A Full Custom ASIC Design for the Real-Time Generation of Lowpass and Bandpass Multiresolution Image Representations

by

Jonathan D. Loewen

A thesis
presented to the University of Manitoba
in partial fulfillment of the
requirements for the degree of
Master of Science
in
Electrical Engineering

Winnipeg, Manitoba, 1987

ISBN 0-315-37468-3

A FULL CUSTOM ASIC DESIGN FOR THE REAL-TIME GENERATION OF LOWPASS AND BANDPASS MULTIRESOLUTION IMAGE REPRESENTATIONS

BY

JONATHAN D. LOEWEN

A thesis submitted to the Faculty of Graduate Studies of the University of Manitoba in partial fulfillment of the requirements of the degree of

MASTER OF SCIENCE

I hereby declare that I am the sole author of this thesis.

I authorize the University of Manitoba to lend this thesis to other institutions or individuals for the purpose of scholarly research.

Jonathan Loewen

I further authorize the University of Manitoba to reproduce this thesis by photocopying or by other means, in total or in part, at the request of other institutions or individuals for the purpose of scholarly research.

Jonathan Loewen

The University of Manitoba requires the signatures of all persons using or photocopying this thesis. Please sign below, and give address and date.

ABSTRACT

Many image processing and analysis tasks can be accomplished efficiently by using multiresolution representations of an image. A fixed computational cost is spent to transform the information content of an image into a multiresolution representation which enables the use of very efficient image processing and analysis algorithms. This thesis describes the design of an application specific integrated circuit (ASIC) which may be cascaded to produce a system capable of generating a Gaussian (lowpass) filtered multiresolution representation and a Difference of Gaussian (bandpass) filtered multiresolution representation of an image in real time. The design of the ASIC is based upon a systolic architecture, however modifications to the architecture are introduced to take advantage of separability of the Gaussian function and hierarchical convolution. High degrees of pipelining and exploitation of the *flow-through* nature of the algorithm enable the ASIC to achieve real-time operation. Real-time operation of the ASIC allows implementation of the multiresolution representation system in the pipeline between detector and downline viewing, storing, or further image processing or analysis steps.

Acknowledgements

I would like to extend my appreciation to my advisor, Prof. H.C. Card for his supervision and assistance during the development of this work.

Financial support from the Natural Sciences and Engineering Research Council of Canada and equipment loans from the Canadian Microelectronics Corporation are gratefully acknowledged.

Table of Contents

List of Figures

List of Tables

CHAPTER 1

Introduction

The computational complexity of an information processing problem is determined, in part, by the representation of the information. Many image processing and pattern recognition tasks can be accomplished more efficiently by using an improved representation of the information in the image. Multiple resolution representations produce successively condensed versions of the information in an image. This thesis describes the implementation in silicon of a scheme for the generation of successively reduced resolution representations of an image by (1) condensing image intensity and by (2) providing increasingly coarse approximations to certain descriptive features by condensing the information about those features.

Multiple resolution representations of images was first suggested by Kelly [1] as a method of *planning* to detect edges. Marr and Hildreth [2], Hanson and Riseman [3], and many others have shown the advantages of multiple resolution representations for detecting edges. Marr and Poggio [4], and Moravec [5] have used multiresolution representations in stereo matching. Crowley [6] accomplishes efficient pattern classification and Burt and Adelson [7] have devised a compact image code using multiresolution representations. The motivation for using a multiple resolution representation is to spend a fixed computational cost to transform the information content of an image into a representation which enables the use of very efficient image processing and analysis algorithms.

Image processing applications pose extremely large information processing problems due to the high resolution (many pixels or data elements per image) and the fast data acquisition frequently required (particularly in the fields of satellite imagery and medical imaging). Multiresolution representations can be used to reduce the computational cost of many image processing operations by using divide-and-conquer principles. For example, feature detection can be performed efficiently at lower resolutions and the results used to constrain the search at higher resolutions. Searching becomes fast since a global region in the original image can be detected using local operators at a low resolution. In parallel hardware implementations of image processing algorithms, local operators require considerably less communication and message passing and thus exhibit faster execution times and simpler control than global operations. In a multiresolution representation, global information is condensed at lower resolutions thus allowing use of efficient local operators to detect and extract the global information. A multiresolution representation therefore enables extraction of both local and global information using simple, fast, and efficient local operators as well as restricting the resolution at which the information is extracted.

1.1. Gaussian-Filtered Multiresolution Representation

Multiple resolution representations provide successively condensed representations of the information in an image. A simple scheme might generate successively reduced resolution representations by averaging image intensities in non-overlapping two by two blocks of pixels and subsampling. Repeated application of this process produces the multiresolution representation shown in Figure 1 where level 0 (l_0) is the original image of size $2^n \times 2^n$; level 1 (l_1) has been averaged and subsampled once producing a $2^{n-1} \times 2^{n-1}$ image. Repeated applications produce exponentially decreasing image sizes until a single pixel image is generated. Stacking these images on top of one

another forms a pyramid and thus some authors refer to this multiresolution representation as a pyramid structure.

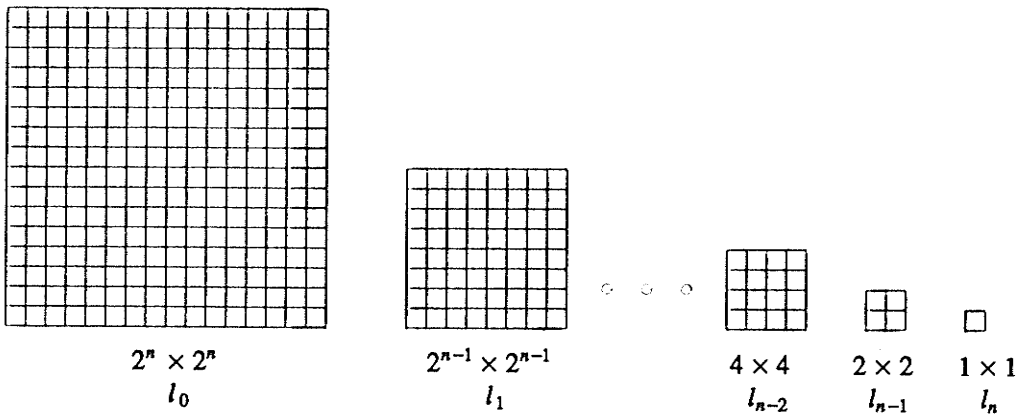


Figure 1: Multiresolution representation of an image.

Averaging image intensities in non-overlapping two by two blocks amounts to low-pass filtering of the image. The low-pass filtering effectively reduces the high frequency content of the image output at each stage and is required to reduce the aliasing error introduced by subsampling. Frequency domain evaluation of a non-overlapping unweighted square region average implementation for low-pass filtering reveals a wide-band characteristic or *ringing* effect. The *ringing* characteristic or high frequency pass-bands beyond the principal pass-band pass some high frequency image content as well as passing certain high frequency noise. The inability to suppress all high frequency noise is a disadvantage since later processing steps such as edge detection typically enhance noise thus producing false edges and distortions. These unwanted features can be avoided by using an overlapping, weighted circular region average in the filtering stage. The optimal averaging filter to be used is determined by two physical considerations. The first is the frequency domain consideration of wide-band response, ringing characteristics and high frequency noise. These problems are avoided by using a filter whose frequency spectrum is smooth and bandlimited; ie. its

variance in the frequency domain should be small. The second consideration is a spatial domain consideration and is due to the fact that the visual world is constructed of *things* which are spatially localized relative to a certain scale. These *things* give rise to intensity changes in an image and consist of light sources, shadows, illumination gradients, changes in orientation or distance, and changes in surface reflectance [2]. The spatial localization of intensity changes is in fact the essence of a multiresolution representation. All of the *things* represented in an image are, at some resolution, spatially localized. Thus, the contributions to each point in the filtered image should arise from a smooth average of nearby points rather than any kind of average of widely scattered points. Furthermore, when physically implemented, an algorithm which averages a local region is less computationally expensive and requires only local communication. Thus, the optimal filter should be smooth and localized in the spatial domain; ie. its variance in the spatial domain will be small. The optimal filter therefore has the conflicting requirements of spatial localization and frequency localization. The best compromise in satisfying these conflicting requirements is obtained using a Gaussian filter [2] described as

$$G(x, y) = \frac{1}{2\pi\sigma^2} e^{\frac{-(x^2 + y^2)}{2\sigma^2}} \tag{1.1}$$

in two dimensions.

Use of a Gaussian filter eliminates the wide-band response and high frequency pass-bands associated with an unweighted square region average. The high frequency content of the image is reduced at each level in the multiresolution representation thus enabling resampling of the image without aliasing errors. Furthermore, intensity changes in an image become localized at some level in the representation. Repeated filtering of an image with Gaussian filters of appropriate standard deviation followed by resampling will generate scaled copies of the original image. The resultant

multiresolution representation has been found appropriate for use in motion analysis, texture analysis, image segmentation, and image property algorithms [8].

1.2. Multiresolution Representation based on Difference of Gaussians

The Gaussian-filtered multiresolution representation effectively provides copies of an image with successively reduced resolution by condensing image intensity. It is also often desired to provide reduced resolution copies, corresponding to increasingly coarse approximations to certain descriptive features, by condensing the information about these features. Such a representation, while still enabling the complete recovery of the original image, has the advantages of data compression and of making the salient information readily available. Representing size-scaled copies of an image with a reduced amount of data will decrease the necessary memory or storage requirements and increase the throughput rate of the system performing the relevant image processing and analysis tasks. Furthermore, these tasks will be simpler and more efficiently implemented. Obtaining a complete, compact description of the most meaningful image information is viewed as the first step in visual information processing [2, 9]. This first step involves representing the intensity changes or *edges* which correspond to the reflectance and illumination of visible surfaces and their orientation and distance relative to the point of observation.

In natural images, intensity changes occur over a wide range of resolutions. To adequately characterize the intensity changes in the image in terms of the physical processes that originated them, the intensity changes must be detected at all resolutions at which they occur. This process suggests characterizing edges within a multiresolution framework. Furthermore, a Gaussian-filtered multiresolution representation is the optimal framework within which to characterize edges [2].

Intensity changes within an image may be detected by comparing intensity values within a limited neighborhood. The rate of change of the intensity values along a path in the image is given by the first derivative. Where the rate of change is large, the absolute value of the first derivative in the direction of change will also be large. For natural images, rapid changes or sharp variations in intensity correspond to physical edges in the objects or surfaces in the image. Thus, extreme values of the first directional derivative localize physical edges. Alternatively, these peaks in the first derivative will correspond to zero-crossings of the second derivative. The intensity changes may then be identified by locating the zero-crossings of

$$D^2[G(x, y) * I(x, y)] \tag{1.2}$$

where D^2 signifies the second derivative in the appropriate direction, $G(x, y)$ corresponds to the Gaussian function used to generate the scaled copies of the original image, $I(x, y)$ is the original image, and * signifies a two-dimensional convolution. By the derivative rule for convolutions,

$$D^2G(x, y) * I(x, y) \tag{1.3}$$

The direction in which the second derivative is taken within the two-dimensional image must also be determined. In smooth images, intensity change near and parallel to an edge will be approximately linear. The linear change stipulates that the second derivative operator with zero-crossings of maximum slope will localize the edge. However, these zero-crossings correspond to the zero-crossings of the Laplacian (∇^2) which is the only orientation-independent second-order differential operator. Use of the Laplacian enables determination of the appropriate zero-crossings for a particular resolution with just one convolution. The process is

$$\nabla^2G(x, y) * I(x, y) \tag{1.4}$$

where the function, $\nabla^2G(x, y)$ is called the Laplacian of a Gaussian [2] or LOG

function and is mathematically expressed as

$$\nabla^2 G(x, y) = \frac{1}{\pi\sigma^4} e^{\frac{-(x^2+y^2)}{2\sigma^2}} \left[\frac{x^2+y^2}{2\sigma^2} - 1 \right] \tag{1.5}$$

The LOG filter is bandpass and responds optimally to a certain range of spatial frequencies of the intensity changes. Implementing the bandpass filter at multiple resolutions effectively reduces the center frequency of the filter as the image resolution decreases. Alternatively, the size of the LOG filter (convolution window size) determines the range of resolution over which it will respond to intensity changes. Thus, intensity changes at different resolutions can be optimally detected by using LOG filters of different sizes. Large filters detect soft edges and overall illumination changes. Smaller filters detect finer detail.

The computational cost and complexity of implementing the LOG filter in hardware [10] can be significantly reduced by approximating the $\nabla^2 G$ function with the difference of two Gaussian functions or DOG function. The $\nabla^2 G$ function is very similar to a DOG function and is in fact the limiting case of the DOG function as σ_1 / σ_2 tends to unity [2]. σ_1 and σ_2 are the standard deviations of the Gaussian functions which are subtracted to form the DOG function. The DOG function is expressed mathematically as

$$DOG(x, y) = \frac{1}{2\pi} \left\{ \frac{1}{\sigma_1^2} e^{\frac{-(x^2+y^2)}{2\sigma_1^2}} - \frac{1}{\sigma_2^2} e^{\frac{-(x^2+y^2)}{2\sigma_2^2}} \right\} \tag{1.6}$$

A computational saving exists since it is now possible to simply subtract consecutive levels of the Gaussian-filtered multiresolution representation to obtain a near-optimal bandpass representation. The resultant bandpass multiresolution representation undergoes data compression by removing pixel to pixel correlations and shifting pixel values

toward zero thus enabling less than eight-bit representations of each pixel. Pixel to pixel correlations are removed when consecutive levels of the Gaussian-filtered multiresolution representation are differenced (see Sec 6.1). This bandpass multiresolution representation has been found appropriate for pattern classification [6], image encoding [7], stereo and motion analysis [4], and other visual information processing applications.

Having determined that a Gaussian multiresolution representation and a Difference of Gaussian multiresolution representation are optimal or near-optimal for many image processing and analysis tasks, we now set about to develop a system capable of producing these representations. The following four chapters detail the development and testing of an application specific integrated circuit (ASIC) which is used to produce these representations. A strict criteria in the design of the ASIC is that it achieves the throughput required by real-time computation. We define real-time computation as the ability to process 30 image frames per second, with an image frame containing 512×512 image elements or pixels where each pixel is represented with eight bits. Other criteria such as area and accuracy are left less rigid in order to make tradeoffs between design criteria and to enable the realization of the ASIC design in the available IC fabrication facilities.

CHAPTER 2

Algorithms and Architectures

The practical value of an algorithm for a problem is ultimately determined by its computational cost. A practical algorithmic solution of a signal processing problem will correspond to an architectural design which allows for extensive pipelining and high degrees of parallelism. It will take advantage of the concurrency present in both the application and the target architecture while observing the requirements of locality and balanced distribution of computation. These requirements enable extensive pipelining of the architecture. The resultant system can therefore hope to meet the very high throughput rate requirements of real-time signal processing applications.

In designing special purpose ASIC's for digital signal processing there are a number of algorithmic issues to be addressed. Almost all digital signal processing algorithms are characterized (1) by the regularity of arithmetic operations (multiplications and additions), (2) by a negligible amount of decision branching, and (3) by a large ratio of computational steps to loop steps during execution. As a result, digital signal processing algorithms tend to exhibit a *flow-through* behaviour with partial results moving from one step to the next with negligible requirements for decision branching or looping. Exploitation of these attributes in the architectural design is necessary to achieve the throughput rates required by real-time computation of the digital signal processing algorithm considered here.

In the present problem the system is being designed to operate in a pipelined fashion between detection and viewing steps. Data is acquired in a continuous stream from the pipe and exits in a continuous stream. Both the algorithm and the architecture must be designed to enable the system computation rate to match the I/O rate of the pipeline. The architecture must also be designed to allow high degrees of pipelining to exploit the flow-through behaviour of the algorithm and in so doing to employ simple, local communications. An FFT implementation and a systolic architecture are evaluated in this chapter in light of these algorithmic and architectural considerations.

2.1. FFT Implementation

FFT implementations are common in digital signal processing applications. The FFT algorithm is derived by decomposing dimensionally-separated discrete Fourier transform pairs to reduce the number of multiply and add operations required. An FFT implementation for this problem instance requires (1) the image to be transformed, (2) a transform of the filter or filters to be hardwired or stored in memory, (3) frequency-domain multiplication of the image and the appropriate filter, and finally (4) the inverse transform of the result.

An FFT requires $\frac{1}{2}N\log_2 N$ multiplications and $N\log_2 N$ additions for a one-dimensional N-point transform. The two-dimensional transform is obtained as a series of one-dimensional transforms; N row transforms followed by N column transforms for an $N\times N$ image. Multiplication in the frequency domain of the image with the filter requires N^2 multiplies. The requirements of the inverse FFT are the same as the forward FFT. The total computational requirements of an FFT implementation therefore are $N^2 + 2N^2\log_2 N$ multiplies and $4N^2\log_2 N$ additions for an $N\times N$ image.

The FFT implementation is independent of the convolution window size in the spatial domain and therefore no advantage can be taken of the fact that a Gaussian filter can be implemented in a hierarchical manner. This necessitates the storage of the entire row or column of the image to complete the appropriate row and column transforms. The repetitive use of a pixel element required by the FFT algorithm necessitates the element to be stored inside the system until the element is no longer required. The memory capacity of the system will therefore influence I/O imposed limitations. For example, performing the N-point fast Fourier transform using an s-point device when N is large and s is small necessitates each subcomputation block to be sufficiently small so that it can be handled by the s-point device. During execution, results of a block must be temporarily sent to the host or some form of memory and later retrieved to be combined with results of other blocks as they become available. To perform an N-point FFT with a device of $O(s)$ memory requires $O(\frac{N \log N}{\log s})$ I/O operations for any decomposition scheme [11]. Thus, the I/O limitations of device to host or device to memory impose an upper bound which is independent of device speed. Furthermore, the entire image must be obtained and stored before the first filtered result can be entirely computed. This imposes an $O(N^2)$ (for an $N \times N$ image) latency or delay when the device is operated in the pipe between detector and further processing or viewing steps.

2.2. Systolic Architecture

A systolic system [12] is an array of processor elements (PE's) which rhythmically compute and pass data through the array. Systolic systems may be one- or two-dimensional with the possibility for data to flow at multiple speeds and in multiple directions through the system. In a systolic design all data, while being *pumped* regularly across the array, can be effectively used in all the PE's. Data flows between cells

in a pipelined fashion and between the system and the outside world only at boundary cells. Thus, the multiple use of each data element within the systolic array allows compute-bound computations to be accelerated without increasing the I/O requirements.

Each processor element in the systolic array performs some simple operation. Thus, the array can be built modularly with simple, regular, local communications between processor elements allowing for very simple data and control flows. I/O and computations are overlapped in this highly pipelined and highly synchronized architecture. Systolic arrays do however require global synchronization (ie. global clock distribution). This may cause clock-skew problems in high-order VLSI system implementations and must be addressed in design and layout stages. Fortunately, for any one-dimensional systolic array, a global clock parallel to the array presents no problem, even if the array is arbitrarily long. The array will operate correctly despite the possibility of a large clock-skew between the two ends. Aside from global synchronization, systolic arrays are modular, have regular interconnects, local comunication, present no difficult synchronization or resource conflict problems and eliminate overhead associated with operations such as address indexing.

A straightforward systolic array implementation of a two-dimensional filter is shown in Figure 2. The size of the array is primarily dependent on the required sidelobe attenuation of the Gaussian low-pass filter. Each computational element in the systolic array contains a weighting coefficient which is equal to the sampled filter coefficient. The filter coefficients do not move. Input data, $x(n)$, flows in one direction through the array and output results, $y(n)$, flow in the opposite direction. The row by row transmission of the image data necessitates the need for shift register buffers at the end of each row in the array to allow the convolution window to overlay the proper region of the image. The size of the shift register is dependent on the array

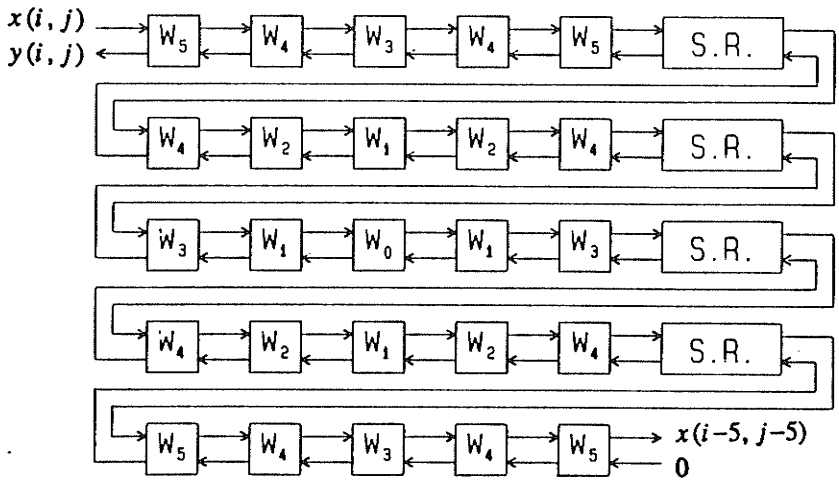


Figure 2: A systolic array implementation of a 5×5 coefficient digital filter with octal symmetry.

size and on the image row length. The convolution is produced by each computational element multiplying its input, $x(n)$, by its weighting coefficient, $w(n)$, then adding this result to the output $y(n)$ from the previous stage, and passing the sum to the next computational element. The latency associated with this implementation is equal to $N \times (M - 1) + M$ where N is the image row length and M^2 is the filter size. Note that the two-dimensional array can be *unfolded* to essentially produce a one-dimensional array. A global clock parallel to the systolic array will enable proper operation despite the possibility of a large clock-skew between the two ends of the array.

Clark and Lawrence [13] have designed a hierarchical system using this method for the generation of a Laplacian of a Gaussian ($\nabla^2 G$) in a multiresolution representation of an image. They have generated multiresolution representations of the image using a half-band low-pass filter (which restricts the maximum frequency to one-half its previous value) and a subsampler followed by $\nabla^2 G$ filters at each resolution. Word lengths of eight bits were used for filter coefficients and image data. A standard deviation $\sigma = \sqrt{2}$ for the $\nabla^2 G$ filter requires 11×11 coefficients for the bandpass filter.

Clark and Lawrence chose a peak sidelobe level of less than -30 db and therefore they required a filter size of 25×25 coefficients for the low-pass filter. Both filters have octal symmetry. Thus not all computational elements are required to perform multiplications, if the input values from each computational element with identical weighting coefficients are first ripple-summed in a cyclical manner, followed by a single multiplication with the coefficient. The ripple summing of input values produces considerable computational savings at the expense of loss of linearity in the systolic array. This latter effect can produce problems with clock-skew, as well as requiring large multipliers since summed input values now require twice as many bits for accurate representation. As results percolate through the systolic arrays they require an increasing number of bits to avoid overflow and retain accuracy. Due to the fact that the low-pass filter requires 625 computational elements and the $\nabla^2 G$ filter requires 121 computational elements, this implementation requires considerable area in a silicon implementation.

The following section describes a design which uses a separable, hierarchical implementation of Gaussian filters to greatly reduce the computational complexity and silicon area requirements.

2.3. Dimensional Separability and Hierarchical Convolution

A straightforward systolic convolution implementation of a Gaussian filter requires M^2 computational elements and $O(M^2N^2)$ multiplies and additions for each image frame, where M^2 is the number of filter coefficients and N^2 is the number of pixels per image frame. Two techniques can be used to reduce both the computational complexity and the number of computational elements in the systolic array: (1) dimensional separability of the two-dimensional Gaussian filter and (2) hierarchical convolution.

Dimensional separability refers to the fact that a two-dimensional Gaussian filter can be implemented with two consecutive one-dimensional convolutions; a horizontal convolution followed by a vertical convolution. Mathematically, a Gaussian function is expressed as

$$G(x, y) = \frac{1}{2\pi\sigma^2} e^{\frac{-(x^2+y^2)}{2\sigma^2}} \tag{2.1}$$

in two dimensions. When dimensionally separated, the function becomes

$$G(x, y) = G(x) \times G(y) = \frac{1}{\sqrt{2\pi}\sigma} e^{-\frac{x^2}{2\sigma^2}} \times \frac{1}{\sqrt{2\pi}\sigma} e^{-\frac{y^2}{2\sigma^2}} \tag{2.2}$$

where $G(x)$ is a one-dimensional horizontal convolution and $G(y)$ is a one-dimensional vertical convolution. Separated in this manner the convolution may be carried out in one of two ways: (1) the vertical and horizontal one-dimensional convolutions operate on the original image and corresponding pixels are multiplied together, or (2) first the horizontal (or vertical) one-dimensional convolution operates on the original image followed by the vertical (horizontal) one-dimensional convolution which operates on the resultant. Both methods give the correct result; however, the second method does not require multiplication of corresponding pixels and is therefore computationally less expensive. Dimensional separability reduces the computational complexity of an $M \times M$ convolution from $O(M^2)$ multiplies and additions to $O(M)$. Further, the number of computational elements in the systolic array is reduced from $O(M^2)$ to $O(M)$. The systolic array in fact becomes two truly one-dimensional arrays, each of length M. Use of one-dimensional systolic arrays enable the use of a parallel clock thus eliminating any clock skew problems.

The second technique, hierarchical convolution, exploits the fact that the convolution of a Gaussian function with another Gaussian function results in a third Gaussian function with different standard deviation. In particular, if $G_i(x)$ with standard

deviation σ_i, is convolved with $G_j(x)$ with standard deviation σ_j, the resultant Gaussian has standard deviation $\sigma = \sqrt{\sigma_i^2 + \sigma_j^2}$. This may be shown as follows. In one dimension let

$$G_1(x) = \frac{1}{\sigma_1\sqrt{2\pi}} e^{-\frac{x^2}{2\sigma_1^2}} \quad , \quad G_2(x) = \frac{1}{\sigma_2\sqrt{2\pi}} e^{-\frac{x^2}{2\sigma_2^2}} \tag{2.3}$$

then

$$G_1(x) * G_2(x) = \frac{1}{\sigma_1\sqrt{2\pi}} e^{-\frac{x^2}{2\sigma_1^2}} * \frac{1}{\sigma_2\sqrt{2\pi}} e^{-\frac{x^2}{2\sigma_2^2}} \tag{2.4}$$

or taking the product of the respective Fourier transforms,

$$\frac{1}{\sqrt{2\pi}} e^{\frac{-u^2\sigma_1^2}{2}} \times \frac{1}{\sqrt{2\pi}} e^{\frac{-u^2\sigma_2^2}{2}} = \frac{1}{2\pi} e^{\frac{-u^2(\sigma_1^2 + \sigma_2^2)}{2}} \tag{2.5}$$

with inverse Fourier transform $\frac{1}{\sigma\sqrt{2\pi}} e^{\frac{-x^2}{2\sigma^2}}$ where $\sigma = \sqrt{\sigma_1^2 + \sigma_2^2}$

Generation of size-scaled copies of the original image is accomplished through convolution with Gaussian functions of increasing standard deviation. Generated hierarchically, the image is convolved consecutively with Gaussian filters where, at each pass, the coefficients of the filter are mapped into a larger sample grid, thereby expanding the size of the filter or equivalently increasing the resultant standard deviation of the Gaussian function. As the standard deviation of the Gaussian function increases, the upper cutoff frequency of the filter decreases, and thus its output can be resampled with coarser spacing without loss of information. The exponential growth in the number of filter coefficients which results from the exponential scaling of size is offset by an exponential growth in distance between points at which the convolution is computed. Each Gaussian-filtered copy of the image may therefore be computed with the same filter as shown in Figure 3, where each output is a Gaussian-filtered copy of

its predecessor. Use of identical filters to generate all size-scaled copies of an image produces a very modular design.

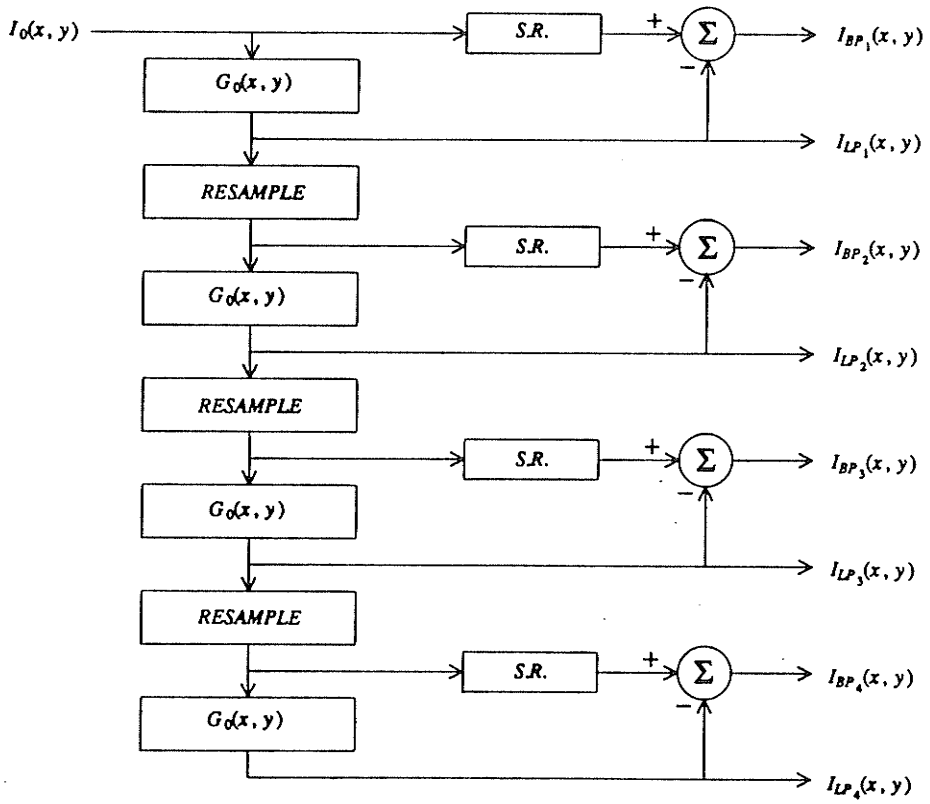


Figure 3: Generation of multiresolution lowpass and bandpass representations.

Multiresolution bandpass or difference of Gaussian representations are derived by subtracting each lowpass image from the resampled version of the previous lowpass image (Figure 3). Subtracting two eight-bit per pixel images in the pipeline requires only a single eight-bit subtractor. Obviously, this method requires considerably less silicon area to implement than a full two-dimensional systolic array. The shift registers are necessary to realign the image pixels due to the latency or delay of the $G_0(x, y)$ systolic convolver. The computational savings of hierarchical convolution is proportional to the depth of hierarchy in the system. As the number of coefficients in

the Gaussian filter $G_0(x, y)$ decreases, the depth of the hierarchy increases for a fixed standard deviation of the Gaussian function. Thus, using fewer coefficients for $G_0(x, y)$ increases the computational saving. However, decreasing the number of coefficients decreases the accuracy with which the sampled Gaussian function is represented. A tradeoff therefore exists between accuracy and computational complexity.

Burt [14] has derived a hierarchical method for the generation of Gaussian-filtered size-scaled copies of an image. In one dimension, the five coefficients 0.05 0.25 0.4 0.25 0.05, represent the sampled Gaussian function. Each node at each level in the hierarchy is obtained as a weighted average of the five coefficients centered on the five nearest neighbors of the corresponding node in the previous level. The sample distance in each level is double that in the previous level, effectively doubling the standard deviation of the resultant Gaussian function. The resampling results in each image in the hierarchy being half as large as its predecessor, since the resampling rate is two (ie. every second pixel). The process may be summarized as follows;

1) define a sampled Gaussian function, G_0 as a width-5 convolution window with coefficients 0.05 0.25 0.40 0.25 0.05

2) convolve the original image I_0 (*size* N) with G_0 producing a Gaussian-filtered output I_{LP_1} (*size* N) (LP = lowpass).

3) resample I_{LP_1} with sample reduction factor 2.

4) convolve this result with G_0 producing I_{LP_2} (*size* $N/2$).

5) resample I_{LP_2} with sample reduction factor 2.

6) convolve this result with G_0 producing I_{LP_3} (*size* $N/4$).

and repeat.

The hierarchically-generated Gaussian-filtered outputs $I_{LP_2} - I_{LP_n}$ correspond to convolving the original image directly with equivalent Gaussian functions $G_1 - G_n$. That is, three convolutions using the Gaussian function G_0 and appropriate resampling produces the output I_{LP_3}. However, the original image could have been convolved with an equivalent Gaussian function G_2, producing the identical output I_{LP_3}. The equivalent Gaussian functions are shown in Figure 4. Each function in the hierarchy has a standard deviation twice that of its predecessor. Each resultant Gaussian-filtered copy of the image therefore has a band limit one octave lower than its predecessor. Sample rate reduction in the hierarchy is in proportion to the band limit reduction, and remains above the Nyquist rate; thus no information is lost in the subsequent multiresolution representation.

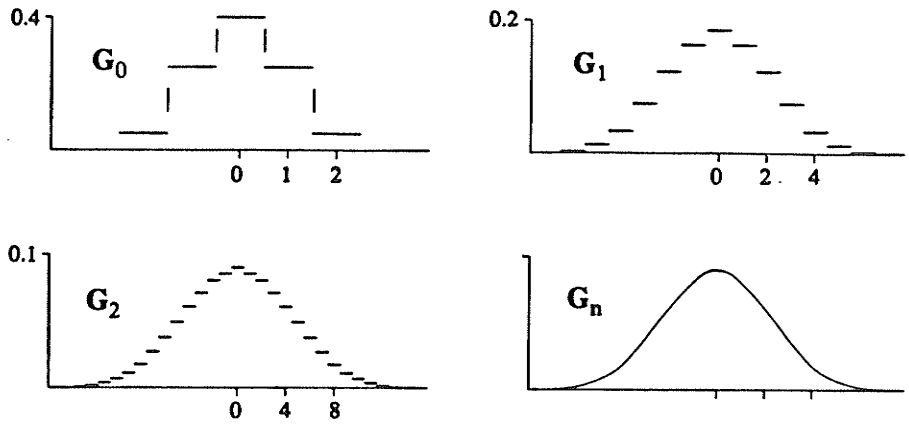


Figure 4: Equivalent Gaussian functions

The dimensional separability of the Gaussian function enables simple expansion of the process to two dimensions. Convolution is performed horizontally, row by row, and then vertically, column by column, at each level in the hierarchy. The convolution at each level is interleaved with resampling which consists of selecting every second

pixel per row and every second row in the image frame thus reducing the image size by a factor of two in both dimensions. The resultant data flow graph (Figure 3) shows the generation of both a lowpass (Gaussian) filtered multiresolution representation and a bandpass (Difference of Gaussian) filtered multiresolution representation.

2.4. Computational Cost and Complexity

The computational cost of the above convolver design is determined for the most part by the number of multiply and addition operations required, the I/O requirements, and the memory or storage requirements. For compute-bound problems such as convolution, systolic arrays have considerable computational cost advantages over FFT implementations in regard to I/O and memory requirements (see Sec. 2.1 and 2.2). The number of multiply and addition operations required by an FFT implementation is independent of convolution window size and is fixed for each size-scaled copy required. The total requirements are $N^2 + 2N^2\log_2 N$ multiplies and $4N^2\log_2 N$ additions for an $N \times N$ image. In a hierarchical systolic implementation as described above, the computational cost is determined by the number of steps needed to compute the equivalent convolution; the number of levels in the hierarchy. However, in most applications, all levels of the multiresolution representation are needed. Thus, since all levels are being generated there is a fixed cost per level. Furthermore, exploiting the symmetry of the sampled Gaussian function coefficients, the process requires three multiplies and four additions per pixel per level. Assuming an $N \times N$ image the total requirements are $3N^2$ multiplies and $4N^2$ additions. Depending on image size N, a hierarchical systolic implementation requires one or two orders of magnitude fewer computational operations than convolution in the frequency domain using the FFT.

Computational complexity is determined in part by the number of bits used to represent the data, the coefficients, and the partial results. In a two-dimensional

systolic array the number of bits required to represent the partial results increases at each computational element through which the result passes. Large word size is therefore required for large convolutions. Alternatively, the number of bits allocated to each computational element determines the size of the convolution which can be computed. A hierarchical implementation enables large convolutions to be broken into smaller convolutions in a divide-and-conquer scheme. As a result the number of bits needed in arithemetic operations is independent of convolution size. Furthermore, the one-dimensional width-5 convolution used has normalized coefficients, ie. $0.05 + 0.25 + 0.40 + 0.25 + 0.05 = 1.0$ Therefore each result is represented by the same number of bits as the original data. Partial results require a maximum of twelve bits to retain accuracy (see Sec. 2.5).

2.5. Systolic Convolver Design

Implementation of the dimensionally-separable hierarchical convolution algorithm using systolic convolvers results in the data flow diagram shown in Figure 5. The data flow diagram, in its entirety, represents the $G_0(x, y)$ (Figure 3) convolver block used for implementation of the Gaussian filters. The one-dimensional systolic convolvers $G_0(x)$ and $G_0(y)$, implement the horizontal and vertical convolutions respectively. These convolvers are identical aside from the fact that pixels are shifted serially through $G_0(x)$, one new pixel per clock cycle, whereas five new pixels are shifted in parallel into $G_0(y)$ on each clock cycle. This is due to the fact that the image data is being transmitted row by row as opposed to column by column. Henceforth, the term G_0 will refer to both $G_0(x)$ and $G_0(y)$.

The row by row transmission of the image data necessitates the need for the shift register buffers shown to allow the convolution window to overlay the proper region of the image. The size of the shift register is dependent on the convolver size and the

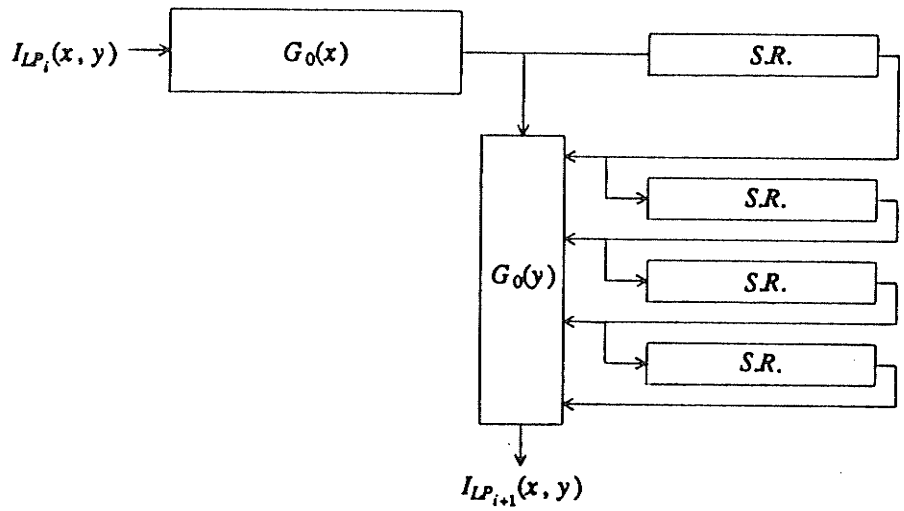


Figure 5: Data flow diagram of the systolic convolution implementation of the dimensionally-separable hierarchical convolution algorithm.

image row length. The systolic convolver, G_0, contains five computational elements corresponding to the five Gaussian function coefficients, 0.05 0.25 0.4 0.25 0.05. For an image row length of N, the required shift register length is $N - 5$. Due to the resampling, N decreases by a factor of two at each level in the hierarchy; thus the shift register length is halved at each level.

The systolic convolver, $G_0(x)$, uses a fan-in design as shown in Figure 6. The weighting coefficients are fixed to their particular computational elements. During a cycle, all x_i's move one cell to the right, multiplications are performed at all cells simultaneously, and their results are fanned-in and summed using an adder to form a new y_i. Essentially, the convolver performs a sum of products operation. The systolic convolver, $G_0(y)$, requires a modification to this design due to the fact that the inputs, x_i, are shifted in parallel into the computational elements. The modifications imply the following design (Figure 7).

The number of multipliers can be reduced by exploiting the symmetry of the Gaussian function. In particular, this is achieved by grouping the equivalent G_0

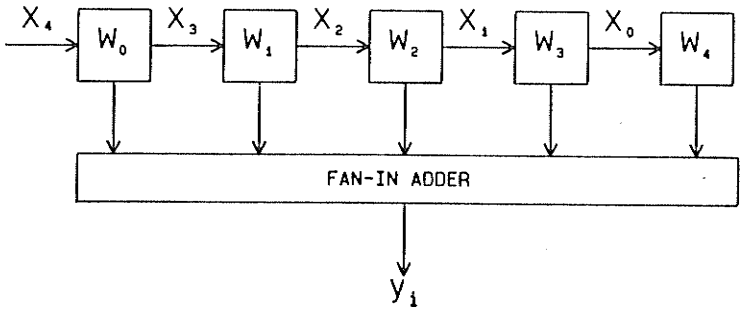


Figure 6: Systolic convolution array where w_i's are fixed, x_i's move systolically, and y_i's are formed as a sum of products via the fan-in adder.

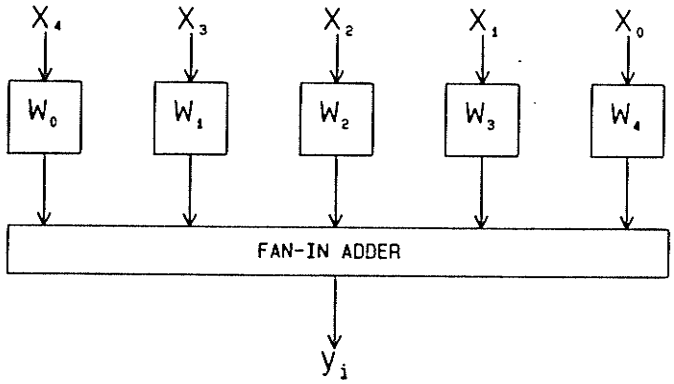


Figure 7: Systolic convolution array showing the parallel input modification to the $G_0(y)$ convolver necessitated by the row by row transmission of image data.

weighting coefficients. This process can be expressed in the following manner. G_0 contains the weighting coefficients 0.05 0.25 0.4 0.25 0.05. The convolution window, at any one instant, will contain five pixels which we label a b c d e. Mathematically, the convolution can be expressed as follows

$$0.05 \times a + 0.25 \times b + 0.4 \times c + 0.25 \times d + 0.05 \times e \tag{2.6}$$

Grouping coefficients gives

$$0.05\ (a + e) + 0.25\ (b + d) + 0.4\ (c). \tag{2.7}$$

The grouping of coefficients requires two additions. Pixel a must be added to pixel e, and pixel b must be added to pixel d leaving a nine-bit result from both additions. At the expense of two eight-bit additions, the required number of multiplications is reduced from five to three. 0.05 must be multiplied with the nine-bit result of $(a+e)$, 0.25 must be multiplied with the nine-bit result of $(b+d)$, and 0.4 must be multiplied with the eight-bit representation of c.

Since the values of the weighting coefficients are fixed and unchanging, the fixed-precision multiplications can be implemented by combinations of binary shift and add operations. Shift and add implementations of fixed-precision multiplications can be mechanized in much smaller silicon area and can also be made considerably faster since multipliers are typically both slow and expensive in area. A parallel multiplier available in the Manitoba CMOS standard cell library [15] requires $1.5 \times 10^6\ \mu m^2$ area ($3\mu m$ *double metal CMOS*) and approximately 500 nanoseconds to perform a single ten-bit by ten-bit multiplication.

A simple method of applying the binary shift and add operations to implement the multiplications is to scale the 0.05 weighting coefficient to unity giving

$$\frac{1}{20}[(a+e)+5(b+d)+8c] \tag{2.8}$$

The weighting coefficients are now represented by the small whole numbers 1, 5, and 8. Multiplication by one is trivial requiring no operation. Multiplication by 8 is implemented by three shifts left which also requires no operation. Finally, multiplication by 5 is implemented by shifting twice left and adding the original which requires a single eleven-bit addition. Thus, a single eleven-bit adder suffices to implement all three multiplications. The resultant three products are then added in the fan-in adder of Figure 6. The fan-in adder may be implemented with an eleven-bit adder and a twelve-bit adder. Due to the 0.05 scaling factor, the output of the fan-in adder must be

divided by 20 in order to retain an eight bit per pixel representation of the image data. Once again this fixed-precision multiplication can be implemented with a series of binary shift and add operations if some small error is allowed. The division by 20 is approximated by $(x/16 - x/64) + (x/16^2 - x/(16 \times 64)) = 51x/1024$ where x is the output of the fan-in adder. The error can be calculated as

$$\frac{1/20 - 51/1024}{1/20} \times 100\% = 0.4\% \tag{2.9}$$

The error in fact represents a scale factor of 0.996, identical for each pixel. The scale factor will not cause overflow since it is less than unity and therefore will be inconsequential.

The divide-by-20 process is physically implemented with two additions by subtracting x shifted six places to the right from x shifted four places to the right. The result is then shifted four places to the right and added to itself. The use of two's complement arithmetic reduces addition and subtraction to a single operation. The divide-by-20 process therefore requires a ten-bit adder and a nine-bit adder. The resultant systolic convolver, G_0, can be represented in block diagram form as shown in Figure 8.

As indicated in Figure 8, seven adders are required to implement the systolic convolver, G_0. Adders *1* and *2* are eight-bit adders generating nine-bit results and are used to group the coefficients a and e, and b and d respectively. Adder *3* combines the result of adder *1* (nine bits) with 8 times pixel c (eleven bits) requiring an eleven-bit adder. Adder *4* implements a multiplication by 5, adding 2 times the result of adder *2* to itself thus requiring an eleven-bit adder. Adder *5* combines the twelve-bit results of adders *3* and *4* and thus a twelve-bit adder is needed. Adders *6* and *7* implement the divide-by-20 operation and are ten- and nine-bit adders respectively. Simplification is possible where the inputs are hardwired to one or zero (ie. use of half

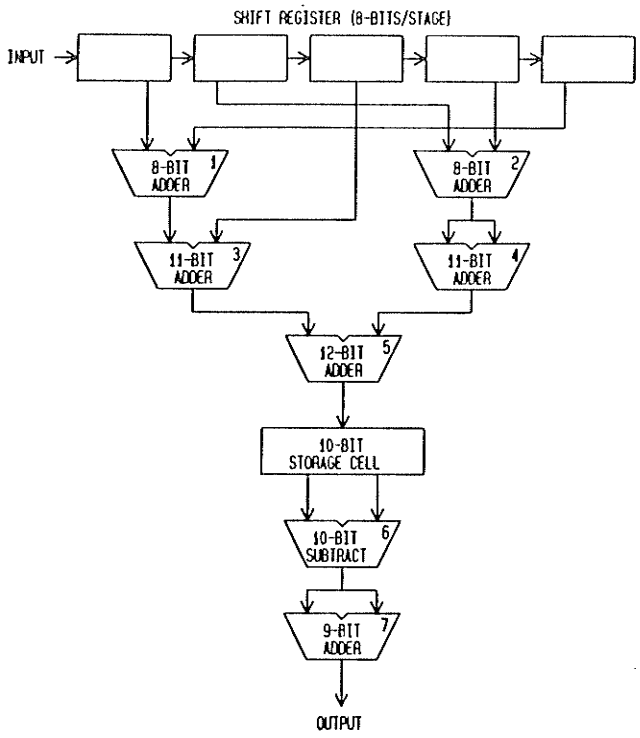


Figure 8: Block diagram of the G_0 systolic convolver.

adders is possible). The sequence in which the additions have been implemented enables adders *1* and *2* and adders *3* and *4* to operate concurrently in addition to the already highly cascaded nature of the circuit.

Due to the fact that the G_0 weighting coefficients are constant, thus enabling the implementation of the algorithm in seven additions, the physical dimensions of the silicon implementation will be considerably smaller than a straightforward implementation using multipliers. Since the building block for a parallel multiplier is a full adder cell, a single eight-by-eight multiplier requires 64 full-adder cells. The multiplier implementation requires roughly three such multipliers as well as four parallel adders to group coefficients and sum the final result. The seven adder solution requires silicon area roughly the size of a single eight-by-eight multiplier. Except for the 0.996 scale factor, the solutions are identical.

Due to the cascaded nature of the layout the data exhibits a sequential flow through the circuit. If the layout is depicted as in Figure 8, the flow is from top to bottom. Since this flow of data is sequential it is possible to incorporate further pipelining in the circuit. A ten-bit storage register placed after adder *5* reduces the computation, and thus the computation time of each block in the pipeline, allowing a faster clock rate to be used. The faster clock rate is achieved at the expense of a greater latency associated with the systolic convolver. However, when large amounts of data are to be processed identically as in this problem instance (and for all systolic convolvers), the benefit of a faster clock rate far exceeds the additional cost in latency.

CHAPTER 3

Design of Adders, Storage Elements, and Multiplexers

The preceding chapter illustrated the design of a systolic convolver with low component count and high throughput. The major computational components used in the G_0 systolic convolver are adders and storage elements. This chapter discusses and contrasts the varying adder schemes available, describes the design of choice, and explains its operation. The chapter then continues with the design of the storage elements used and an explanation of their operation. The final section discusses the manner in which data is multiplexed into and out of the chip. Multiplexing of data is necessary due to the limited I/O of the pad frame. Each cell design is a full custom design built specifically for use in the systolic convolver.

3.1. Ripple Carry Adders

The G_0 systolic convolver requires seven adders which vary from eight-bit to twelve-bit additions. Addition of two multi-bit binary numbers may be implemented either in serial or in parallel. The serial method uses a single full adder to generate the sum and carry outputs from corresponding bits in the addend and augend. The carry is required to be stored after each bit-addition and depending on the implementation it may be required to supply registers for the addend, augend, and sum. The resultant silicon area required by the storage elements reduces the area advantage of the serial method. Furthermore, because of the sequential nature of the addition, a

serial adder is slow. Since the rate at which data is clocked through the convolver is determined by the speed of operation of the adder, serial adders were considered to be too slow to meet the throughput rate requirements of the convolver.

Parallel addition requires one full adder for each pair of corresponding bits in the addend and augend. All sums are computed simultaneously since all full adders operate in parallel. This concurrency enables parallel addition to be faster than serial. There are a number of options in terms of the type of parallel adder that can be implemented. The choices available include ripple carry adders, lookahead carry adders, and encoded addition adder schemes. Lookahead carry adders and encoded addition adder schemes are typically faster than ripple carry adders; however, any encoded adder scheme or lookahead carry adder requires considerablely more area to implement in silicon than does a ripple carry adder. A comparison between ripple carry adders and carry lookahead adders follows.

For the present problem, ripple carry adders have two advantages over carry lookahead adders. The first is reduced area. Ripple carry adders require approximately 30% of the silicon area of carry lookahead adders. The second advantage is more subtle and is a result of the cascaded layout of the convolver. Depending on the word-length of the parallel adders and the cascade depth in the layout, cascaded ripple carry adders can in some cases be faster than carry lookahead parallel adders. The timing sequence of a cascaded ripple carry adder is contrasted with that of lookahead carry adders in Figure 9.

Associated with each full adder is a computation time τ corresponding to the time between presentation of augend, addend, and carry-in and generation of the sum and carry-out. Since the full adder corresponding to the LS bit in the second bank of adders is presented with augend, addend, and carry-in τ seconds after the full adder corresponding to the LS bit in the first bank, there is only a single τ second delay

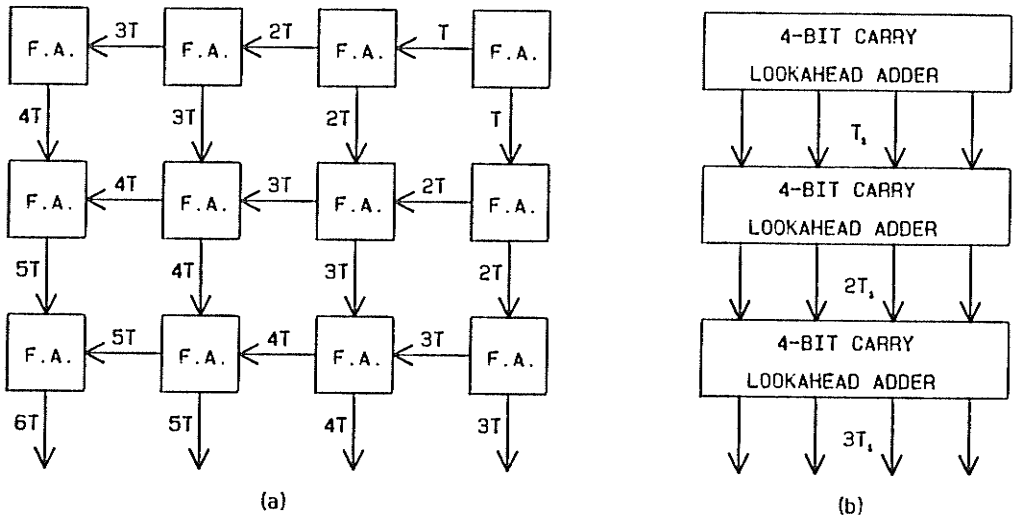


Figure 9: Timing sequence of (a) a cascaded ripple carry adder (F.A. = full adder) and (b) a lookahead carry adder.

between their respective outputs. This is true for all corresponding full adders in the two banks and thus there is only a τ second delay between adder banks in a cascaded layout. The total computation time of a cascaded layout is $C_t = \tau(b + d - 1)$, where τ is the computation time of a single full adder, b is the number of adder banks, and d is the number of bits per adder bank. The computation time τ, of a single full adder is determined by the charging (or discharging) time on the carry propagate path which consists of a metal wire, a transmission gate, and two inverters. For the full adder used in this design, $\tau = 6$ *nsec* (see Sec. 3.1.1). In contrast, carry lookahead circuits typically calculate propagate and generate signals for each corresponding bit in the addend and augend and use these signals to generate the input carrys four bits at a time. The circuit will then use a ripple carry from four-bit bank to four-bit bank. The computation time, τ_l, per four-bit bank is determined by the propagation of the input signals through the circuitry shown in Figure 10. Adding the indicated computation times associated with each gate along the critical path determined the required computation time $\tau_l = 20$ *nsec*. Thus, $\tau_l/\tau = 20/6$. For $d = 4$, the cascaded ripple carry layout will require less computation time when

$$\tau(b + 3) < \tau_1 \times b$$

$$b + 3 < \frac{\tau_1 \times b}{\tau}$$

$$(\tau_1/\tau - 1)\, b > 3$$

$$b > \frac{3}{\tau_1/\tau - 1} = \frac{3}{20/6 - 1} = 9/7 \tag{3.1}$$

or the number of adder banks is greater than or equal to 2.

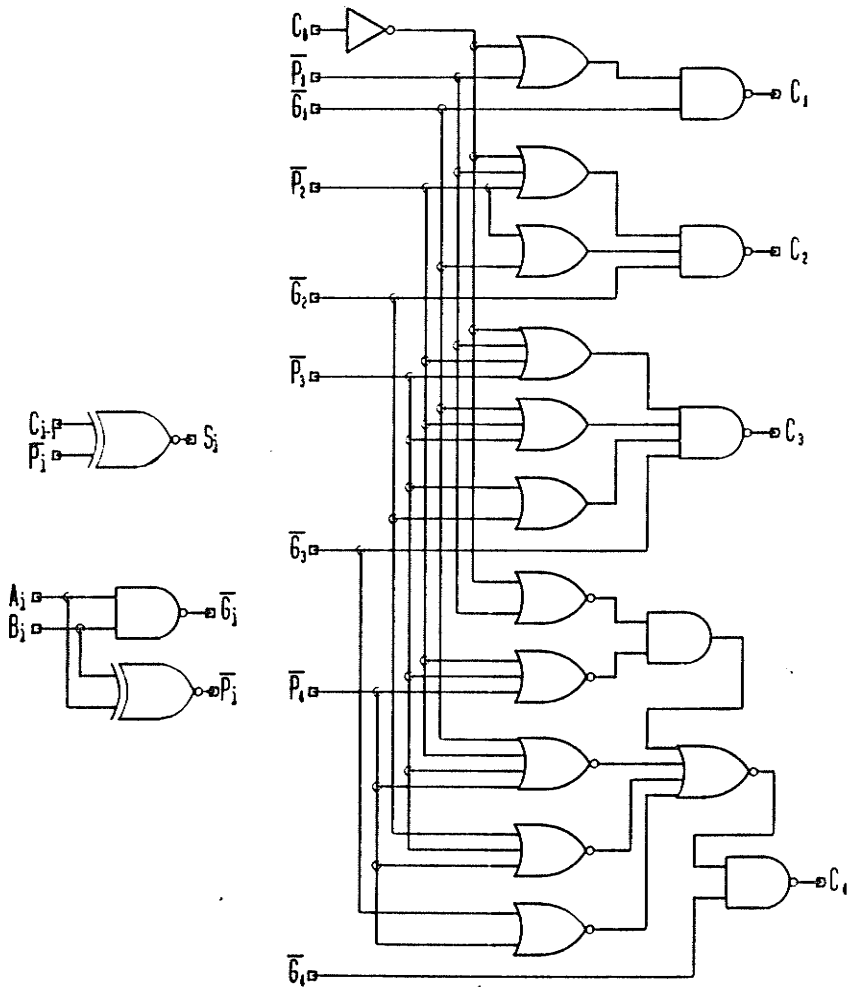


Figure 10: Logic diagram of a carry lookahead circuit.

3.1.1. Full Adder Design

For this problem instance, the best adder layout in terms of cost and performance is the cascaded ripple carry adder. For a ripple carry adder, the performance is determined by the propagation of the carry signal as it *ripples* through the carry chain. The critical path is the carry chain and this path must be optimized to obtain the best performance. The following design of a full adder cell optimizes the carry chain of a ripple carry adder. Optimization reduces the carry chain to a single transmission gate with associated wiring at each full adder stage. The truth table of a full adder is shown below.

A_i	B_i	C_{i-1}	S_i	C_i
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

Table 1: Full adder truth table.

At each full adder stage i in the carry chain, the inputs A_i and B_i will determine whether the carry-in, C_{i-1}, will propagate on to the next stage or if a new carry-out, C_i, will be generated. As can be seen from the truth table, if A_i *EXOR* B_i is true, then $C_i = C_{i-1}$ and the carry-in can propagate thru the i_{th} stage. If A_i *EXOR* B_i is false a new carry-out, C_i must be generated. If A_i *EXOR* B_i is false then $A_i = B_i = C_i$ (see Table 1). Generation of C_i is simply a matter of feeding either A_i or B_i to C_i and propagation is a matter of feeding C_{i-1} to C_i. The sum, S_i is generated as A_i *EXOR* B_i *EXOR* C_{i-1}. The resultant full adder logic circuit is shown in Figure 11.

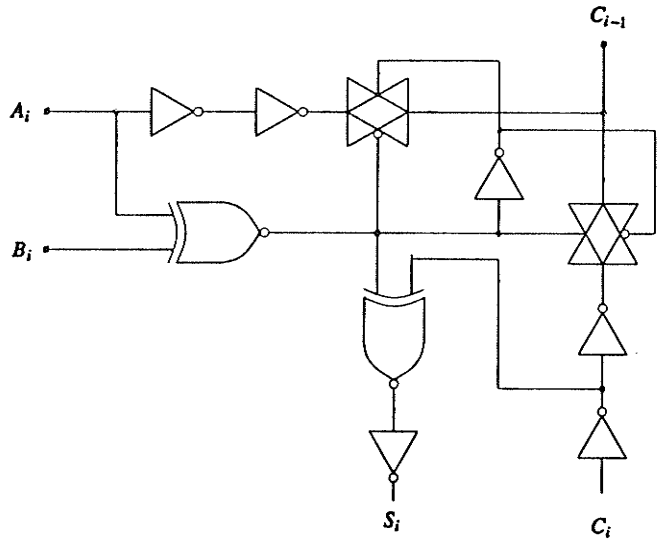


Figure 11: Logic circuit of a full adder stage with optimized carry chain and buffered I/O.

Once the inputs A_i and B_i arrive, each stage computes whether to propagate C_{i-1} or A_i as the carry-out and sets the transmission gates appropriately. Since this can be computed in parallel at each stage in the adder bank, the worst case add will occur when the carry-out from the LS bit full adder must propagate through the entire carry chain. This requires the signal to propagate through one transmission gate per stage and the associated wiring on the carry chain. The RC component associated with the carry chain will necessitate the use of repeaters or drivers at every third or fourth adder stage. The drivers are necessary since the RC time constant or, equivalently, the charge and discharge time of the carry chain limits the speed with which the carry signal propagates through the carry chain. Due to the lack of modularity introduced by using drivers in this manner, a simple buffer is used on the carry chain at every stage rather than employing drivers at every third or fourth stage. This enables the full adder stages to be linked together without regard for the loading on the carry chain. Furthermore, since the adder banks are cascaded in this design, such that the sum of one adder bank becomes the input to a following adder bank, it is advantageous to keep the carry propagation time through each full adder stage equivalent. Use of

drivers at every third or fourth adder stage will cause unequal carry propagation times through the stages. To avoid delays, the corresponding drivers in each bank must be vertically aligned throughout the overall cascaded circuit. This is impossible in this layout and therefore input buffering on the carry chain is used at each full adder stage. A criteria in the design of the full adder is to enable its use in a modular fashion when building the varying length adder banks. This requires appropriate buffering of inputs and outputs. Since the sum of one adder bank becomes the input to a following adder bank, the sum output of each full adder is driven with a double-sized inverter. The driver enables the input load of the following full adder stage to be charged or discharged in a reasonable amount of time. The resultant layout of the ripple carry full adder used is shown in Figure 12.

The layout is a full custom design requiring 156 μm by 95 μm using 3 μm double layer metal CMOS technology. The circuit requires 24 transistors with the output sum being driven by a double-sized inverter. The power bus (gnd and vdd) runs in parallel along the bottom of the layout. This enables the full adder stages to be linked together in a row to form the adder banks without requiring an extra power bus to be run to each adder bank. All I/O ports in the layout are exported with a via (metal-1 to metal-2 contact) and no metal-2 wiring is used in the layout. This enables intercell wiring using the metal-2 layer and allows the wires to be run over the top of the full adder stages greatly reducing the wiring complexity and the wiring area.

3.1.2. Half Adder Design

Half adders can be used when either the carry-in, the addend bit, or the augend bit input to a stage is always zero. A half adder therefore has only two inputs, in_0 and in_1. The sum and carry-out are computed as $SUM = in_0\ EXOR\ in_1$ and $C_{out} = in_0\ AND\ in_1$. Figure 13(a) shows the logic circuit of the half adder. The

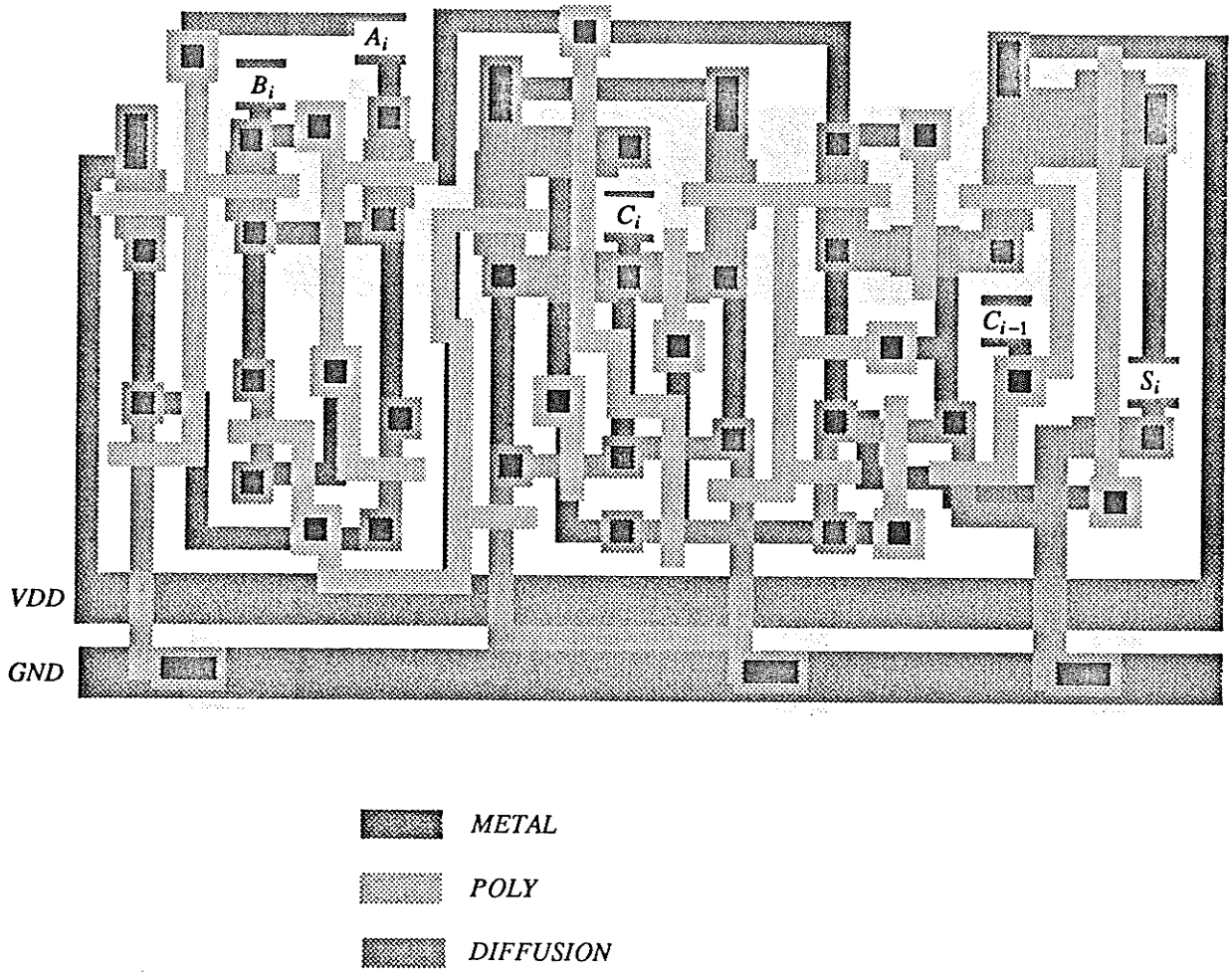
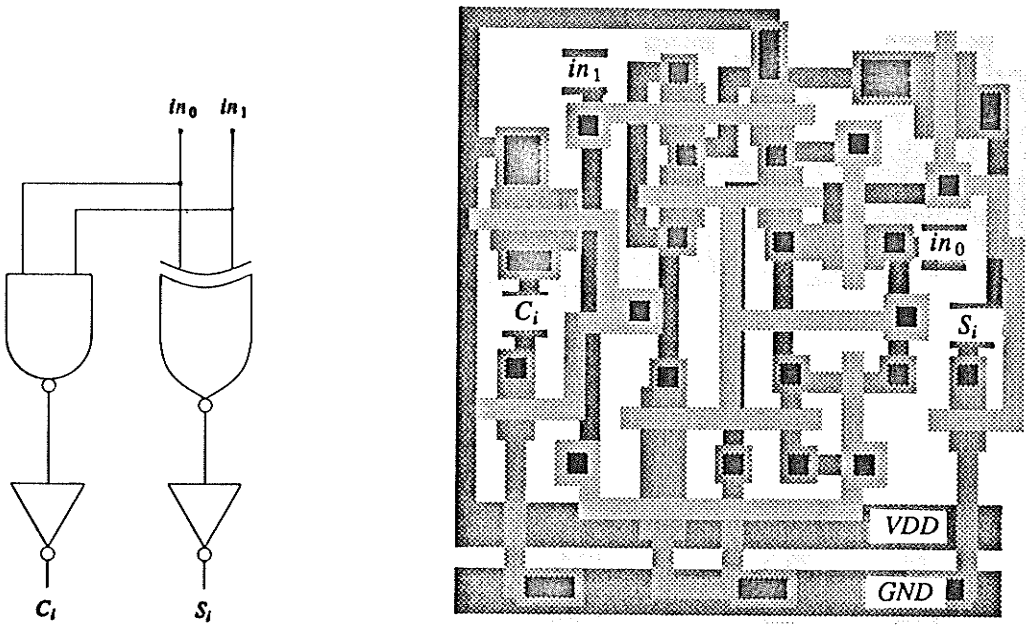


Figure 12: CMOS layout of a full adder cell.

AND gate is implemented with a NAND and an inverter. The EXOR gate is implemented with an EXNOR and an inverter. Use of the inverters enable the outputs to be properly buffered. In particular, the sum output inverter is double-sized for reasons explained above. Figure 13(b) shows the silicon layout which is a full custom design requiring 89 μm by 94 μm using 3 μm double layer metal CMOS technology. The layout requires 14 transistors and in a similar way to the full adder layout, the power bus runs along the bottom of the layout and all I/O ports are exported with a metal via. No metal-2 layer wiring is used.

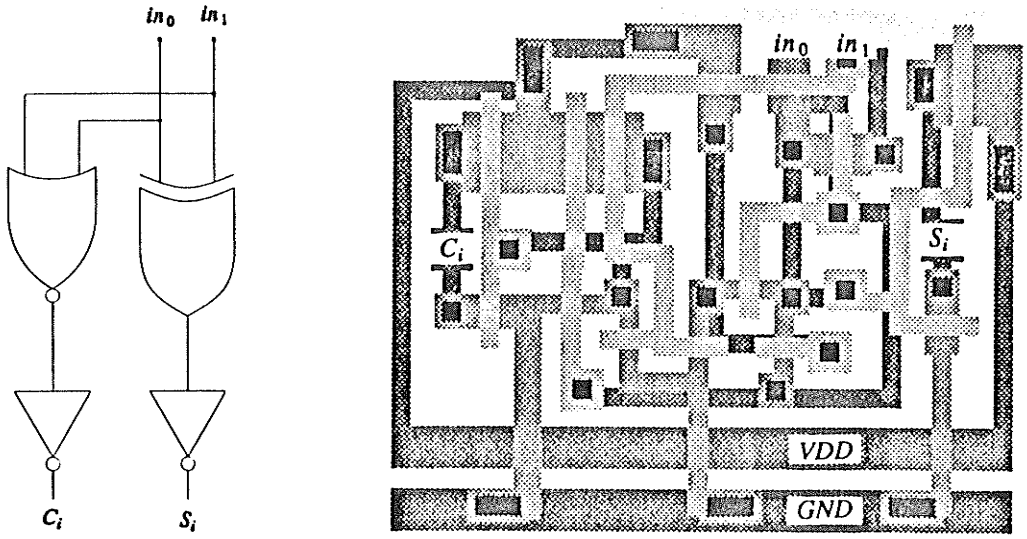


(a) (b)

Figure 13: (a) Half adder logic circuit and (b) CMOS layout.

3.1.3. Half Subtractor Design

Half subtractors are used when either the carry-in, the subtrahend bit, or the minuend bit input to a stage is always one. A half subtractor therefore has only two inputs, in_0 and in_1. The sum and carry-out are computed as $SUM = \overline{in_0\ EXOR\ in_1}$ and $C_{out} = in_0\ OR\ in_1$. Figure 14(a) shows the logic circuit of the half subtractor. The OR gate is implemented with a NOR and an inverter. The sum output is generated with an EXOR gate and a double-sized inverter for driving down-line devices. Figure 14(b) shows the silicon layout which is a full custom design requiring 104 μm by 86 μm using 3 μm double-layer metal CMOS technology. The layout requires 14 transistors and as with previous layouts the power bus runs along the bottom of the layout and all I/O ports are exported with a metal via. No metal-2 layer wiring is used within the cell.



(a) (b)

Figure 14: (a) Half subtractor logic circuit and (b) CMOS layout.

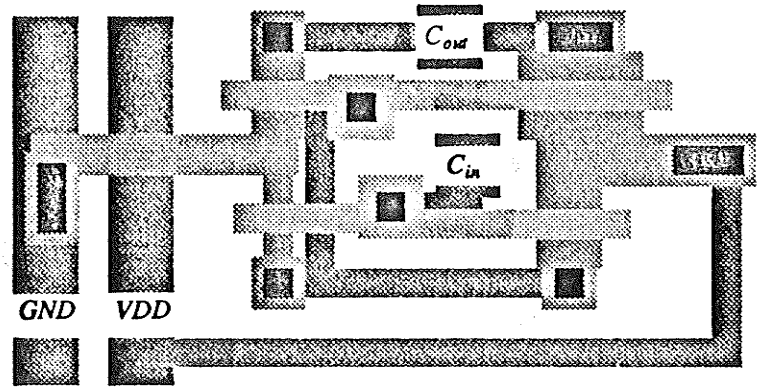


Figure 15: CMOS layout of the carry-out driver.

3.1.4. Carry-out Driver Design

The carry-out of each full adder stage is designed considering the optimization of the carry chain and is unbuffered. The MS bit carry-out from an adder bank therefore will be unbuffered. Since this carry-out will, in some situations, be required to drive down-line devices, it is necessary to use a driver to enable the drive capability of the carry-out to match the load requirements. The silicon layout is shown in Figure 15.

The driver is essentially two inverters cascaded with the second inverter being double-sized to increase its drive capability. The layout requires 47 μm by 83 μm in 3 μm double-layer metal CMOS technology.

3.1.5. Adder Bank Design

As indicated in Section 2.5, the G_0 systolic convolver requires six adder banks and one subtractor bank. A further eight-bit subtractor bank is required to implement the differencing of images to produce the bandpass representation. All adder and subtractor banks are built primarily with the full adder cell, however in some instances the carry-in, augend, or addend is required to be hardwired to one or zero. In these cases use of half adders, half subtractors, or simple wires replace the full adder cell and require less silicon area to implement. As indicated above, all adder cells have been designed to be linked together in a row to form the adder banks. The carry chain is wired using the metal-2 layer and is run over the adder stages thereby keeping its RC component to a minimum. The requirements and design of each adder and subtractor bank is explained in this section. Reference is made to Figure 8 to identify each adder bank uniquely.

Adder banks 1 and 2 (Figure 8) are used to exploit the symmetry of the low-pass filter coefficients. Both are eight-bit adders and are identical. Representing the five image elements in the convolver as $a \quad b \quad c \quad d \quad e$, adder bank 1 is required to add $a_7 \; a_6 \; a_5 \; a_4 \; a_3 \; a_2 \; a_1 \; a_0$ to $e_7 \; e_6 \; e_5 \; e_4 \; e_3 \; e_2 \; e_1 \; e_0$, where $a_7 \cdots a_0$ and $e_7 \cdots e_0$ are the eight-bit representations of image elements a and e respectively. Adder bank 2 is required to add $b_7 \cdots b_0$ to $d_7 \cdots d_0$. No single bit of any of the image elements is known a priori and thus the only simplification which can be made is the use of a half adder for the LS stage. Since the carry-out is required to drive a down-line adder stage it is driven using a carry-out driver. The resultant adder

bank is shown in Figure 16(a). The silicon layout requires 1114 μm by 95 μm.

Adder bank 3 adds the output of adder bank 1 with eight times image element c. The required addition is

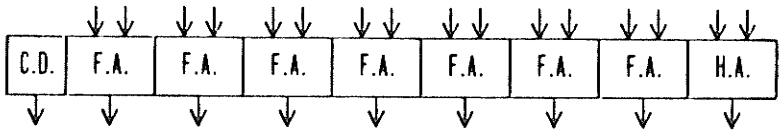
	0	0	1_8	1_7	1_6	1_5	1_4	1_3	1_2	1_1	1_0	*addend*
+	c_7	c_6	c_5	c_4	c_3	c_2	c_1	c_0	0	0	0	*augend*
3_{11}	3_{10}	3_9	3_8	3_7	3_6	3_5	3_4	3_3	3_2	3_1	3_0	Σ

3_2 to 3_0 are equivalent to 1_2 to 1_0 respectively and therefore the full adders in these stages are replaced with wires. The carry-out of the *3rd* stage is 0 and the fourth stage can thus be implemented with a half adder. The two MS bits of the addend are zero, however the carry-out of the previous stage is not known a priori and half adders are therefore required for the two MS stages. Once again a carry-out driver is used after the MS stage. Adder bank 3 is shown in Figure 16 (b). The silicon layout of adder bank 3 requires 989 μm by 95 μm.

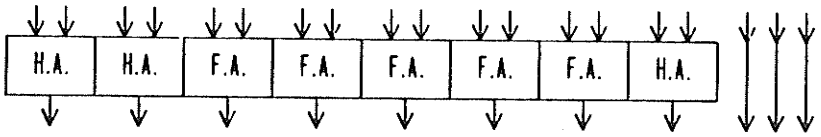
The fourth adder bank generates five times the output of adder bank 2. This is accomplished as follows.

	0	0	2_8	2_7	2_6	2_5	2_4	2_3	2_2	2_1	2_0	*addend*
+	2_8	2_7	2_6	2_5	2_4	2_3	2_2	2_1	2_0	0	0	*augend*
4_{11}	4_{10}	4_9	4_8	4_7	4_6	4_5	4_4	4_3	4_2	4_1	4_0	Σ

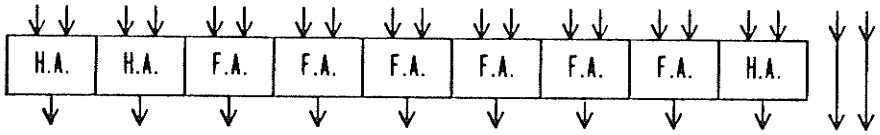
Since bits 4_1 and 4_0 are equivalent to 2_1 and 2_0 respectively, the full adders in these stages are replaced with wires. The carry-out of the second stage is zero and the third stage is implemented with a half adder. As with adder bank 3, the two MS stages are implemented with half adders and the carry-out is driven. Adder bank 4 is shown in Figure 16(c). Its silicon layout requires 1136 μm by 95 μm.



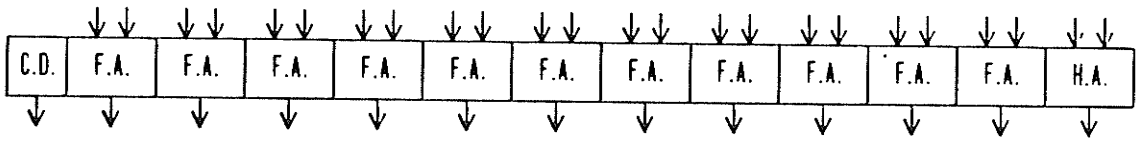
(a)



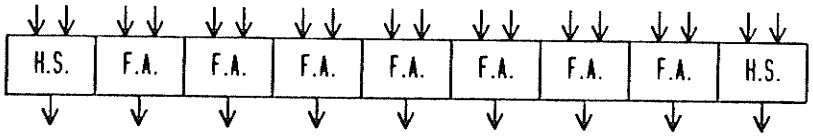
(b)



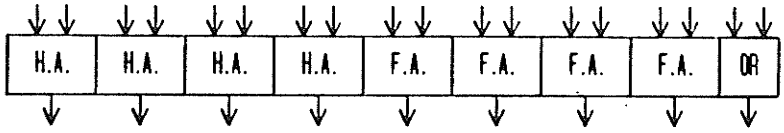
(c)



(d)



(e)



(f)

Figure 16: Adder bank layouts. F.A. = full adder, H.A. = half adder, H.S. = half subtractor, C.D. = carry-out driver

The fifth adder bank adds the outputs of adders 3 and 4 as follows.

	3_{11}	3_{10}	3_9	3_8	3_7	3_6	3_5	3_4	3_3	3_2	3_1	3_0	*addend*
+	4_{11}	4_{10}	4_9	4_8	4_7	4_6	4_5	4_4	4_3	4_2	4_1	4_0	*augend*
5_{12}	5_{11}	5_{10}	5_9	5_8	5_7	5_6	5_5	5_4	5_3	5_2	5_1	5_0	Σ

The LS stage is implemented with a half adder and all other stages require full adders. The carry-out is driven with an output carry driver giving the configuration for adder 5 shown in Figure 16(d). Adder bank 5 requires 1596 μm by 95 μm for its silicon lay-out.

Adder bank six implements the first stage of the divide-by-20 process, which is $x/16 - x/64$ (where x is the output of adder 5). This process constitutes a subtraction. The subtraction is implemented using 2's complement arithmetic which requires adding the minuend to the 1's complement of the subtrahend and using a carry-in of one on the LS stage of the adder. The 1's complement of the subtrahend is generated by inverting each bit of the subtrahend. Inverted values of the subtrahend are available from the storage cell which is positioned in the circuit immediately upline from adder 6 (Figure 8). The subtraction unit therefore requires only a full adder for each stage. The subtraction may be depicted as follows.

														1	*carry–in*
	5_{12}	5_{11}	5_{10}	5_9	5_8	5_7	5_6	5_5	5_4	. 5_3	5_2	5_1	5_0	0	0	*minuend*
+	1	1	$\overline{5_{12}}$	$\overline{5_{11}}$	$\overline{5_{10}}$	$\overline{5_9}$	$\overline{5_8}$	$\overline{5_7}$	$\overline{5_6}$	. $\overline{5_5}$	$\overline{5_4}$	$\overline{5_3}$	$\overline{5_2}$	$\overline{5_1}$	$\overline{5_0}$	*subtrahend*

The carry-in of one is indicated over the LS stage and the decimal points indicate the shift right processes used to implement the divide-by-16 and divide-by-64. The six fractional decimal places shown would require excessive silicon area to implement as well as requiring extra time during execution. The five LS stages of the addition are therefore omitted and the following process is implemented:

											1	*carry–in*
	5_{12}	5_{11}	5_{10}	5_9	5_8	5_7	5_6	5_5	5_4	.	5_3	*minuend*
+	1	1	$\overline{5_{12}}$	$\overline{5_{11}}$	$\overline{5_{10}}$	$\overline{5_9}$	$\overline{5_8}$	$\overline{5_7}$	$\overline{5_6}$	.	$\overline{5_5}$	*subtrahend*
1	0	6_7	6_6	6_5	6_4	6_3	6_2	6_1	6_0	.	6_{-1}	Σ

Omission of the five LS stages means the output will be greater by an additive factor of between 0.0 and 0.5. Since the divide-by-20 process implemented here involves a scale factor of 0.996, the additive factor will result in a more accurate result and is therefore beneficial not only as an area and time saving but also to increase accuracy. A larger additive factor cannot be used since it would then be impossible to obtain a zero output.

In 2's complement subtraction an output carry of one indicates a positive result. As shown, the output carry is always high since the result is always positive ($x/16 - x/64$ is always positive for positive x). The sum out from the MS stage is shown as zero. It is always zero since the output of the subtractor can be no larger than 239 which requires only eight bits for a binary representation. A maximum of 239 is due to the fact that the largest output from adder 5 is 5100 since $5100 = 20 \times 255$, where 20 is the initial scale factor and 255 is the largest pixel value possible for an eight-bit representation. Finally, $239 = 5100/16 - 5100/64$. Since the output carry and the sum out of the MS stage are known a priori, no hardware is necessary for the MS stage. The sixth adder bank therefore requires a half subtractor for the LS stage, seven full adders, and a half subtractor to generate 6_7. Adder bank six is implemented as shown in Figure 16(e). The silicon layout of adder bank six requires 1224 μm by 95 μm.

The seventh adder bank implements the second stage of the divide-by-20 process and accomplishes the following:

										1	*carry–in*
	6_7	6_6	6_5	6_4	6_3	6_2	6_1	6_0	.	6_{-1}	*minuend*
+	0	0	0	0	6_7	6_6	6_5	6_4	.	6_3	*augend*
	G_{i_7}	G_{i_6}	G_{i_5}	G_{i_4}	G_{i_3}	G_{i_2}	G_{i_1}	G_{i_0}			Σ

The eight-bit result shown represents the output of the G_0 convolver. The carry-out is always zero since $239 + 239/16 < 256$. An LS stage input carry of 1 is used to round-up the output rather than truncate since the fraction can no longer be carried. No sum output is required from the LS stage and the carry-out is generated as 6_{-1} OR 6_3 by using a simple OR gate. Stages two through five require full adders and stages six through nine are implemented with half adders. The configuration of adder bank 7 is shown in Figure 16(f). Its silicon layout requires 971 μm by 95 μm.

One final subtractor unit is necessary. An eight-bit subtractor is required to take the difference of consecutive lowpass filtered results, thus producing a bandpass filtered output. An adder bank subtracts the output of the $G_i(x, y)$ systolic convolver from the output of the previous stage, $G_{i-1}(x, y)$. The output of the subtractor can be either positive or negative and is represented with nine bits in 2's complement form where the carry-out indicates whether the result is positive or negative. It therefore becomes necessary to change the representation to sign-magnitude form using eight bits to represent the magnitude and an additional bit to indicate the sign. Two conditions exist; (1) the carry-out of the adder bank is high, or (2) it is low. If the carry-out is high, the result is positive and in the correct sign-magnitude form. If the carry-out is low, the result is negative and in 2's complement form. Condition (2) requires the 2's complement of the output to be taken. This is accomplished by inverting the eight magnitude bits and incrementing once, leaving the result in sign-magnitude form. There are now two results in sign-magnitude form; (1) the output of the adder, and (2) its 2's complement. If the carry-out is high the first result is selected and if the carry-

out is low the second result is selected. This is accomplished by using the carry out as the select line to seven 2-1 multiplexers, one for each corresponding pair of bits of the two results. The LS stage output of the 2's complement is always S_0 and therefore no multiplexer is required. The operation of the subtractor unit may be summarized as follows.

									1	*carry–in*
		G_{i-1_7}	G_{i-1_6}	G_{i-1_5}	G_{i-1_4}	G_{i-1_3}	G_{i-1_2}	G_{i-1_1}	G_{i-1_0}	*minuend*
stage 1	+	$\overline{G_{i_7}}$	$\overline{G_{i_6}}$	$\overline{G_{i_5}}$	$\overline{G_{i_4}}$	$\overline{G_{i_3}}$	$\overline{G_{i_2}}$	$\overline{G_{i_1}}$	$\overline{G_{i_0}}$	*subtrahend*
	S_8	S_7	S_6	S_5	S_4	S_3	S_2	S_1	S_0	Σ
									1	*carry–in*
stage 2	+	$\overline{S_7}$	$\overline{S_6}$	$\overline{S_5}$	$\overline{S_4}$	$\overline{S_3}$	$\overline{S_2}$	$\overline{S_1}$	$\overline{S_0}$	Σ *inverted*
		S'_7	S'_6	S'_5	S'_4	S'_3	S'_2	S'_1	S'_0	*2's complement*

stage 3 If S_8 is high select the *stage* 1 output else select the *stage* 2 result.

Stage 1 of the subtractor unit is implemented using a half subtractor for the LS stage and full adders for the remaining stages. The inverted bit values of the subtrahend are available from the output of the $G_i(x, y)$ convolver and therefore inverters are not required. Stage 2 is implemented using an inverter at each stage as well as seven half adders for stages one through seven. Stage 3 uses the carry-out of stage 1 (S_8) as the select line to seven 2-1 multiplexers enabling the selection of the correct result. The subtractor unit is physically implemented as shown in Figure 17. The silicon layout of the subtractor unit requires 1508 μm by 353 μm.

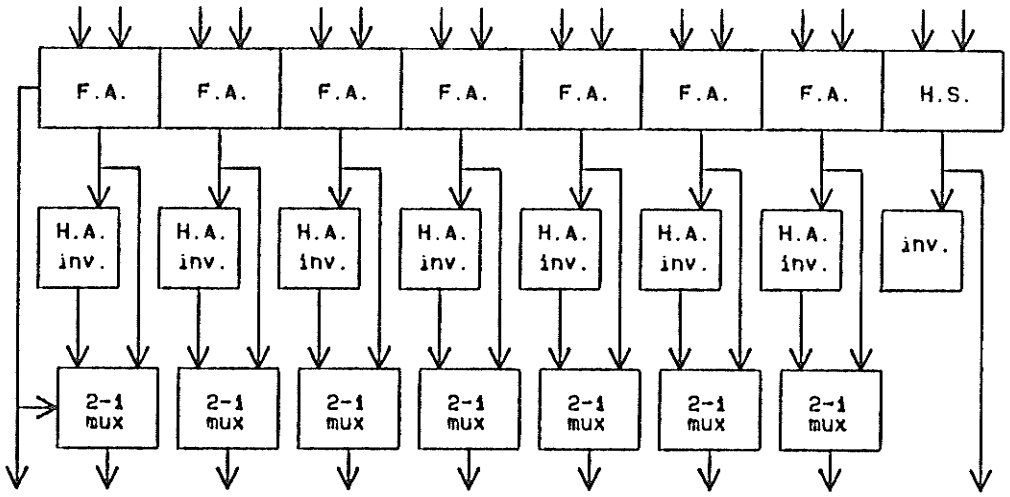
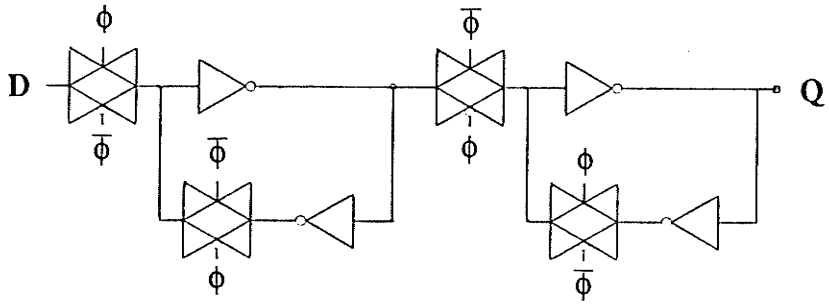


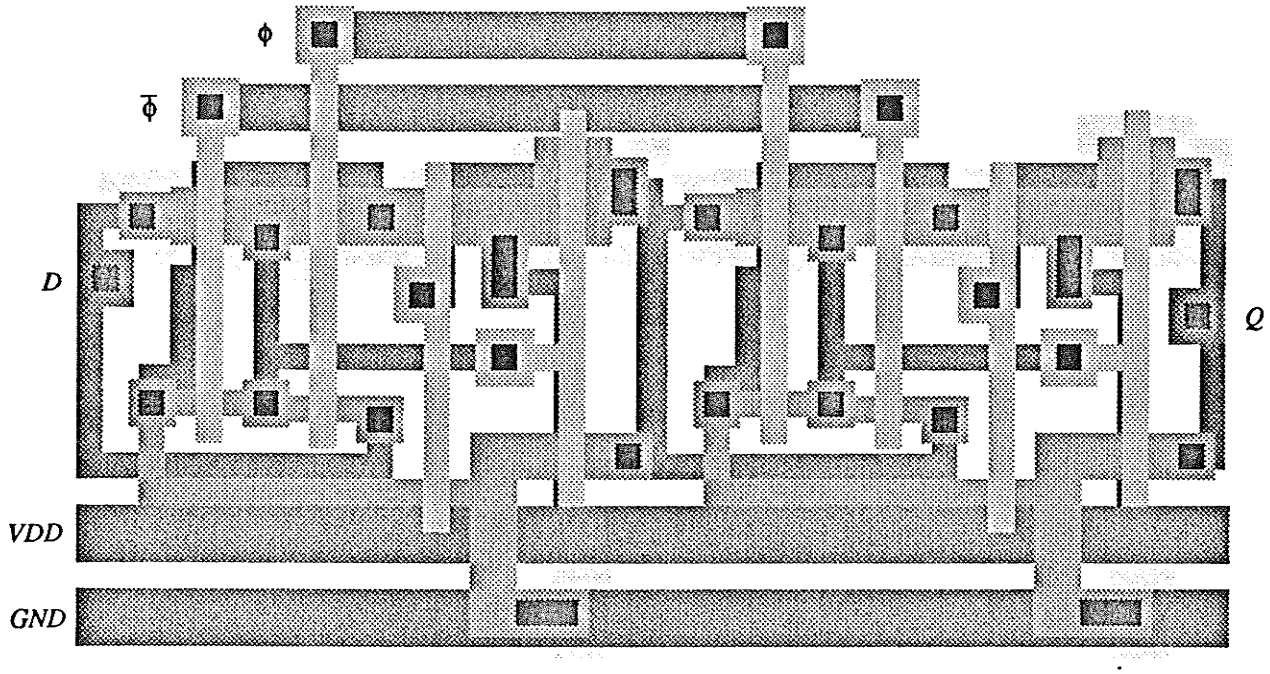
Figure 17: Subtractor unit layout

3.2. Flip-Flop Design

Each computational element in a systolic array requires some form of storage to hold inputs, weighting coefficients, and partial results as the data percolates through the array. A fan-in adder systolic array, however, eliminates the need for storage of partial results since, at each time step in the computation, all partial results are summed and the resultant sum exits from the one-dimensional systolic convolver. No storage is necessary for weighting coefficients since all weighting coefficients are hardwired in the design. Therefore, each computational element requires storage to hold a single input element. Each image pixel is represented by eight bits requiring eight bits of storage at each computational element. Because the input data percolates through the array, the data must be shifted from each computational element to the next on each clock cycle. Storage elements holding corresponding bits in each computational element are linked together forming shift registers to enable the data to percolate through the array. An edge-triggered D-type flip-flop (Figure 18(a)) is used as the basic storage element since, when linked together, D-type flip-flops become the basic delay circuit for a shift register.



(a)



(b)

Figure 18: D-type flip-flop (a) logic diagram and (b) CMOS layout

A potential problem with D-type flip-flops is that a *clock race* condition exists when ϕ is high and $\overline{\phi}$ overlaps it due to skew (ϕ and $\overline{\phi}$ are nonoverlapping clock phases). If this occurs the D input and feedback signal will *fight* to determine the new value on the input latch. Skew between the clock phases must be eliminated by ensuring the RC time constant of the ϕ and $\overline{\phi}$ phases is balanced. This is accomplished, in part, by balancing the loads of the ϕ and $\overline{\phi}$ phases of the silicon layout of the D-type

flip-flop. Figure 18(b) shows that the polysilicon wires and gates associated with the clock phases have equal RC components. The length of these polysilicon wires has been kept as short as possible to reduce the loading on the clock drivers. Furthermore, the clock phases are run in a parallel bus along the top of the flip-flop in the metal-1 layer using 6 μm width wires, enabling the flip-flop to be linked in a row to form multi-bit storage cells.

The silicon layout is a full custom design requiring 139 μm by 79 μm using 3 μm double-layer metal CMOS technology. The circuit requires 16 transistors as shown. The power bus runs along the bottom of the layout enabling the flip-flops to be linked in a row to form multi-bit storage cells without requiring an extra power bus to run to each storage cell bank. All I/O ports in the layout are exported with a metal via and no metal-2 layer wiring is used in the layout enabling inter-cell wiring using the metal-2 layer.

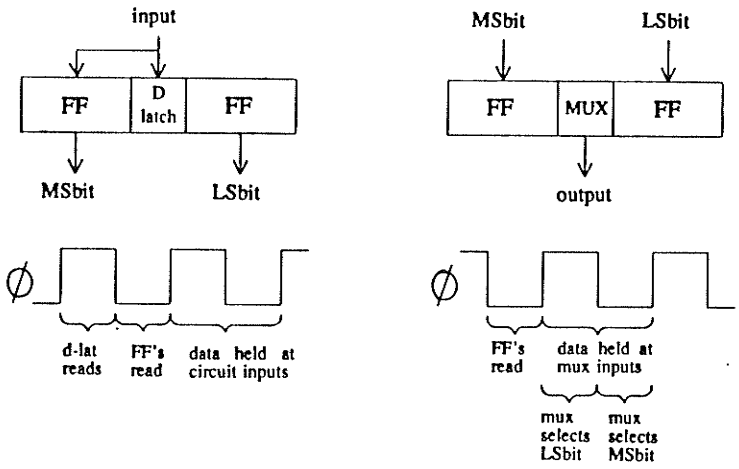
3.3. Multiplexer Design

Multiplexing of the I/O data is required due to the I/O limitations of the available pad frame within which the design is fabricated. The limited number of pads requires two I/O data bits to be multiplexed through each pad on every clock period. Input data is latched in using D-latches and output data is multiplexed out using 2-1 multiplexers. The D-latch requires ϕ and $\overline{\phi}$ clock phases and the 2-1 multiplexer requires *select* and $\overline{select}$ lines. Using the ϕ and $\overline{\phi}$ phases as the multiplexer select lines enables the I/O data to be multiplexed in and out of the design without the need for further clock phases.

The input data is latched in as shown in Figure 19(a). When ϕ is high the LS bit (of the two bits to be multiplexed into this input) is available on the input line and is consequently latched into the D-latch. The flip-flop inputs are not on. When ϕ goes

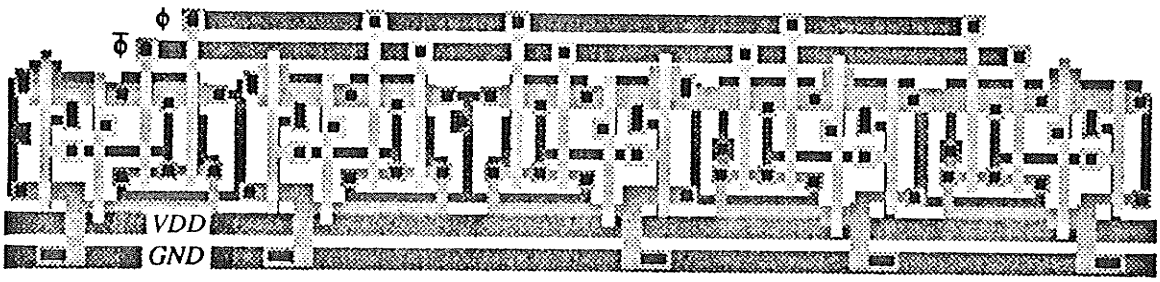
low the MS bit becomes available on the input line. The D-latch is latched and is no longer reading the input line. Both flip-flops are now reading their inputs. The flip-flop associated with the MS bit latches the data on the input line and the flip-flop associated with the LS bit latches the data held at the output of the D-latch. The LS and MS bits associated with the input line are held at the outputs of the two flip-flops for a full clock period and are therefore available to the down-line logic for the full clock period. The silicon layout of the input multiplexing circuitry is shown in Figure 20(a). The layout consists of two flip-flops and a D-latch where the D-latch is essentially half a flip-flop. The entire layout requires 340 μm by 79 μm in 3 μm double-metal CMOS technology.

Output data is multiplexed as shown in Figure 19(b). Two flip-flops hold the LS and MS bits (associated with the particular output line) at their outputs for a full clock period. The outputs are fed to the input of a 2-1 multiplexer. When ϕ is high the multiplexer selects the LS bit making it available on the output line and when ϕ is low the MS bit is selected. The silicon layout of the output multiplexing circuitry is shown in Figure 20(b). The layout consists of two flip-flops and a 2-1 multiplexer requiring a total of 350 μm by 79 μm in 3 μm double-metal CMOS technology.

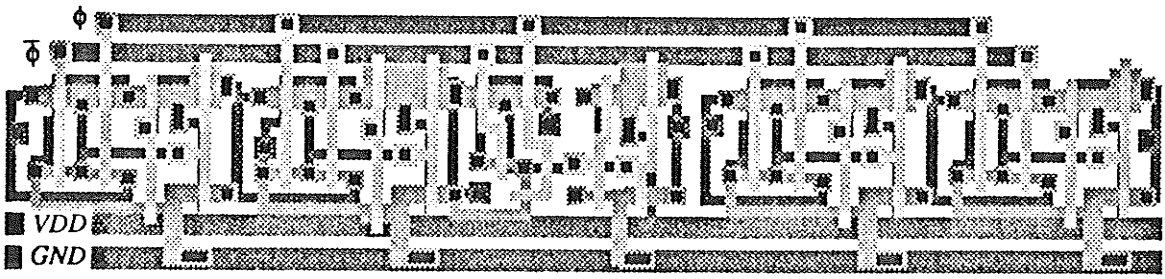


(a) (b)

Figure 19: Circuitry necessary for (a) input multiplexing and (b) output multiplexing



(a)



(b)

Figure 20: CMOS layout of (a) input multiplexer and (b) output multiplexer

3.4. Simulation

At the time of design, the only available simulator for the double-metal technology was CSIM, a switch-level simulator. A switch-level simulator is limited to functional verification of the layout, ie. to verify correct logic and wiring. Each device described in this chapter was simulated using CSIM to verify correct functionality. The full adder stage is a very crucial element in the design since it is the main building block for the adder and subtractor banks. Poor electrical performance of the full adder stage will be magnified in the electrical performance of the adder and subtractor banks and consequently in the entire chip. Verification of the electrical performance of the full adder stage under loaded conditions was therefore considered important. A timing-level simulator was therefore used to verify the performance of the full adder stage. The simulation was accomplished by duplicating the layout of the full adder cell using single-layer metal 3 μm CMOS technology for which a timing-level simulator was available. An eight-bit adder bank was designed using the full adder cell. Outputs of the adder bank were loaded using the inputs to a further adder bank. Simulation tests were then run using the timing-level simulator. Best case eight-bit additions were accomplished in 15 nanoseconds for output logic levels of 0.5V low and 4.5V high. Worst-case additions occur when the entire carry chain must be driven from the LS stage, the entire carry chain must change logic level, and each output load must change logic level. The worst-case eight-bit additions were accomplished in 48 nanoseconds for output logic levels of 0.5V low and 4.5V high. This simulation indicates each full adder stage requires 48/8 = 6 nanoseconds to compute a worst-case addition when configured in cascaded adder banks under loaded conditions.

CHAPTER 4

Floor Plan and Layout

The previous chapter described the design and layout of the necessary computational elements or blocks used in the final layout. This chapter deals with the placement of the blocks establishing the final floor plan of the design. The placement of the blocks is influenced by a number of criteria including power bus, clock bus, and data bus wiring considerations.

4.1. Floor Plan

Each computational block within the design requires power bus connections in order to function. The power bus must therefore be connected to every element within the design. The RC component of the power bus itself must be kept minimal to enable the bus to properly sink and source charge at each computational block. The bus is therefore wired using only metal, and the VDD line is never allowed to cross the GND line. When data buses are required to cross the power bus, the data buses, if necessary, use metal vias and no break is made in the power bus. Minimal crossover of buses is achieved by wiring power buses in a *finger* fashion. This method requires power buses to run in parallel at regular intervals across the layout with all VDD lines connected on one side of the layout and all GND lines connected on the other. The analogy of *fingers* is obvious when the left and right hand fingers are interleaved with palms toward one's face. No crossover of VDD and GND lines occur using this method and the power bus is available throughout the design.

All flip-flops, D-latches, and multiplexers within the design require clock bus connections. As with the power bus, the clock bus is wired only in metal to keep the RC product to a minimum. Data buses are broken when required to cross the clock bus. The clock bus is run in parallel across the design but only where it is required by flip-flops, D-latches, and multiplexers. Both phases of the clock bus are connected to the parallel buses on the same side of the layout. The ϕ and $\overline{\phi}$ wires will thus cross over each other at certain instances and require metal vias. However, connecting the clock phases on the same side of the layout is necessary to avoid skew between the clock phases (ie. ϕ and $\overline{\phi}$ must remain nonoverlapping).

The amount of wiring, the number of vias, and the length of wires used by a layout influences the simplicity with which it is built, the silicon area it requires, and the performance of the layout. To increase the simplicity of inter-block wiring, computational blocks are placed in the floor plan near blocks with which they share I/O. This tends to limit the length of the wires required and the number of crossovers between data buses. Computational blocks are oriented such that, where possible, the input and output of connecting blocks align in a one-to-one correspondence, thus eliminating the need for a data bus to cross over itself. Data buses are routed in such a way that crossover between buses is kept to a minimum. Crossover of data buses requires metal vias and the number of vias must be kept to a minimum since vias require more area, have a larger RC component, and are less fault tolerant than a simple metal wire.

Figures 3 and 8 show block diagrams of the entire multiresolution representation system and the G_0 systolic convolver respectively. Along with the area and I/O requirements of the individual computational blocks listed in the previous chapter, these block diagrams were used to layout a floor plan which stipulates the required area and I/O lines. Figure 21 shows the floor plan consisting of the three blocks ($G_0(x)$ convolver, $G_0(y)$ convolver, and subtractor unit) necessary to generate both a

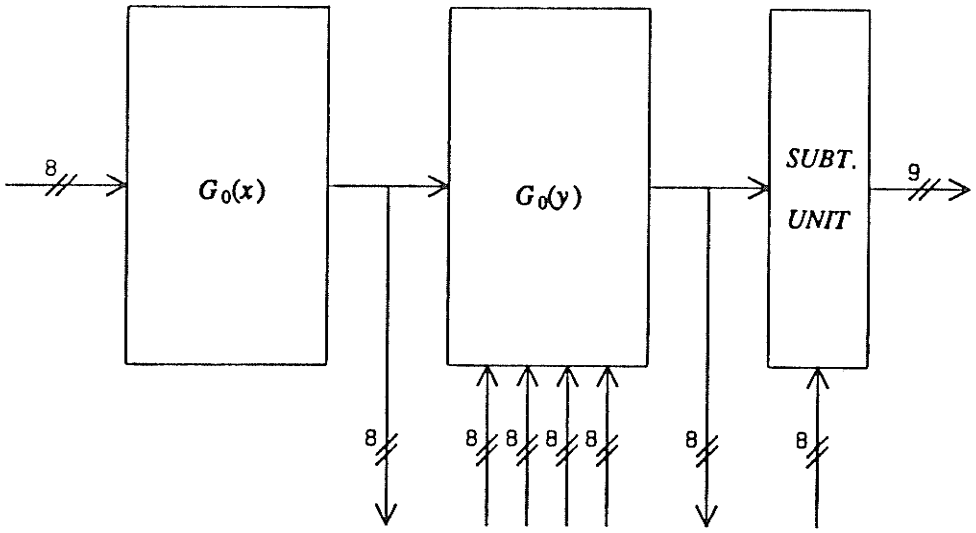


Figure 21: Floor plan showing computational blocks and I/O requirements.

lowpass and a bandpass representation of an image. Implementation of these blocks on a single chip enables a multiresolution representation system to be built by cascading a number of these chips together and adding the required number of shift registers at each level in the hierarchy. Shift registers are not implemented on chip since the number required is reduced by a half at each level in the hierarchy. Inclusion of the shift registers on chip would therefore negate the multiple use of the one chip to create the entire multiresolution system. Furthermore, it would restrict the image size with which the system could be used. Exclusion of the shift registers from the chip however, requires an additional five eight-bit I/O lines. The chip therefore requires two eight-bit data buses for the $G_0(x)$ convolver (one in and one out), five eight-bit data buses for the $G_0(y)$ convolver (four in and one out), and one eight-bit input data bus and a nine-bit sign-magnitude result from the subtractor unit. The number of I/O data lines required is $8 \times 8 + 9 = 73$. In addition, the layout requires four pads for the VDD, GND, ϕ, and $\overline{\phi}$ lines. Since the pad frame used contains only 40 I/O pads the need to multiplex I/O data becomes evident. To reduce the number of required pads

to 40, each eight-bit data bus is multiplexed in or out of the chip using four pads. The two MS bits of the magnitude result of the subtractor unit are ignored, and the remaining six bits are multiplexed out using three pads. Dropping the two MS bits can be done without loss of information since the subtraction operation removes pixel to pixel correlations, shifting pixel values toward zero and allowing six-bit representations of each pixel (see Sec. 6.1). The I/O pad requirements then become $8 \times 4 + 3 + 1 + 4 = 40$ pads. All input data buses are recieved on chip using an input multiplexer bank which also holds the data in flip-flops for a full clock period. This enables the input data to be available to the required computational elements for a full clock period at each time step in the computation. Output buses use output multiplexer banks which present each data bit to the output pad for half a clock period. A detailed floor plan of input and output multiplexer banks, adder banks, and the subtractor unit, as well as the required data bus wiring between these computational blocks is shown in Figure 22.

The placement of the computational blocks allows for the floor plan design criteria to be satisfied. 1) The power buses are run in a *finger* fashion at regular intervals throughout the layout. There are essentially two columns of computational banks in the floor plan. These columns of banks are placed in such a way to enable the aligning of the power bus which runs along the bottom of each bank. Each power bus therefore runs across the entire layout allowing VDD to be connected on the left side of the layout and GND to be connected on the right. Both VDD and GND are wired using only the metal-1 layer. 2) The clock bus is run in parallel across the design only where it is required. Both clock phases are connected on the same side of the layout to eliminate skew between the phases. Clock buses running horizontally are wired using the metal-1 layer only allowing metal-2 data buses to be run vertically over top. The vertical clock bus is run in the metal-2 layer enabling it to cross over the VDD

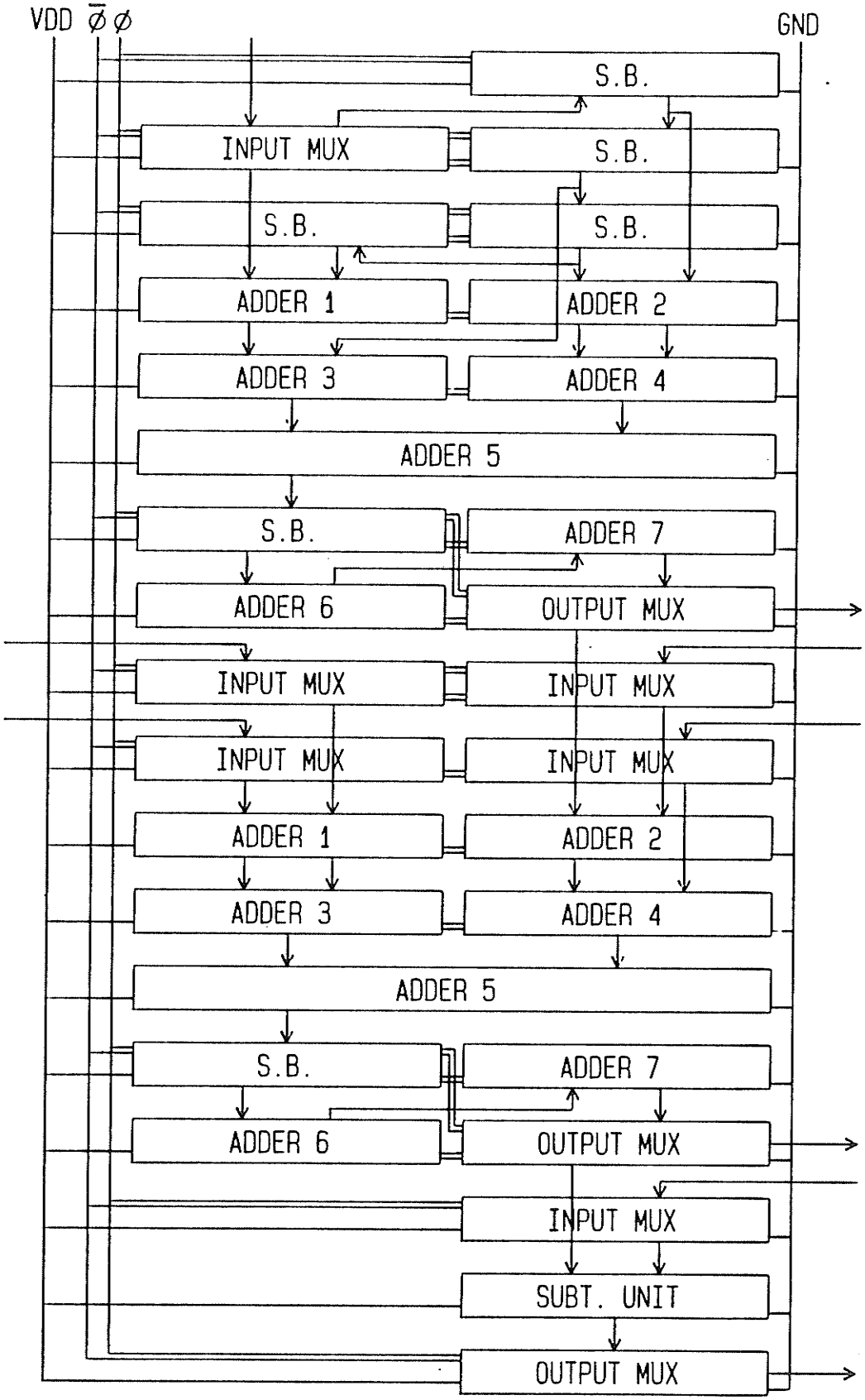


Figure 22: Detailed floor plan showing computational block placement and bus wiring.

wires without the need for metal vias. 3) Computational banks are placed in such a way as to eliminate all global wiring and to minimize data bus crossover. Corresponding bits in the adder banks are for the most part aligned vertically, therefore data bus cross over is only necessary when a data bus is required to connect one column to the other.

All vertical data buses are run in the metal-2 layer enabling the buses to cross directly over computational banks. This is possible since the cells which comprise the computational banks were designed without metal-2 wiring. Crossing over computational banks allows data buses to be far shorter and have far less crossover than would otherwise be possible since the buses would then be required to snake around computational banks and vertical alignment of adder banks could not be used to minimize crossover.

To determine the necessary area requirements of the floor plan is straightforward. All computational banks are approximately the same size. An average size is about 1200 μm by 90 μm. Allowing an additional 30 μm of width for wiring at each bank gives an average size of 1200 μm by 120 μm. Since the floor plan consists of two columns and 18 rows of computational banks, a rough estimate of the required area may be given as (1200 + 1200) by (120 $\times$ 18) or 2400 μm by 2160 μm. The pad frame used has outside dimensions of 4500 μm by 4500 μm with usable interior layout area of 3900 μm by 3900 μm. The floor plan layout should, therefore, when laid out in silicon, fit easily into the pad frame.

4.2. CMOS Layout

The layout of the design is shown in Figure 23 with the final pad-frame layout and pad connections shown in Figure 24. 2715 μm by 2192 μm of silicon area are required by the layout. The circuit contains 5264 transistors. Most evident in the

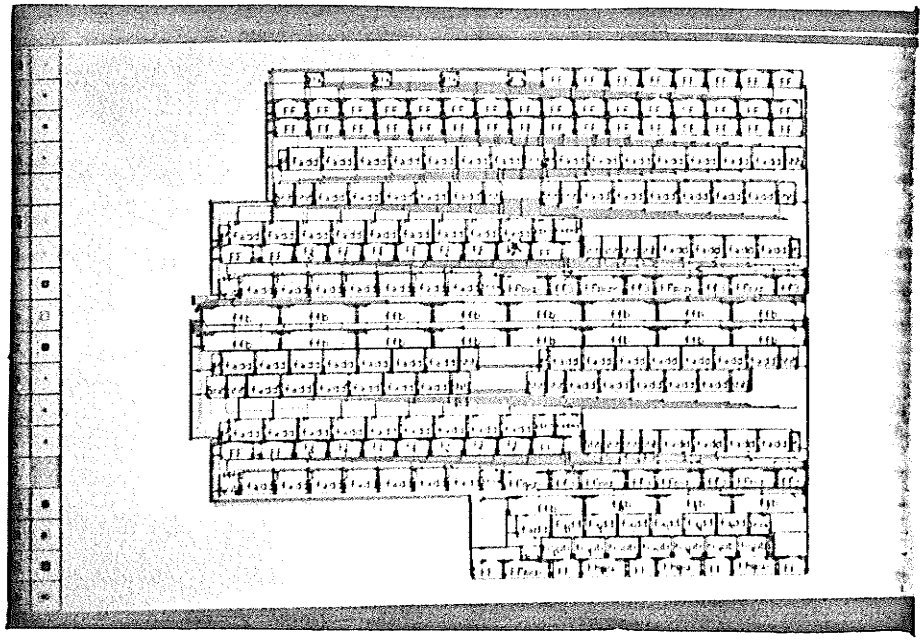


Figure 23: Layout of the ASIC

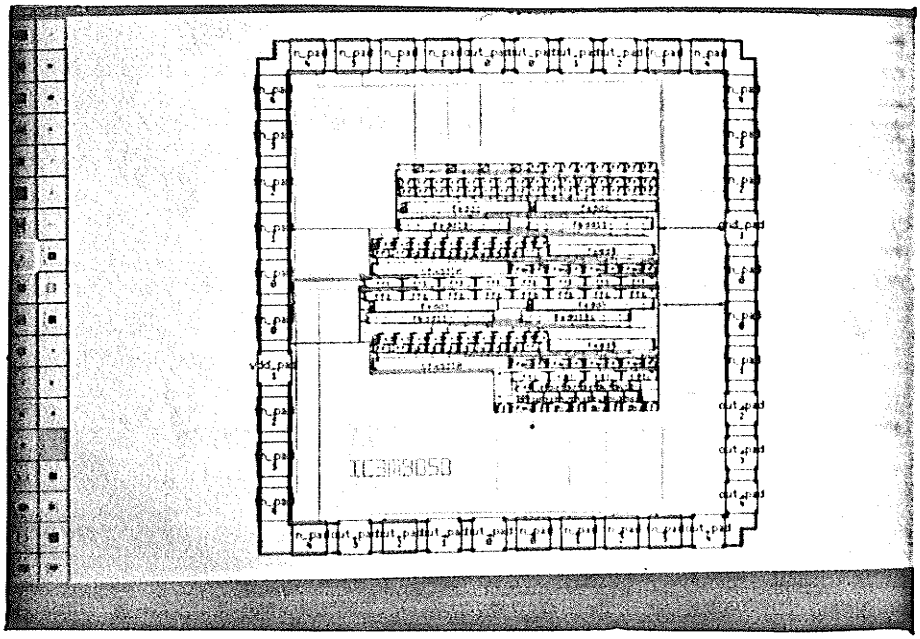


Figure 24: Final pad-frame layout.

layout is the small percentage of wire area used by the design. This is largely due to the efficient placement of the computational banks and the fact that the banks have been designed to allow metal-2 wiring to be run across them. All wiring has been done using only metal-1 and metal-2 layers thus minimizing the RC component of the wires. All data buses are local and therefore relatively short thus negating the need for large drivers or repeaters that would be necessary on global buses.

The final layout of the ASIC enables the design of the multiresolution representation system shown in Figure 25. Cascading the ASIC as shown results in an extremely modular system requiring the simple addition of more ASIC's and shift registers to generate further low-resolution levels in the representation. Resampling of the image data between levels in the hierarchy is acheived by using the clocking scheme shown. The scheme allows each ASIC to read every second pixel and every second image row from the output of the ASIC above it.

4.3. Timing

The architecture of the ASIC is based on a systolic architecture. Therefore all pipeline blocks in the systolic array are synchronized with a global clock and each block receives new inputs and calculates a new output on every clock pulse. Consequently, on every clock pulse, the ASIC receives one new image element and outputs two image elements corresponding to the lowpass filtered output and the bandpass filtered output. There are five pipeline blocks in the ASIC; two each in the $G_0(x)$ and the $G_0(y)$ convolvers and one in the subtractor unit.

The shift registers which allow the convolution window to overlay the correct region of the image can be thought of as a second dimension of pipelining. Both dimensions of pipelining influence the timing sequence of the ASIC. A timing diagram illustrating the flow of data through the ASIC is shown in Figure 26.

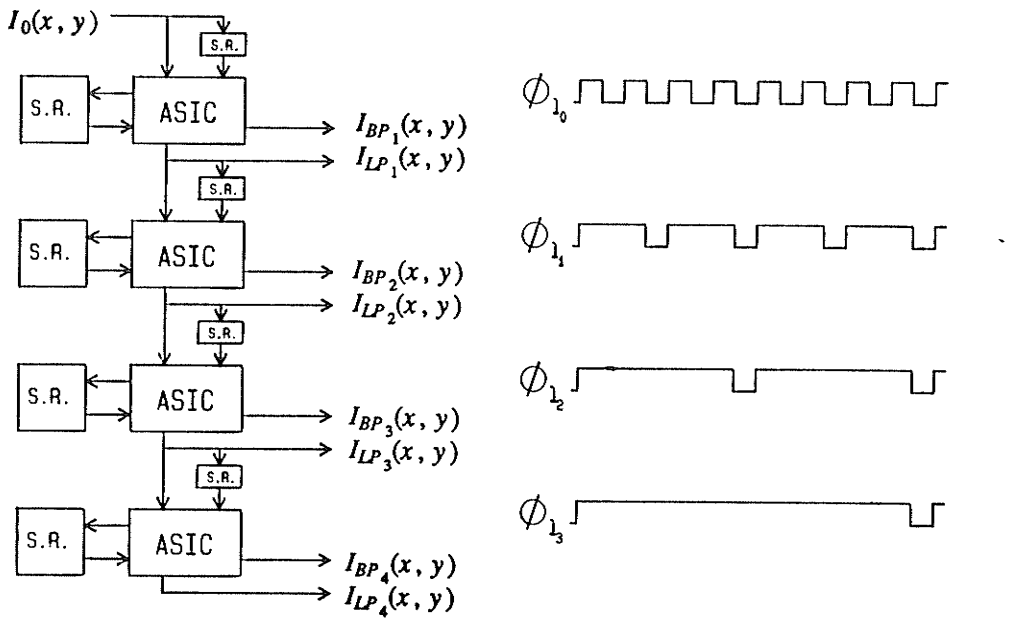


Figure 25: Cascading the ASIC to produce a multiresolution representation system. $I_0(x, y)$ is the original image, $I_{LP_i}(x, y)$ and $I_{BP_i}(x, y)$ are lowpass and bandpass representations of $I_0(x, y)$. The clocking scheme shown effectively resamples the image data between levels in the hierarchy.

$I_{LP_{i-1}}(m, n)$ is the input image element to the level i ASIC, $I'_{LP_i}(m, n)$ is an image element which has been processed by the $G_0(x)$ convolver at level i, $I_{LP_i}(m, n)$ is an image element which has been processed by the $G_0(y)$ convolver at level i, and $I_{BP_i}(m, n)$ is an image element corresponding to the level i bandpass filtered output. $I_{LP_i}(x, y)$ and $I_{BP_i}(x, y)$ are generated 6 and 7 clock periods respectively after the $I_{LP_i}(x + 2, y + 2)$ input is read. The resultant latency of the system is equal to $(2^i N + (2 + 6) \times 2^{i-1})$ clock periods for the $I_{LP_i}(x, y)$ output and $(2^i N + (2 + 7) \times 2^{i-1})$ clock periods for the $I_{BP_i}(x, y)$ output where N is the number of image elements per image row and i is the level in the hierarchy.

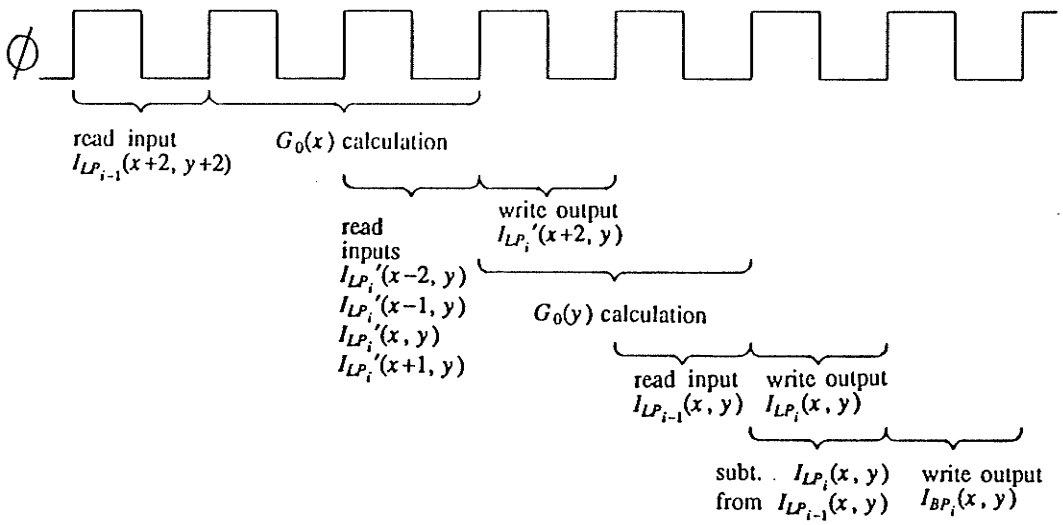


Figure 26: Timing diagram showing timing sequence of the computation

4.4. Simulation

Simulation of the layout is relatively straightforward since the design employs only synchronous logic. The only available simulator is a switch-level simulator called CSIM. It is therefore possible to verify only that the correct logic and wiring has been implemented in the layout process. The design is composed of several functional blocks including the $G_0(x)$ convolver, the $G_0(y)$ convolver, and the subtractor unit. Evaluation of the outputs of each of these functional blocks during simulation reduces the required number of test cases needed in order to exhaustively test all data paths in the circuit. Exhaustive testing at this level using a switch-level simulator is relatively useless since the logic of each smaller building block or cell within the design has been verified by simulation previously. Therefore it is only useful to verify that the cells have been wired together correctly. Approximately 100 test cases were simulated all of which produced the correct outputs.

The design criteria of real-time processing influenced both the algorithmic and arcitectural design. Since it is impossible to determine, using switch-level simulation,

whether or not this criteria has been met, some estimation of the computation time required by the layout was needed before the chip was fabricated. A reasonably good estimation was made using the worst-case computation time of the full adder cell which was acquired from the timing-level simulation (Sec. 3.4). Since the layout uses a pipelined architecture, data can only be clocked through the layout at the rate at which the data can be clocked through the pipeline block with the largest computation time. The block with the largest computation time is the G_0 convolver block. An estimate of its required computation time may be found as follows. The data must propagate through full and half adders as shown in Figure 27. The worst-case computation time through each adder cell has been simulated at $t = 6$ *nanoseconds* (Sec. 3.4).

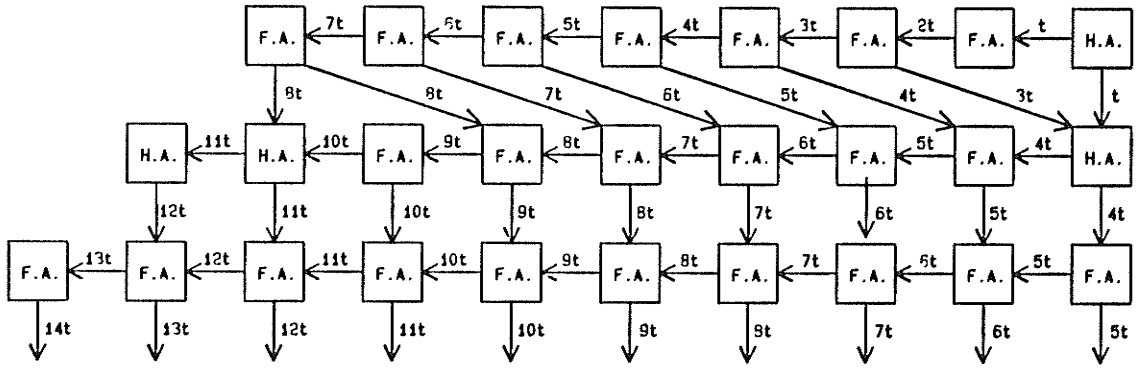


Figure 27: Worst-case computation time estimate for the G_0 convolver.

The longest path in Figure 27 passes through 14 full and half adder cells. Thus, $14 \times 6 = 84$ *nanoseconds*. After computation the output must be read by a storage cell. The flip-flops used for the storage cell require a 6 *nanosecond* set-up time, giving $84 + 6 = 90$ *nanoseconds* for the worst-case computation time. All other pipeline blocks in the layout require less computation time. Therefore, a rough estimate for the frequency at which data may be clocked through the layout is $\frac{1}{90 \times 10^{-9}} = 11.1\ Mhz$.

At this rate the system will process a 512×512 image in $90 \times 10^{-9} \times 512 \times 512 + latency$. The number of image frames the system can process per second is independent of the latency. For a 512×512 image, the multiresolution system will process $\frac{1}{90 \times 10^{-9} \times 512 \times 512}$ or 42 image frames per second. This is well above TV rates of 30 image frames per second.

CHAPTER 5

Testing

Five ASIC's were fabricated and available for testing purposes. A photomicrograph of a chip is shown in Figure 28. Each ASIC was tested by generating test vectors on an HP 8180A data generator and analyzing the outputs using an HP 8182A data analyzer. All ASIC's were functionally tested for correct operation using test vectors. Each working ASIC was then tested for its maximum operating frequency under worst-case computational conditions.

The testing of each ASIC is simplified by two results of its architectural design. Firstly, the design employs only synchronous logic and thus dynamic faults are eliminated. Secondly, block testing of the chip is possible since outputs are available at three different points in the computational logic of the design. Block testing reduces the combinational explosion of testing by reducing the number of test vectors required to ensure a given percentage of internal nodes at each particular block are correct. Outputs are available at three points in the ASIC; after the $G_0(x)$ block, after the $G_0(y)$ block, and after the subtractor unit.

Each ASIC has 20 input lines and thus 40 inputs to its combinational logic. The outputs of the ASIC are dependent on its inputs and its internal state. The internal shift register of the $G_0(x)$ systolic convolver has 32 bits of storage resulting in internal states which influence the outputs. These internal states must be clocked in before each test case can be run. Exhaustive testing of the combinational logic will therefore require $2^{40} \times 2^{32} = 2^{72}$ test cases. Block testing reduces the necessary number of test

cases to $2^{40} + 2^{40} + 2^{16}$. Although block testing greatly reduces the number of test cases, clearly, exhaustive testing is impossible. It is therefore required to generate a minimal number of test cases to verify (1) the correct behavior of the ASIC and (2) the correct operation of a given percentage of internal nodes.

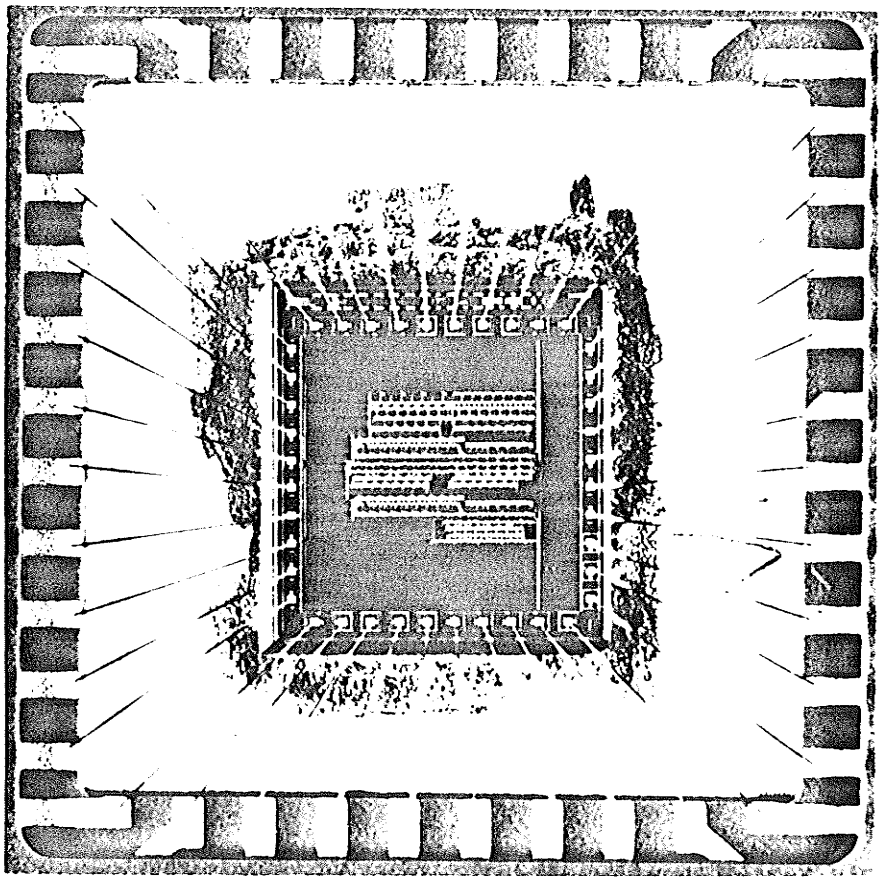


Figure 28: Photomicrograph of the ASIC

The correct operation of some internal nodes can be accomplished by running three simple test cases.

Case (1). All inputs are set to zero and the data is clocked through to the outputs. The outputs of the $G_0(x)$ and $G_0(y)$ blocks should both be zero. The magnitude of the subtractor unit should be zero and its sign bit should be one indicating a positive result. All internal nodes corresponding to sum and carry-out nodes in the adder banks

must be zero for correct operation. Carry-out nodes in the subtractor bank will be high with sum nodes being low.

Case (2). All inputs are set to one and the data is clocked through to the outputs. The outputs of the $G_0(x)$ and the $G_0(y)$ blocks should both equal 254 (not 255 because of the 0.996 scale factor). The magnitude of the subtractor unit will be one and its sign bit will also be one (255 - 254 = 1). Most internal nodes corresponding to sum and carry-out nodes in the adder banks will be one. The states of the sum and carry-out nodes in the subtractor banks will have both high and low logic levels.

Case (3). All inputs are left high except for the subtractor unit inputs which are set low. The magnitude of the subtractor unit output will be 254 and its sign bit will be zero indicating a negative result.

These three test cases eliminate the possibility of stuck at faults at all output nodes and some internal nodes. Secondly, correct propagation of the carry signal through the carry chain in each adder and subtractor bank is verified. Due to the operational verification determined by these test cases, further test was considered necessary using only a small number of random test cases. Fifteen random test cases were run on one of the ASIC's which had correct results from the first three test cases.

Maximum operating frequency under worst-case computational conditions is determined by applying input vectors to the ASIC which correspond to the worst-case computational conditions. The operating frequency of the ASIC will be limited by the maximum operating frequency of adder banks 1 through 5 in both the $G_0(x)$ convolver and the $G_0(y)$ convolver. Worst-case conditions occur when the carry signal is required to propagate a maximum distance through the carry chain in all adder banks. A worst-case computation will occur when each bit of pixels a and c are ones and all bits of pixels b, d, and e are zero (see Section 2.5 for explanation of pixel labeling). This causes the carry signal to propagate through the entire carry chain in adder bank

1, through three-quarters of adder bank 3's carry chain, and through three-quarters of the carry chain in adder bank 5. Computation times of adder banks 2 and 4 are irrelevant since they operate in parallel with adder banks 1 and 3 respectively and have a computation time no worse then the worst-case computation time of adder banks 1 and 3.

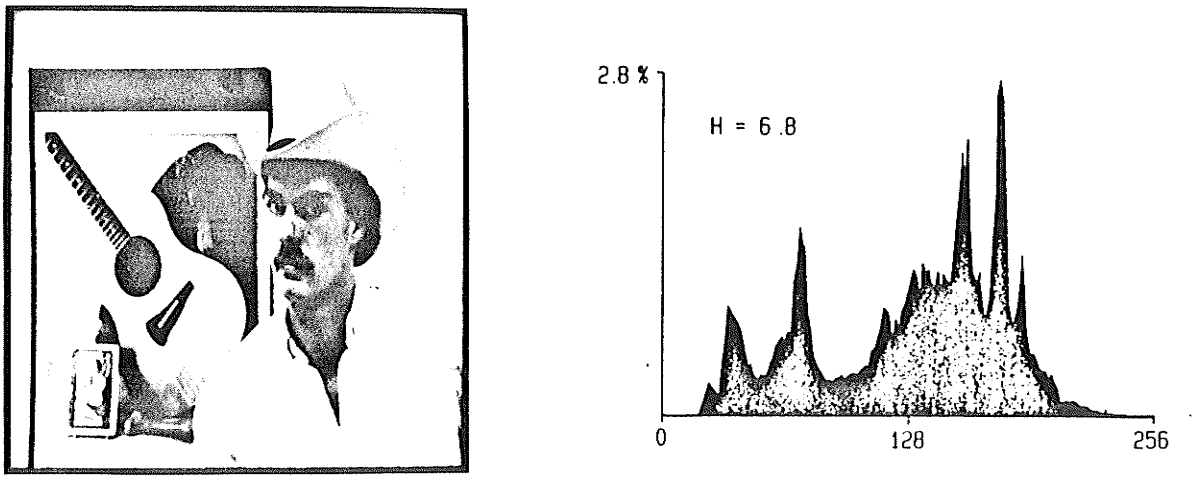
Of the five ASIC's tested, one did not function properly. The faulty ASIC powered-up but produced erroneous outputs when test vectors were clocked in. The remaining four tested correctly under the three test cases mentioned above and tested correctly under worst-case conditions. One of the four had a maximum operating frequency under worst-case conditions of 10 *Mhz*. The remaining three operated correctly at 12 *Mhz*. The maximum operating frequency is the highest frequency at which logical testing produces correct results. One ASIC was then tested using the fifteen random test cases. All results were correct.

CHAPTER 6

Simulation and Applications

6.1. Simulation

Computer simulation of the multiresolution representation system was accomplished using a 512 by 512 image where each pixel is represented with eight bits. The first four levels of both the lowpass and bandpass representations were generated in the simulation. The programs used for the simulations are listed in Appendix A. The test image and its histogram are shown in Figure 29(a) and (b) respectively.



(a) **(b)**

Figure 29: (a) The test image, (b) histogram of the test image.

It is a digitization of a scene containing both sharply outlined objects and shadowy objects. Since it is an unaltered digitization of a real scene, it contains noise. Figure 30 shows the lowpass-filtered and subsampled outputs I_{LP_1} thru I_{LP_4}. Figures 31(a)-(d)

show each of these filtered copies with pixel sizes magnified for ease of viewing. The reduction of information at each level is evident.

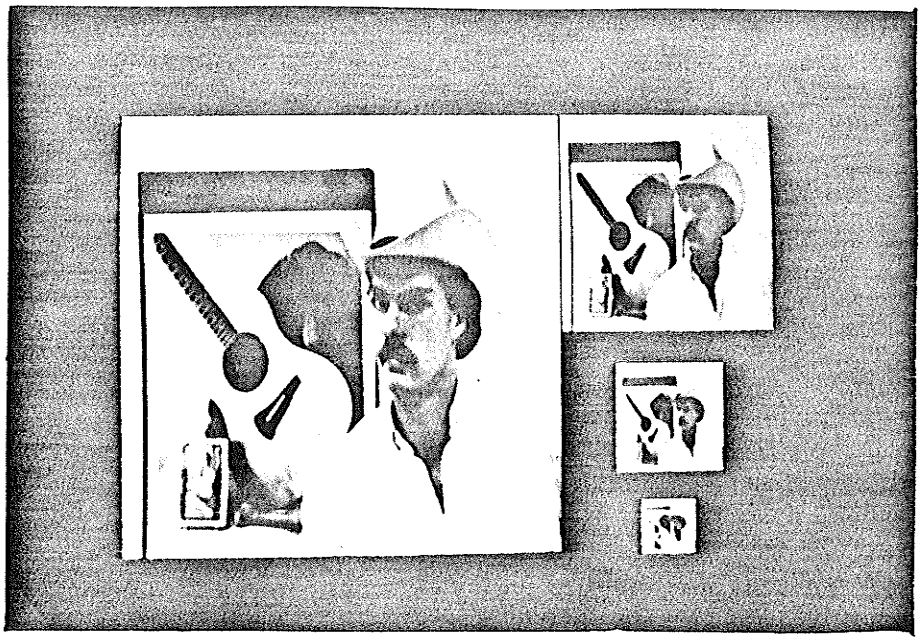


Figure 30: Lowpass-filtered representations I_{LP_1} thru I_{LP_4}.

Figure 32 shows the bandpass filtered outputs I_{BP_1} thru I_{BP_4}. Generation of these outputs has removed pixel-to-pixel correlations and has shifted pixel values towards zero. This compression of data is evident from the histograms of I_{BP_1} thru I_{BP_4} shown in Figure 33. Pixel values are clustered about zero and have both positive and negative values between -128 and +128. To accommodate the negative pixel values for viewing purposes, an offset has been added linearly to the gray scale to enable a zero pixel value to have a gray scale level of 128, a pixel of 128 to have a gray scale level of 256, and a pixel value of -128 to have a gray scale level of 0. For viewing purposes, histogram equalization was performed on each bandpass image. Figures 33(a)-(d) show these results with pixel sizes magnified. The consecutively-reduced center frequency of the passband is evident in these figures. I_{BP_1} accentuates the high frequency

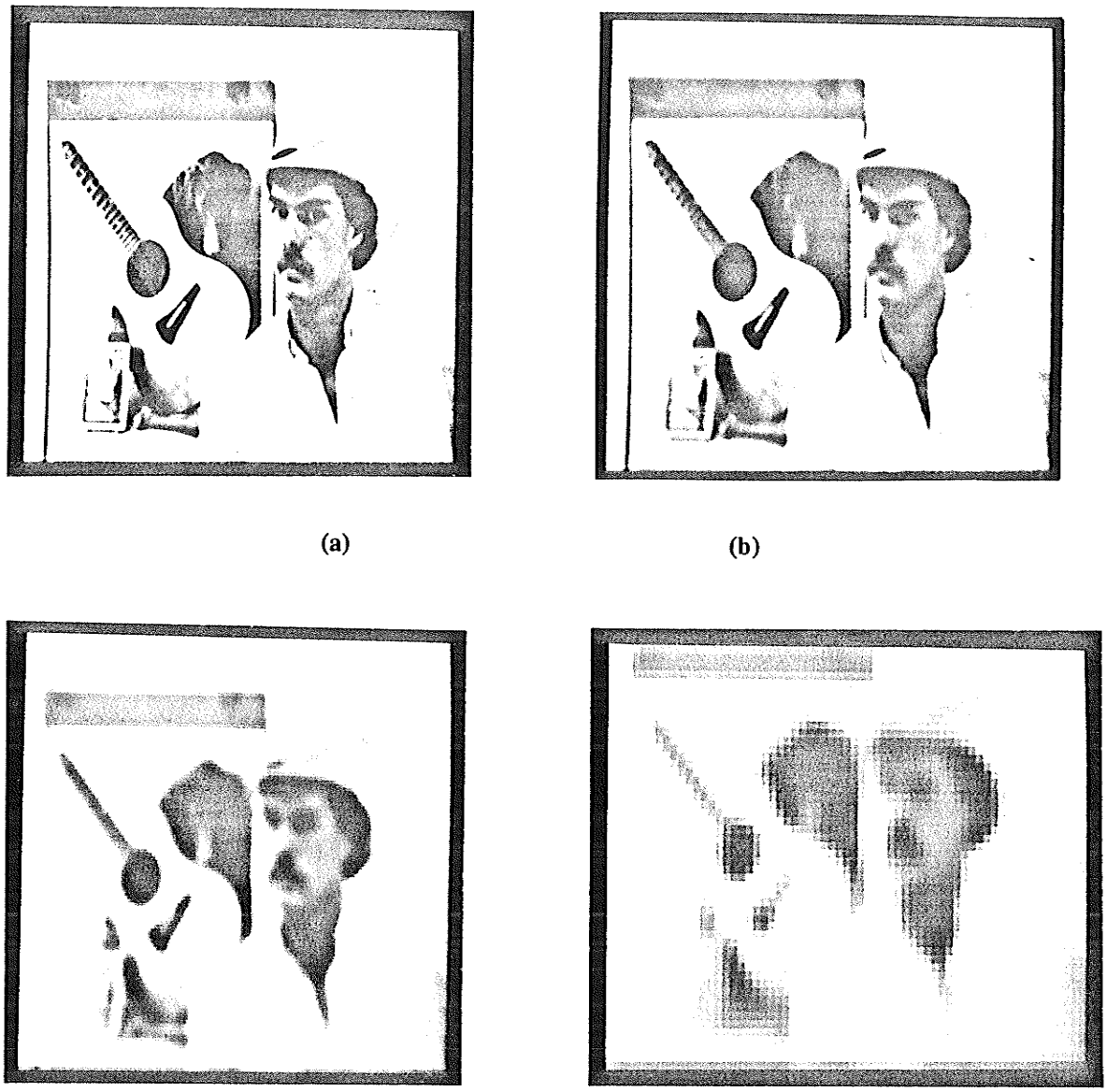


Figure 31: Pixel sizes magnified. (a) I_{LP_1}, (b) I_{LP_2}, (c) I_{LP_3}, (d) I_{LP_4}.

component of the original image whereas I_{BP_4} portrays the overall shape and size of the objects within the image. The overtone medium-level gray of each bandpass image is due to the fact that only the relevant image information at each passband is represented allowing considerable data compression. The entropy, H, given as $H = \sum_{0}^{k} p(r_k) \log_2 p(r_k)$, represents the minimum number of bits per pixel required to

exactly encode the image (k is the number of gray levels and $p(r_k)$ is probability of the k^{th} gray level).

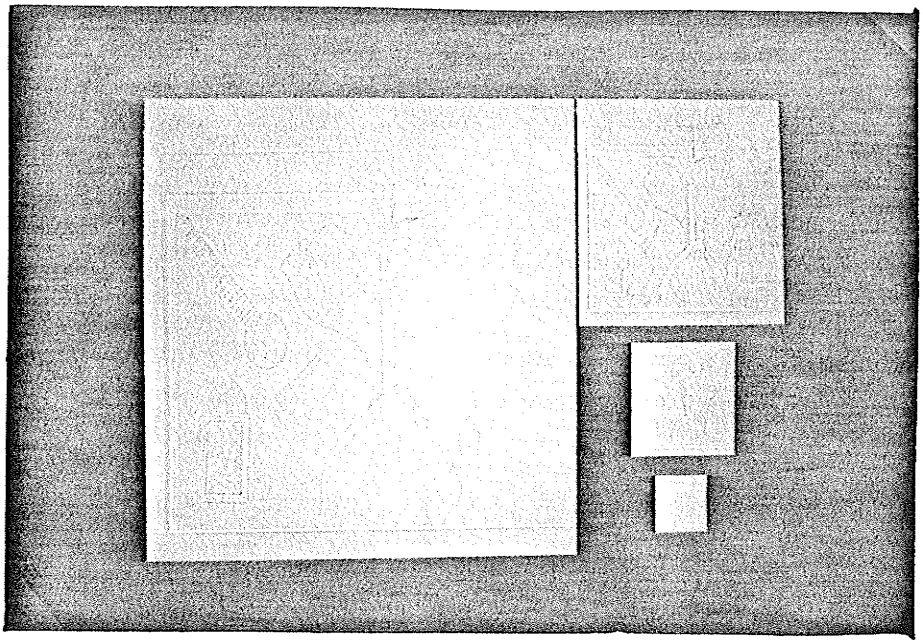


Figure 32: Bandpass-filtered representations I_{BP_1} thru I_{BP_4}.

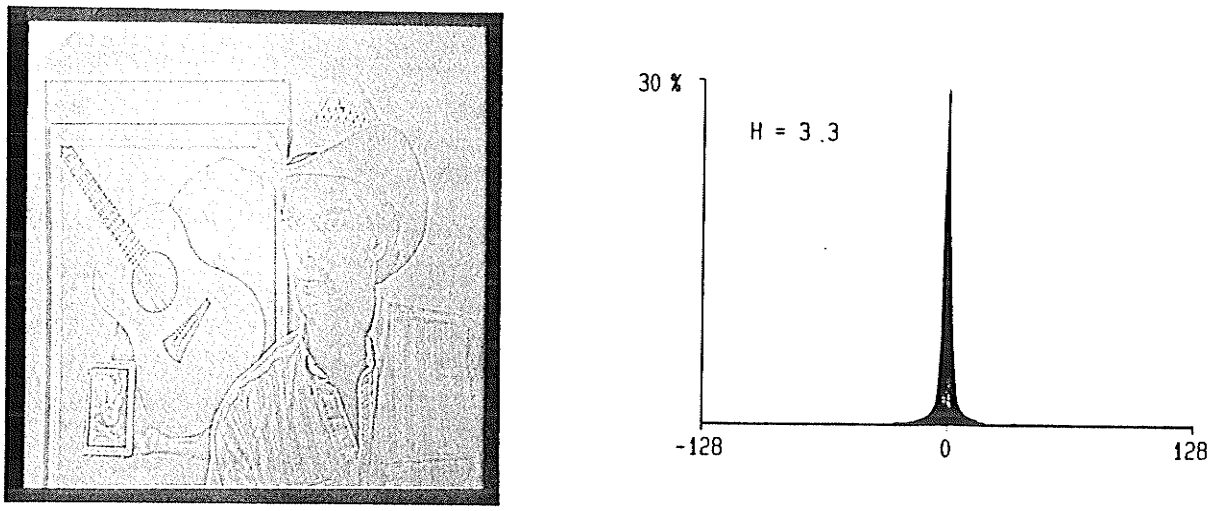
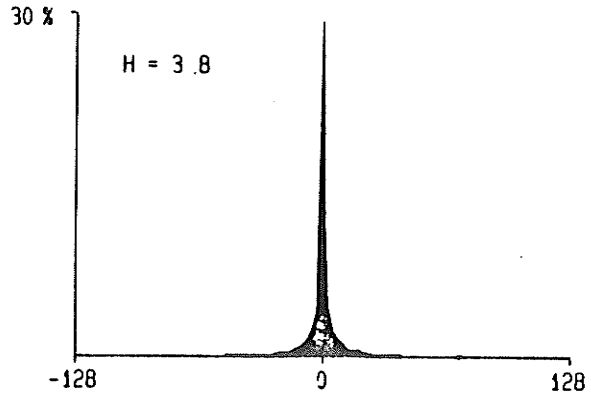
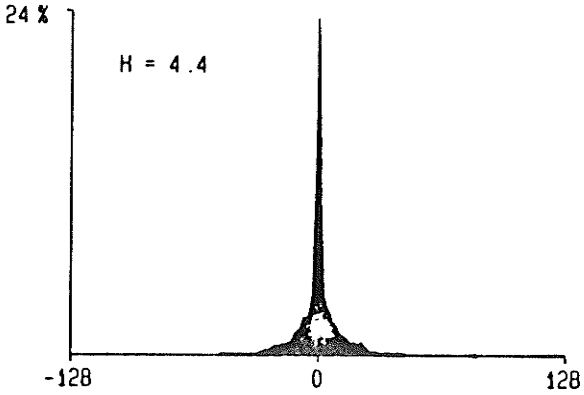


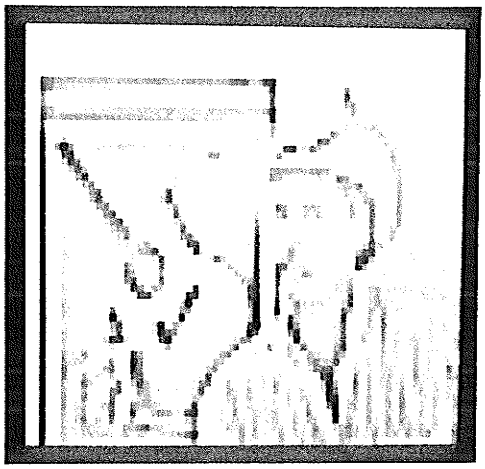
Figure 33(a)

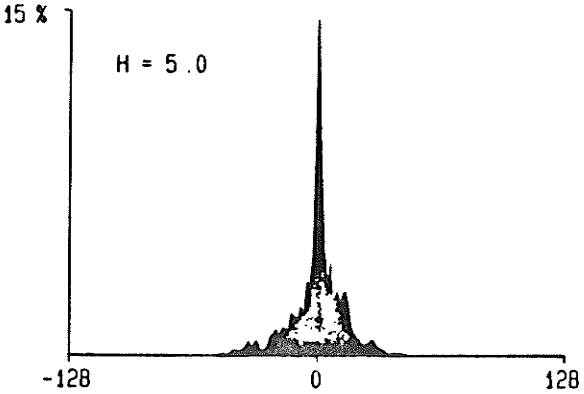


(b)



(c)





(d)

Figure 33: Histogram equalized bandpass filtered representations with pixel sizes magnified. (a) I_{BP_1} and histogram, (b) I_{BP_2} and histogram, (c) I_{BP_3} and histogram, (d) I_{BP_4} and histogram.

An important attribute of a multiresolution representation is completeness: it should be possible to recover the original image from its multiresolution representation. The bandpass multiresolution representation has been generated as follows;

$$I_{BP_1} = I_0 - I_{LP_1}$$

$$I_{BP_2} = resample\,(I_{LP_1}) - I_{LP_2}$$

$$I_{BP_3} = resample\,(I_{LP_2}) - I_{LP_3}$$

$$I_{BP_4} = resample\,(I_{LP_3}) - I_{LP_4}$$

The original image I_0, may be reconstructed from the first four bandpass level representations and fourth lowpass representation as follows;

$$I_{LP_3} = expand\,(I_{BP_4} + I_{LP_4})$$

$$I_{LP_2} = expand\,(I_{BP_3} + I_{LP_3})$$

$$I_{LP_1} = expand\,(I_{BP_2} + I_{LP_2})$$

$$I_0 = I_{BP_1} + I_{LP_1}$$

OR

$$I_0 = I_{BP_1} + expand\,(I_{BP_2} + expand\,(I_{BP_3} + expand\,(I_{BP_4} + I_{LP_4})$$

The expansion process produces I_{LP_i} $(size\ m \times m)$ from the images $I_{BP_{i+1}}$ $(size\ m/2 \times m/2)$ and $I_{LP_{i+1}}$ $(size\ m/2 \times m/2)$. Addition of corresponding pixels in images $I_{BP_{i+1}}$ and $I_{LP_{i+1}}$ produces the resampled version of I_{LP_i}. The remaining pixel values of I_{LP_i} which were removed in the resampling process may be calculated from $I_{BP_{i+1}}$ and $I_{LP_{i+1}}$. A one-dimensional $I_{BP_{i+1}}$ contains $m/2$ equations in m unknowns and a one-dimensional $I_{LP_{i+1}}$ contains $m/2$ equations in the same m unknowns. Thus, in total there are m equations in m unknowns and the remaining pixels may be calculated

exactly. Since the two-dimensional process is separable into consecutive one-dimensional processes, the I_{LP_i} image may be reconstructed. This method however, is computationally expensive and is not practicable. An approximate method of reconstruction was used were each resampled pixel was interpolated from its two horizontal or vertical neighbors, where use of horizontal or vertical neighbors depends upon the location of the pixel in the resampled grid. This process is both computationally simple and efficient, and is very easily implemented in real-time with the appropriate hardware. Figure 34 shows the reconstructed image. Only close scrutiny on a high-resolution monitor reveals any degradation from the original image.

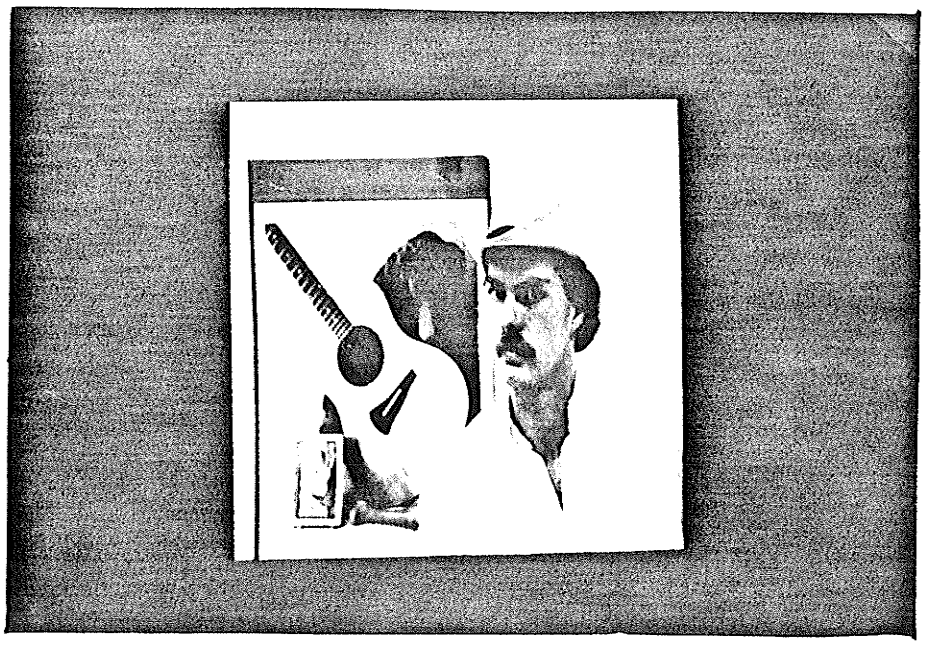


Figure 34: The reconstructed image.

6.2. Applications

Of the numerous applications applicable to lowpass and bandpass multiresolution representations, two are briefly explained here. The first uses the representation for

image data compression and the second application produces a multiresolution representation of the intensity changes or *edges* in an image.

6.2.1. Image Data Compression

Neighboring pixels within an image are highly correlated and therefore direct representation of the pixels results in redundant information being represented. A process which decorrelates the image pixels will enable a compressed data representation of the image. Decorrelation of image pixels may be achieved through predictive or transformation techniques. Transformation techniques encode pixels in blocks and involve image transforms or solutions to large sets of simultaneous equations. Predictive techniques encode pixels sequentially by subtracting the predicted value of a pixel from its actual value. Predictive coding is computationally simple and inexpensive. Transformation coding is computationally expensive; however, it offers greater data compression.

There are essentially two types of predictive coding: causal and noncausal. Causal prediction predicts a pixel's value using only previously-encoded pixel values. Noncausal prediction uses symmetric neighborhoods centered about each pixel and thus yields greater data compression. The method of data compression described here uses multiresolution symmetric neighborhoods to predict each pixel value noncausally.

If I_0 is the image to be encoded, then each pixel's predicted value will be an expanded version of the resampled I_{LP_1}. The expansion process constitutes an interpolation using simple local averaging. Each predicted pixel value is subtracted from the pixel's original value to obtain the prediction error. The array of prediction error values, E_1, essentially represent a bandpass filtered copy of I_0. Encoding E_1 (*size* $n \times n$) and I_{LP_1} (*size* $n/2 \times n/2$) rather then I_0 (*size* $n \times n$) results in a net

data compression since E_1 is largely decorrelated. Further data compression is achieved by applying the same process to I_{LP_1}. Iteration of the process produces a multiresolution representation of predictive error images $E_1, E_2, ..., E_N$, where $E_N = I_{LP_N}$. I_0 is recovered without error from this representation by expanding each predictive error image to size $n \times n$ and summing, ie. $I_0 = \sum_0^N expand(E_i)$. Expansion is identical to the expansion used to obtain the predicted values of each pixel. Further data compression is possible by quantizing pixel values if quantization errors are allowed by the application. Burt and Adelson [7] have used this method to encode images with negligible degradation at less than one bit per pixel.

The required expansion process used to produce the prediction pixel values and expand the prediction error images is considerably less computationally expensive then the ASIC designed for the generation of the lowpass representations, I_{LP_i}. Thus, a system could be constructed to encode and decode noncausally in real time.

6.2.2. Image Edges

As indicated in chapter 1 the first step in visual information processing involved representing an images intensity changes or edges. The zero-crossings of a Difference of Gaussian multiresolution representation localize image edges in a near optimal manner. Figure 35 shows the zero-crossings of the bandpass levels I_{BP_1} thru I_{BP_4}. Figures 36(a)-(d) show each image with pixel sizes magnified for ease of viewing. Zero-crossings may be found very efficiently using the sign bit of each pixel value in the bandpass representation.

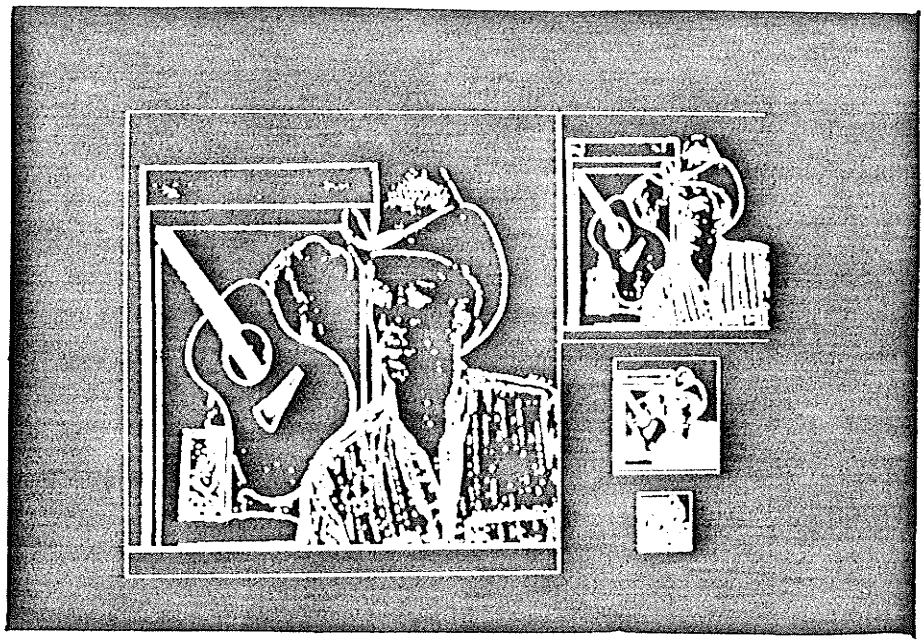
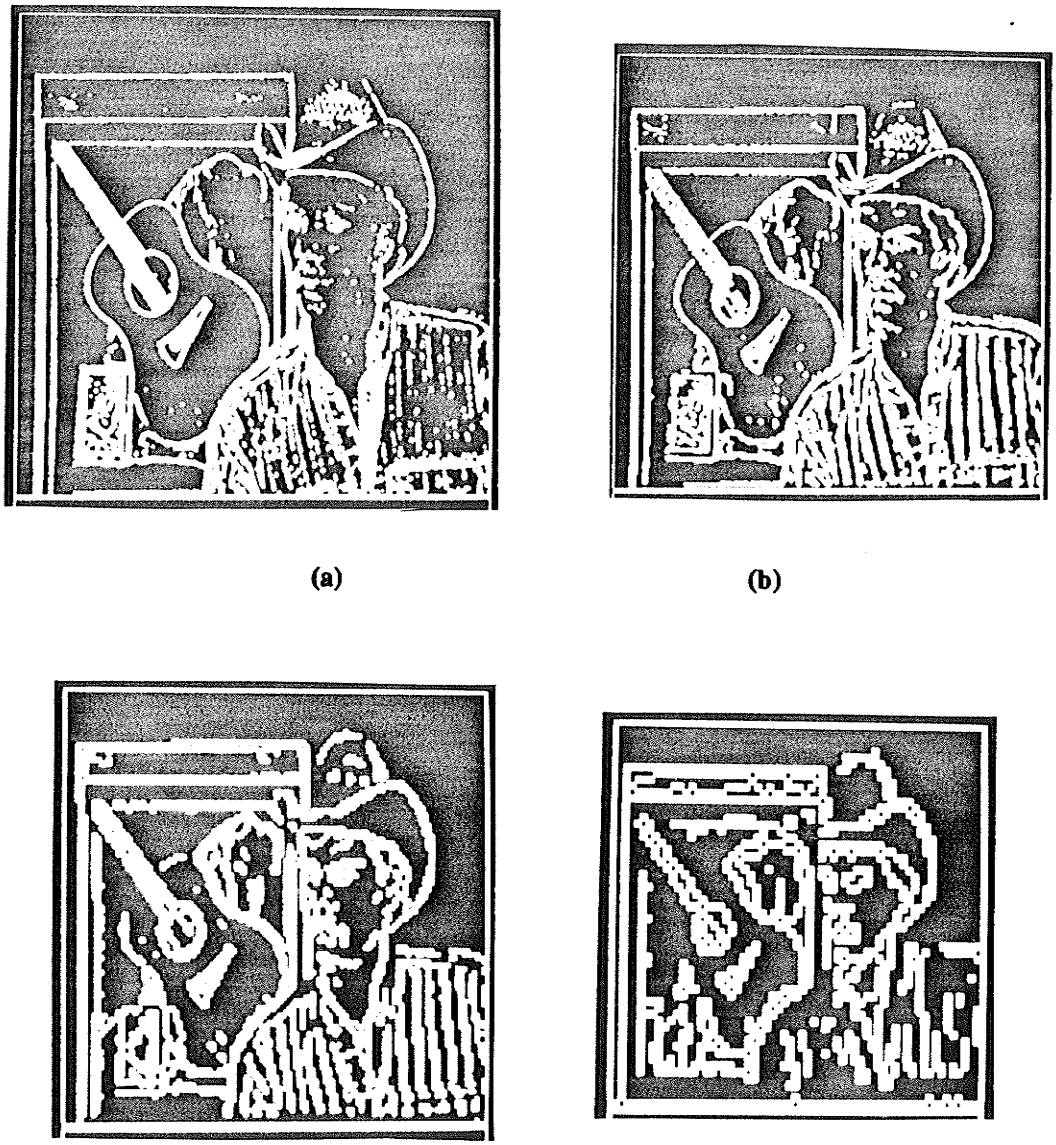


Figure 35: Multiresolution zero-crossing representation.



(a) (b)

(c) (d)

Figure 36: Multiresolution zero-crossing representation with pixel sizes magnified.

CHAPTER 7

Conclusions

Many image processing and analysis tasks can be accomplished efficiently by using multiresolution representations of an image. A full-custom ASIC enables the design of a system capable of transforming the information content of an image into a multiresolution representation in real time.

A number of techniques have been used to reduce the computational complexity and the cost of the ASIC design. Separability of the Gaussian function reduced the number of computational elements used by the systolic array from M^2 to $2M$ and reduced the number of multiplications and additions for each image frame from $O(M^2N^2)$ to $O(MN^2)$ where N^2 is the number of pixels per image frame. Hierarchical convolution eliminated the use of large bit representations of partial results by reducing large convolutions to smaller convolutions in a divide-and-conquer scheme. Thus, only a small fixed-size convolution is necessary, enabling the repetitive use of just one ASIC in cascaded fashion to produce a complete multiresolution representation system. The symmetry of the Gaussian function and the fixed nature of the filtering coefficients further allow the separated convolutions to be implemented with a small number of additions and with no multiplications.

Other techniques have also been adopted to increase the performance of the layout. Exploitation of the *flow-through* nature of the algorithm allows a second dimension of pipelining to be incorporated in the design, thus increasing the throughput of the system. The full custom design of the ASIC allowed for optimization of all

critical paths and of the basic cells from which the system was built, particularly within the adder banks. Each cell was designed to allow the overall layout to be compact, to require only local data-bus wiring, and to allow appropriate power- and clock-bus wiring. The resultant ASIC was therefore able to achieve real-time operation under test conditions.

References

[1] M.D. Kelly, 'Edge Detection in Computers by Using Planning', in *Machine Intelligence,* B. Meltzer and D. Mitchie, eds. 1971.

[2] D.B. Marr and E. Hildreth, 'Theory of Edge Detection', *Proc. R. Soc. London,* B 207, pp. 187-217, 1980.

[3] A.R. Hanson and E.M. Riseman, 'Visions: A Computer System for Interpreting Scenes', in *Computer Vision Systems,* A.R. Hanson and E.M. Riseman, eds., New York: Academic, 1985.

[4] D.B. Marr and T. Poggio, 'A Computational Theory of Human Stereo Vision', *Proc. R. Soc. London,* B 204, pp. 301-328, 1979.

[5] H. Moravec, 'Obstacle Avoidance and Navigation in the Real World by a Seeing Robot Rover', Ph. D. dissertation, Stanford Univ., Sept. 1980.

[6] J.L. Crowley, 'Multiple Resolution Representation and Probabilistic Matching of 2-D Gray-Scale Shape', *IEEE Trans. on Patt. Ana. and Mach. Int.,* Vol. PAMI-9, No. 1, pp. 113-121, Jan. 1987.

[7] P.J. Burt and E. Adelson, 'The Laplacian Pyramid as a Compact Image Code', *IEEE Trans. of Comm.,* Vol. COMM-31, No. 4, pp. 532-540, Apr. 1983.

[8] A. Rosenfeld, *Multiresolution Image Processing and Analysis,* Springer-Verlag, New York, 1984.

[9] V. Torre and T.A. Poggio, 'On Edge Detection', *IEEE Trans. on Patt. Ana. and Mach. Int.,* Vol. PAMI-8, No. 2, pp. 147-163, March 1986.

[10] C.D. Thompson, 'Area-Time Complexity for VLSI', *Proc. 11th Annual ACM Symp. Theory of Computing,* ACM Signact, pp. 81-88, 1975.

[11] J.W. Hong and H.T. Kung, 'I/O Complexity: The Red-Blue Pebble Game', *Proc. 13th Annual ACM Symp. Theory of Computing,* ACM Signact, pp. 326-333, May 1981.

[12] H.T. Kung and C.E. Leiserson, 'Algorithms for VLSI Processor Arrays', in C.A. Mead and L. Conway, *Introduction to VLSI Systems,* Addison Wesley, pp. 271-292, 1980.

[13] J.J. Clark and P.D. Lawrence, 'A Hierarchical Image Analysis System Based Upon Oriented Zero Crossings of Bandpassed Images', in *Multiresolution Image Processing and Analysis,* A. Rosenfeld, ed., Springer-Verlag, New York, pp. 148-168, 1984.

[14] P.J. Burt, 'Fast Filter Transforms for Image Processing', *Computer Graphics Image Processing,* Vol. 16, pp. 20-51, 1981.

[15] H.C. Card, P.D. Hortensius, and R.D. McLeod, *Standard Cells for Custom VLSI Chip Design,* Dept. of Elec. Eng., Technical Report TR87-1, Univ. of Man., 1987.

Appendix A: Program Listings

```
C*************************************************************
C*  THIS PROGRAM PRODUCES A LOWPASS, A BANDPASS, AND THE     *
C*  BANDPASS ZEROCROSSING MULTIRESOLUTION REPRESENTATION     *
C*  OF A 512 BY 512 IMAGE.  FOUR RESOLUTION CHANNELS ARE     *
C*  GENERATED IN EACH CASE.                                  *
C*************************************************************
C
      integer*2 img(512,512), fimg(512,512),n,i,j,val,a,b,c
      character imag(512,512)
C
C     read in image
C
      open(10,file='cowboy',form='unformatted')
      open(11,file='bp1',form='unformatted')
      open(12,file='bp2',form='unformatted')
      open(13,file='bp3',form='unformatted')
      open(14,file='bp4',form='unformatted')
      open(15,file='lp1',form='unformatted')
      open(16,file='lp2',form='unformatted')
      open(17,file='lp3',form='unformatted')
      open(18,file='lp4',form='unformatted')
      open(19,file='zc1',form='unformatted')
      open(20,file='zc2',form='unformatted')
      open(21,file='zc3',form='unformatted')
      open(22,file='zc4',form='unformatted')
C
      read(10) n,n
      do 25 i=1,512
         read(10)(imag(i,j),j=1,512)
   25 continue
C
C     decode the ascii image to integer format
C
      do 26 i=1,512
      do 26 j=1,512
         img(i,j)=ichar2(imag(i,j)) + 1
   26 continue

      n=512
      call gauss(img,fimg,n)
      call stoimg(img,imag,n)
      write(15) n,n
      do 27 i=1,n
         write(15)(imag(i,j),j=1,n)
   27 continue
      call stoimg(fimg,imag,n)
      write(11) n,n
      do 28 i=1,n
         write(11)(imag(i,j),j=1,n)
   28 continue
      call zcross(fimg,n)
      call stoimg(fimg,imag,n)
      write(19) n,n
      do 29 i=1,n
         write(19)(imag(i,j),j=1,n)
   29 continue
      n=256
      call subsample(img,n)
      call gauss(img,fimg,n)
      call stoimg(img,imag,n)
      write(16) n,n
      do 30 i=1,n
         write(16)(imag(i,j),j=1,n)
   30 continue
      call stoimg(fimg,imag,n)
      write(12) n,n
      do 31 i=1,n
         write(12)(imag(i,j),j=1,n)
   31 continue
      call zcross(fimg,n)
      call stoimg(fimg,imag,n)
      write(20) n,n
      do 32 i=1,n
         write(20)(imag(i,j),j=1,n)
   32 continue
      n=128
      call subsample(img,n)
      call gauss(img,fimg,n)
      call stoimg(img,imag,n)
      write(17) n,n
      do 33 i=1,n
```

```
         write(17)(imag(i,j),j=1,n)
   33 continue
      call stoimg(fimg,imag,n)
      write(13) n,n
      do 34 i=1,n
         write(13)(imag(i,j),j=1,n)
   34 continue
      call zcross(fimg,n)
      call stoimg(fimg,imag,n)
      write(21) n,n
      do 35 i=1,n
         write(21)(imag(i,j),j=1,n)
   35 continue
      n=64
      call subsample(img,n)
      call gauss(img,fimg,n)
      call stoimg(img,imag,n)
      write(18) n,n
      do 36 i=1,n
         write(18)(imag(i,j),j=1,n)
   36 continue
      call stoimg(fimg,imag,n)
      write(14) n,n
      do 37 i=1,n
         write(14)(imag(i,j),j=1,n)
   37 continue
      call zcross(fimg,n)
      call stoimg(fimg,imag,n)
      write(22) n,n
      do 38 i=1,n
         write(22)(imag(i,j),j=1,n)
   38 continue
      stop
      end
C
C*********************************************************************
C* this subroutine subsamples the image producing an n/2 by n/2 image ***
C*********************************************************************
C
      subroutine subsample(img,n)
C
      integer*2 img(512,512),i,j,n
C
      do 12 i=1,n
      do 12 j=1,n
         img(i,j)=img(i*2,j*2)
   12 continue
      return
      end
C
C*********************************************************************
C* this subroutine converts the image into ascii format *******
C*********************************************************************
C
      subroutine stoimg(mg,imag,n)
C
      integer*2 mg(512,512),n
      character imag(512,512)
C
      do 13 i=1,n
      do 13 j=1,n
         imag(i,j)=char(mg(i,j) - 1)
   13 continue
      return
      end
C
C*********************************************************************
C* this subroutine performs the two-dimensional convolution **
C*********************************************************************
C
      subroutine gauss(img,fimg,n)
C
      integer*2 img(512,512),fimg(512,512),x(512,512)
      integer*2 i,j,a,n
C
      do 10 i=1,n
         x(i,1)=0.7*img(i,1)+0.25*img(i,2)+0.05*img(i,3)
         x(i,2)=0.3*img(i,1)+0.4*img(i,2)+0.25*img(i,3)+0.05*img(i,4)
         x(i,n-1)=0.05*img(i,n-3)+0.25*img(i,n-2)+0.4*img(i,n-1)
     +   +0.3*img(i,n)
```

```
         x(i,n)=0.05*img(i,n-2)+0.25*img(i,n-1)+0.7*img(i,n)
       do 10 j=3,n-2
          x(i,j)=0.4*img(i,j)+0.25*(img(i,j-1)+img(i,j+1))
     +    +0.05*(img(i,j-2)+img(i,j+2))
    10 continue
C
       do 11 j=1,n
          a=img(1,j)
          img(1,j)=0.7*x(1,j)+0.25*x(2,j)+0.05*x(3,j)
          fimg(1,j)=a-img(1,j)+129
          a=img(2,j)
          img(2,j)=0.3*x(1,j)+0.4*x(2,j)+0.25*x(3,j)+0.05*x(4,j)
          fimg(2,j)=a-img(2,j)+129
          a=img(n-1,j)
          img(n-1,j)=0.05*x(n-3,j)+0.25*x(n-2,j)+0.4*x(n-1,j)
     +    +0.3*x(n,j)
          fimg(n-1,j)=a-img(n-1,j)+129
          a=img(n,j)
          img(n,j)=0.05*x(n-2,j)+0.25*x(n-1,j)+0.7*x(n,j)
          fimg(n,j)=a-img(n,j)+129
          do 51 i=3,n-2
             a=img(i,j)
             img(i,j)=0.4*x(i,j)+0.25*(x(i-1,j)+x(i+1,j))
     +    +0.05*(x(i-2,j)+x(i+2,j))
          fimg(i,j)=a-img(i,j)+129
    51 continue
       do 11 i=1,n
          if (fimg(i,j).gt.256) then
             fimg(i,j)=256
          end if
    11 continue
       return
       end
C
C*************************************************************************
C* this subroutine detects zero crossings of the bandpass image         *
C*************************************************************************
       subroutine zcross(fimg,n)
C
       integer*2 fimg(512,512),n,i,j,g,y(3,512),thres,neg
       logical*2 val,dow,up,lft,rgt,tlc,trc,blc,brc
C
       do 41 i=1,2
       do 41 j=1,n
          y(i+1,j)=fimg(i,j)
    41 continue
       do 40 i=2,n-1
          do 42 g=1,n
            y(1,g)=y(2,g)
            y(2,g)=y(3,g)
            y(3,g)=fimg(i+1,g)
    42    continue
       do 40 j=2,n-1
          if(n .eq. 512) then
             thres=(128)
          else
             thres=129
          end if
          val=(y(2,j) .ge. thres)
          up=(y(1,j) .ge. thres)
          dow=(y(3,j) .ge. thres)
          rgt=(y(2,j+1) .ge. thres)
          lft=(y(2,j-1) .ge. thres)
          tlc=(y(1,j-1) .ge. thres)
          trc=(y(1,j+1) .ge. thres)
          blc=(y(3,j-1) .ge. thres)
          brc=(y(3,j+1) .ge. thres)
          neg=0
          if(up) neg=neg+1
          if(dow) neg=neg+1
          if(rgt) neg=neg+1
          if(lft) neg=neg+1
          if(tlc) neg=neg+1
          if(trc) neg=neg+1
          if(blc) neg=neg+1
          if(brc) neg=neg+1
          if((val).or.(up.and.dow.and.lft.and.rgt.and.tlc.and.trc
     +      .and.blc.and.brc).or.((.not.up).and.(.not.dow).and.
     +      (.not.lft).and.(.not.rgt)).or.(neg.eq.7)) then
            fimg(i,j)=1
          else
```

```
            fimg(i,j)=256
          end if
   40 continue
      return
      end
```

```
C*************************************************************
C* THIS PROGRAM RECONSTRUCTS A 512 BY 512 IMAGE FROM ITS     *
C* FIRST FOUR BANDPASS LEVEL REPRESENTATIONS AND ITS FOURTH   *
C* LEVEL LOWPASS REPRESENTATION.                             *
C*************************************************************
C
      integer*2 img(512,512),img2(512,512),img3(512,512),n,i,j,x,y,val
      character imag(512,512)
C
C     read in image
C
      open(10,file='bp1',form='unformatted')
      open(11,file='bp2',form='unformatted')
      open(12,file='bp3',form='unformatted')
      open(13,file='bp4',form='unformatted')
      open(14,file='lp4',form='unformatted')
      open(15,file='rebuilt',form='unformatted')
C
      read(13) n,n
      do 20 i=1,n
         read(13)(imag(i,j),j=1,n)
   20 continue
      call decod(imag,img,n)
      read(14) n,n
      do 21 i=1,n
         read(14)(imag(i,j),j=1,n)
   21 continue
      call decod(imag,img2,n)
      do 22 i=1,n
      do 22 j=1,n
         img2(i,j)=img(i,j)+img2(i,j)-129
   22 continue
      call expand(img2,img3,n)
      read(12) n,n
      do 23 i=1,n
        read(12)(imag(i,j),j=1,n)
   23 continue
      call decod(imag,img,n)
      call comb(img,img2,img3,n)
      call expand(img2,img3,n)
      read(11) n,n
      do 24 i=1,n
         read(11)(imag(i,j),j=1,n)
   24 continue
      call decod(imag,img,n)
      call comb(img,img2,img3,n)
      call expand(img2,img3,n)
      read(10) n,n
      do 25 i=1,n
         read(10)(imag(i,j),j=1,n)
   25 continue
      call decod(imag,img,n)
      call comb(img,img2,img3,n)
      call stoimg(img2,imag,n)
      write(15) n,n
      do 26 i=1,n
         write(15)(imag(i,j),j=1,n)
   26 continue
      stop
      end

C
C*************************************************************************
C* this subroutine decodes the ascii image into integer format *******
C*************************************************************************
C
      subroutine decod(imag,mg,n)
C
      integer*2  mg(512,512),n,i,j,val
      character  imag(512,512)
C
      do 30 i=1,n
      do 30 j=1,n
         mg(i,j)=1 + ichar2(imag(i,j))
   30 continue
      return
      end
C
C*************************************************************************
C* this subroutine converts the image into ascii format *******
C*************************************************************************
```

```
C
      subroutine stoimg(mg,imag,n)
C
      integer*2 mg(512,512),n
      character imag(512,512)
C
      do 40 i=1,n
      do 40 j=1,n
         imag(i,j)=char(mg(i,j) - 1)
   40 continue
      return
      end
C
C*****************************************************************
C* this subroutine combines the lowpass image with the ********
C* bandpass image forming the next lowpass level image ********
C*****************************************************************
C
      subroutine comb(img,img2,img3,n)
C
      integer*2 img(512,512),img2(512,512),img3(512,512),n,i,j,x,y,m
C
      do 50 i=1,n
      do 50 j=1,n
         img2(i,j)=img(i,j) - 129 + img3(i,j)
         if(img2(i,j) .lt. 1) img2(i,j)=1
   50 continue
      return
      end
C
C*****************************************************************
C* this subroutine expands an n x n image into a 2n x 2n image ******
C*****************************************************************
C
      subroutine expand(img2,img3,n)
C
      integer*2 img2(512,512),img3(512,512),n,i,j,o,m,x,y
C
      do 60 i=1,n
      do 60 j=1,n
         img3(i*2,j*2)=img2(i,j)
   60 continue
      o=n*2
      m=2
      img3(1,1)=img3(2,2)
      do 70 i=2,o
         x=mod2(i,m)
         if(x .EQ. 0) then
            img3(i,1)=img3(i,2)
         else
            img3(i,1)=0.5*(img3(i-1,2)+img3(i+1,2))
         end if
   70 continue
      do 80 j=2,o
         y=mod2(j,m)
         if(y .EQ. 0) then
            img3(1,j)=img3(2,j)
         else
            img3(1,j)=0.5*(img3(2,j-1)+img3(2,j+1))
         end if
   80 continue
      do 90 i=2,o
        x=mod2(i,m)
        do 90 j=2,o
          y=mod2(j,m)
          if((x .eq. 1) .and. (y .eq. 0)) then
            img3(i,j)=0.5*(img3(i-1,j)+img3(i+1,j))
          else if((x .eq. 0) .and. (y .eq. 1)) then
            img3(i,j)=0.5*(img3(i,j-1)+img3(i,j+1))
          else if((x .eq. 1) .and. (y .eq. 1)) then
            img3(i,j)=0.25*(img3(i-1,j-1)+img3(i+1,j-1)
     +      +img3(i-1,j+1)+img3(i+1,j+1))
          end if
   90 continue
      return
      end
```