x_n, y_n) are the dominant points. The first and second blocks use the two end points to compute $\tilde{\theta} = tan^{-1}(-\frac{x_n - x_0}{y_n - y_0})$, the initial estimate of θ. The Hough transform of (x_0, y_0) is computed around $\tilde{\theta}$ in the third block. The last block computes the p values of other pixels by simple addition or subtraction and accumulates votes to estimate (θ, p) for the contour segment.

For the computation of the rotation angle, the vote accumulator is used to compute the votes on $\Delta\theta$ to obtain the initial estimate of the rotation angle, $\tilde{\phi}$ such that a one to one correspondence can be established between $\{(\theta, p)\}'_{t_a}$ and $\{(\hat{\theta}, \hat{p})\}'_{t_b}$. Then, the divider is used to average the differences between the corresponding angles θ (in $\{(\theta, p)\}'_{t_a}$) and $\hat{\theta}$ (in $\{(\hat{\theta}, \hat{p})\}'_{t_b}$) for a better estimate of the rotation angle ϕ. For the computation of the translation vector, the second subunit in the master unit of the HT processor computes the translation terms so that the intrinsic terms and $\Delta p(\theta_i)$ can be computed. Then, the divider computes the slopes and $y-$intercepts. The vote accumulator computes the votes in the $x - y$ plane (inverse HT) to obtain $(\hat{c}_x, \hat{c}_y)$. Finally, the translation vector $(\hat{c}_x - c_x, \hat{c}_y - c_y)$ is obtained using a simple subtractor.

4 Preprocessor

The key modules in the preprocessor are the "boundary smoothening module", the "contour tracer", the "contour integrator", and the "dominant point detector". Here, we briefly describe these modules.

4.1 Boundary Smoothening Module

We implement the boundary smoothening module with a state machine having 40 states where each state corresponds to a distribution of pixels in a 3×3 window centered around the current edge pixel [14]. If a combination of 9 pixels matches with any one of the 40 states, the detected noise pixel is removed, i.e. "1" is replaced by "0" (see Figure 6 (a)) or a pixel is added, i.e. "0" is replaced by "1" to smoothen the boundary (see Figure 6 (b)). Such a state machine can be easily implemented using a NAND decoder. Whenever the 9 pixel values from the shift registers (positive edge triggered) and latches (transparent on clock='1', see Figure 7) match with one of the 40 states, the corresponding output line goes high (i.e. a state is detected) to reset or set one of the 9 pixel values (corresponding to removing noise or smoothening the boundary). These signals (detected states) that are used to remove or add the same pixel are "OR" together and then gated with the inverted clock to clear or preset the corresponding flipflop in the shift register (see Figure 7). Since we use a 3×3 window, border pixels ("0"s) are added to each image frame to make its size equal to 258×258.

We work directly on the raster input pixels to smoothen the boundary. Hence, two line delays are required. The two line delays are implemented using two RAMs (see Figure 7). Each RAM is a dual-port RAM and stores 258 pixels or bits with every stored bit accessed by its $x-$coordinate $(0 - 257)$ as the address. The $x - y$ coordinates of all the pixels in one image frame are determined by two 9-bit counters, the x-counter and the y-counter. Within every clock cycle, each RAM reads out the stored pixel in the first half of the cycle and writes the input pixel to the same address in the second half. Thus, there are 258 clock cycles between the time when a pixel is stored into the RAM and the time when it is read out of the RAM (i.e.the RAM acts like a $258-$bit shift register or a line delay). The architecture of the boundary smoothening module is shown in Figure 7.

4.2 Contour Tracer

In this motion estimation system, the image frame is partitioned into 16 subimages (subimage (i, j), $0 \leq i \leq 3$, and $0 \leq j \leq 3$), and the contours in each subimage (64×64 pixels) are generated in parallel. Next, contour integration is done to link the contours in each subimage. The contour tracer consists of 16 processors with each processor assigned to a subimage for contour tracing and one postprocessor (contour integrator) to do contour integration. The architecture of the contour tracer is shown in Figure 8.

The switch control signal $enable_{ij}$ ($0 \leq i \leq 3$, and $0 \leq j \leq 3$) is determined by the $x - y$ coordinates of the pixels of subimage (i, j). Each subimage processor has a state machine (NAND decoder) with 28 states and is responsible for tracing the contours of a subimage. First, the processor operates directly on the raster input pixels to detect the starting pixels of all the contours. Once a match occurs on any of the 28 states and the next edge pixel (assuming counterclockwise tracing direction) is inside the subimage, a starting pixel is detected and its $x - y$ coordinates are stored in the "starting point memory". Once all the subimage pixels are input and stored in the "subimage memory", the detection of starting pixels ends, and the contour tracing procedure begins. To reduce the number of starting pixels and make contour integration simple, we only search the starting pixels around the border of one subimage (assuming that an object is not completely located within a subimage), and specify the counterclockwise tracing direction. The above method can also avoid repeated contour tracing to reduce the complexity. It should be noted that if no starting pixel is detected in a subimage, then no further operations will be undertaken in the corresponding processor.

Example 1: Consider a 2-D object shown in Figure 9 that extends into four subimages, subimage (i, j), subimage $(i, j-1)$,subimage $(i+1, j-1)$, and subimage $(i+1, j)$. The $x - y$ coordinates of a boundary pixel P_i is (x_i, y_i) and the corresponding chain code is C_i. The starting pixels for subimage (i, j), subimage $(i, j-1)$, subimage $(i+1, j-1)$, and subimage $(i+1, j)$ are P_0 (coordinates (x_0, y_0)), P_{k+1} (coordinates (x_{k+1}, y_{k+1})), P_{q+1} (coordinates (x_{q+1}, y_{q+1})), and P_{m+1} (coordinates (x_{m+1}, y_{m+1})), respectively. □

During contour tracing, if a match occurs on any of the 28 states, the next edge pixel (which is the center pixel of the next 3×3 window), and the chain code of the current edge pixel relative to the previous edge pixel (in the contour) are both determined and stored in the "contour memory".

Once the boundary of a subimage is reached, the end pixel of a contour is detected and stored in the "end point memory". The address of an end point (end index) in the contour memory is also stored in the "end index memory". The contour tracing procedure then starts from the next starting pixel.

Example 2: Consider the same object as described in Example 1. At the end of the contour tracing, the contents of the contour memories, the end point memories, and the end index memories of subimage (i, j), subimage $(i, j-1)$, subimage $(i+1, j-1)$, and subimage $(i+1, j)$, respectively are shown in Figure 10. □

The address generator for the "subimage memory" (SMAG) has to generate 9 addresses of the pixels of the 3×3 window that is centered at the current edge pixel in 9 clock cycles. The architecture of SMAG is shown in Figure 11. The timing diagram of the six control signals, xs_0, xs_1, xs_2, ys_0, ys_1, and ys_2, is illustrated in Figure 12. The control logic consists of two 3-bit shifters (SR) with the initial value of each SR set to "001".

4.3 Contour Integrator

The architecture of the contour integrator is shown in Figure 13 and Figure 14. Each contour segment is represented by two values, the start index (equal to the previous end index added by one) and the end index. Both indices are the addresses in the contour memory for one subimage. First, all the starting pixels of each subimage, A_1, B_1, and so on are loaded into memory 1. All the subimage indices, start indices, end indices and related end pixels, A_2, B_2, and so on are loaded into memory 2. Then, this unit links two contours by checking if the starting pixel of one contour equals the end pixel of another contour. Once a match occurs, the subimage index (i, j), start index and end index are stored into the "sorted contours memory" (memory 3). Last, the contour pixels and chain codes from the start index to the end index are loaded from the contour memory in that subimage specified by the subimage index (i, j) to the "final contour memory". Once the last contour segment is loaded into the final contour memory, the task of contour integration is completed.

Example 3: Consider the same object as described in Example 1. At the end of the contour integration, the contents of the memory 1, memory 2, the memory 3, and the final contour memory are shown in Figure 15 and Figure 16. □

4.4 Dominant Point Detector

Algorithm: Dominant points are high curvature points. In order to detect them correctly, the region of support has to be determined precisely [16]. The region of support can be a large corner or a convex/concave region. In our earlier work, we have developed a new and efficient algorithm that detects the region of support in a single pass during contour tracing [14]. This algorithm uses the local properties of the curve to determine the size of the region of support. The local properties of the curve are specified by three parameters n_1, n_2, and n_3 [14]. If the number of consecutive pixels with the same chain code is $> n_1$, these pixels are effective and contribute to the region of support. n_2 specifies the minimum number of effective pixels to represent one side of a large corner, and n_3 specifies the minimum number of effective pixels to complete a small corner or a convex/concave region[4]. These three parameters play a role similar to that of σ in the Gaussian kernel used in scale-space mapping. Here, we rephrase the algorithm in [14] by describing it in terms of searching for groups of pixels with specific chain codes.

Each region of support consists of several groups of effective pixels with each group corresponding to a chain code. Let a group of effective pixels corresponding to the chain code "i" be defined as the i-group. We also define the count of i-group as the total number of pixels in the i-group (i.e. the count of i-group is $\geq n_1$). If the count of i-group is $\geq n_2$ and the count of j-group (encountered later than i-group), where $|i-j|$ or $8-|i-j| = 1$ or 2, also reaches n_2, then a large corner is detected. For a sharp corner, the count of i-group can be either $\leq n_2$ or $\geq n_2$. Once the count of j-group (encountered later than i-group), where $|i-j|$ or $8-|i-j| = 3$ or 4, reaches n_3, a convex or concave region is detected. For either of the above two cases, the region of support is specified by the front pointer (FP) of the i-group (where FP is the position of the first pixel in the i-group) and the rear pointer (RP) of the j-group. After the count of i-group reaches n_2, FP moves such that

[4]For a 256×256 image, n_1 is 2 or 3, n_2 is ≈ 10, and n_3 is ≈ 6.

the number of pixels between FP and the last pixel of i-group remains at n_2. RP is the position of the n_2th pixel or the n_3th pixel in the j-group. Such definitions of FP and RP ensure reduction in the size of the region of support. For instance, let 0-group be the first group in a region of support. Then, the possible chain codes for the last group to complete a large corner are 6, 7, 1, and 2. Similarly, if 0-group is the last group in a region of support, the possible chain codes for the first group to start a large corner are 6, 7, 1, and 2. For a convex or concave region, the possible chain codes for the last (or the first) group in a region of support with 0-group corresponding to the first (or the last) group are 3, 4, and 5. It should be noted that for a convex/concave region, once the count of j-group reaches n_3, we may have more than one i-group to start a region of support. In this case, the i-group that has the largest FP is selected to reduce the size of the region of support. Figure 17 summarizes the rules for determining the chain codes for a group of pixels to start or to complete a region of support. The hardware design of detecting the region of support thus follows the rules specified by Figure 17.

Hardware Design: The dominant point detector consists of two parts. The first part detects the region of support, and has 8 cells corresponding to the 8 chain codes. The architecture of a cell for chain code 0 (0-cell) in the first part is shown in Figure 18. Each cell for a chain code i (the reference chain code) has a counter and related logic (specified by n_1, n_2, and n_3) to generate three control signals: contrl-A_i, contrl-B_i, and contrl-C_i. These three control signals are used to determine if the number of consecutive pixels with the same chain code, i.e. the count can contribute to a large corner (count $\geq n_2$) or a convex/concave region (count $\geq n_3$). The front pointer (FP) and rear pointer (RP) (addresses in the "final contour memory") are generated using the rules specified by Figure 17. Once a RP is generated, the corresponding chain code (the reference chain code) is also output to each cell (the status-in) to generate the signal, reset-cfp. This signal then resets the counts and FPs of those cells with different reference chain codes than the status-in. Then, the detection of the next region of support starts. It should be noted that contrl-A_i is connected to the logic circuits for FP so that the number of effective pixels between the FP and the latest input pixel (having the same chain code) remains at n_2.

The second part in the dominant point detector computes the curvature of each pixel within a region of support using the k-curvature method [16]. The address of the middle point, $\lfloor \frac{FP+RP}{2} \rfloor$ is used as the reference address. If the address of an input pixel is less than or equal to the reference address and the curvature of this input pixel is greater than or equal to the maximum stored curvature, then the maximum stored curvature is updated and the address of this input pixel is stored. On the other hand, if the address of an input pixel is greater than the reference address, then the maximum stored curvature and the stored address are updated only when the curvature of this input pixel is greater than the maximum stored curvature. Hence, the detected dominant point has the maximum curvature and is located near the central portion in a region of support. The architecture of the second part in the dominant point detector is shown in Figure 19.

5 Hough Transform (HT) Processor

The input to the HT processor is a contour segment: (x_0, y_0), (x_1, y_1), ..., (x_n, y_n) (loaded from the "final contour memory"), where (x_0, y_0) and (x_n, y_n) are the dominant points. The key modules in the HT processor are the "divider", the "inverse tangent unit", the "master unit", and the "vote accumulator". Here, we briefly describe each of these modules.

5.1 Divider

The divider computes the quotient, $Q = -\frac{NUM}{DENOM}$, where NUM and $DENOM$ are both positive. The following equation derived using the CORDIC technique [17] is used to compute Q in $m+1$ iterations (or clock cycles).

$$Q_{i+1} = Q_i + DENOM * \delta_i * 2^{k-i}, \tag{4}$$

where $i = 0, 1, \ldots m$, $Q_0 = NUM$, and $k = 2, 5$ or 7.[5] δ_i is determined by the MSBs of Q_i and $DENOM$. Then, $Q = -\frac{NUM}{DENOM} \cong \sum_{i=0}^{m} \delta_i * 2^{k-i}$. The architecture of the divider is shown in Figure 20. The "Logic" to generate δ_i is simply a two-input XNOR gate.

The initial estimate of θ is computed using the formula $\tilde{\theta} = tan^{-1}(-\frac{x_n - x_0}{y_n - y_0})$. If $-\frac{x_n - x_0}{y_n - y_0} \geq 8$,

[5] $k = 2$ for the computation of $\tilde{\theta}$, $k = 5$ for the computation of ϕ and $k = 7$ for the computation of slopes, intercepts, (c_x, c_y), and (c'_x, c'_y).

$\tilde{\theta}$ is close to 90^o. Hence it is good enough for initial estimates to use 8 iterations, i.e. $m = 7$ to compute $-\frac{x_n - x_0}{y_n - y_0}$. In this case, $NUM = x_n - x_0$, $DENOM = y_n - y_0$ and $k = 2$.

5.2 Inverse Tangent Unit

For $Q \geq 1$ or $Q \leq -1$, we use a look up table to determine $\tilde{\theta}$. First, Q is made positive. If $Q \geq 8$, then $\tilde{\theta} = 83^o$. However, if $Q < 8$ but ≥ 4, then $\tilde{\theta} = 76^o$. In such cases, the $\tilde{\theta} = \sum_{j=0}^{7} \delta_j * \theta_j$, where $\theta_j = tan^{-1} 2^{-j}$, is solved to determine δ_js. If Q is negative, $\tilde{\theta}$ is replace by $180^o - \tilde{\theta}$. For $-1 < Q < 1$, we can directly solve the following equation, $Q \cong \sum_{j=0}^{7} \delta_j * 2^{-j}$ to determine δ_js. The δ_js are sent to the master unit. The architecture of the inverse tangent unit is shown in Figure 21.

5.3 Master Unit

This unit consists of two identical subunits with each subunit implementing the following two equations: $p_{j+1} = p_j + \delta_j * 2^{-j} * q_j$ and $q_{j+1} = q_j - \delta_j * 2^{-j} * p_j$, $j = 0, \ldots, 7$ [17]. These two subunits work in parallel to save the overall computation time and reduce storing complexity. The first subunit computes the p value at $\tilde{\theta}$ for (x_0, y_0), i.e. $p_0(\tilde{\theta}) = x_0 cos\tilde{\theta} + y_0 sin\tilde{\theta}$ by using $p_0 = x_0$ and $q_0 = y_0$. The second subunit computes $sin\tilde{\theta}$, and $cos\tilde{\theta}$ in parallel by using $p_0 = 1$ and $q_0 = 0$. It takes 8 clock cycles to compute these values. Let the quantization level $\Delta\theta$ be such that $tan(\Delta\theta) = 2^{-6}$, i.e. $\Delta\theta \cong 0.9^o$. Then, the other p values for (x_0, y_0) at $\tilde{\theta} \pm \Delta\theta$, $\tilde{\theta} \pm 2\Delta\theta$,..., and $sin(\tilde{\theta} \pm \Delta\theta)$, $cos(\tilde{\theta} \pm \Delta\theta)$, $sin(\tilde{\theta} \pm 2\Delta\theta)$, $cos(\tilde{\theta} \pm 2\Delta\theta)$ ... can be easily computed in the first and second subunits. Each of these values is computed in one clock cycle. The architecture of the subunit is shown in Figure 22. A logarithmic barrel shifter shifts p_j and q_j to the right by j bits in the jth iteration.

These p values as well as the sine and cosine of the corresponding angles are transmitted to the vote accumulator (see below) and used as initial values to compute p values for other pixels in the contour segment.

5.4 Vote Accumulator

This unit consists of a linear array of cells where each cell computes the p values corresponding to some θ value for all the pixels in the contour segment. For instance, if the range of θ is $\tilde{\theta} - l * \Delta\theta$ to $\tilde{\theta} + l * \Delta\theta$, then this unit consists of $2l + 1$ cells. Each cell stores the three values transmitted from the master unit, $p_0(\theta)$, $sin\theta$, and $cos\theta$, where $p_0(\theta) = x_0 cos\theta + y_0 sin\theta$. If the contour segment is (x_0, y_0), (x_1, y_1), ..., (x_n, y_n), where (x_0, y_0) and (x_n, y_n) are the dominant points, then the cell corresponding to θ computes $p_1(\theta)$ for (x_1, y_1), $p_2(\theta)$ for (x_2, y_2), ..., and $p_n(\theta)$ for (x_n, y_n). Since the $x - y$ coordinates of pixels are integers $(0 - 257)$, the p value of the jth pixel p_j, is computed from the p value of the $(j-1)$th pixel p_{j-1}, by $p_j = p_{j-1} + b_{j1} cos\theta + b_{j2} sin\theta$, where θ is the angle assigned to that cell, and b_{j1}, b_{j2} are determined from the difference in the $x - y$ coordinates of the jth and $(j-1)$th pixel. b_{j1}, b_{j2} can take on values -1, 0 and 1. Once a p value is computed, it is latched as the address to the "vote memory" which stores the votes, and the corresponding vote is incremented and stored back in the same address. In each evaluation, the updated vote is compared to the maximum stored vote (initial value $= 0$) and the larger vote is stored as the maximum vote. The coresponding p is also stored. In the end, each cell detects the largest vote and its corresponding p. A comparison of these values among the cells is used to to obtain the (θ, p) for one contour segment. The architecture of the vote accumulator is shown in Figure 23. Control signals so, $s1$, $s2$, $s3$, $s4$, and $s5$ are generated from the difference in the $x - y$ coordinates of the two consecutive pixels and connected to every cell to determine b_{j1} and b_{j2}. For instance, if $x_j = x_{j-1} + 1$ and $y_j = y_{j-1} - 1$, then $s1$ and $s3$ become high and $p_j = p_{j-1} + cos\theta - sin\theta$. The circuit for computing p_j from p_{j-1} is shown in Figure 24.

5.5 Estimation of the Rotation Angle

One cell of the vote accumulator is used to compute the votes on $\Delta\theta$ to obtain the initial estimate of the rotation angle $\tilde{\phi}$. The computed $\Delta\theta$ is latched as the address to the "vote memory" and the corresponding vote is incremented and stored back in the same address. In each evaluation, the updated vote is compared to the maximum stored vote (initial value $= 0$) and the larger vote is stored as the maximum vote. The register that holds $\Delta\theta$ is also updated. At the end of the process, $\tilde{\phi}$ is the value that is stored in the $\Delta\theta$ register, Since the differences between any two corresponding angles θ and $\hat{\theta}$ are not exactly equal (refer to Section 2), the detected peak does not always correspond to the correct rotation angle. Hence, $\tilde{\phi}$ is used as the initial estimate of the rotation angle, ϕ. We use the circuit shown in Figure 25 to compare $\hat{\theta}_j$ with $\theta_i + \tilde{\phi} \pm \Delta\tilde{\phi}$. If the output goes high, $\hat{\theta}_j$ is a rotated version of θ_i by approximately $\tilde{\phi}$ (i.e. $\hat{\theta}_j$ is matched with θ_i). In the end of the procedure, we establish a one to one correspondence between matched $\{(\theta, p)\}_{t_a}$ (denoted as $\{(\theta, p)\}'_{t_a}$) and $\{(\hat{\theta}, \hat{p})\}_{t_b}$ (denoted as $\{(\hat{\theta}, \hat{p})\}'_{t_b}$). The differences, $\Delta\theta$s between the corresponding angles θs (in $\{(\theta, p)\}'_{t_a}$) and $\hat{\theta}$s (in $\{(\hat{\theta}, \hat{p})\}'_{t_b}$) are averaged for a better estimate of the rotation angle, ϕ. In this case, the variables in eqn.(4) are $m = 7$, $k = 5$, $NUM = \sum_{i=1}^{n_\phi} \Delta\theta$, and $DENOM = -n_\phi$, where n_ϕ is the number of matched (θ, p) pairs.

5.6 Estimation of the Translation Vector

The translation vector is computed using the θ and p values in matched $\{(\theta, p)\}'_{t_a}$ and $\{(\hat{\theta}, \hat{p})\}'_{t_b}$. For the computation of the votes in the $x - y$ plane (inverse HT), the translation terms, intrinsic terms, Δp values, slopes, intercepts, and the centroid of the polygonal approximation of the test object, (c'_x, c'_y) are computed.

- Translation term: The second subunit (in the master unit of the HT processor) computes the translation term, $T(\theta_i) = t_0 cos(\theta_i - \alpha_0)$ by using $p_0 = t_0$ and $q_0 = 0$, where $i = 1, \ldots, n_\phi$. $\theta_j - \alpha_0$ is sent to the inverse tangent unit to generate δ_js. If $(\theta_j - \alpha_0) > 90^o$, then $(\theta_j - \alpha_0 - 90^o)$ is used to generate δ_js and $p_0 = 0$ and $q_0 = -t_0$.

- Intrinsic term: The intrinsic term, $p_0(\theta_i)$, is calculated for each θ by subtracting the corresponding translation term, $T(\theta_i)$ from $p(\theta_i)$ of the reference object.

- Δp values: $\Delta p(\theta_i) = \hat{p}(\hat{\theta}_i) - p_0(\theta_i)$.

- Slopes: The divider computes the slopes m_is, where $m_i = \frac{-cos\hat{\theta}_i}{sin\hat{\theta}_i}$ and $i = 0, 1, \ldots, n_\phi$. In this case, the variables in eqn.(4) are $k = 7$, $m = 14$, $NUM = cos\hat{\theta}_i$, and $DENOM = sin\hat{\theta}_i$.

- Intercepts: The divider computes $y-$intercepts b_i, where $b_i = \Delta p_i / sin\hat{\theta}_i$ for $\theta_i > \hat{\theta}_i$ or $b_i = \Delta p_i/(-sin\hat{\theta}_i)$ for $\theta_i < \hat{\theta}_i$. Hence, in the computation of $y-$intercepts, the variables in eqn.(4) are $k = 7$, $m = 14$, and $NUM = \Delta p_i$. For $DENOM$, it is equal to $-sin\hat{\theta}_i$ for $\theta_i > \hat{\theta}_i$ and $sin\hat{\theta}_i$ for $\theta_i < \hat{\theta}_i$.
- (c'_x, c'_y): The divider computes $c'_x = \frac{\sum_{i=1}^{n'} x'_i}{n'}$ and $c'_y = \frac{\sum_{i=1}^{n'} y'_i}{n'}$, where (x'_i, y'_i) is the dominant point in the test object (at time t_b) and n' is the number of dominant points. The variables in eqn.(4) are $k = 7$, $m = 7$, $NUM = \sum_{i=1}^{n'} x'_i$ and $DENOM = -n'$ for c'_x, and $k = 7$, $m = 7$, $NUM = \sum_{i=1}^{n'} y'_i$ and $DENOM = -n'$ for c'_y.

Once the above values are computed, the vote accumulator computes the votes in the region around (c'_x, c'_y) with x values in the range from $c'_x - l$ to $c'_x + l$ and y values in the range from $c'_y - \Delta y$ to $c'_y + \Delta y$, where Δy specifies the range of y values. Each cell computes the y values corresponding to some x value in the specified range. If the computed y values is outside the specified range, the maximum stored vote and the corresponding y value are not updated. A comparison of these stored votes among the cells is done to obtain $(\hat{c}_x, \hat{c}_y)$, the refined center point of the test object at time t_b. Finally, the translation vector $(\hat{c}_x - c_x, \hat{c}_y - c_y)$ is obtained using a simple subtractor, where (c_x, c_y) are the coordinates of the centroid of the polygonal approximation of the obect at time t_a. The computation of (c_x, c_y) is similar to (c'_x, c'_y), which is done at the end of processing the reference object at time t_a.

6 Experimental Results

In this section, two examples are presented to illustrate the proposed 2-D motion estimation system. The experimental results are obtained by simulating the complete hardware system (gate level) using the Quicksim tool of Mentor Graphics. The motion estimation results of the same examples obtained from running the C code of the proposed algorithm in [14] (software implementation) are also included for comparison.

Example 4:

Figure 26 shows the reference object $C1$ (at time t_a) and the test object $C2$ (at time t_b after translation by $(104, 44)$ and rotation by 20 degrees). The center of the reference object $C1$ is at $(70, 137)$ and the center of the test object $C2$ is at $(174, 181)$. The seven dominant points detected for $C1$ and $C2$ are indexed on the contours (see Figure 26). The straight-line approximation of the contour between two dominant points is computed with quantization intervals of $\Delta p = 1.0$, and $\Delta\theta = 1^o$. The $\{(\theta, p)\}$ and $\{(\hat{\theta}, \hat{p})\}$ for $C1$ and $C2$ are listed in Figure 27.

The estimated rotation angle $\tilde{\phi}$ corresponding to the highest vote is 20^o. The refined rotation angle $\phi = 20.25^o$ is obtained by averaging the angles around $\tilde{\phi}$ with the maximum deviation $\Delta\tilde{\phi} = 5^o$. Based on this rotation angle, a one-on-one correspondence between $\{(\theta, p)\}$ and $\{(\hat{\theta}, \hat{p})\}$ is established.

The centroid of the polygon approximation of the reference object $C1$ is $(c_x, c_y) = (67, 133)$. The centroid of the test object $C2$ is first approximated by the centroid of its polygon approximation, $(c'_x, c'_y) = (177, 185)$ and is then refined by computing the inverse Hough transform of $\{(\theta, \Delta p)\}$. By doing such a coarse to fine search, the voting in the $x - y$ plane is computed in a small area (20×20 pixels) around $(177, 185)$. The refined $(\hat{c}_x, \hat{c}_y)$ corresponding to the highest vote count is equal to $(172, 181)$ and the translation vector is $(105, 48)$. Figure 28 summarizes the motion estimation results. The comparison indicates that the experimental results obtained from the hardware implementation are quite satisfactory. □

Example 5:

Figure 29 shows the reference object $C3$ (at time t_a) and the test object $C4$ (at time t_b after translation by $(-125, -34)$ and rotation by 15 degrees). The center of the reference object $C3$ is at $(189, 174)$ and the center of the test object $C4$ is at $(64, 140)$. The dominant points detected for $C3$ and $C4$ are indexed on the contours (see Figure 29). The straight-line approximation of the contour between two dominant points is computed with quantization intervals of $\Delta p = 1.0$, and $\Delta\theta = 1^o$. The $\{(\theta, p)\}$ and $\{(\hat{\theta}, \hat{p})\}$ for $C3$ and $C4$ are listed in Figure 30.

The estimated rotation angle $\tilde{\phi}$ corresponding to the highest vote is 10^o, and the refined rotation angle ϕ is equal to 12.25^o. The maximum deviation $\Delta\tilde{\phi} = 8^o$. For the reference object $C3$, $(c_x, c_y) = (189, 174)$. For the test object $C4$, $(c'_x, c'_y) = (61, 143)$, and the refined $(\hat{c}_x, \hat{c}_y)$ is equal to $(60,145)$. The translation vector is thus equal to $(-129,-32)$. The voting in the $x-y$ plane is computed in a small area (20×20 pixels) around $(61,143)$. Figure 31 summarizes the motion estimation results. The experimental results obtained from Example 4 and 5 indicate that the difference between the simulation results of hardware and software implementations is caused by the limited precision of hardware implementation (i.e. fixed number of bits for angles, slopes, and so on) and the accompanying accumulated errors. In addition, the 2-D motion estimation system has larger errors for an object with a curved boundary than for a simple polygon. This is because the discretization errors for curved boundaries are larger than for straight boundaries. Thus, the displacements in the positions of the dominant points are larger for curved boundaries and the numbers of detected dominant points for the same object with a curved boundary at two different times (after rotation and translation) can sometimes be even different (see Example 5).

7 Conclusion

In this paper, we have presented a 2D motion estimation system based on the straight-line Hough transform. The system is based on the motion estimation technique of [14] that represents objects uniquely by straight-line approximations of the boundary $\{(\theta, p)\}$, and estimates the motion parameters from shifts in the $\theta - p$ space. The system has been implemented using Mentor Graphics tools. The experimental results obtained from the hardware implementation have been presented. A comparison with the software implementation indicates that the experimental results are quite accurate. Moreover, the 2-D motion estimation system has larger errors for an object with a curved boundary than for a simple polygon. This is because the discretization errors for curved boundaries are larger than for straight boundaries.

References

[1] K.M.Yang et. al., "A family of VLSI designs for the motion compensation block matching algorithm," *IEEE Trans. Circuits and Systems*, vol. 36, pp. 1317-1325, Oct. 1989.

[2] C.H.Hsieh and T.P.Lin, "VLSI architecture for block matching motion estimation algorithm," *IEEE Trans. Circuits and Systems*, vol. 2, pp. 169-175, June 1992.

[3] Y.S.Jehng et. al., "An efficient and simple VLSI tree architecture for motion estimation algorithms," *IEEE Trans. Signal Process.*, vol. 41, pp. 889-899, Feb. 1993.

[4] C.L.Wang et. al., "A high-throughput flexible VLSI architecture for motion estimation," *IEEE ICASSP*, vol 5, pp. 3295-3298, 1995.

[5] M.C.Lu and C.Y.Lee, "Semi-systolic array based motion estimation processor design," *IEEE ICASSP*, vol 5, pp. 3299-3302, 1995.

[6] Y.S.Jehng et. al., "Realization of array architectures for video compression algorithms," Proceedings of *ISCAS*, pp. 1672-1675, 1992.

[7] G.Gupta and C.Chakrabarti, "Architectures for hierarchical and other block matching algorithms," *IEEE Trans. Circuits and Systems for Video technology,* pp. 477-489, Dec. 1995.

[8] D.Casasent and R.Krishnapuram, "Curved object location by Hough transformations and inversions," *Patt. Recogn.,* vol. 20, no. 2, pp. 181-188, 1987.

[9] D.C.W.Pao, H.F.Li, and R.Jayakumar, "Shapes recognition using the straight line Hough transform: theory and generalization," *IEEE Trans. on PAMI*, vol. 14, no. 11, pp. 1076-1089, Nov. 1992.

[10] K.Hanahara et. al., "A real time processor for the Hough transform", *IEEE Trans. PAMI*, vol. 10, pp. 121-125, 1988

[11] F.M.Rhodes et. al., "A monolithic Hough transform processor based on restructurable VLSI", *IEEE Trans. PAMI*, vol-10, pp. 106-110, 1988

[12] A.Antola et. al., "A chip-set for the Hough transform", *VLSI Signal Processing VI*, pp. 129-138, 1993.

[13] J.Schonfeld and P.Pirsch, "Image processing board for real-time extraction of line symbols from video sequences," VLSI Signal Processing VI, 1993, pp. 30-38.

[14] H.L.Li and C.Chakrabarti, "Motion estimation of two-dimensional objects based on the straight-line Hough transform: A new approach," *Pattern Recognition*, vol. 29, no. 8, pp. 1245-1258, 1996.

[15] H.L.Li and C.Chakrabarti, "Hardware implementation of a 2-D motion estimation system based on the Hough transform", Proceedings of *ISCAS*, vol. 4, pp. 260-263, 1996.

[16] C.H.Teh and R.T.Chin, "On the detection of dominant points on digital curves," *IEEE Transactions on PAMI*, vol. 11, no. 8, pp. 859-872, 1989.

[17] J.E.Volder, "The CORDIC trigonometric computing technique," *IRE Trans. Electron. Comput.* pp. 330-334, Sep. 1959.

[18] I.Agi et. al., "A VLSI processor for parallel contour tracing," *IEEE Trans. Signal Process.*, vol. 40, pp. 429-438, Feb. 1992.

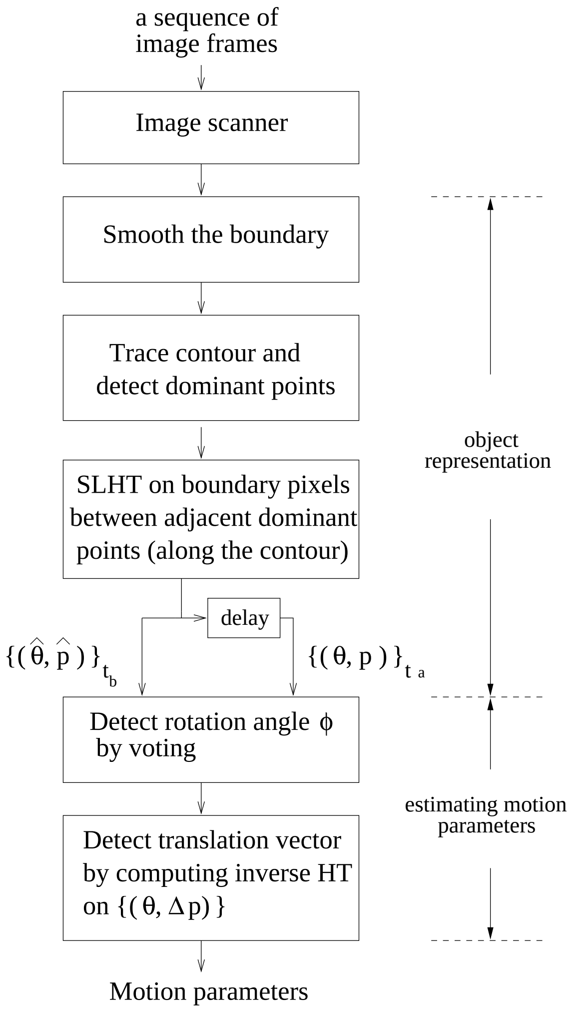

Figure 1: Block diagram of the motion estimation algorithm.

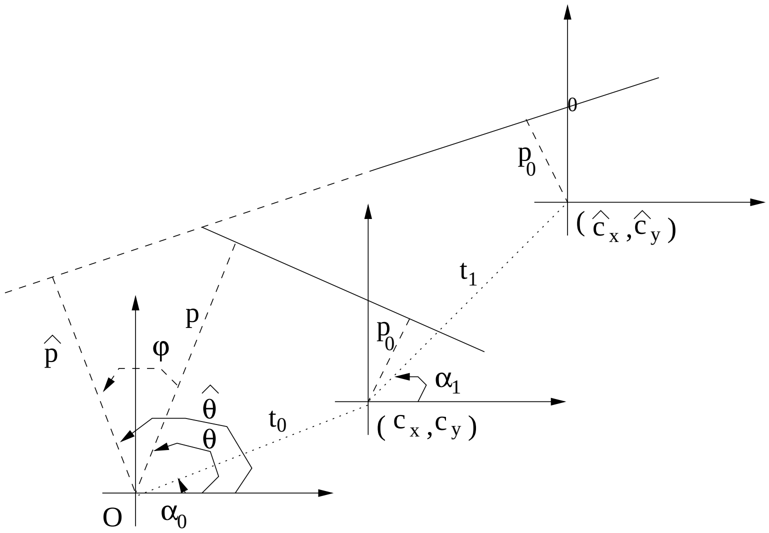

Figure 2: SLHT of a line segment (θ, p) after translation by $(\hat{c_x} - c_x, \hat{c_y} - c_y)$ and rotation by ϕ.

	preprocessing	straight line approximation of the boundary	motion estimation
proposed method	$O(N^2)$	$O(\hat{m}_\theta * P)$	$O(n^2 + b)$
Pao et al's method	$O(N^2 + m_\theta * P)$		$O(m_\theta^2 * K_D + N^2)$

Figure 3: Comparison of the computational complexities of Pao et al's method and our method for motion estimation.

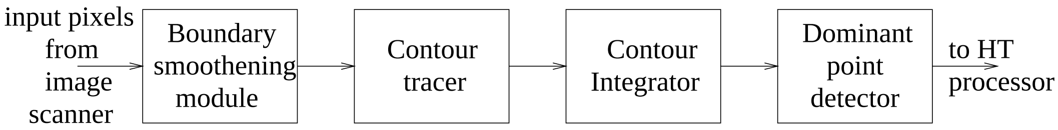

Figure 4: Block diagram of the preprocessor.

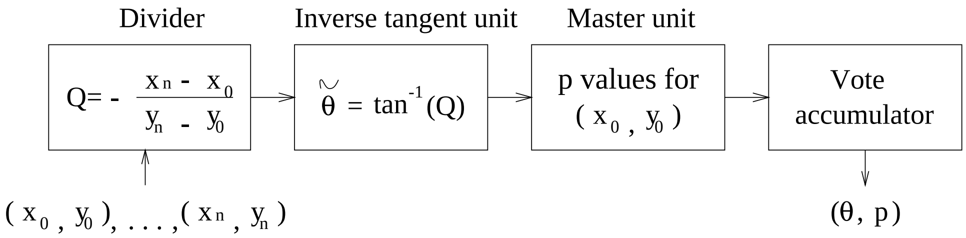

Figure 5: Block diagram of the HT-processor.

0	1	0
1	1	1
1	1	1

(a)

1	0	1
1	1	1
1	1	1

(b)

Figure 6: (a) Once this state is detected, the noise pixel (the circled one) is removed. (b) Once this state is detected, a pixel is added to the circled position.

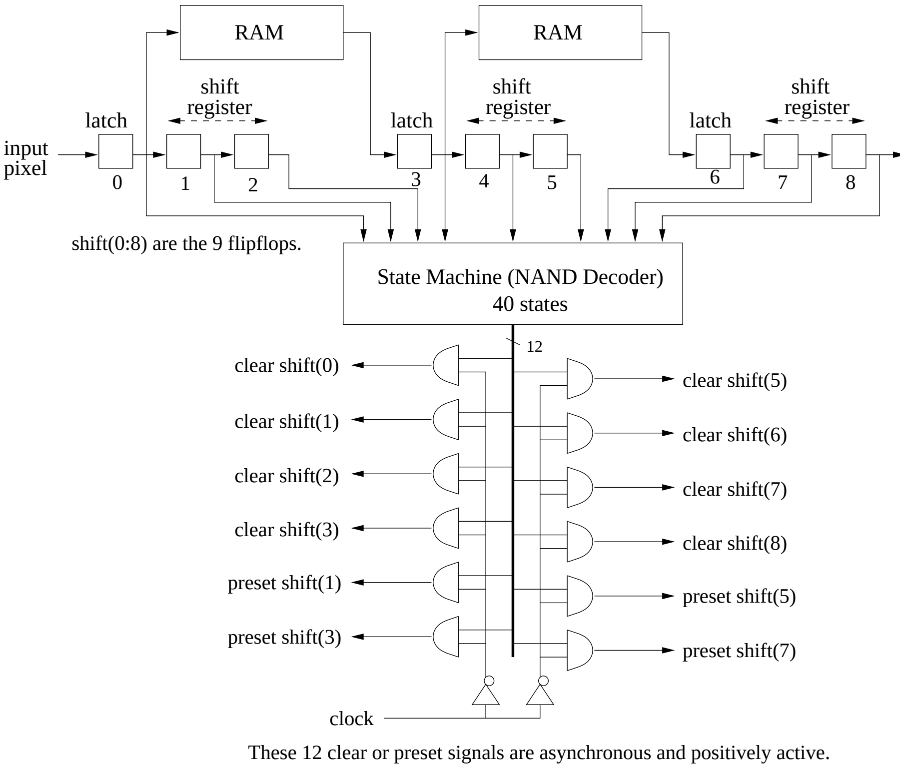

Figure 7: Architecture for boundary smoothening module.

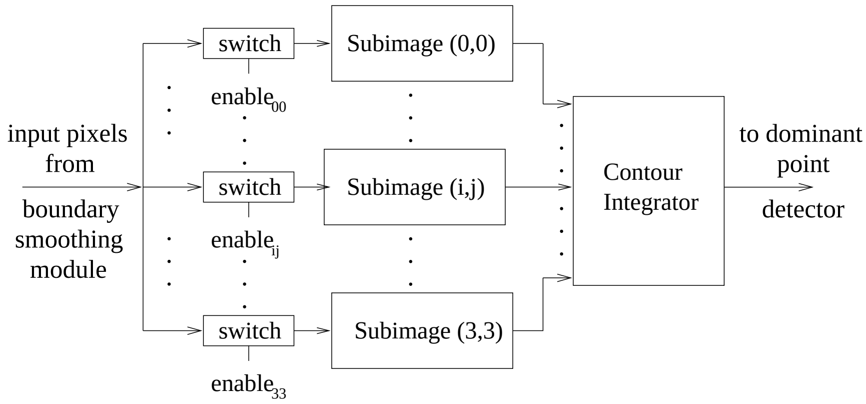

Figure 8: Contour tracer architecture.

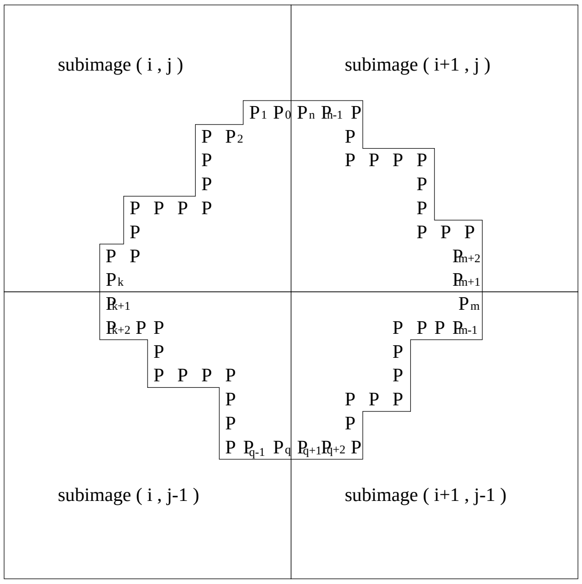

Figure 9: The 2-D object in Example 1.

subimage (i , j)		subimage (i , j-1)		subimage (i+1 , j-1)		subimage (i+1 , j)	
contour memory		contour memory		contour memory		contour memory	
0	(x_1 , y_1) C_0	0	(x_{k+2} , y_{k+2}) C_{k+1}	0	(x_{q+2} , y_{q+2}) C_{q+1}	0	(x_{m+2} , y_{m+2}) C_{m+1}
1	(x_2 , y_2) C_1	1	(x_{k+3} , y_{k+3}) C_{k+2}	1	(x_{q+3} , y_{q+3}) C_{q+2}	1	(x_{m+3} , y_{m+3}) C_{m+2}
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮
k-1	(x_k , y_k) C_{k-1}	q-k-2	(x_q , y_q) C_{q-1}	m-q-2	(x_m , y_m) C_{m-1}	n-m-2	(x_n , y_n) C_{n-1}
k	(x_{k+1} , y_{k+1}) C_k	q-k-1	(x_{q+1} , y_{q+1}) C_q	m-q-1	(x_{m+1} , y_{m+1}) C_m	n-m-1	(x_0 , y_0) C_n
end point memory (x_{k+1} , y_{k+1})		end point memory (x_{q+1} , y_{q+1})		end point memory (x_{m+1} , y_{m+1})		end point memory (x_0 , y_0)	
end index memory k		end index memory q-k-1		end index memory m-q-1		end index memory n-m-1	

Figure 10: The results of the contour tracing.

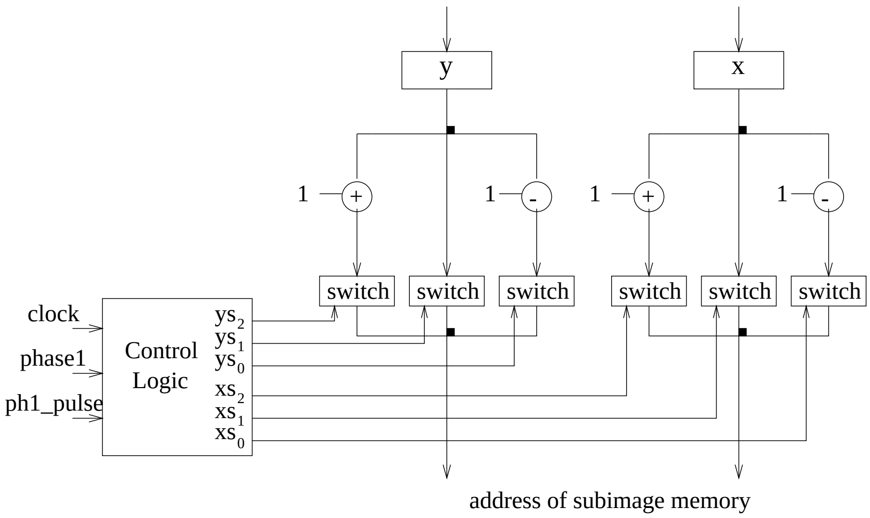

Figure 11: Architecture of the address generator for the subimage memory.

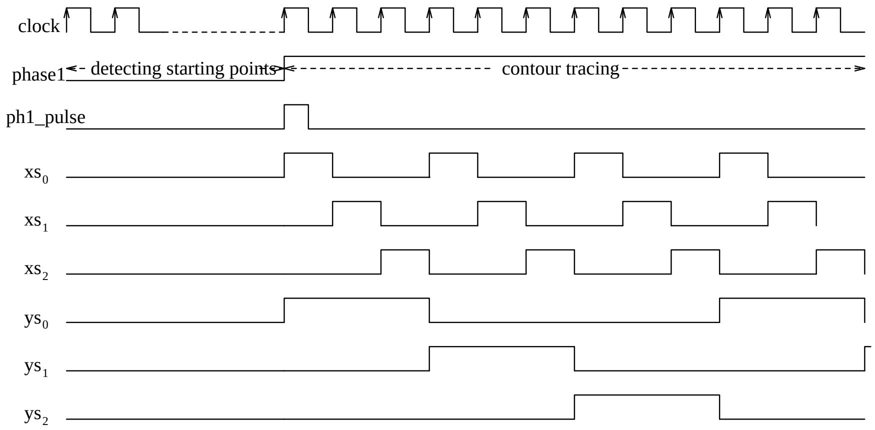

Figure 12: The timing diagram of the six control signals.

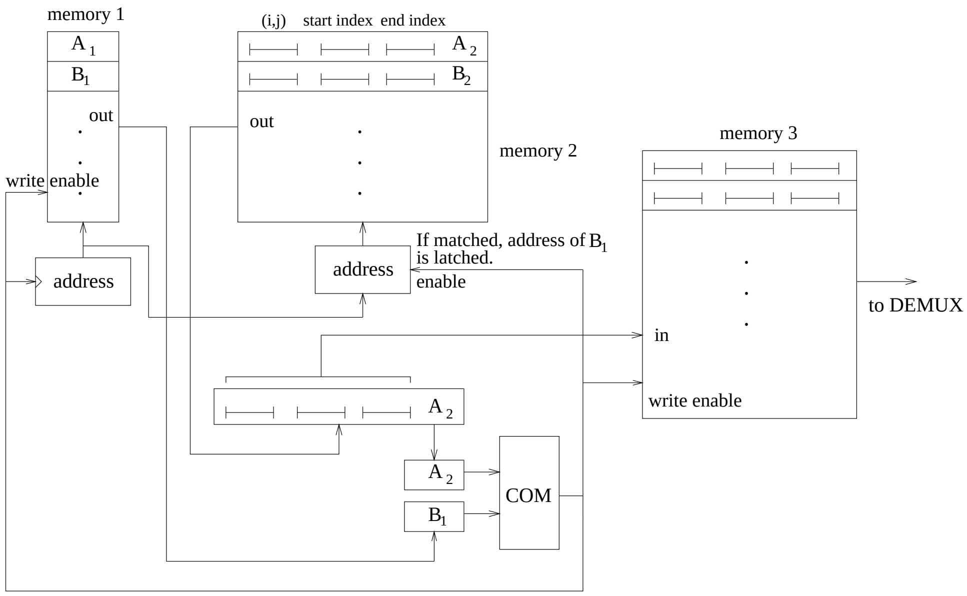

Figure 13: Contour integrator, Part I.

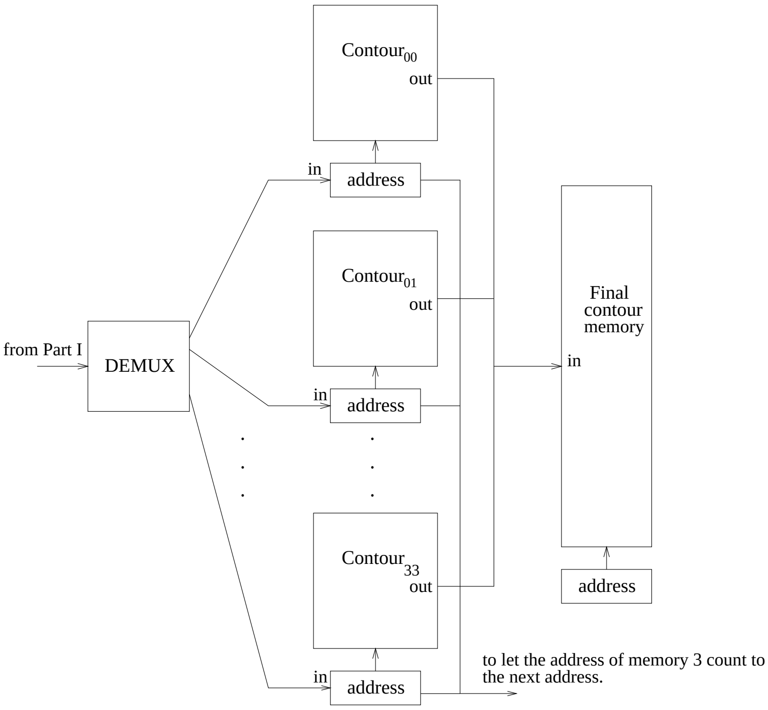

Figure 14: Contour integrator, Part II.

	memory 1	memory 2				memory 3		
	starting pixels	subimage indices	start indices	end indices	end points	subimage indices	start indices	end indices
0	(x_0, y_0)	(i, j)	0	k	(x_{k+1}, y_{k+1})	(i, j)	0	k
1	(x_{m+1}, y_{m+1})	$(i+1, j)$	0	n-m-1	(x_0, y_0)	$(i, j-1)$	0	q-k-1
2	(x_{k+1}, y_{k+1})	$(i, j-1)$	0	q-k-1	(x_{l+1}, y_{l+1})	$(i+1, j-1)$	0	m-q-1
3	(x_{q+1}, y_{q+1})	$(i+1, j-1)$	0	m-q-1	(x_{m+1}, y_{m+1})	$(i+1, j)$	0	n-m-1

Figure 15: The results of the contour integration.

final contour memory		
0	(x_1, y_1)	C_0
1	(x_2, y_2)	C_1
⋮	⋮	
k-1	(x_k, y_k)	C_{k-1}
k	(x_{k+1}, y_{k+1})	C_k
k+1	(x_{k+2}, y_{k+2})	C_{k+1}
k+2	(x_{k+3}, y_{k+3})	C_{k+2}
⋮	⋮	
q-1	(x_q, y_q)	C_{q-1}
q	(x_{q+1}, y_{q+1})	C_q
q+1	(x_{q+2}, y_{q+2})	C_{q+1}
q+2	(x_{q+3}, y_{q+3})	C_{q+2}
⋮	⋮	
m-1	(x_m, y_m)	C_{m-1}
m	(x_{m+1}, y_{m+1})	C_m
m+1	(x_{m+2}, y_{m+2})	C_{m+1}
m+2	(x_{m+3}, y_{m+3})	C_{m+2}
⋮	⋮	
n-1	(x_n, y_n)	C_{n-1}
n	(x_0, y_0)	C_n

Figure 16: The contents of the final contour memory.

	Obvious corner	Convex or concave region
0-group	6, 7, 1, 2	3, 4, 5
1-group	2, 3, 0, 7	4, 5, 6
2-group	0, 1, 3, 4	5, 6, 7
3-group	1, 2, 4, 5	0, 6, 7
4-group	2, 3, 5, 6	0, 1, 7
5-group	3, 4, 6, 7	0, 1, 2
6-group	4, 5, 0, 7	1, 2, 3
7-group	0, 1, 5, 6	2, 3, 4

Figure 17: Chain codes for front and rear pointers corresponding to different groups.

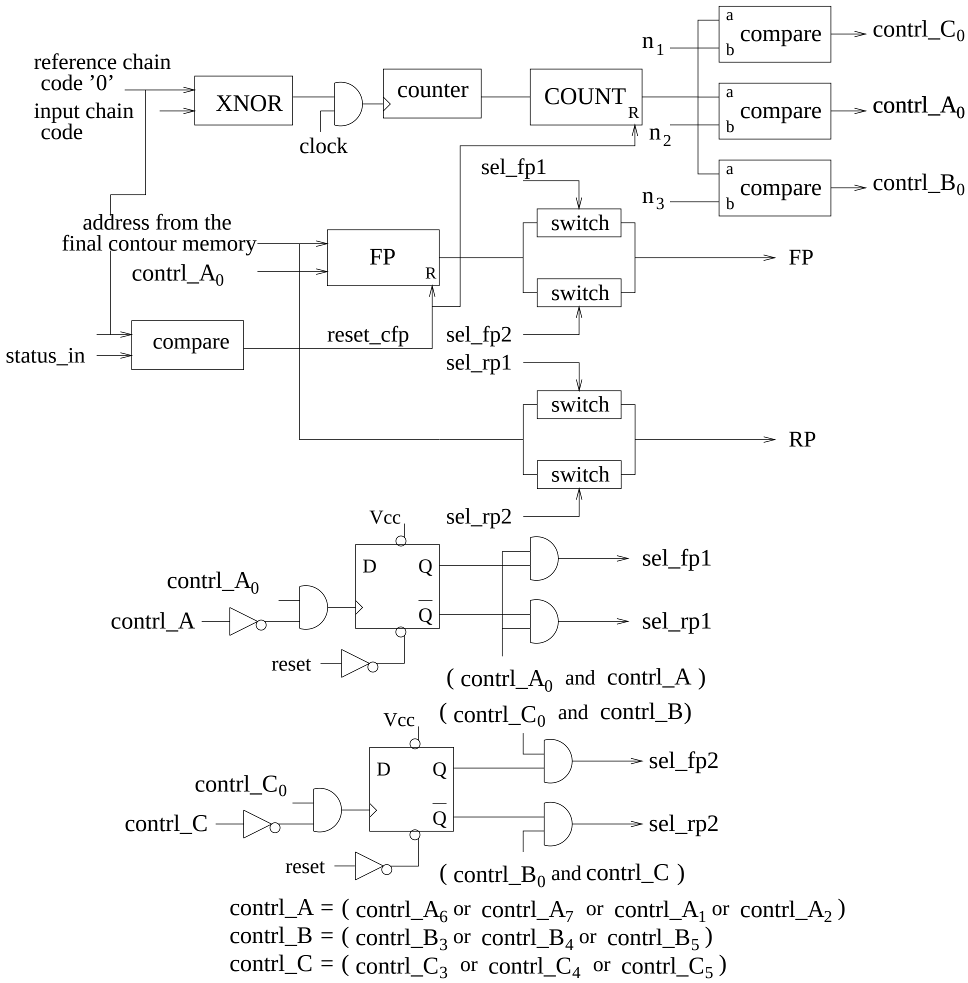

Figure 18: Architecture of the 0-cell.

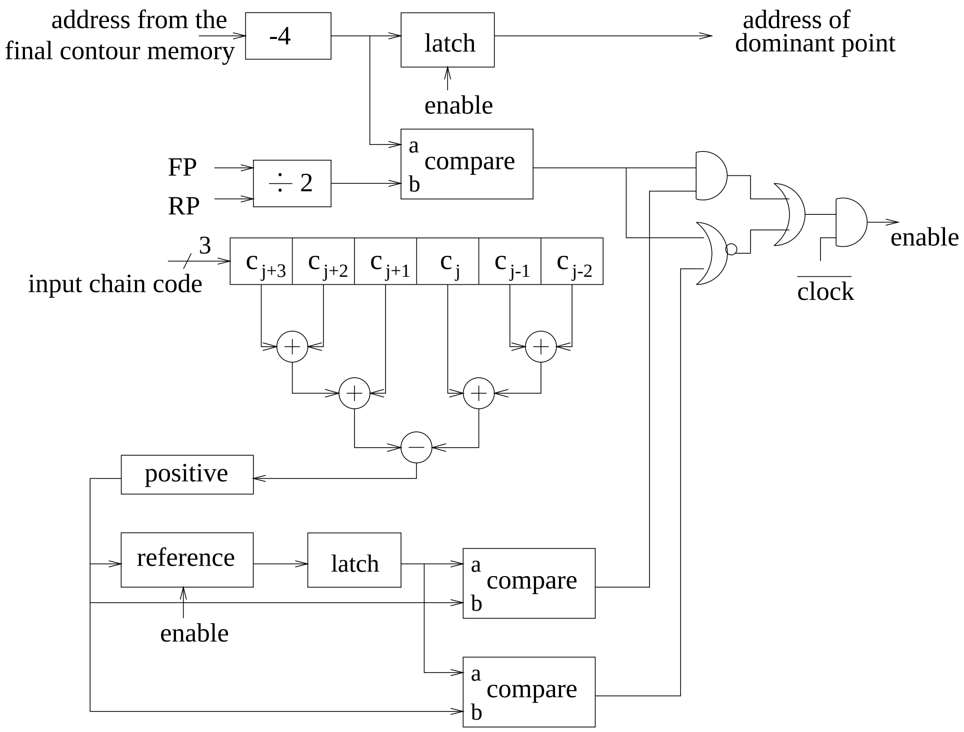

Figure 19: Architecture of the second part in the dominant point detector.

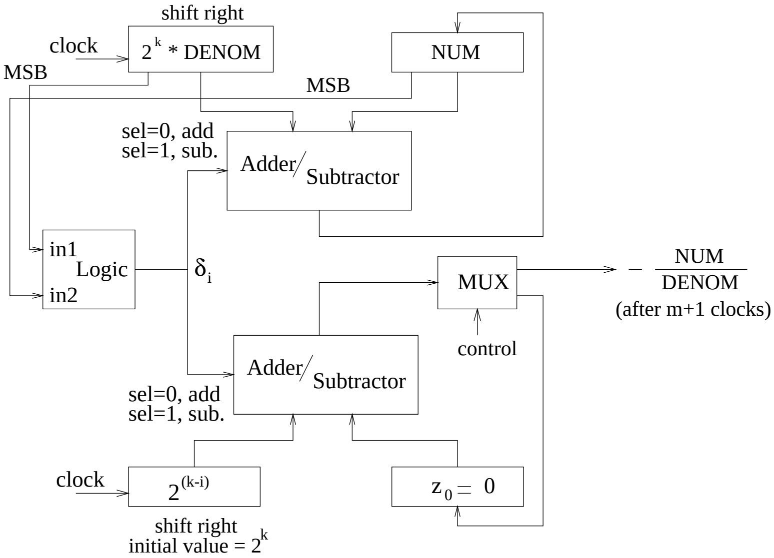

Figure 20: Architecture of the divider.

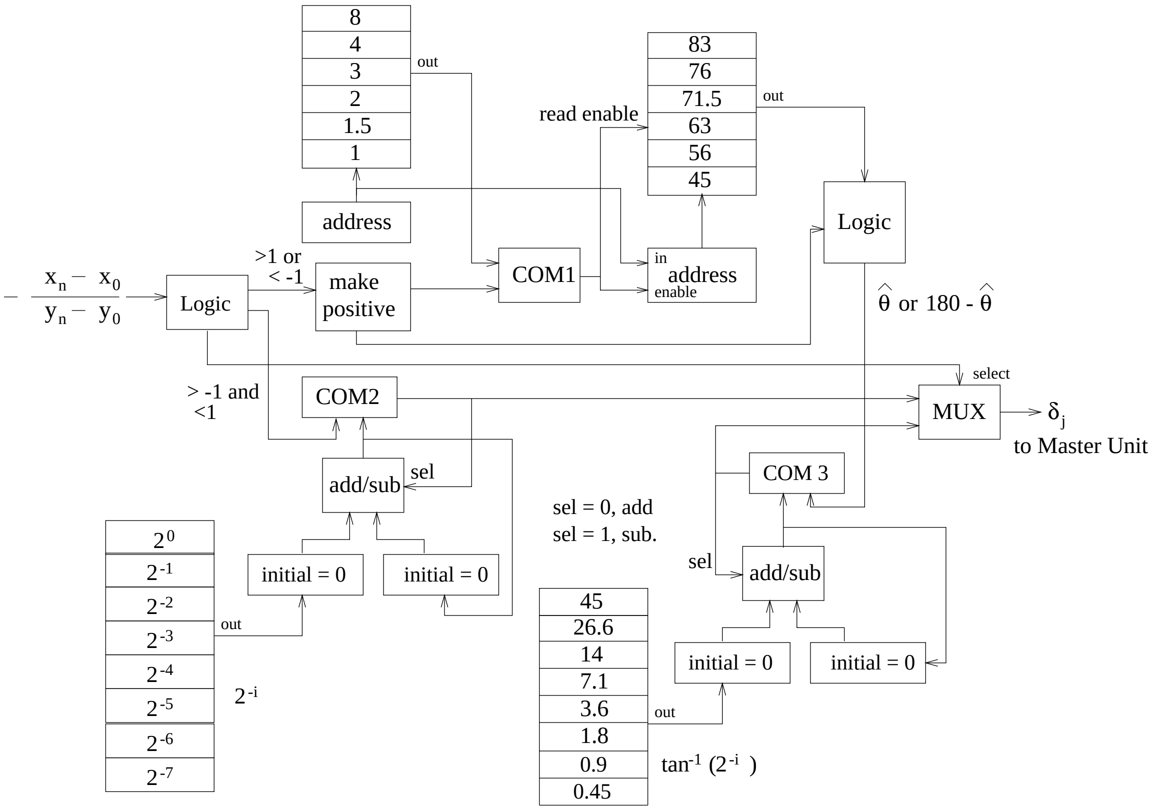

Figure 21: Architecture of the inverse tangent unit.

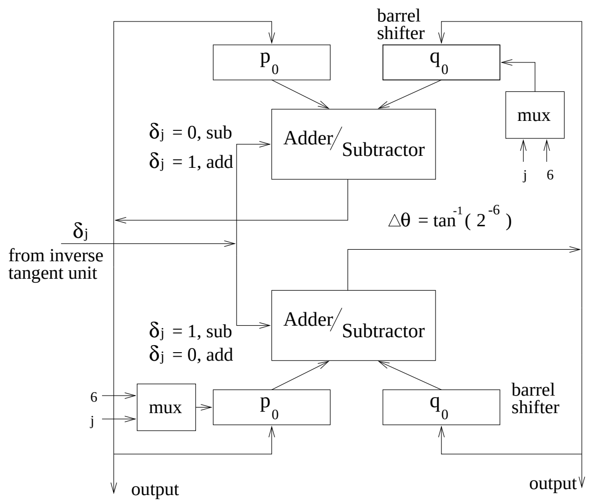

Figure 22: Architecture of the subunit in the HT-processor.

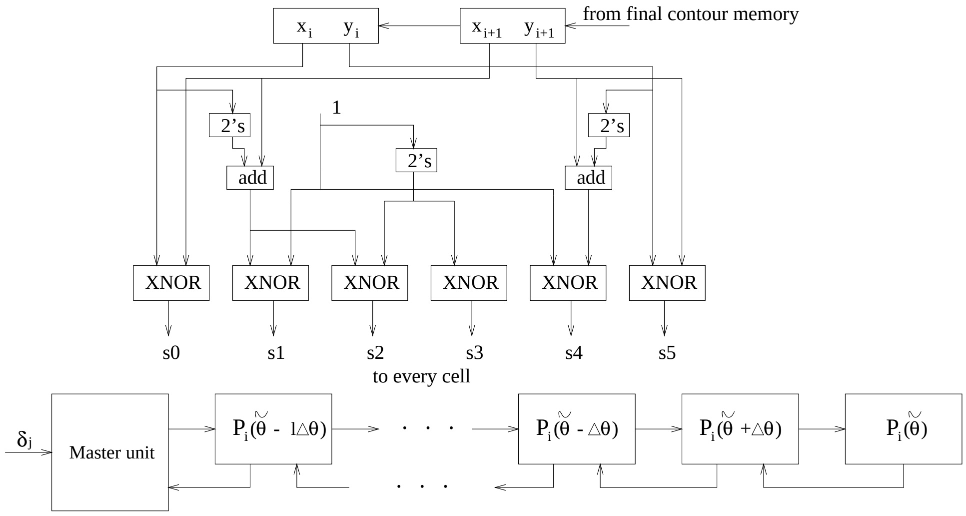

Figure 23: Architecture of the vote accumulator.

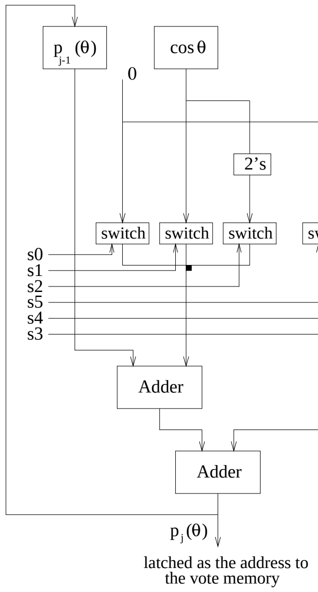

Figure 24: The circuit for computing p_i from p_{i-1}.

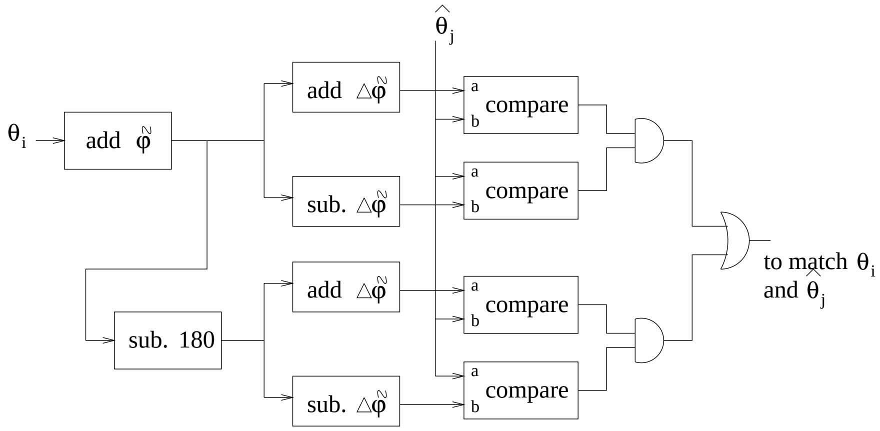

Figure 25: The circuit for matching θ_i and $\hat{\theta}_j$.

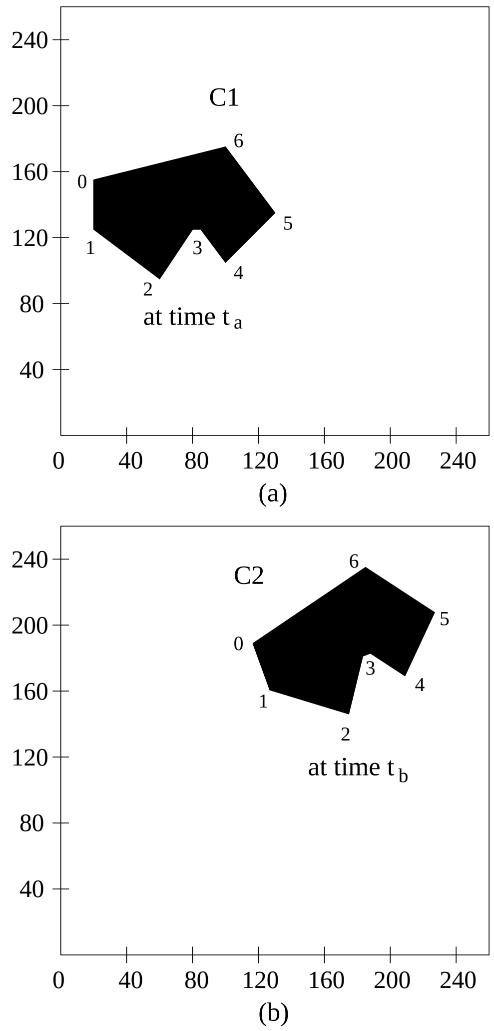

Figure 26: Example 4: (a) C1: reference object at time t_a (b) C2: test object at time t_b.

	Hardware implementation				Software implementation			
	C1		C2		C1		C2	
	p	θ	$\hat{p}$	$\hat{\theta}$	p	θ	$\hat{p}$	$\hat{\theta}$
1	20	0	164	17	17	0	164	15
2	110	52	186	71	112	53	189	70
3	11	143	-128	163	3	146	-134	165
4	143	38	251	56	145	40	247	59
5	5	134	-133	156	7	133	-124	155
6	184	35	295	55	185	36	292	57
7	148	103	126	128	145	104	85	124

Figure 27: Example 4: $\{(\theta, p)\}$ values of $C1$ and $\{(\hat{\theta}, \hat{p})\}$ values of $C2$.

	$(\hat{C}_x, \hat{C}_y)$	(C_x, C_y)	translation vector	error vector	rotation angle	error
hardware implementation	(172 , 181)	(67 , 133)	(105 , 48)	(1 , 4)	20.25	0.25
software implementation	(173.5 , 180)	(70 , 137)	(103.5 , 43)	(-0.5 , -1)	19.72	-0.28
actual values	(174 , 181)	(70 , 137)	(104 , 44)		20	

Figure 28: Example 4: Summary of the motion estimation results obtained from both hardware and software implementations.

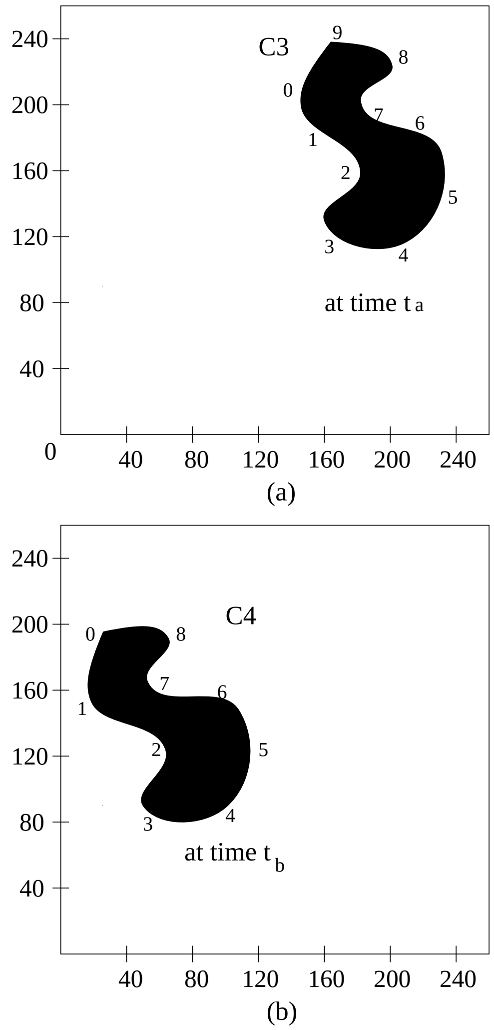

Figure 29: Example 5: (a) C3: reference object at time t_a. (b) C4: test object at time t_b.

	Hardware implementation				Software implementation			
	C3		C4		C3		C4	
	p	θ	$\hat{p}$	$\hat{\theta}$	p	θ	$\hat{p}$	$\hat{\theta}$
1	156	8	-10	173	-126	174	0	174
2	239	59	144	69	243	52	141	62
3	-105	164	-76	182	-98	154	-54	176
4	162	82	79	91	166	71	91	82
5	-68	135	-6	135	-66	134	-8	135
6	218	0	105	1	227	0	125	5
7	213	72	164	85	237	73	167	84
8	-75	169	12	166	-60	150	20	156
9	296	63	199	91	301	56	204	81
10	1	144			0	145		

Figure 30: Example 5: $\{(\theta, p)\}$ values of $C3$ and $\{(\hat{\theta}, \hat{p})\}$ values of $C4$.

	$(\hat{C}_x, \hat{C}_y)$	(C_x, C_y)	translation vector	error vector	rotation angle	error
hardware implementation	(60 , 145)	(189 , 177)	(-129 , -32)	(-4 , 2)	12.25	-2.75
software implementation	(63 , 141.5)	(189 , 174)	(-126 , -32.5)	(-1 , 1.5)	13.5	-1.5
actual values	(64 , 140)	(189 , 174)	(-125 , -34)		15	

Figure 31: Example 5: Summary of the motion estimation results obtained from both hardware and software implementations.