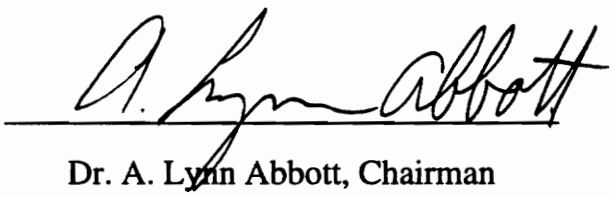
Median and morphological filtering of images in real time using an FPGA-based custom computing platform

by

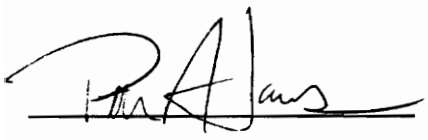
Adit Tarmaster

Thesis Submitted to the Faculty of the
Virginia Polytechnic Institute and State University
in partial fulfillment of the requirement for the degree of
Master of Science
in
Electrical Engineering

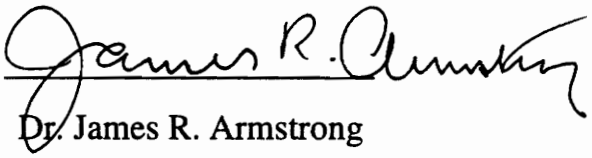
APPROVED:



Dr. A. Lynn Abbott, Chairman



Dr. Peter M. Athanas



Dr. James R. Armstrong

April 1994

Blacksburg, Virginia

Median and morphological filtering of images in real time using an FPGA-based custom computing platform

by

Adit Tarmaster

Dr. A. Lynn Abbott, Chairman

Electrical Engineering

(ABSTRACT)

This thesis describes the design and implementation of real-time image-processing tasks on a custom computing platform called Splash-2. The tasks that have been implemented are image histogram generation, median filtering using 3×3 neighborhoods, and morphological dilation and erosion. These problems are computationally intensive, involving large amounts of data. The problems are especially difficult when the images need to be processed at real-time rates, typically 30 frames per second. Splash-2 is a reconfigurable FPGA-based attached processor featuring several programmable processing elements and programmable communication paths. Although not designed specifically for image-processing applications, it possesses architectural properties that make it well suited for high speed computations and data transfer rates that are characteristic of this class of problems. This thesis discusses the design process which has been used to map the tasks to Splash-2, and presents results which demonstrate the effectiveness of custom computing platforms for high performance image processing.

ACKNOWLEDGEMENTS

I would like to thank Dr. A. Lynn Abbott, who supported this research. He has been of great help in his capacity as graduate advisor. I would also like to express my gratitude to my graduate co-advisor, Dr. Peter M. Athanas, without whose help this work would not have been accomplished. Also, special thanks to Dr. James R. Armstrong for being on my graduate committee.

I appreciate the backing and assistance of all the "VTSplashers" who have been with me through thick and thin. Special thanks to Ramana, Brad, Nabeel, Jeff, Luna, Rob, Al and Jim.

Finally, I would like to thank my parents without whose support I wouldn't be here.

Table of Contents

List of Illustrations

Chapter 1. Introduction

1.1 Motivation

Most image processing tasks are computationally intensive. Even tasks that are conceptually simple, such as template matching and edge detection, require operations to be performed repeatedly at every location in an image. Because of the large number of picture elements (pixels) associated with a single image, many computations are needed.

Image processing is especially challenging when processing must be performed at real-time rates, typically 30 frames/second. Traditionally, real-time processing has required expensive application-specific hardware. Custom circuits are often designed to accomplish specific tasks optimally, but this approach sacrifices generality.

This thesis describes the use of *Splash-2* [1] for real-time image processing. Splash-2 is a custom computing platform developed at the Supercomputing Research Center in Bowie, Maryland. The fundamental components of the system are an array of field programmable gate arrays [2] which can be configured to perform a wide variety of tasks. For some problems, Splash-2 performs as well as application-specific circuits.

The goal of this research has been to demonstrate that Splash-2 can be programmed to perform several different image processing tasks at real-time rates. Splash-2 requires an unconventional design approach which is described in this thesis. By

implementing real-time processing of image data, the potential of adaptive computing platforms has been demonstrated.

1.2 Contributions of this research

Four different image-processing tasks have been implemented on Splash-2 during the course of this research. The specific tasks that have been successfully demonstrated are image histogram generation, median filtering using 3×3 neighborhoods, and gray-scale morphological dilation and erosion. The algorithms to perform these tasks, the problem partition to map these algorithms to the Splash-2, and the VHDL descriptions for the individual components of the Splash-2 array have been developed.

Median filtering and the morphological operations are of particular interest because they are nonlinear operations, presenting unique challenges when compared with more common convolutional filtering. A novel algorithm is developed to obtain 3×3 image neighborhoods on the Splash-2. The problem partition to map this algorithm to the Splash-2 architecture is designed and implemented.

The tasks are performed in real time with data coming in from a camera at a frame rate of 30 per second. The system therefore processes "live" image data. The results are displayed on a monitor. Data processed at slower rates may be directed to a file on the Sun SPARC-2 workstation which serves as the host to the Splash-2.

1.3 Organization of thesis

The thesis is organized as follows. Chapter 2 presents the relevant concepts of image processing. The real-time image processing tasks that have been implemented on

Splash-2 are described here. Chapter 3 describes architectures that were previously designed for the purpose of high performance image processing. The Splash-2 architecture is described in Chapter 4. The hardware details of the real-time vision system developed at Virginia Tech are provided here. This includes the Splash-2 processor which is used with a Sun SPARC-2 host. Chapter 5 describes the design process for implementing applications on Splash-2. This chapter covers the software environment available to the applications programmer, the VHDL design process, the simulation and synthesis tools used.

Chapter 6 gives a detailed description of the design of the image histogram generator. Problem partitioning, data flow and interprocessor communication, and logic circuits for the processing elements are provided in this chapter. Similar design details for the median filter and morphological filter implementations are provided in Chapter 7 and Chapter 8, respectively.

Chapter 9 serves as a conclusion to the thesis. It comments on the performance of the system and highlights the limitations of the design methodology.

Appendix A gives the entity declarations for the Splash-2 processing elements. Appendix B gives a port description of the Splash-2 entities which are relevant to this work. Finally, Appendix C provides sample VHDL files for the components of the Splash-2 system which have been modified to implement the particular image-processing tasks.

Chapter 2. Image processing - background and concepts

2.1 Introduction

The fundamental unit in image processing is the picture element, or *pixel*. An image is typically composed of a two-dimensional array of pixels. For *grayscale* images, each pixel has one of K values (usually 2^N), which are strictly ordered. These values may be thought of as the integers 0 to K-1. The image array may be represented by $I(r, c)$, where r and c are the row and column location of the pixel, respectively. This is shown in Figure 1. A camera generates pixels in *raster* order, in which the pixels in the image are scanned from left to right and from top to bottom in a "snake-like" fashion.

Often, in the case of 2D grayscale images, the pixel values represent a third dimension. The image is then referred to as a 3D image with two *spatial* dimensions and one *value* dimension which holds grayscale data. This thesis is concerned only with grayscale images, with a typical size of 512 rows $\times$ 512 columns, and with each pixel represented by an 8-bit grayscale value.

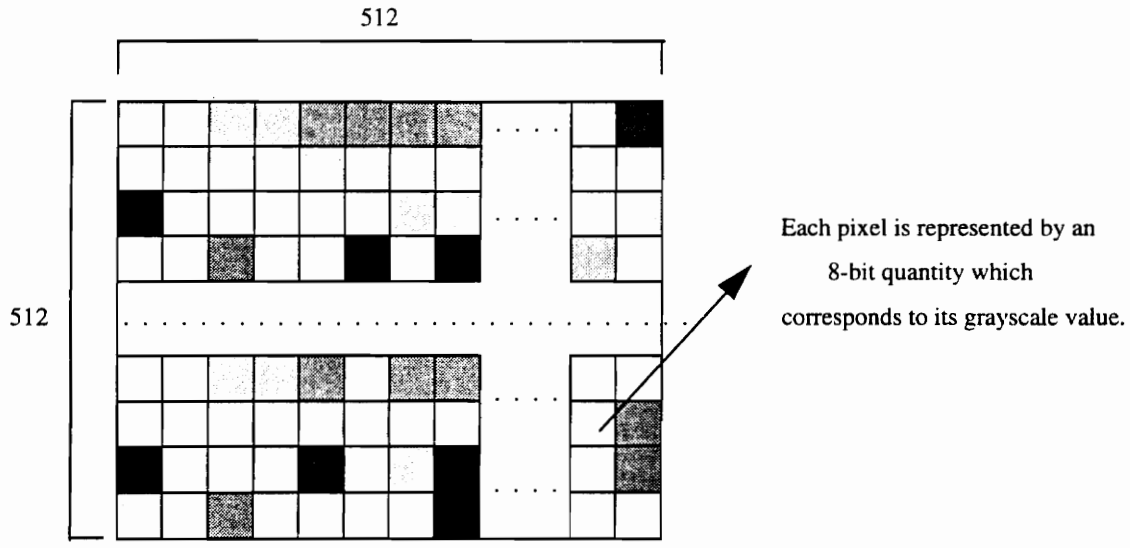


Figure 1. Example of a grayscale image. A typical size is 512 rows × 512 columns. The top-left corner pixel is at location (0, 0), while the bottom-right corner pixel is at location (511, 511). The gray value of each pixel within the image is represented by an 8-bit quantity.

Image operations may be broadly classified into several generic classes [3]. An operation in the *combination* class takes two images of a given type and produces a new image of the same type. This is accomplished by combining each pair of elements from the input images into a new element. The *generation* class are those which create a new image of a given type from scratch. The *transformation* class takes one input image of a given type and transforms it into another image of the same type. The transformations may be distinguished as *point* and *neighborhood* transformations. In case of point transformations, the transformation depends only on the value of an individual pixel. In case of neighborhood transformations, the transformation depends on a set of neighbors of each pixel (usually but not necessarily nearest neighbors). The *measurement* class takes in an image of a given type and reduces it to a number or distribution representing a

measurement on the image. The *conversion* class includes those operations which take an image of a given type and convert them to a different type.

In this research, we demonstrate image processing operations that may be categorized into the classes just described. The image histogram generation may be classified as a measurement operation as it measures the intensity distribution of the image. The median and morphological filters may be classified as neighborhood transformations. This work therefore demonstrates the effectiveness of Splash-2 for two classes of image processing tasks.

2.2 Histogram of an image

The histogram of a grayscale image contains intensity distribution statistics about the image. The histogram provides the number of pixels that are associated with any gray value in the image [4]. Histogram data is often used for image processing tasks such as *thresholding*. Thresholding distinguishes pixels that have higher gray values from pixels that have lower gray values and the histogram data helps in deciding the level at which this distinction can be made. Thresholding is useful in applications such as region detection, region labeling, and conversion of grayscale images to binary images.

The histogram h of a digital image I is defined by $h(m) = \#\ \{(r, c) \mid I(r, c) = m\}$, where m represents each gray level value and $\#$ is the operator that counts the number of elements in a set [4].

The algorithm for computing a histogram is shown in Figure 2. H is a vector array dimensioned from 0 to MAX, where 0 is the value of the smallest possible gray level and MAX is the value of the largest (255 in this thesis). I is a two-dimensional array, dimensioned 0 to (ROWSIZE-1) by 0 to (COLSIZE-1) that holds a gray level image.

```
procedure Histogram(I, H);
for i = 0 to MAX do                    // Initialize the histogram to zero.
   H(i) = 0;

for r = 0 to ROWSIZE-1 do              // Compute values by accumulation.
   for c = 0 to COLSIZE-1 do
      begin
         grayval = I(r, c);
         H(grayval) = H(grayval) + 1;
      end
   end for
end for

end Histogram
```

Figure 2. Algorithm for computing an image histogram. $H(i)$ is an array which stores the number of pixels at each gray level i.

An example histogram for a small 8 row × 8 column image is shown in Figure 3. The histogram is represented both in tabular and bar-graph forms. There are only 16 possible gray-levels for the sample image shown here. Therefore, the histogram array has dimensionality 16. Realistic images typically have 256 grayscale values.

2.3 Median Filtering

The median filter is used widely for the reduction of noise and periodic interference patterns in images [5]. It is useful for removing sporadic noise without blurring the edges and yet retaining the monotone changes.

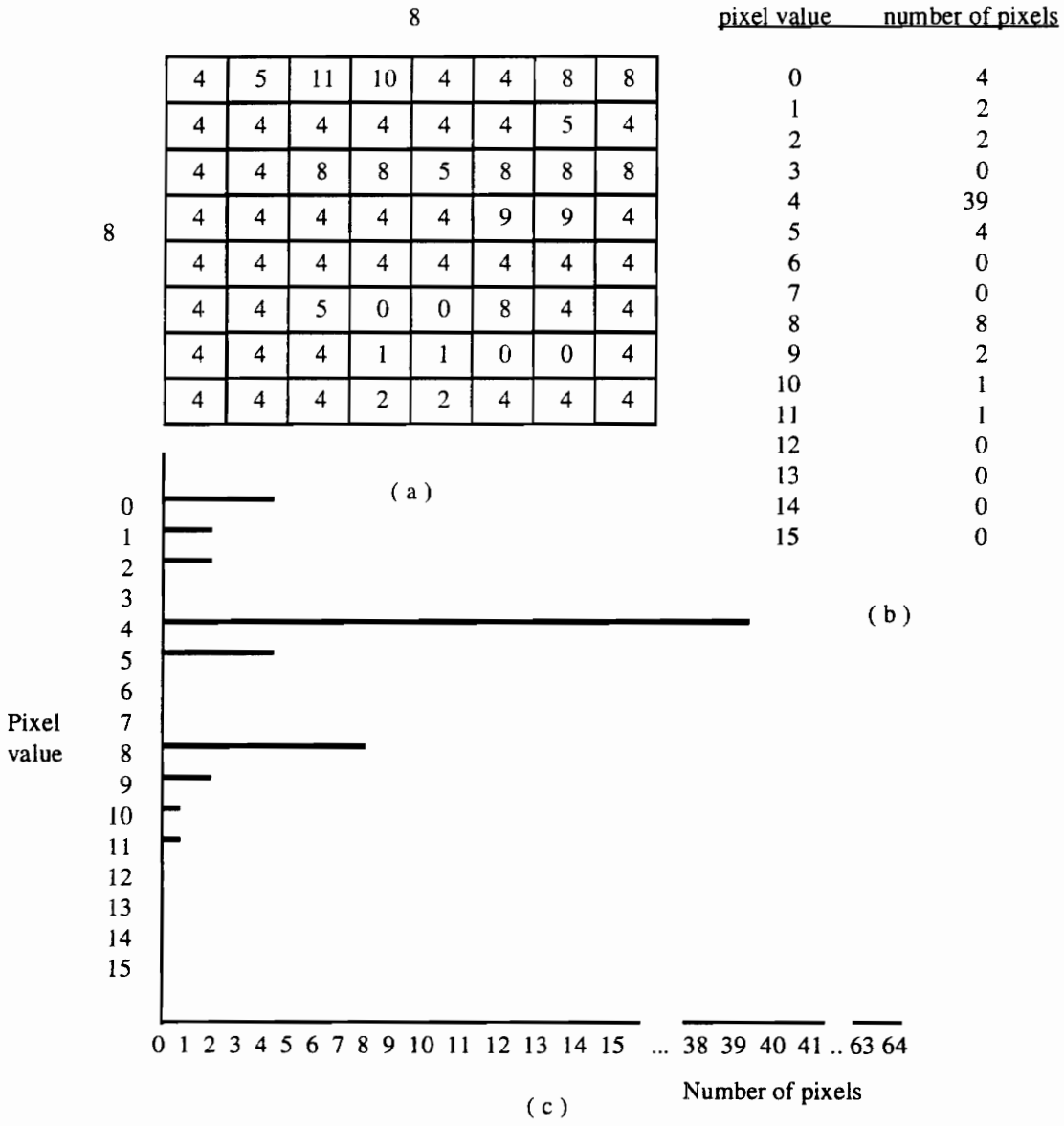


Figure 3. Example of an image histogram. (a) An 8×8 grayscale image. Each pixel has one of sixteen possible gray values. (b) The histogram of this small image is given in tabular form. (c) The histogram is displayed as a bar graph. The length of the bar is an indication of the number of pixels in the image at a particular gray value. In the above example, there are 39 pixels with a gray value of 4.

Median filtering is often used for noise cleaning [3, 4, 5, 6]. It uses neighborhood spatial coherence and neighborhood pixel value homogeneity as its basis. Noise-cleaning techniques detect lack of coherence and replace the incoherent pixel by something more spatially coherent. This is done by using some or all of the pixels in a neighborhood containing the given pixel or by averaging or smoothing the pixel value with others in an appropriate neighborhood.

The median filtering operation may be stated mathematically in the following manner. Let $f_0, f_1, \ldots, f_{N-1}$ represent the intensity values for input image I within an N-point neighborhood about the point (i, j) in the image. These values are ordered so that

$$f_K \leq f_{K+1}.$$

The output image R is determined as

$$R(i, j) = f_{(N-1)/2} \qquad \textit{(for odd } N\textit{)}$$

$$R(i, j) = \frac{1}{2}\left(f_{(N/2)-1} + f_{(N/2)}\right) \qquad \textit{(for even } N\textit{)}$$

In most image-processing applications, rectangular neighborhoods are assumed.

Figure 4 illustrates the concept of a 3×3 neighborhood operation. The shaded 3×3 window is assumed to "slide" over I producing an output value for R at each location of the window. For median filtering, the value of the pixel at any location in R is the median of the nine values in the 3×3 window with center at that position in I. Two window positions are shown in the figure, with corresponding positions highlighted in R. For an input image of size 512×512, approximately 262,144 nine-point median values need to be extracted to produce R.

The median filter does a good job of estimating the true pixel values in situations where the underlying neighborhood trend is flat or monotonic and the noise distribution has flat tails. It is effective for removing impulsive noise. However, when the

neighborhood contains fine detail such as thin lines, they are distorted or lost. Corners can be clipped. It can produce regions of constant or nearly constant values that are perceived as patches, streaks, or amorphous blotches. Such artifacts may suggest boundaries that really do not exist.

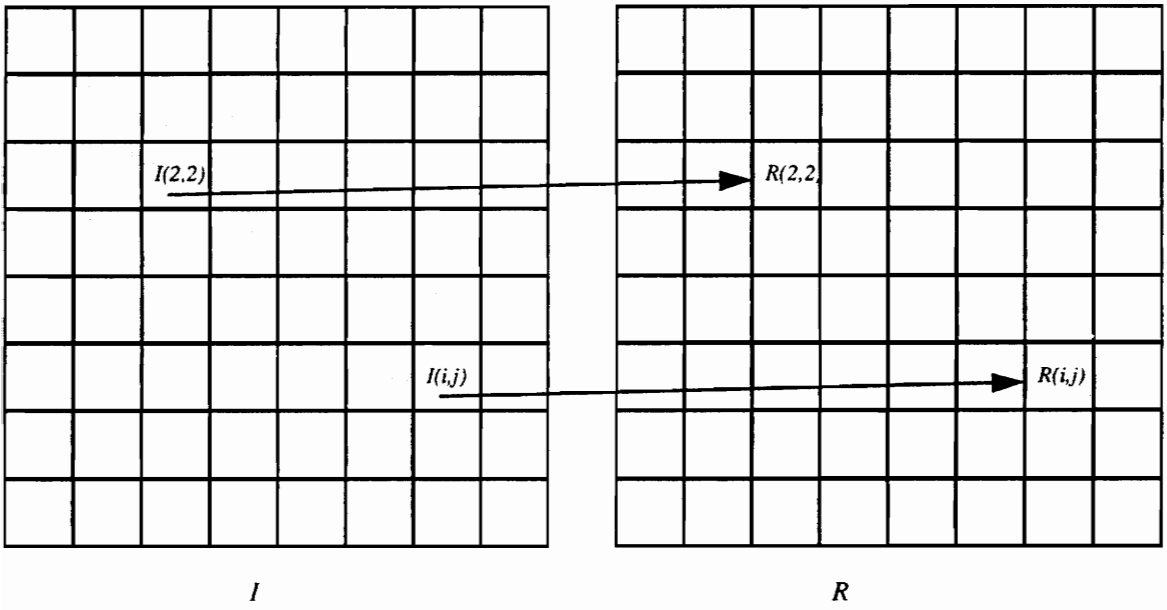


Figure 4. Concept of a 3×3 window-based operation. For the median filter, the value of $R(i, j)$ is the median of the 9 pixels of I which lie within the 3×3 window with center at $I(i, j)$.

2.4 Morphological filtering

2.4.1 Introduction

Mathematical morphology [7] is a set-theoretic approach to image processing and analysis which considers images to be sets in the underlying support space. It manipulates

them using set-based operators like union and intersection. Though originally developed for binary imagery, it has since been extended to gray level and other types of images [8]. This section is concerned with two grayscale morphological operations, known as erosion and dilation, which have been implemented on Splash-2.

The word *morphology* refers to the study of form and structure. The morphological approach is generally based upon the analysis of an image in terms of some predetermined geometric shape known as a *structuring element*. Essentially, the manner in which the structuring element "fits" in the image is studied. The morphological approach to image analysis has been constructed to answer questions about shape in a rigorous mathematical way. These operations provide for the systematic alteration of the geometric content of an image while maintaining the stability of important geometric characteristics. A well-developed morphological algebra exists.

Morphological operations can be employed for many purposes, including edge detection, segmentation, and enhancement of images. An entire class of morphological filters exist that are employed in place of standard linear filters. Whereas linear filters sometimes distort the underlying geometric form of an image, morphological filters leave much of that form intact.

Figure 5 shows the gray level set combination operations used in morphology. The union operator is interpreted as the maximum and is used to combine different types of high intensity structures in the gray image topography. It can be used to select the pixel value from all of several input images which satisfy some criterion to the greatest extent, again as expressed by its gray value. The gray level intersection or minimum is generally used to put together low intensity structures which have been selected independently, or to select each pixel value from whichever image satisfies some criterion to the least extent.

Morphological operators such as dilation and erosion are *duals* of each other. This means that we can get the equivalent of dilation by performing its dual operation, erosion, on the complement image, and taking the complement of the result. Stated differently, when we perform the first operation on the foreground, we are at the same time performing the second (dual) operation on the background, and vice versa.

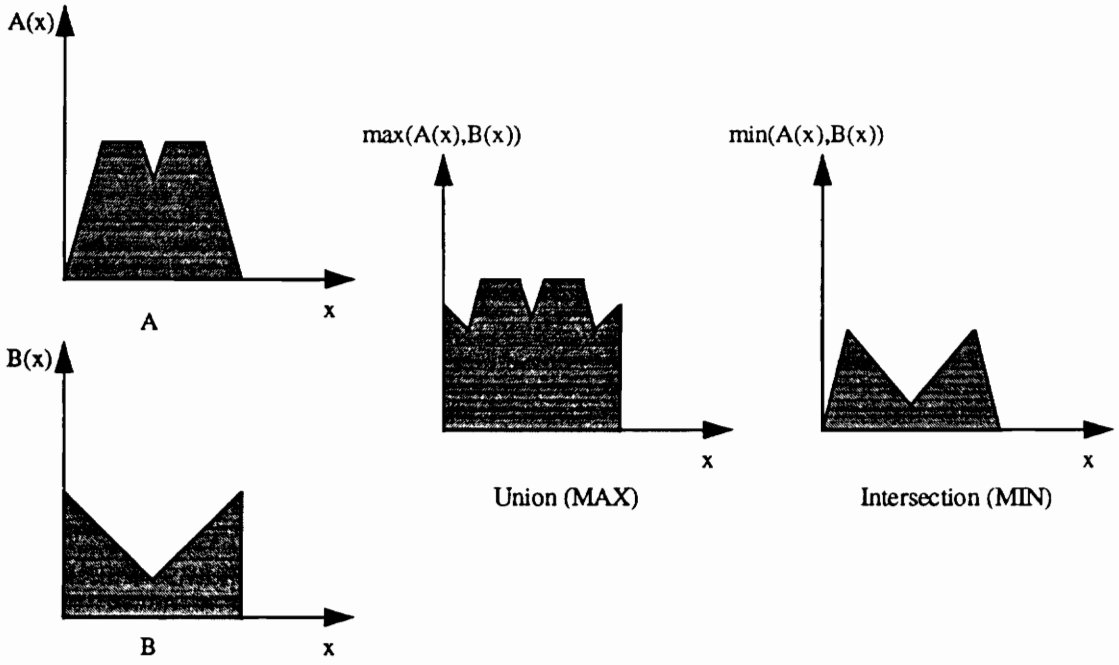


Figure 5. Gray level set combination operations. The union (max) and intersection (min) are shown for the images A and B. A and B represent one-dimensional images in the continuous domain. The vertical axis represents the grayscale value of each pixel in the image.

2.4.2 Dilation

Dilation is a morphological transformation that combines two sets by using vector addition of set elements. The binary dilation of F by K is denoted by $F \oplus K$ and is defined by

$$F \oplus K = \{x \in E^N \mid x = f + k \text{ for some } f \in F \text{ and } k \in K\}$$

where F and K are sets in E^N, denoting the Euclidean N-space.

Typically, F represents an image and K represents a structuring element, which is a shape that acts on F.

For computational purposes, grayscale dilation can be accomplished by taking the maximum of a set of sums. Instead of doing the summation of products as in convolution, we do a maximum of sums. These calculations for a sample function f are shown in Figure 6.

Let $f: F \rightarrow E$ and $k: K \rightarrow E$. Then $f \oplus k: F \oplus K \rightarrow E$ can be computed by using

$$(f \oplus k)(x) = max\{f(x - z) + k(z) \mid z \in K,\ x - z \in F\}$$

In terms of the original set F, the dilation returns all of the locations where the origin of K can be placed such that it has a non-empty intersection with F. In another sense, the dilation leaves as background only those locations where K fits completely in the background, so it tells us something about the 'containment' of K in the background.

	K		
y	1	0	-1
$k(y)$	36	0	36

	F			
x	4	5	6	7
$f(x)$	19	23	9	29

y	1	0	-1
$f(4 - y)$	-	19	23
$k(y)$	36	0	36
$f(4 - y) + k(y)$	-	19	59
max $f(4 - y) + k(y)$			59

y	1	0	-1
$f(6 - y)$	23	9	29
$k(y)$	36	0	36
$f(6 - y) + k(y)$	59	9	65
max $f(6 - y) + k(y)$			65

y	1	0	-1
$f(5 - y)$	19	23	9
$k(y)$	36	0	36
$f(5 - y) + k(y)$	55	23	45
max $f(5 - y) + k(y)$			55

y	1	0	-1
$f(7 - y)$	9	29	-
$k(y)$	36	0	36
$f(7 - y) + k(y)$	45	29	-
max $f(7 - y) + k(y)$			45

x	4	5	6	7
$f(x)$	19	23	9	29
dilation	59	55	65	45

Figure 6. Calculations for a grayscale dilation. F is a one-dimensional image which is dilated with the structuring element K, as shown at the top. A maximum of a set of sums is computed to obtain the value of each pixel in the new image. These computations are shown in the four blocks at the center. The values for $(f \oplus k)(x)$ are shown at the bottom. (Adapted from [4].)

Another analogy to explain dilation is as follows. We can think of each pixel in the 2D support as having a number of little cubes stacked on top of it, representing its height or gray value. These cubes remain at the end of the operation if it does *not* represent a location where the *stelt* or structuring element could be placed to fit entirely above the surface (in the background). It is very interesting to note that while the structuring elements work primarily according to criteria related to containment in the x and y dimensions, their effect is to modify the Z level or gray dimension, almost as a side effect.

Dilation by disk structuring elements corresponds to isotropic swelling or expansion algorithms common to binary image processing. Dilation by small squares (3× 3) is a neighborhood operation easily implemented by adjacency connected array architectures and is the one that many image-processing people know by the name "fill", "expand", or "grow".

In mathematical morphology, structuring elements are considered to be probes or special tools for looking at images. The simplest structuring element is a single point. If the point is directly on the origin then we get an identity transformation or exact copy of the original; if not, the shape of the original set will be maintained but it may be translated and/or shifted in the gray level dimension. Other common structuring elements are a pair of points, a line structuring element which consists of a contiguous linear sequence of points with the origin at the middle, diamond or rhombus shapes, squares, octagons, disks and rings. The structuring element is chosen depending on the task and the input image. Once chosen, it is made to interact with the image, and the changes observed. Because of what we know of the interaction, we can deduce something about the shapes in the original image.

Dilation is *extensive* since it always gives us more than what we had at the beginning. This property is important in that if we are searching for an operation which will give us a certain result which contains the set we have at present, we can restrict our search to just the extensive ones.

2.4.3 Erosion

Erosion is the morphological dual of dilation. It is the morphological transformation that combines two sets by using vector subtraction of set elements. The erosion of F and K is denoted by $F \ominus K$ and is defined by

$$F \ominus K = \{x \in E^N \mid x + k \in F \text{ for every } k \in K\}$$

where F and K are sets in E^N, denoting Euclidean N-space.

Whereas dilation can be represented as a union of translates, erosion can be represented as an intersection of the negative translates. (Refer again to Figure 5.) The erosion transformation is popularly conceived of as a shrinking of the original image. In set terms, the eroded set is often thought of as being contained in the original set. A transformation that has this property is called *antiextensive* since it always gives us less than what we had earlier.

Computationally, grayscale erosion can be accomplished by taking the minimum of a set of differences. Let $f : F \rightarrow E$ and $k : K \rightarrow E$. Then $f \ominus k : F \ominus K \rightarrow E$ can be computed by using

$$(f \ominus k)(x) = \min\{f(x + z) - k(z) \mid z \in K\}$$

These calculations for a sample f are illustrated in Figure 7.

We can use the same analogy as in the case of dilation to explain the concept of erosion. One can imagine the gray level surface as a landscape, and then slide the structuring element over it. Each pixel in the 2D support has a number of little cubes stacked on top of it, representing its height or gray value. These cubes will remain after an erosion if they represent locations where the stelt could be centered such that it would fit entirely below the surface.

	K		
y	1	0	-1
$k(y)$	6	0	6

	F			
x	4	5	6	7
$f(x)$	19	23	9	29

y	-1	0	1
$f(4 + y)$	-	19	23
$k(y)$	6	0	6
$f(4 + y) - k(y)$	-	19	17
min $f(4 + y) - k(y)$			17

y	-1	0	1
$f(6 + y)$	23	9	29
$k(y)$	6	0	6
$f(6 + y) - k(y)$	17	9	23
min $f(6 + y) - k(y)$			9

y	-1	0	1
$f(5 + y)$	19	23	9
$k(y)$	6	0	6
$f(5 + y) - k(y)$	13	23	3
min $f(5 + y) - k(y)$			3

y	-1	0	1
$f(7 + y)$	9	29	-
$k(y)$	6	0	6
$f(7 + y) - k(y)$	3	29	-
min $f(7 + y) - k(y)$			3

x	4	5	6	7
$f(x)$	19	23	9	29
erosion	17	3	9	3

Figure 7. Calculations for a grayscale erosion. F is a one-dimensional image which is eroded with the structuring element K, as shown at the top. A minimum of a set of differences is computed to obtain the value of each pixel in the new image. These computations are shown in the four blocks at the center. The values for $(f \ominus k)(x)$ are shown at the bottom. (Adapted from [4].)

Alternatively, think of gray value of the erosion at any pixel as the maximum value for which the structuring element centered at that point and level still fits entirely within the foreground under the surface. This is computed by taking the minimum of the gray surface translated by all the points of the structuring element. For flat stelts, erosion returns the minimum value within the region covered by the structuring element. Gray

stelts modify this by biasing the neighbors by different constants before taking the minimum.

Erosion is useful in that it tells us about containment in the foreground, since it returns all the locations where we can place the origin of structuring element, K, such that K completely fits inside of F. Alternatively, the area or volume left after the erosion expresses the probability that a shape K placed at random in the space is completely contained within F.

A very important property of erosion and dilation is that the structuring elements can be decomposed into smaller, simpler elements [9]. The practical importance of such structural decomposition is that architectures may be developed for small 3×3 windows which are concatenated into a pipeline to construct the equivalent of using larger stelts. This helps get effectively much larger structuring element sizes. However, the drawback of this approach is in getting structuring elements that are not well-rounded because the 3 ×3 window limits the number of slant angles that can be produced.

2.4.4 Opening and closing of images

Morphological opening and closing of are operations that are formed by performing dilation and erosion in sequence. These operators can produce powerful filtering effects on images.

An opening is an erosion followed by a dilation, while a closing is a dilation followed by an erosion. Note that the structuring element used is the same for both the erosions and dilations in the above cases.

Opening and closing may be defined as

$$F \circ K = (F \ominus K) \oplus K \qquad \textit{Opening}$$

$$F \bullet K = (F \oplus K) \ominus K \qquad \textit{Closing}$$

Opening and closing are dual operations. Opening is anti-extensive, while closing is extensive. Openings and closings tell us about the sizes of things - the sizes of objects in the foreground, or spaces in the background. They let us classify pixels according to the size and shape of the region to which they belong, rather than according to their distance from the foreground or background.

Openings result in removal of the foreground protrusions and small regions which are smaller than the structuring element [10]. The reverse operation of closing will result in closing small gaps or holes in the image foreground.

Chapter 3. Architectures for image processing

3.1 Real-time image processing requirements

Image processing involves a large quantity of data. Typical images consist of 512 rows × 512 columns for a total of 262,144 pixels. Each pixel in a grayscale image is commonly represented by an 8-bit pixel value. To process an image, all this data has to be accepted, processed, and output. This is especially challenging for a real-time system, where the input rate is typically 30 frames per second.

Many image processing tasks tend to be computationally intensive. To match the needs of computing in real time, these tasks have to be partitioned so that pipelining or parallelism is achieved. This, however, increases the inter-processor communication and I/O overheads.

Conventional general-purpose machines are not suited to meet the high I/O requirements of the complex image processing tasks, nor are they equipped for parallel computation that are required in many vision-related tasks.

Several parallel processing systems have been proposed for image processing. Mesh architectures [11, 12] provide significant speedup after images are loaded, but the I/O limitations are severe in these systems. Pipelined machines [13, 14, 15, 16] can accept image data in real-time from a camera but these suffer from the problem of being difficult

to reconfigure for different processing tasks. These machines also suffer from the problem of not performing fast enough if certain computationally intensive, and hence slow, subtasks in the pipeline cannot be further partitioned.

3.2 Architectures for image processing

To meet the specific computational demands of image processing applications, various architectures have been proposed and developed over the past years. This section describes several representative machines.

3.2.1 MESH - an architecture for image processing

The MESH system [12] is a two-dimensional SIMD array developed at the University of Oxford. The 2D array is an appropriate architecture for computer vision since it reflects the topology of the vision problem. The image pixels are mapped to the processor array on a one-to-one basis. The large two-dimensional array of processors is implemented by using Ultra Large Scale Integration (ULSI) or Wafer Scale Integration (WSI). These technologies provide the means to put down a large array of processors on silicon. The SIMD controller broadcasts its instructions to all the processors in the array. All the processors then carry out the same instruction on their data.

The MESH architecture is appropriate for algorithms, such as image filtering, which are based on local nearest-neighbor operations and where all pixels have the same instruction executed on them.

A typical problem with large chips is the presence of defects that occur during fabrication, wafer testing, mounting, and during the lifetime of the IC. The architecture uses a novel fault-tolerant strategy to enhance yield and improve reliability. This is done by testing the processors on power-up and identifying the faulty ones. The array is then reconfigured, and used for computation. Thus, the architecture uses hardware tolerance along with software tolerance to provide a robust parallel image-processing system.

3.2.2 PASM - a reconfigurable multimicrocomputer system for image processing

PASM [17] is a special purpose, dynamically reconfigurable, large-scale multimicroprocessor system. It can be partitioned to operate several independent SIMD and/or MIMD machines of varying sizes. A variety of image-processing problems have been implemented on it.

The PASM architecture consists of a system control unit which controls and co-ordinates the activities of the other components. It has a parallel computation unit which contains $N = 2^n$ processors, N memory modules, and an interconnection network. The processors are microprogrammable and a memory module is connected to each processor to form a processing element. There are microcontrollers, which are microprogrammable microprocessors, acting as the control units for the processors in SIMD mode and orchestrating their activities in the MIMD mode. One of the projects uses a 1024-PE architecture to demonstrate image processing tasks. A 16-PE prototype system, based on the Motorola MC68010 microprocessor, is also developed for the same purposes.

The dynamically reconfigurable nature of the architecture and the constituent microprogrammable processing elements make the PASM suitable for a wide range of image processing tasks.

3.2.3 Massively Parallel Processor (MPP)

The Massively Parallel Processor [18], completed by Goodyear Aerospace Corp., is an example of a bit-serial SIMD array processor. It is a 128×128 array of one-bit microprocessors, operating with a clock cycle of 100 ns.

This machine is very efficient when processing small images if the number of transformation cycles is greater than the number of cycles required to load and unload the image. It also has certain disadvantages. The number of processors required and the interconnections required is large. Because of bit-serial processing, the number of instructions needed to perform a transformation increases. Error detection and fault tolerance is poor. I/O, being serial, dominates the total processing time and reduces the overall efficiency.

3.2.4 Cytocomputer

Operations like binary mathematical morphology are easily implemented in cellular logic image processors like the Cytocomputer [13]. Cytocomputers are pipelined neighborhood processors specialized for implementing tasks such as morphology through iterative neighborhood operations. Cytocomputers have been developed by the Environmental Research Institute of Michigan (ERIM).

The system utilizes mathematically supported neighborhood processing stages with modules for image storage, multiple-image combinations and high-speed transfer. It operates under control of a host microcomputer and cellular processing is performed in the pipeline of raster subarray neighborhood processors.

Each processing element stage of the cellular computer contains a 3×3 cellular register and the associated lookup table in conjunction with storage registers for two lines of image data. The contents of each lookup table is different from that of the others. Therefore, most of the lookup tables are loaded separately from the host computer. The cellular computer built by ERIM consists of a total of 88 binary image processing stages.

Cytocomputers are useful for image processing tasks when operating a known algorithm repetitively. The lookup tables are fixed and are loaded at initialization. The overhead in the lookup table loading procedure is significant. This is a disadvantage when the system is used in an environment where algorithms are frequently modified.

3.2.5 Other architectures

There have been a number of architectures proposed and implemented for image processing purposes. These include machines, such as the Cellular Logic Image Processor (CLIP) [19], which have been used effectively to demonstrate binary mathematical morphology and image neighborhood operations. Pipeline architectures for morphologic image analysis have been proposed in [14]. A complete image processing system on a single, 2-slot, VMEbus card, has been developed by Datacube [15]. The system, called MaxVideo 200, is capable of processing large images through multiple, simultaneous pipes operating at 40 MHz and 20 MHz. The system features Datacube VSIM (Virtual Surface Image Memory), a high performance memory processor. Users can configure the MaxVideo 200 with any number of these memory processors to best match their processing/cost parameters.

A heterogeneous parallel processor, consisting of three different, tightly-coupled, parallel processors, has been developed at the University of Massachusetts. Known as the UMass Image Understanding Architecture (IUA) [20], it is designed to address the differing requirements of the low, intermediate, and high levels of a knowledge-based computer vision system. The low-level processor is a reconfigurable mesh-connected array of bit-serial processors operating in SIMD modes. Its purpose is to process image data and extract features that can be represented in a symbolic form. The intermediate-level processor operates in MIMD mode, and consists of an array of digital signal processors. The high-level processor is a coarse-grained MIMD system that supports knowledge-based processing. The first generation of the IUA had 4K low-level processors, 64 intermediate-level processors, and a single high-level processor. The next generations have retained the three-level structure, but the architecture of each level has been enhanced. The density of processors has been increased and I/O features have been improved. The software system for the third generation of IUA supports FORTH, C, and C++ library functions.

The use of application specific integrated circuits and programmable gate arrays is becoming increasingly popular. In case of the later, the applications designer has the flexibility to program the programmable gate arrays and thus a variety of image processing tasks can be performed with the same hardware resources. A programmable and flexible array boundary processor for a fine grain mesh-connected SIMD array of 1000 FPGA-based processing elements has been developed at the University of Bologna [21] as part of an artificial vision system.

Chapter 4. Splash-2 : a real-time custom computing platform

4.1 The VTSplash system

To demonstrate processing of image data using the Splash-2, a real-time image processing system has been established at Virginia Tech. This laboratory system, called *VTSplash*, uses a Splash-2 attached processor which serves a Sun SPARC-2 host.

Figure 8 shows the laboratory setup of the real-time vision system. The complete system consists of a video camera, a Splash-2 attached processor, two custom interface boards, a commercial frame grabber, a video monitor and a Sun SPARC-2 workstation.

The camera produces an RS-170 video signal which is digitized at 9.8 MHz. The custom digitizer reformats the video stream and passes it to Splash-2 at 10 MHz. After processing, the output of Splash-2, which may be a real-time video data stream, or some other form of information, is presented to another custom board. This board formats the data. Once formatted, the data is presented to a second frame grabber card (a commercial Data Translation DT2867LC frame grabber) which accepts these images and displays them on a color monitor. The Sun SPARC-2 workstation serves as the VTSplash host and is responsible for configuring Splash-2 and establishing communication paths. The Sun SPARC-2 workstation does not participate or interfere with the real-time video

stream. It may, however, be used to download the processed data at nonreal-time clock rates.

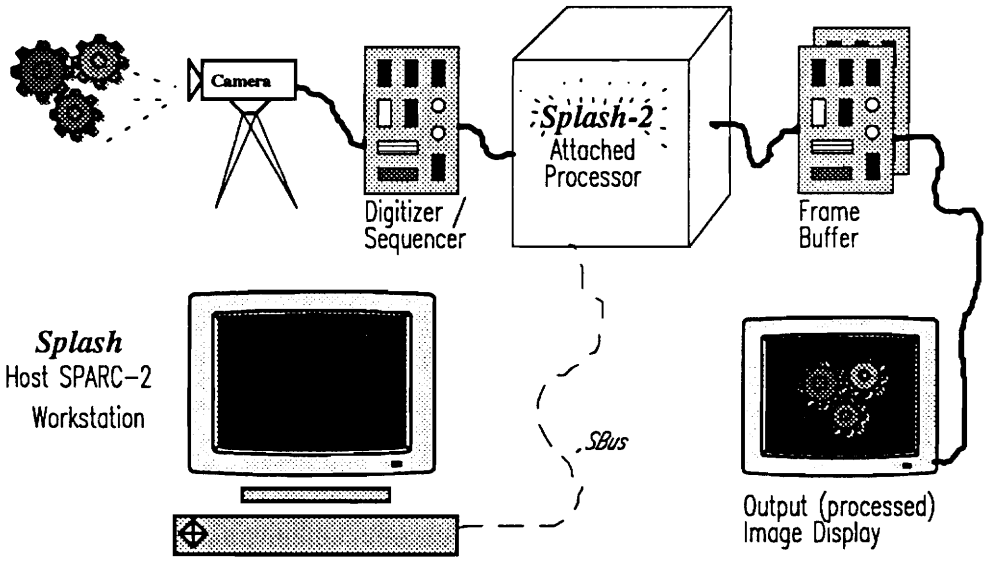


Figure 8. VTSplash - laboratory setup for real-time vision system. The Splash-2 processor serves a SPARC-2 host.

4.2 The Splash-2 system

Splash-2 [1] is a second generation custom computing attached processor designed by the Supercomputing Research Center in Bowie, Maryland. The architecture of Splash-2 is designed to accelerate the solution of problems which exhibit at least modest amounts of temporal parallelism (pipelining) or data parallelism (single instruction multiple data stream) [22]. Splash-2 offers an attractive alternative to traditional architectures. The reconfigurable nature of Splash-2 provides the performance of ASIC hardware, while

preserving the general-purpose nature of being able to accommodate a wide variety of tasks.

Splash-2 is classified as an *attached processor* since it is intended to append a host machine through an expansion bus. It differs from a *coprocessor* in that it does not reside directly on the host processor bus. Splash-2 has been designed with an SBus interface, and currently serves a Sun SPARC-2 host.

The schematic for the Splash-2 system is shown in Figure 9. A Splash-2 system consists of one to fifteen Splash array boards, an interface board and a Sun SPARC-2 workstation host. Each array board contains sixteen processing elements (PEs), X1 through X16, arranged in a linear array, and fully connected via a 16X16 crossbar switch, which is regulated by the seventeenth FPGA control element, X0. The sixteen processing elements are arranged in a linear array with each connected to its neighbors to the left and right via 36-bit wide data paths.

The Splash-2 interface board performs the functions of system control, DMA to and from host memory, interrupt handling and generation, data stream pre-processing and data stream post-processing, and clock control. It also matches the 32-bit data width of the host to the 36-bits used by Splash. Details of the Splash-2 Interface board are provided in [22].

Each processing element within a Splash-2 processing board (identified by the *Xn's* in the Figure 9) consists of an FPGA [2] and attached memory. The Xilinx XC4010 FPGA consists of a two-dimensional array of 400 configurable logic blocks that can be connected internally by general interconnection resources. Details about FPGAs are provided in section 4.3. The memory attached to each Xilinx FPGA is a fast 256KX16 static memory. Figure 10 shows schematic for a processing element within a Splash-2 array board.

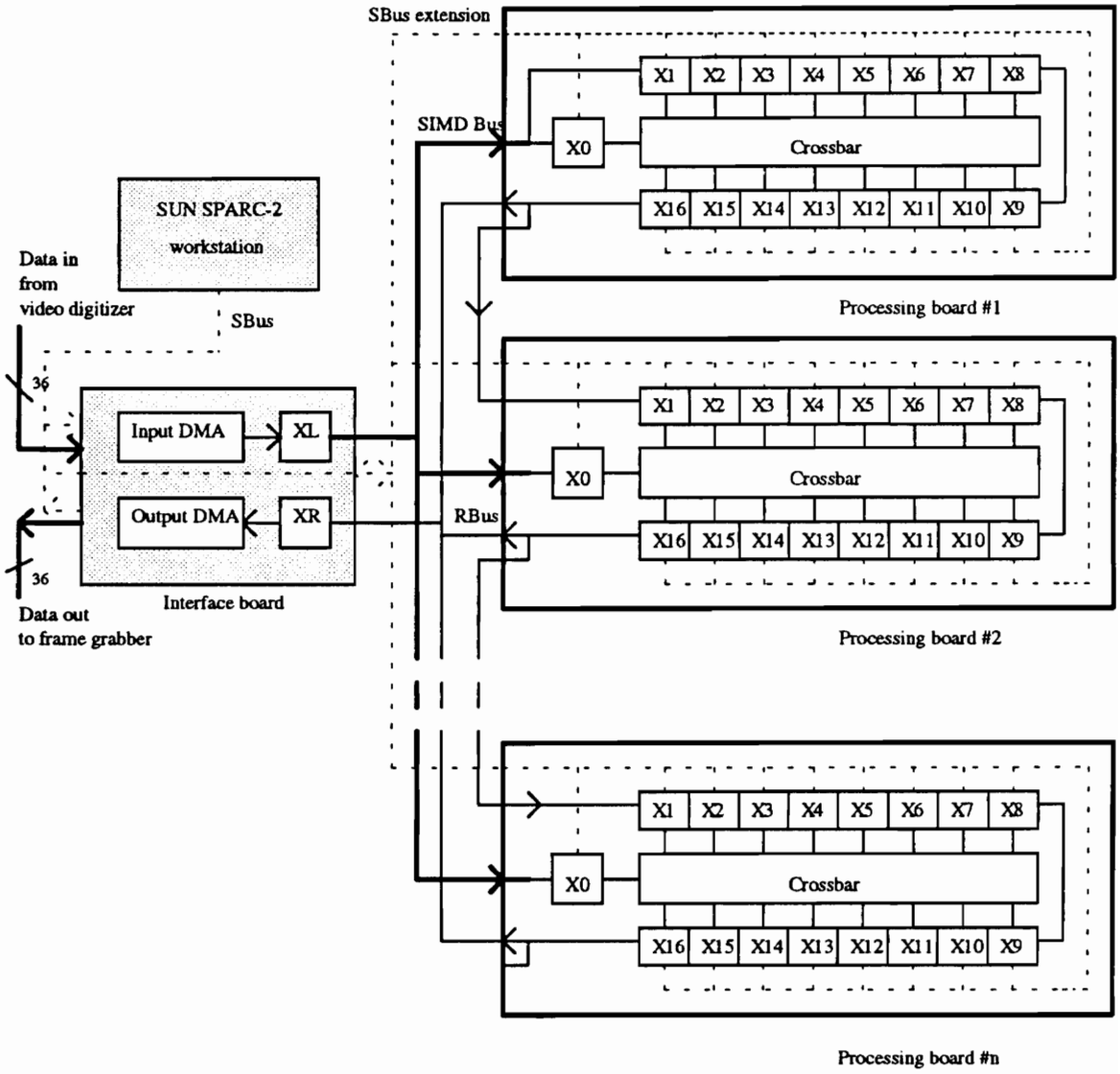


Figure 9. The Splash-2 attached processor system. There may be one or more Splash-2 boards, each consisting of sixteen processing elements (X1-X16) and a control element X0. The data flows from X1 to X16 of each board in a systolic fashion. Direct communication between each pair of PEs on a board is also possible via the crossbar. The video digitizer and the frame grabber are not a part of the Splash-2 processor.

Each of the FPGAs (X0 through X16) is connected to its memory through an 18-bit address bus, a 16-bit bi-directional data bus, one read signal, and one write signal.

Because the memory timing is controlled off chip, the interface as seen by the FPGA is purely synchronous with the clock. To perform a read operation, the FPGA drives an 18-bit address and a logic '1' on the read signal at the rising edge of the clock. The addressed word of memory is guaranteed to be available to be latched by the PE on the next rising edge of the clock. On the same edge that data from one transaction is latched, a new address can be presented to start the next cycle, thus allowing back to back pipelined reads. To perform a write operation, the FPGA drives the 18-bit address, a logic '1' on the write signal, and the data to be written, all on the same rising edge of the clock.

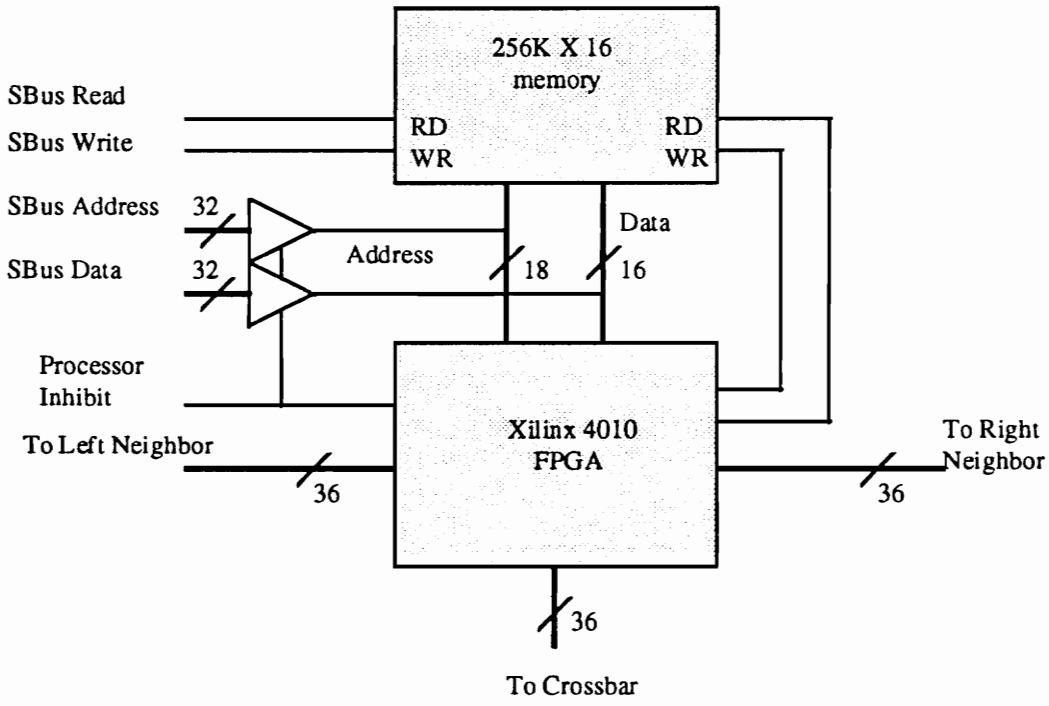


Figure 10. Splash-2 processing element. The PE consists of a Xilinx 4010 FPGA and local 256K × 16 memory.

Two constraints must be observed by an application program using the Splash-2 hardware. One is that the read operation cannot be followed by a write operation on consecutive clock edges. The read control signal must be de-asserted atleast one clock

cycle before the write signal is asserted. Secondly, if both read and write control signals are asserted simultaneously, the results are unpredictable.

It should also be noted that the SBus is extended to each of the Splash-2 array boards, allowing the host direct access to the external memory (Refer Figure 9). However, the host and the FPGA computing element cannot access the Splash-2 memory simultaneously.

Each FPGA has three 36-bit bi-directional data paths, one each to the left and right neighboring processing elements, and one to the crossbar switch. In addition, a 16-bit path exists between the FPGA and its static RAM. The input data stream to the array is provided by XL on the interface card through the 36-bit SIMD bus to X0 of each processor board and to X1 of the first processor board. The output data stream is produced from multiple array boards which are linked together by extending the output data stream from the X16 of one board to the X1 of the next. The last board connects to RBus.

The control paths between the Sun SPARC-2 host and the application program running on Splash-2 consists of a set of handshake registers, two on each Splash-2 array board, a global AND/OR mechanism, a broadcast signal, direct access to the on-board memory, and an interrupt mechanism [1, 22].

The two handshake registers on each Splash-2 array board may be used to perform asynchronous communication between the computing elements and the host. Handshake register 1 (*HS1*) contains 17 bi-directional bits, one to each of the 17 computing elements on the board. The direction of *HS1* is controlled by a bit in the Splash-2 board control register. Handshake register 2 (*HS2*) contains a single bit connected to all 17 computing elements. The direction of *HS2* is input to the computing elements only.

Two separate paths are provided to perform global AND/OR reduction across all of the processing elements in the system. *XP_GOR_Result* and *XP_GOR_Valid* are two bi-directional signals which connect every PE on the Splash-2 board to the control element X0. X0 has two output signals connected in a wire-AND configuration to the corresponding outputs of the X0 chips of every board and these values are available to the host in a control register on the interface board.

A single bit on the Interface board connected to the control element X0 and writable by the host is used as the *broadcast bit.* X0 has a separate output signal connected to all the PEs.

Each of the PEs has an *interrupt signal* which is ANDed with the corresponding bit of a 17-bit mask register to form a 17-bit interrupt status register, which is readable by the host.

The central crossbar is built from nine Texas Instruments SN74ACT8841 16X16 4-bit crossbar chips. The crossbar switch is capable of storing up to eight separate, dynamically selectable configurations, with each configuration specifying a different set of connections among the sixteen ports. The crossbar allows point to point, multicast and broadcast communication between all of the processing elements on each processing board. The configuration in use in any given cycle is selected by the control element X0, and may be changed as frequently as once per clock cycle.

Each of the sixteen PEs is connected to the crossbar by a 36 bit bi-directional data path and a 5 bit uni-directional control path. The data path is subdivided into five sections, four bytes and one nibble. Each of these sections may be controlled independent of the others. The set of crossbar configurations for an application program are programmed through a simple text file which is parsed at run-time and is used to generate the programming for the nine crossbar chips.

4.3 Field Programmable Gate Arrays

Field Programmable Gate Arrays or FPGAs consist of an array of uncommitted elements that can be interconnected using programmable resources. The interconnections between the elements are user-programmable [2].

Figure 11 shows the conceptual diagram of a typical FPGA. The interconnect comprises segments of wire, the segments being of variable lengths. Programmable switches serve to connect the logic blocks to the wire segments, or one wire segment to another.

Logic circuits are implemented in an FPGA by partitioning the logic into individual logic blocks. The logic blocks are then interconnected as required by programmable switches. The structure and content of a logic block are called its architecture. The Xilinx XC4000 series of FPGAs have Configurable logic blocks (CLB) which utilize a two-stage arrangement of look-up tables that yields a greater logic capacity per CLB than in previous versions of the Xilinx family of FPGAs. The CLB has two outputs, which may be either combinational or registered.

The structure and content of the interconnect in an FPGA is called its routing architecture. The routing architecture consists of both wire segments and programmable switches. In the Xilinx XC4000 family of FPGAs used in the Splash-2 architecture, routing is done by Single Length Lines and Double Length Lines. The former set are used for relatively short connections or those that do not have critical timing requirements. The Double Length Lines pass through half as many switch matrices and offer lower routing delays.

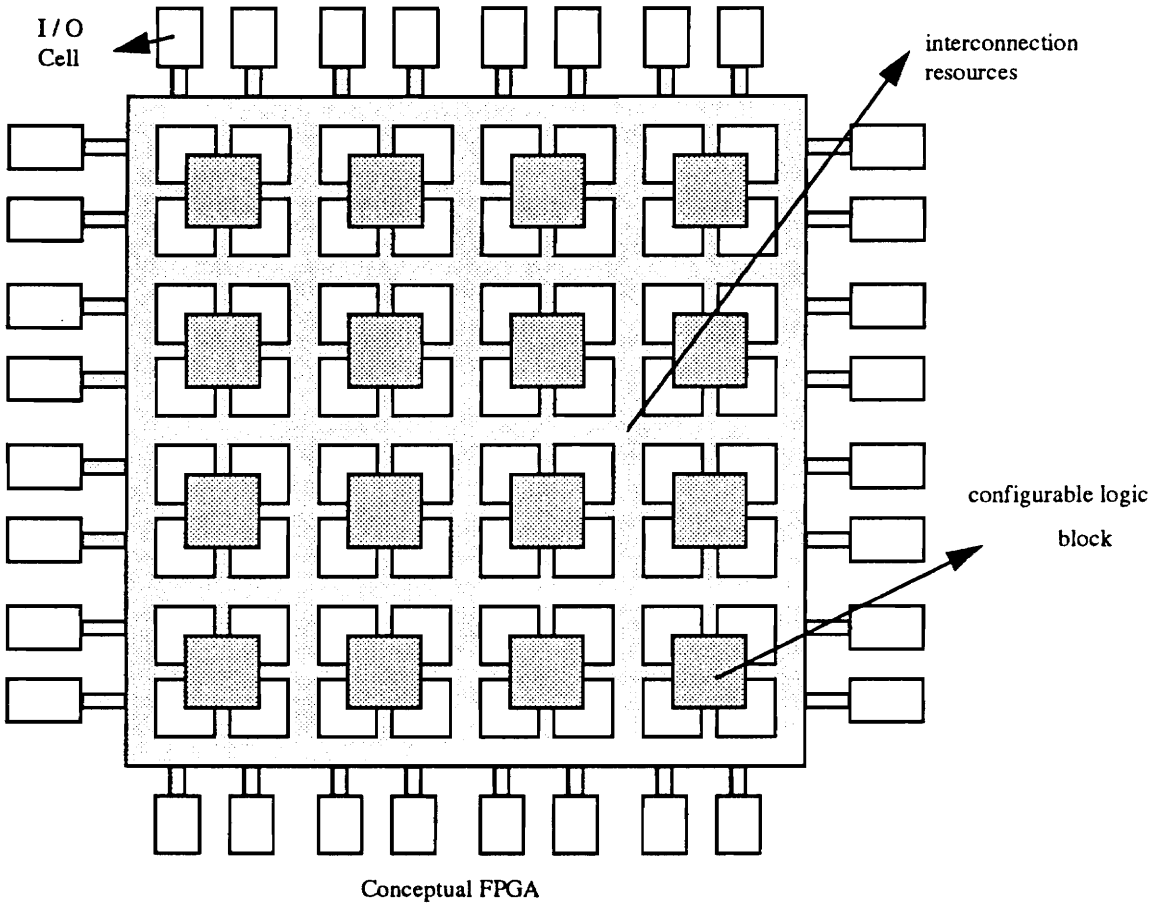


Figure 11. Concept of an FPGA structure. The FPGA consists of configurable logic blocks (CLBs) which are connected by general-purpose programmable interconnection resources and programmable switches.

For the XC4010 series of Xilinx FPGAs used in the Splash-2 processor array, the number of CLBs is 400 and the equivalent gate count approximately 10,000. The CLB of the XC4010 is more powerful and more flexible than the CLBs of the previous Xilinx series of FPGAs. The XC4010 CLB can accommodate up to three logic functions - designated as functions "F", "G", and "H". Figure 12 shows the XC4000 CLB structure.

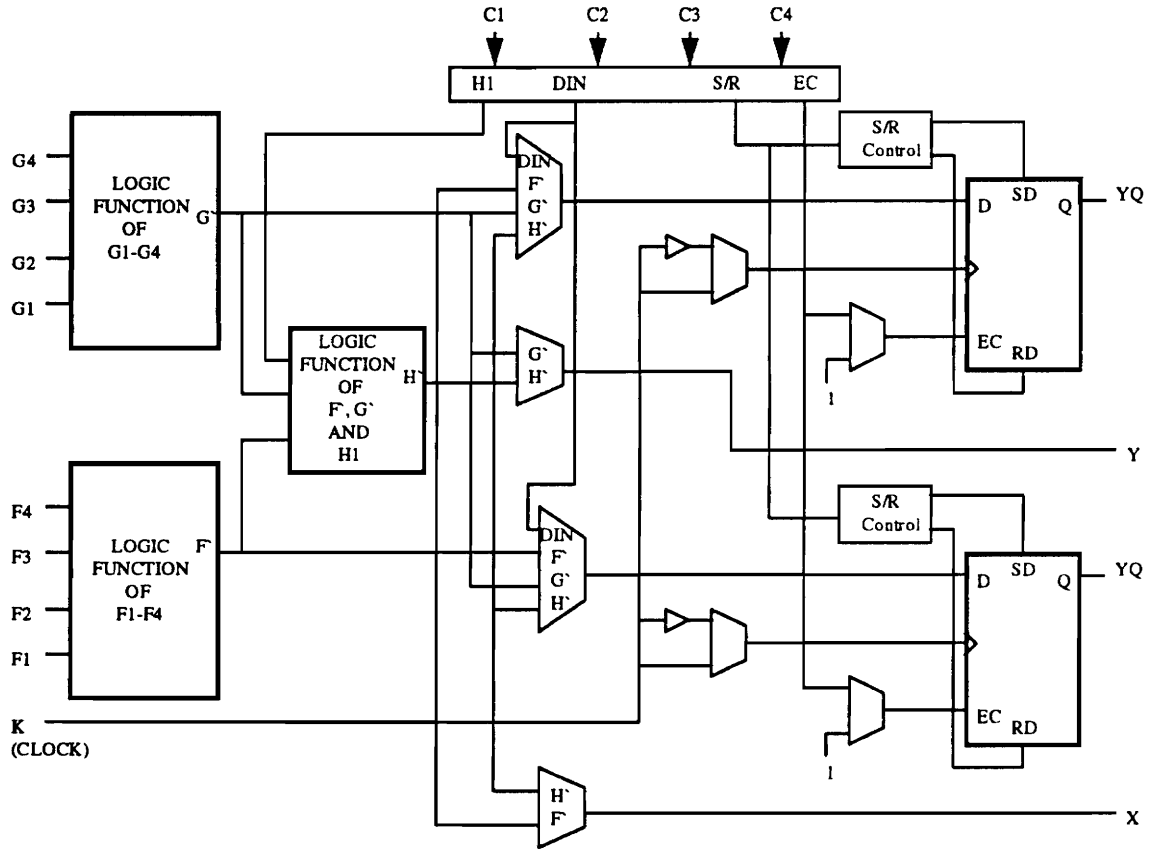


Figure 12. Configurable Logic Block structure for the Xilinx XC4000 series FPGA. There are 400 such CLB in a XC4010 FPGA used in Splash-2. The logic that can be implemented in such a device is equivalent to approximately 10,000 gates.

The Xilinx XC4010 FPGA has 160 Input/Outputs and the Maximum RAM bits are 12,800. It combines architectural versatility, on-chip RAM, increased speed and gate complexity with abundant routing resources and sophisticated software, to achieve fully automated implementation of complex, high performance designs [23].

The design of the CLBs, coupled with that of the interconnection resources, should be such as to facilitate the implementation of a large number of digital logic circuits.

Chapter 5. Design methodology for Splash-2

5.1 Introduction

The Splash-2 provides the programmer with the freedom to configure each of the processing elements. This chapter describes the Splash-2 design process, the Splash-2 simulation environment, the synthesis tools used to map the high-level designs to the Xilinx FPGA devices, and the design verification and debugging facilities available to the Splash-2 applications programmer.

5.2 The Splash-2 design process

The programming environment for the Splash-2 custom computing platform is complete and fully functional [22]. Applications can be developed within this environment in a well defined design process. The Figure 13 illustrates the basic design steps followed in the development of a typical Splash-2 application.

The first step in the design of an application for the Splash-2 is the defining of the problem. A sound and well understood *problem definition* facilitates the design process.

The next step is to model the task with behavioral descriptions. *Behavioral modeling* for the application is typically done using the C programming language or using

behavioral VHDL models. This modeling is done with two objectives. The first is to understand and verify the defined problem. The second reason for modeling the applications is for obtaining sample results which may be used to validate the designs.

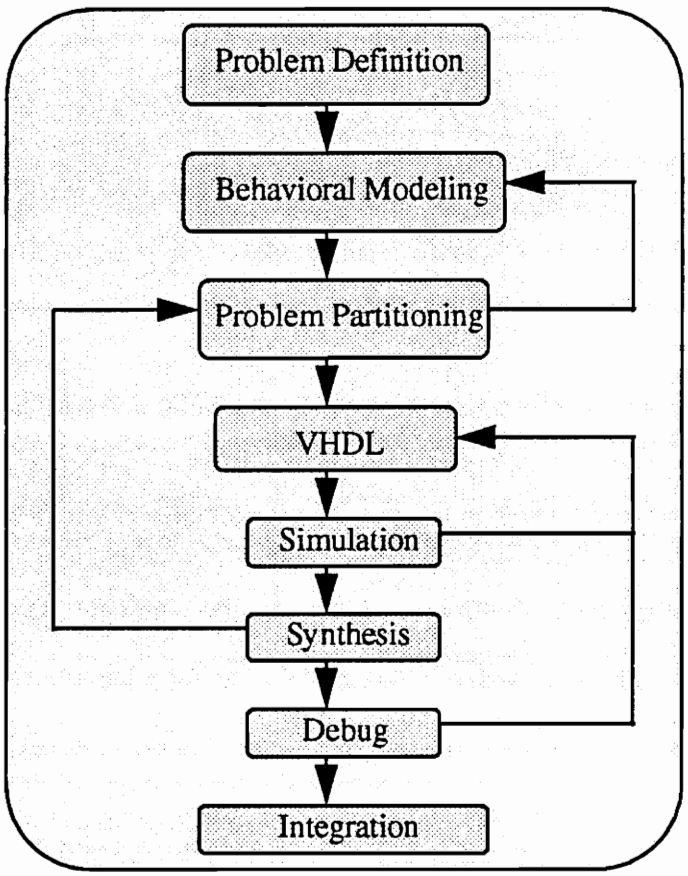


Figure 13. The Splash-2 application design process. The design process starts with defining the problem, and ends with synthesized designs that can be downloaded to the Splash-2. The arrows in the reverse direction represent the iterations needed in the design process to obtain correct results.

The next stage in the Splash-2 application design process is *problem partitioning*. Many applications though easy to model using behavioral descriptions may not map to the Splash-2 architecture directly. This is where problem partitioning becomes significant. There are many factors which must be considered when mapping an application to the

Splash-2 board(s). Some of the primary factors to be considered are *time*, *area*, and *communication complexity.*

The time complexity relates to how much computation is desired per clock cycle. This factor may constrain the amount of logic circuitry that can be designed for computation within the system clock cycle.

The area factor relates to how much of the reconfigurable resources should be allocated to a given computation, and to the total available reconfigurable resources within each processor board and within each of the seventeen processing elements on each board. Area becomes a primary concern in case of applications that are computationally intensive or have a large number of recursive operations.

When the problem is partitioned to map more than one processing element or more than one Splash-2 board, the communication overheads increase proportionally. There is a good interprocessor and I/O communication facility provided in the Splash-2 architecture. However, this facility may be limiting as the need for data transfers and communication increases. Some of the limitations that all Splash-2 designs must comply with are:

- a maximum of 36-bits is allowed for the data paths into and out of the processing elements on the left, right and crossbar ports.
- the data-bus width for the local memory in each processing element is 16-bit.
- communication between the Splash-2 boards is limited to a 36-bit data path, along with several 1-bit global signals.

Though the data path widths are adequate for most general applications, not all applications map easily to these communication constraints, and tough design trade-offs must be considered. Often, a designer must wait until after the synthesis step before he can confirm whether a given problem partition is feasible.

The next step after partitioning the problem is to model it using VHDL. The Splash-2 applications programmer has a number of options to describe the designs. One of the approaches is to use behavioral VHDL descriptions to design the processing elements. Another is to develop structural models and model each entity within the structure using VHDL. The best supported design environment for the Splash-2 is with the Synopsys VHDL simulation and synthesis tools. There are however other approaches as well such as the FPGA design tools like XBLOX.

Once the processing elements are programmed using VHDL, the system needs to be simulated using the Splash-2 simulator (described in section 5.3). For the many image processing tasks discussed in the thesis report, simulation times for a complete frame input often stretched to several days of CPU time per run on a Sun SPARC-10 workstation.

Designs modeled in VHDL and simulated using the Splash-2 simulator are synthesized for actual runs on the Splash-2 hardware. (Details of the Splash-2 synthesis tools, environment, and process are provided in section 5.4.)

It is found that the actual propagation delays in the Xilinx FPGAs are highly sensitive to the outcome of the placement and routing process, and can have a disturbing effect on the behavior of the application. In such cases, the results of Splash-2 runs on actual hardware may not match the results from simulations. This is because the simulation results are based on VHDL models developed prior to placement and routing; and hence are barren of signal propagation annotation. To counter these problems, a powerful debugging tool has been built for the Splash-2. This tool is the *T2 interactive debugger* [22]. It has features that allow monitoring internal state variables and tracing. It should be noted that because of the problems encountered at the synthesis stage which may not have arisen at the simulation stage, the design synthesis steps include the

debugging stage as well. This stage may take several iterations before the design runs correctly on the Splash-2 hardware.

Once the image processing operations are designed and performing satisfactorily on the Splash-2 hardware, they must be integrated within the body of the application. A C library has been developed by the SRC which facilitates communication between a host program and the attached processor.

5.3 The Splash-2 simulation environment

5.3.1 VHDL

The VHSIC Hardware Description Language (VHDL) is an industry standard modeling language for the design and description of electronic systems [24]. VHDL was standardized by the IEEE (Standard 1076-1987) in 1987. It is a formal notation intended for use in all phases of the creation of electronic systems. Because it is both machine readable and human readable, it supports the development, verification, synthesis, and testing of hardware designs, the communication of hardware design data, and the maintenance, modification, and procurement of hardware [25].

VHDL is a powerful medium to represent designs in both the structural and behavioral domains. A structural domain is one in which a component is represented as an interconnection of more primitive components. A behavioral domain is one in which a component is represented by describing its input/output response [26]. Descriptions in both the domains are acceptable for the design process. In this research work, the behavioral domain has been chosen. This is because the designs can be altered easily and the logic optimization tools have more freedom when behavioral models are used.

In VHDL, a given logic circuit is represented as a design entity which consists of an interface description and one or more architectural bodies. The interface description names the entity and describes its inputs and outputs in terms of the mode of the signal ("in" or "out" or "inout") and the type of the signal (*bit* or *bit_vector* or *integer*). The behavior of the entity is specified by an architectural body. The Figure 14 shows a sample VHDL description of the Xilinx 4010 processing part.

```
library SPLASH2;
use SPLASH2.TYPES.all;
use SPLASH2.SPLASH2.all;
use SPLASH2.COMPONENTS.all;

ENTITY Xilinx_Processing_Part IS
        Generic(
                BD_ID    : Integer := 0;          -- Splash Board ID
                PE_ID    : Integer := 0           -- Processing Element ID
        );

        Port (XP_Left          : inout DataPath;
                XP_Right       : inout DataPath;
                XP_Xbar        : inout DataPath;
                XP_Xbar_EN_L   : out   Bit_Vector(4 downto 0);
                XP_Clk         : in    Bit;
                XP_Int         : out   Bit;
                XP_Mem_A       : inout MemAddr;
                XP_Mem_D       : inout MemData;
                XP_Mem_RD_L    : inout RBit3;
                XP_Mem_WR_L    : inout RBit3;
                XP_Mem_Disable : in    Bit;
                XP_Broadcast   : in    Bit;
                XP_Reset       : in    Bit;
                XP_HS0         : inout RBit3;
                XP_HS1         : in    Bit;
                XP_GOR_Result  : inout RBit3;
                XP_GOR_Valid   : inout RBit3;
                XP_LED         : out Bit
        );

END Xilinx_Processing_Part;
```

```
-- Architecture Left_to_Right feeds all 36 bits of data
-- from the Left port to the Right port. Data is latched
-- in the Left port on the first cycle, and latched in the
-- Right port the next cycle, for a total of 2 cycles delay.

ARCHITECTURE Left_to_Right OF Xilinx_Processing_Part IS
        signal PassThru: Bit_Vector(DATAPATH_WIDTH-1 downto 0);
        signal Left: Bit_Vector(DATAPATH_WIDTH-1 downto 0);
        signal Right: Bit_Vector(DATAPATH_WIDTH-1 downto 0);

BEGIN

        PROCESS
        BEGIN
                WAIT UNTIL XP_Clk'Event and XP_Clk = '1';

                Pad_Input (XP_Left, Left);
                Pad_Output (XP_Right, Right);

                Right <= PassThru after 2 ns;
                PassThru <= Left after 2 ns;
        END PROCESS;

        XP_Xbar             <= TriState (XP_Xbar);
        XP_Mem_A            <= TriState (XP_Mem_A);
        XP_Mem_D            <= TriState (XP_Mem_D);
        XP_Mem_RD_L         <= '1';
        XP_Mem_WR_L         <= '1';
        XP_HS0              <= 'Z';
        XP_GOR_Result       <= '0';
        XP_GOR_Valid        <= '0';
        XP_Int              <= '0';
        XP_Xbar_EN_L        <= "11111";

END Left_to_Right;
```

Figure 14. Sample VHDL description for the Xilinx processing element entity. The first section is the entity declaration. This is followed by the architecture body which describes the function of the entity.

The above description represents a design for a 4010 Xilinx processing element. The entity header declares objects used for the communication between a design entity and its environment. The entity is declared as *Xilinx_Processing_Part* in the example shown. The generic list defines generic constants whose values may be determined by the

environment [25]. The port list in the formal port clause defines the input/output ports of the design entity. The above example has two generic constants *BD_ID* and *PE_ID*, and 18 port declarations.

The architectural body, named *Left_to_Right* in the sample program, specifies the relationships between the inputs and outputs of the design entity. The name distinguishes architecture bodies associated with the same entity declaration. Thus, more than one architecture body may exist corresponding to a given entity declaration.

The statement part contains statements that describe the internal organization and/or operation of the block defined by the design entity. In the above example, the value of signal *Left* is assigned to *Passthru* and that of *Passthru* assigned to *Right* only when the clock makes a transition from 0 to 1 (rising edge of the clock). The description therefore represents a circuit that transfers the value of signal *Left* to the signal *Right* at every rising edge of the clock.

This description can be synthesized to obtain netlists and finally bit-file design descriptions that may be downloaded to the actual Xilinx FPGA. For more details about VHDL and its syntax, refer to [24, 25, 26, 27].

5.3.2 The Splash-2 VHDL simulator

The Splash-2 application programs developed in VHDL need to be verified for correct functionality. This can be done using the Splash-2 simulator. The simulator is a hierarchical model of the Splash-2 system comprising a set of VHDL models for each of the components of the system [22]. When an application program is simulated, it is able to interact with the system exactly as it would with the physical hardware. The simulator

is based on commercial tools and a full source level debugging interface is available to the user.

The Splash-2 simulator is written in VHDL and has been ported to commercial simulation tools from Model Technology Inc. and Synopsys Inc. We shall discuss only the simulator for the Synopsys tools.

Synopsys Inc. provides an interactive tool for compiling, simulating, and debugging VHDL programs [28]. The Synopsys simulator runs on the Sun SPARC-station under either X Windows or OpenWindows 3.0. The simulator has both a command line interface and a graphical DBX-like interface.

5.3.2.1 Environment

A script file *startup*, when sourced, sets up the environment for the Synopsys simulator. This script sets several environmental variables which identify the locations of executable and library files. A shell variable *$SYNOPSYS* identifies the location of the base directory of the Synopsys tools. A Synopsys provided shell script is then sourced which modifies the user's search path and creates several more environment variables. It sets the environment variable *$VUOF* to point to the default VHDL User option file. This file establishes bindings of a number of internal simulator variables and contains the mappings from logical library names to actual directories.

As mentioned previously, the Splash-2 Simulator is a collection of VHDL components which, together with the user provided application code, models the behavior of the Splash-2 system. The simulator structure is the same as the actual Splash-2 architecture. The top level entity is the *Splash_System*, which contains components for the Interface board, and the collection of Splash-2 boards. The *Splash_System* entity defines all of the top level signals, such as the host SBus and the Splash SIMD bus.

The *Interface_Board* entity contains components for the input and output FIFOs and the XL and XR chips, and is responsible for generating the system clock. The FIFO components read and write ASCII data files. The *Splash2_Boards* entity contains one or more component instantiations for the Splash-2 array boards. The set of signals used to interconnect the Xilinx parts on each board and the previous/next inter-board connections are declared here. The slot ID for each board is also declared at this level. The *Splash2_Board* entity contains component instantiations for the programmable Xilinx FPGAs, their associated memories, and the crossbar. The control functions such as selection of the source of X1, generating the memory write enable pulse, and checking timing restrictions on memory accesses are performed at this level.

A file *system.vhd* contains the entity and architecture for the top level *Splash_System* component. The Splash-2 system is configured by the file *config.vhd.* The first part of this file configures the interface board. *Input_file1* represents the name of the file to be read by the input FIFO 1. The default value is a null string. Similarly, *Input_file 2* represents the name of the file to be read by the input FIFO 2. The *Output_file1* and *Output_file2* represent the names of the files to be written by output FIFO 1 and output FIFO 2 respectively. *File_type* defines the format of the output files. The default value is Bin though hex, bin may be chosen. *Clock_Freq* determines the frequency of the system clock in MHz. The default value is 20 MHz.

The component for the Splash board within *Splash_System* is named *Splash. Number_of_boards* determines the number of Splash-2 array boards in the system. By default, this value is 1. The array of Splash-2 boards is generated with the identifier *SBOARDS.* Within the generate statement, the *Splash2_Board* component is instantiated as *BD.* The architecture of the *Splash2_Board* may either be a user supplied behavioral

model from the current working library or the default structural model which instantiates individual FPGA components.

The structure architecture of *Splash2_board* instantiates the following components:-

- a model of the central crossbar called *Splash_Crossbar.*
- the FPGA control part (X0) called *Xilinx_Control_Part.*
- the FPGA processing parts, X1 through X16 called *Xilinx_Processing_Part.*
- the external memories associated with X0-X16 called *Memory_Part.*

The *Splash_Crossbar* component has a single generic parameter called *config_file.* This is a string specifying the name of the file containing up to eight crossbar configurations.

The *Memory_Part* component has a single generic parameter called *Load_file* which is a string specifying the name of the file containing directives for pre-loading the contents of the memory.

The *Xilinx_Processing_Part* components and their associated memories are instantiated with a VHDL generate statement with the identifier *XPARTS.*

The *Splash-2 library* contains a set of VHDL packages which may be useful for the development of VHDL application code for Splash-2. These are further grouped according to their functionality.

The *TYPES* package contains the definitions of the data types used for inter-chip communication in the Splash-2 system. The *SPLASH2* package contains a variety of constants, data types, and functions which are specific to either the Splash-2 architecture or the Splash-2 simulator. These include constants such as *Mem_Width, Datapath_Width.* The *COMPONENTS* package contains a set of components and procedures useful in writing Splash-2 applications. These include the procedures such as "Pads" which can be

used as an interface between the tristate (RBit3 type) signals external to the Xilinx chips and the standard logic levels (Bit type) internal to the chips. The most used Pad procedures include the *Pad_Input*, *Pad_Output*, *Pad_InOut*, and *Pad_XBar*. The *ARITHMETIC* package provides support for performing signed and unsigned arithmetic on bit vectors. The *HMACROS* package contains component declarations and simulation models for the Xilinx provided Hard Macros. When using hard macros, the names of the components must exactly match the names of the macros as given in the Xilinx XC4000 Macro library Manual.

Xilinx_Processing_Part is the name of the basic processing element of the Splash-2 system, Xilinx chips X1 through X16. A detailed summary of each of the ports of this entity is given in the Appendix B. The entity declaration for the control part on the Splash-2 board is called the *Xilinx_Control_Part*. The summary of each port for the entity is given in the Appendix B.

The file containing the setting information for the crossbar may be passed to the crossbar model in the Splash-2 configuration file "config.vhd". A sample crossbar file is shown in Appendix C. The memory model used by the Splash-2 simulator allows the user to specify a set of initial contents for the on board memory from an ASCII input file. The name of the file is passed as a generic parameter to the entity *Memory_Part*. To specify an initialization file for a memory, the file name must be passed to the appropriate Memory_Part through the *Load_File* generic in the "config.vhd" file. A sample initialization file is shown in the Appendix C.

5.3.2.2 Execution

To simulate the VHDL file within the Synopsys environment, the command issued is *vhdlan*. This command analyzes each design unit into the logical library WORK. A

mapping for the target library must exist in a user option file. If more than one VHDL file is analyzed as part of a design, the order in which the files are analyzed may be important. The *vhdlan* produces a number of intermediate files that are subsequently used by the simulator. A separate ".sim" file is produced for each entity, for each architecture, for each package, for each package body, and for each configuration analyzed. One ".mra" is produced for each entity analyzed, and contains a list of the architectures corresponding to that entity.

Two interfaces exist for the Synopsys VHDL System Simulator. The first is a command line interface, invoked by the command *vhdlsim*. The other is a graphical interface which may be invoked by the command *vhdldbx*. The simulation tools offer many features such as waveform traces for the signals, single stepping, and break-points, which are useful to the applications developer for debugging the VHDL programs and verifying the designs. The details of these simulation commands and their features are provided in [28].

5.4 The Splash-2 synthesis environment

The Splash-2 application programs which are programmed in VHDL are synthesized to FPGA logic designs. The current Splash-2 synthesis process relies upon the Synopsys Design Compiler (version 2.2b) logic synthesis package.

The synthesis steps, starting from high-level VHDL descriptions, include conversion from VHDL to EDIF netlist format, EDIF to XNF and XNF to bitfile format which may be downloaded to the Xilinx FPGA chips.

5.4.1 VHDL to EDIF logic synthesis

The logic synthesis using the Synopsys tools may be invoked in three ways; using the script *vhdl2edif*, the command line based program *dc_shell*, and the graphical *design_analyzer*. All three interfaces take a VHDL program through three processes. The first is reading the source code (VHDL source code is analyzed). Thereafter, logic optimization or synthesis is performed. Finally the output is written to a file in EDIF netlist format.

5.4.2 EDIF to XNF conversion

The EDIF netlist produced by the Synopsys synthesis tools is translated into the Xilinx Netlist Format (XNF) by the program *edif2xnf*. In the process of translating the netlist, the program also maps the VHDL port names to physical pins, synthesizes IOB logic and global clock buffers, attempts to exploit flip-flop clock enable inputs, identifies hard macro references, and inserts the startup block to connect the global tristate and global reset signals.

5.4.3 Xilinx placement and routing functions

The placement and routing for the designs specified in XNF format is done using software tools provided by Xilinx Inc., the FPGA vendor. The tools used for placement and routing are called *ppr*. The placed and routed netlist is converted into the binary FPGA configuration format by the program *makebits*. A graphical tool for manual editing of the FPGA configurations is also available which is called *XACT*.

Details of the Synopsys design synthesis tools are available in [22, 27, 28] while information about the Xilinx technology tools mentioned in the chapter is provided in [23, 29].

Chapter 6. Histogram generation on Splash-2

The histogram, as described in Chapter 2, is an intensity distribution statistic about the image. It tells us how many pixels are associated with any grayscale value in the image [4]. Implementing a histogram on the Splash-2 is useful for displaying this intensity distribution and also for many other applications that find this data useful.

6.1 Problem partitioning

It is possible to produce histograms using a single processing element on the Splash-2 array board. However, more processing elements are needed for real-time operation. The amount of data that needs to be transferred to and from the processing element is large and a single Splash-2 processing element does not have adequate data path and memory bandwidths to meet these requirements. This problem is resolved by partitioning or dividing the task between more than one processing element.

Ideally, the task should be divided between the processing elements so that the processing elements are almost identical in their design. This reduces the programming effort and also minimizes the time to implement the design. Figure 15 shows how the problem has been partitioned for the image histogram generator application.

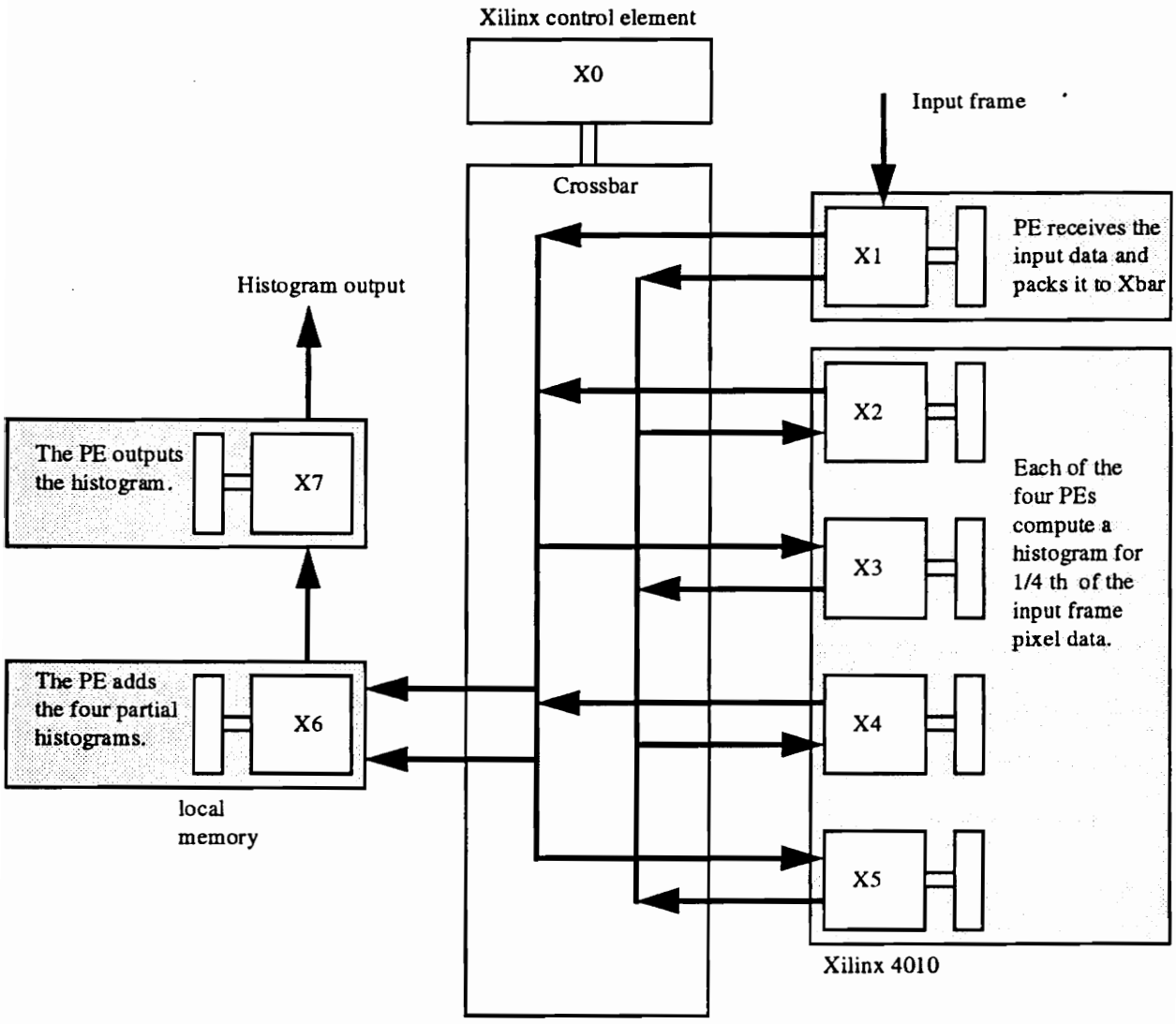


Figure 15. The design partition for image histogram generator. X1 is the input PE, X2-X5 are the PEs which compute the histogram, X6 performs a summation of the four partial histograms obtained from X2-X5, and X7 is the output PE. The control element X0 switches the crossbar configuration every clock cycle.

Data to the Splash-2 processor array is input in the form of 8-bit pixel values, one pixel being received every clock cycle. Every valid pixel data is received with the *valid bit* (bit 35) of the data path set to a '1'. The image is received in raster-scan form and the start of a new frame is indicated by *start of frame* bit set to 1.

The data received, a pixel every clock cycle, needs to be reformatted such that four pixels are available simultaneously. This is required because four processing elements (X2-X5) compute an image histogram in parallel, each using one-fourth of the image data. This task of packing four consecutive pixels together is done by the first processing element on the Splash-2 board (X1).

The first processing element on the Splash-2 board (X1) is programmed such that it accepts the input data at the system clock rate (one 8-bit pixel data every clock cycle). The data is received on its 36-bit wide left port and is packed such that four pixels are output to the crossbar every four clock cycles. The effective data rate remains unaltered. The packed data is transferred to the crossbar port every fourth clock cycle.

Four processing elements (X2-X5) are programmed to compute the histogram. The design of each is almost identical, differing only in the position of the input/output bytes on the crossbar. X2 receive the input pixels on D7-D0 of the crossbar port, X3 received them on D15-D8, while X4 and X5 receive them on D23-D16 and D31-D24 respectively. Each receives one 8-bit pixel value every four clock cycles. X2 and X4 output the histogram results on D31-D16 of the crossbar bus (in alternate clock cycles) while X3 and X5 output them on D15-D0 of the crossbar bus (also in alternate clock cycles).

Figure 16 shows the logical functions that each of X2-X5 are programmed to perform in real time. Each processing element reads the memory location whose address matches the received input pixel and increments the memory contents. The updated value

is then written back to the same location. This operation is done repeatedly in an eight clock-cycle sequence until the entire input image is received. An entry for every pixel within the image is registered in the local memory of the four processing elements. Since there are four processing elements performing the same task, each processing element effectively computes the histogram for only one-fourth of the input frame. This procedure is marked as state 1 in Figure 16. Thereafter, the processing elements enter state 2.

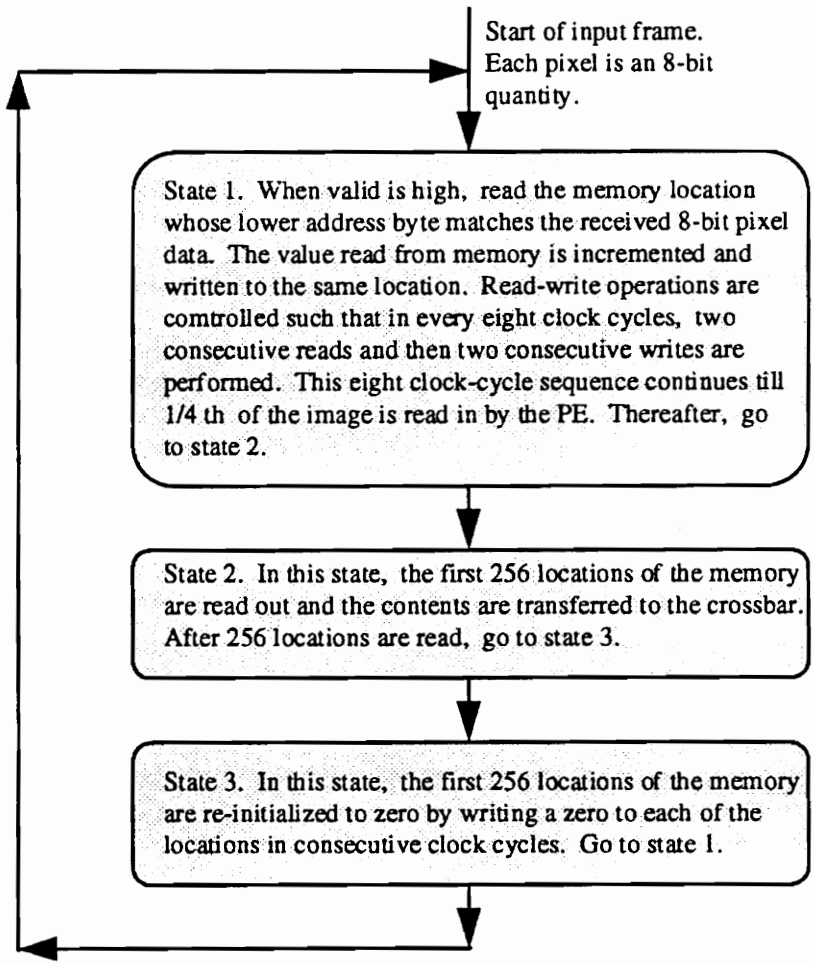


Figure 16. Logic state diagram for the processing elements X2-X5. The PEs are in State 1 while the input frame is being received. They are in State 2 and State 3 in between input frames.

In the state 2, the processing elements read out the first 256 memory locations of their local memory in a series of consecutive read operations. Every value read out is associated with the distribution of pixels in the frame according to their grayscale values, i.e. they represent the histogram for the input frame. The 256 16-bit values computed by each of the four processing elements are output to the crossbar and received by processing element X6. The PE then switches to state 3.

After the 256 values corresponding to 256 grayscale levels are read out, the local memory contents need to be re-initialized to zero so that a new frame may be processed for generation of a histogram. This is done in state 3. 256 consecutive write operations are initiated and a zero is written to each of the first 256 locations of memory.

This three state sequence is repeated by X2-X5 for every frame received.

The partial histogram values from the four PEs X2-X5 are received by the processing element X6. X6 adds four 16-bit values to form an 18-bit value. The Figure 17 shows how the processing element X6 is programmed.

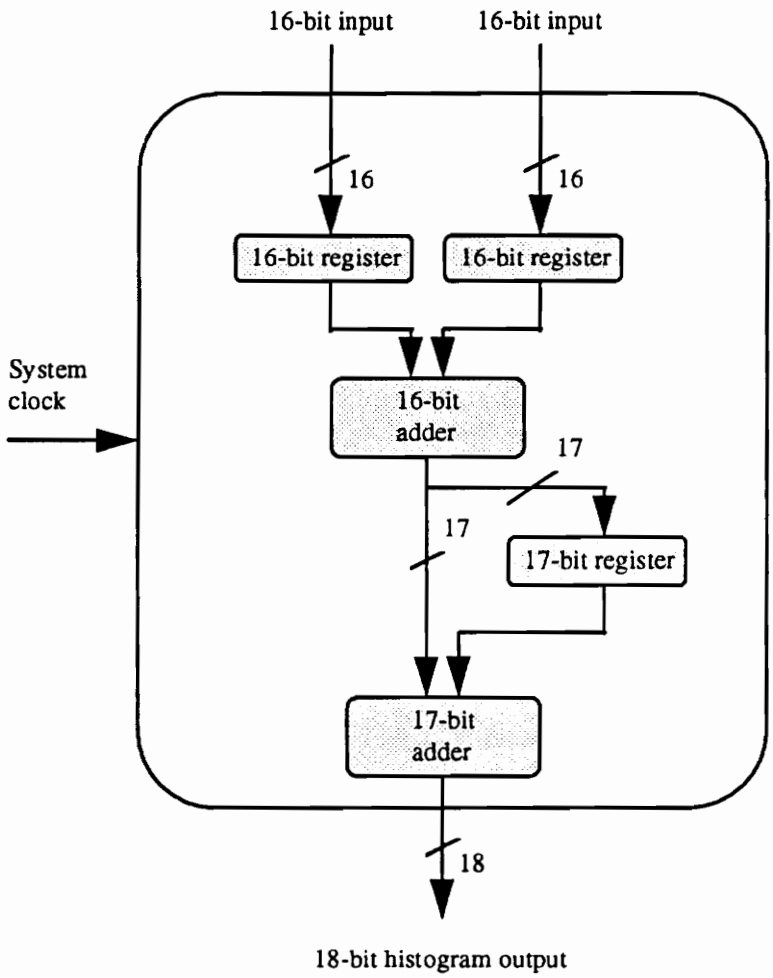


Figure 17. The logic programmed within processing element X6. The design is synchronous with the registers getting updated at only the clock edges. Two 16-bit quantities are added to form a 17-bit quantity in every clock cycle. Two consecutive 17-bit results are added to form 18-bit quantities which are the required histogram values.

PE X16 receives 256 sets of four 16_bit values to represent each gray level in the image. Thus, 256 18-bit values are generated for every frame which form the final histogram. This data is made available as the end result (to be displayed on the monitor as a bar-graph) or as input data to other applications that rely on an image histogram for their processing.

The design implemented does not have overflow problems because 16 bits are sufficient to represent the maximum count that any gray value may have, for one-fourth of the total pixel set in the image.

6.2 Results

The histogram that is generated by the above design consists of 256 18-bit values, each value representing the number of pixels in the image with a particular gray value. The first 18-bit quantity corresponds to the number of pixels with 0 grayscale value while the last corresponds to the number of pixels with a grayscale value of 255.

The design is implemented in two forms. In the first, the histogram output is provided directly as 18-bit quantities for use by other applications. In the second variation of the design, each 18-bit quantity is converted to an equivalent 9-bit quantity and this 9-bit value is displayed on the monitor. Each 9-bit value is displayed as a bar on the display monitor, the length of the bar determined by the 9-bit quantity. The maximum possible length of a bar is 512 pixels which correspond to a row on the monitor.

The histogram is displayed on the monitor as a bar-graph. (Refer to Figure 3 in Chapter 2.) The gray values with a large number of image pixel entries are displayed as long bars while the values with few pixel entries are seen as much shorter bars. The output, displayed on the monitor, is colored and the results are demonstrated with real-time frame data captured from the camera. The design works at a full system speed of 10 MHz with frames being received at the rate of 30 per second.

Figure 18 shows a 512X512 grayscale image that has been input to the Splash-2 processor. The resulting histogram computed by Splash-2 is represented as a bar-graph in

Figure 19. The height of the bars indicate the number of pixels in the image with particular gray values.

The above results have been obtained with clock speeds much lower than the real-time rate of 10 MHz. This is because the system is not designed to output the processed images to a file on the Sun SPARC-2 workstation at real-time rates. A single image stored in a file on the Sun SPARC-2 is used to input the image. The design has, however been tested in real-time, with frame rates of 30 per second, and the histograms viewed on the monitor in the form of bar graphs.

The histogram output matches that obtained by Splash-2 simulations, and by a C program which performs the same task. The C program was found to take 0.866 seconds to compute the histogram for a 512×512 image on a Sun SPARC-2 workstation. The same program took approximately 0.669 seconds on a Sun SPARC-10 workstation. In comparison, the Splash-2 takes only 0.027 seconds[1] which is a significant improvement in performance. A speedup of approximately four is obtained over a one processor design due to concurrent processing.

[1]This time is computed assuming real-time operation with clock rate of 10 MHz and image size of 512 × 512. This does not account for pipeline delays and delay to configure the PEs. For the image shown, the clock rate was much slower (software clock generated from the Sun SPARC-2). Therefore, the time to process the image was 1.44 seconds.

Figure 18. Input image for the histogram. This grayscale image is of size 512 rows $\times$ 512 columns.

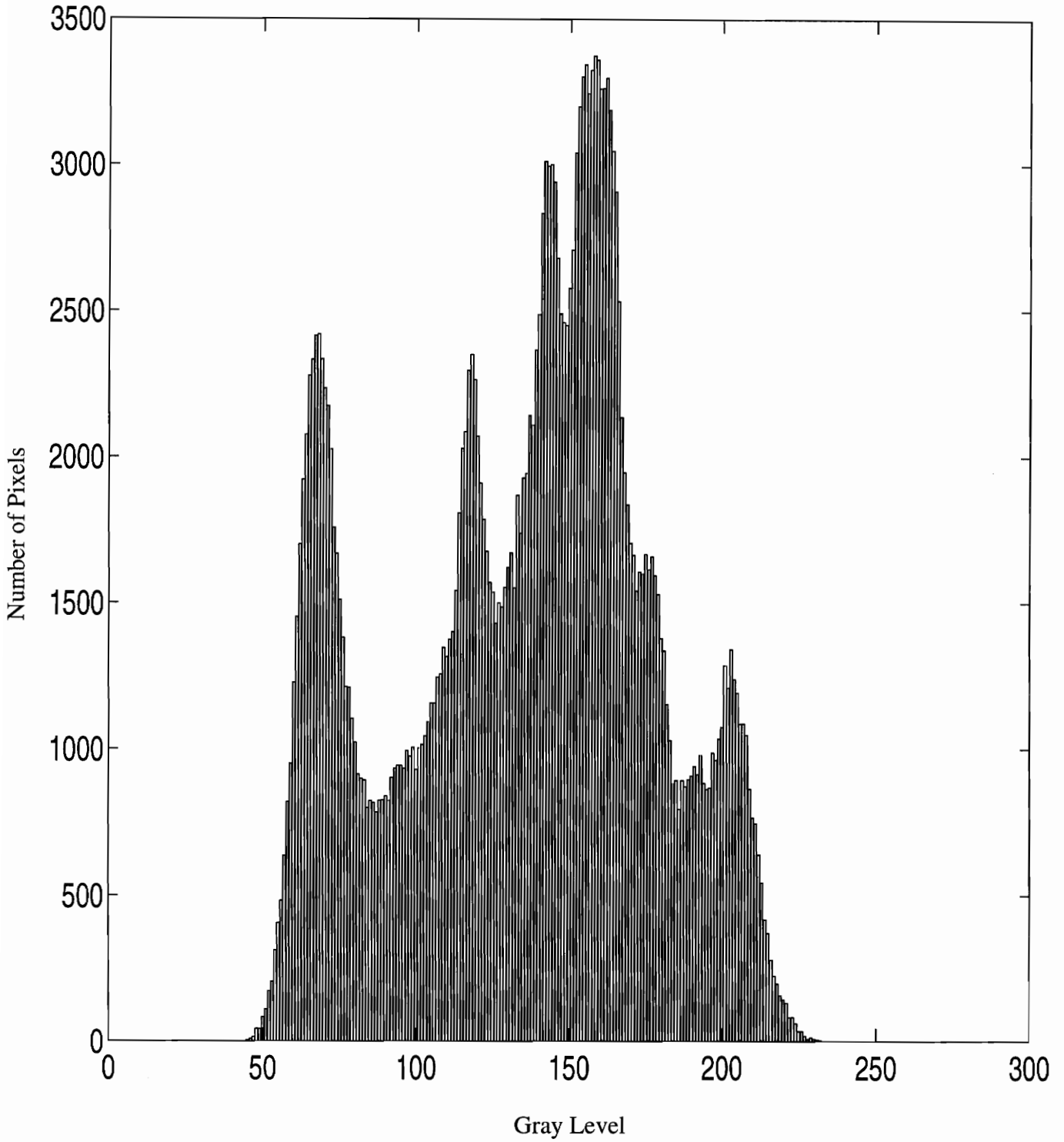


Figure 19. Histogram of image in Figure 18. The histogram computed is represented as a bar graph. Each bar represents the number of pixels in the image with a particular gray value.

Chapter 7. Median filtering on Splash-2

Real-time median filtering, using 3×3 image neighborhoods, has been implemented on Splash-2. This chapter gives the design and implementation details for this application. The design correctly processes 512×512 images. (Refer to Section 2.3 for a description of median filtering.)

7.1 Problem Partitioning

The median filter has been implemented on Splash-2 as a single board design. The design and data flow within the Splash-2 array board are shown in Figure 20. The design makes available all the pixels in a 3×3 window simultaneously so that a combinational sort can be performed on them. The median is then chosen from the sorted values.

We recall that the input image pixels are presented to Splash-2 in raster order. The Xilinx processing element X1 receives one pixel per clock cycle, where the system clock frequency is 10 MHz. The task of storing the input image is so divided that six processing elements are required for the purpose. Each needs to receive the input pixel stream at the same time. This requires the input pixels to be rearranged such that every four consecutive input pixels are packed together to form a 32-bit data word. This packing of input pixels, and transferring the resulting data stream to the crossbar, is done by the

processing elements X1 and X2. The packed input data is broadcast to X3-X8, once every four clock cycles. The effective input data rate remains unaltered.

Figure 21 shows the control logic for Xilinx processing elements X3 and X4, Figure 22 shows the same for processing elements X5 and X6 while Figure 23 shows how X7 and X8 are programmed.

Processing elements X3-X8 are responsible for storing and retrieving the image pixels in local memory. This storage is organized such that all the pixels within a 3×3 window may be accessed simultaneously. The arrangement for storage and retrieval is illustrated in Figure 24.

Let $I(i, j)$ represent the pixel value stored at row i and column j. Pixels are presented left to right for each row (j = 0 to 511), and top to bottom (i = 0 to 511). The first four pixels, $I(0, 0)$, $I(0, 1)$, $I(0, 2)$, $I(0, 3)$ are directed by X2 simultaneously to X3 and X4. $I(0, 0)$ and $I(0, 1)$ are stored in the first location of X3's memory while $I(0, 2)$ and $I(0, 3)$ are stored in the first location of X4's memory. Two pixels are packed into each 16-bit memory location. The next four pixels $I(0, 4)$ - $I(0, 7)$ are stored in similar fashion in the second locations of X3 and X4. This process iterates for one complete row, as is illustrated in Figure 22.

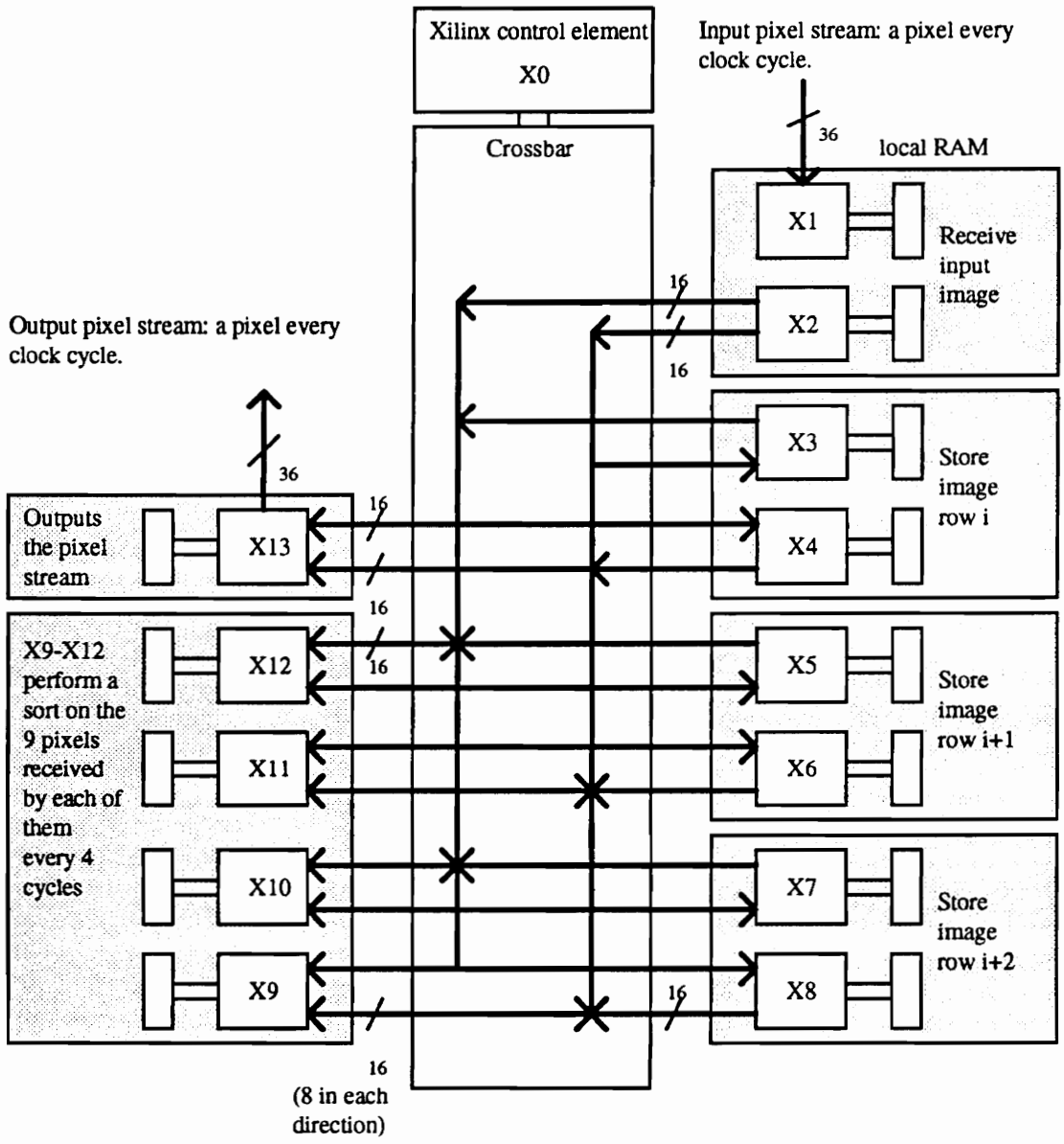


Figure 20. Problem partition for the median filtering operation. PEs X1 and X2 are programmed to receive the input pixel stream, pack it and transfer the pixels to the crossbar. X3-X8 store and retrieve the frames from memory. X9-X12 are used for sorting the pixels and computing the median. X13 outputs the frame with output valid bit. The crossbar configuration is changed dynamically every clock cycle and this is controlled by the control element X0.

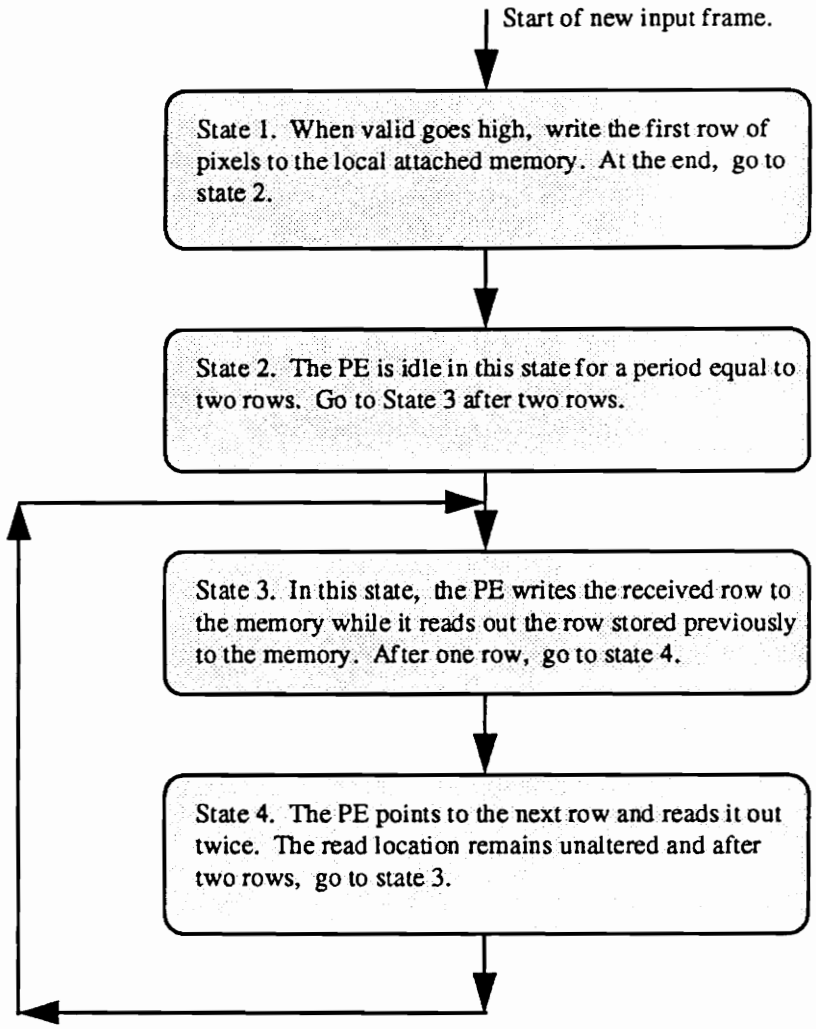


Figure 21. The logic for which the processing elements X3-X4 are programmed. This sequence of logical operations is followed for every input frame.

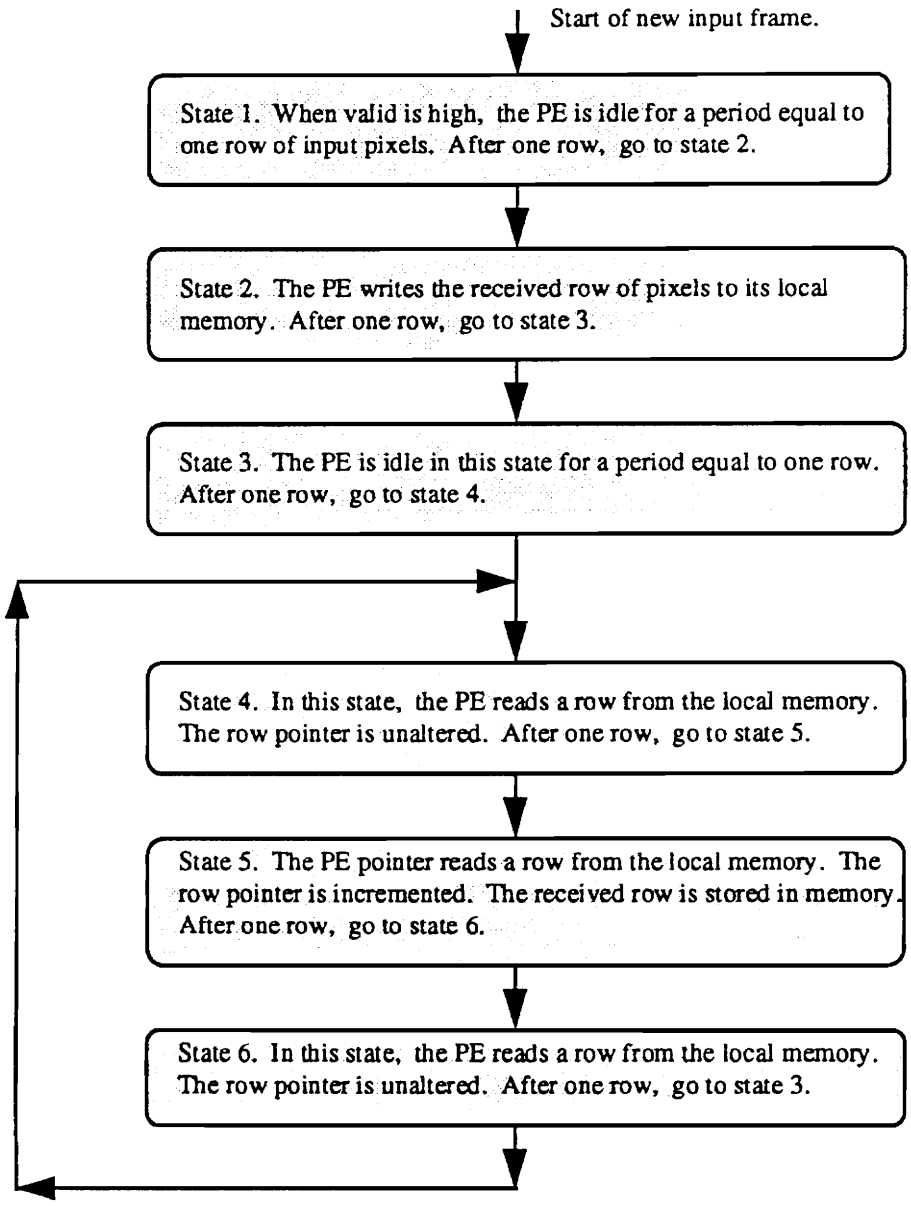


Figure 22. The logic for which the processing elements X5-X6 are programmed. This sequence of logical operations is followed for every input frame.

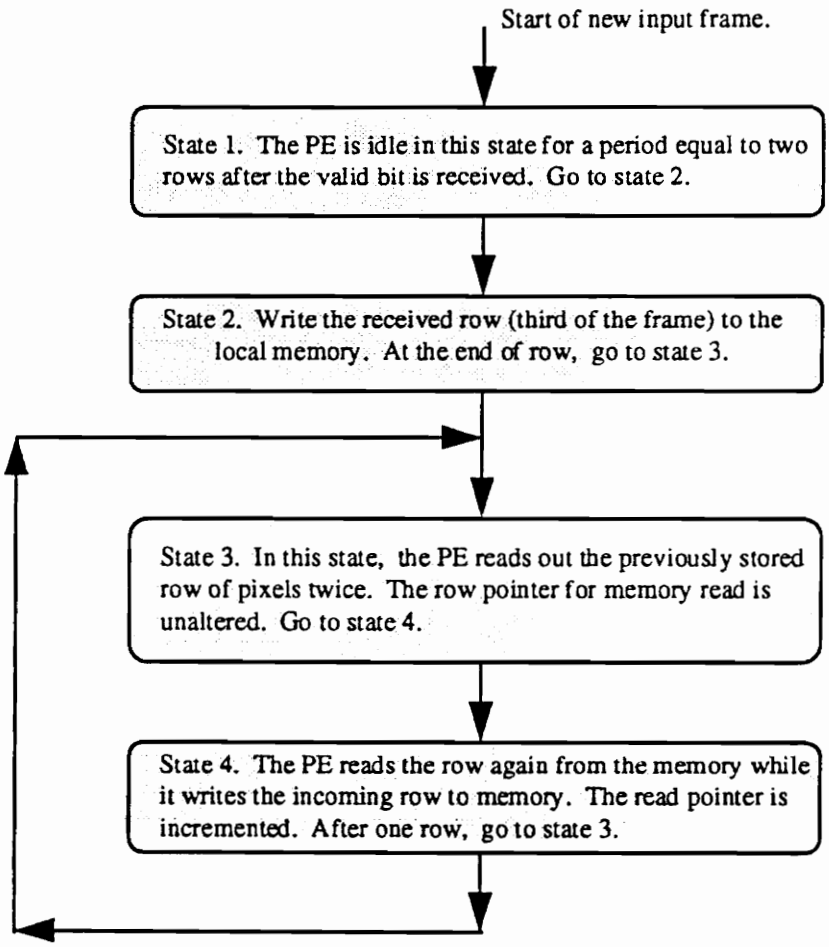


Figure 23. The logic for which the processing elements X7-X8 are programmed. This sequence of logical operations is followed for every input frame.

The second row of the image is stored similarly into the local memory of X5 and X6. The third row is stored in the memory of X7 and X8. This sequence repeats with the fourth row being stored in memories of X3 and X4, the fifth in X5 and X6, the sixth in X7 and X8 and so on, until the entire image has been captured.

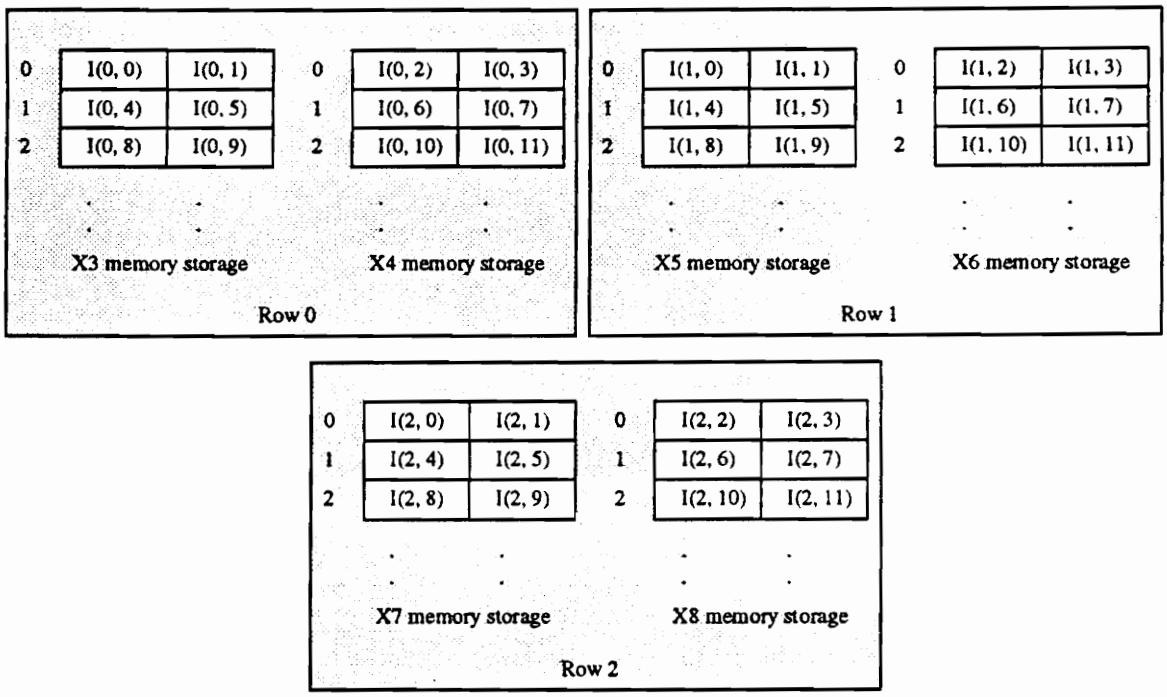


Figure 24. Arrangement to store the frames in the memory of X3-X8. Rows 1, 4, 7 etc. of the image are stored in memory of X3-X4. Rows 2, 5, 8 etc. are stored in the memory of X5-X6, while the rows 3, 6, 9 etc. are stored in memory of X7-X8. While a row is being stored in a memory of a PE, the previously stored row is retrieved for computational purposes.

The retrieval of the stored pixels begins as soon as three rows have been received.

As soon as the first three rows are stored in the memory of X3-X8, all six processing elements (X3-X8) perform a read operation from the first location of their local memory. With two pixels packed within each memory location, the six processing elements are capable of concurrently accessing a total of twelve pixels. At this point, data corresponding to a 3×4 window is available for processing. (Refer to Figure 25.) The 3 ×4 window referred to here lies within the range $i = 0$ to 2 and $j = 0$ to 3. Two complete 3×3 windows lie within this 3×4 window and may therefore be processed at once.

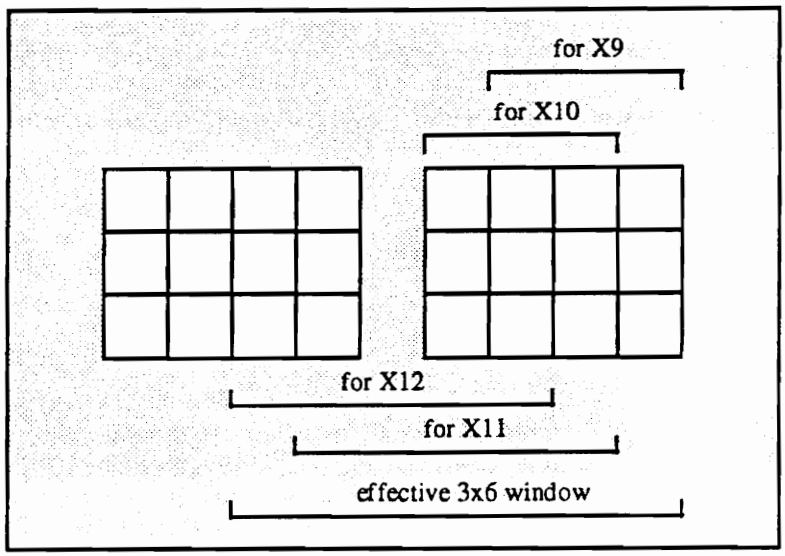


Figure 25. A 3X4 window consists of four 3X3 windows. This is possible because the two rightmost columns of pixels of the previous 3X4 window are retained in the PEs to effectively form a 3X6 window. The pixels within these four 3X3 windows are sent to X9-X12, as illustrated.

The two right-most columns of data in the window (j = 2 and 3) are stored in registers internal to the FPGAs. This storage helps form two additional 3X3 windows every time a 3X4 window is formed. This is done as follows.

In the subsequent read cycle, four new pixels for each of the first 3 rows (j = 4 to 7) are read from memory. Since two columns have been stored in internal FPGA registers, the effective window size is 3X6 instead of 3X4. (Refer to Figure 25). Four 3 X3 windows may be formed from this window and thus four median values may be computed simultaneously.

This process continues with the 3X4 window sliding 4 pixels to the right in every read operation. Once the window reaches the extreme right border of the image (j = 488 to 511), it "wraps" around in a "snake-like" fashion such that it moves one row to the bottom and starts from the leftmost border. The process of sliding right is resumed. This

procedure continues for the entire frame and the pixels within each window are delivered to the processing elements X9-X12 which process them to compute a median value.

Figure 26 shows the design for processing elements X9 and X10 while Figure 27 and Figure 28 show the designs for the processing elements X11 and X12 respectively.

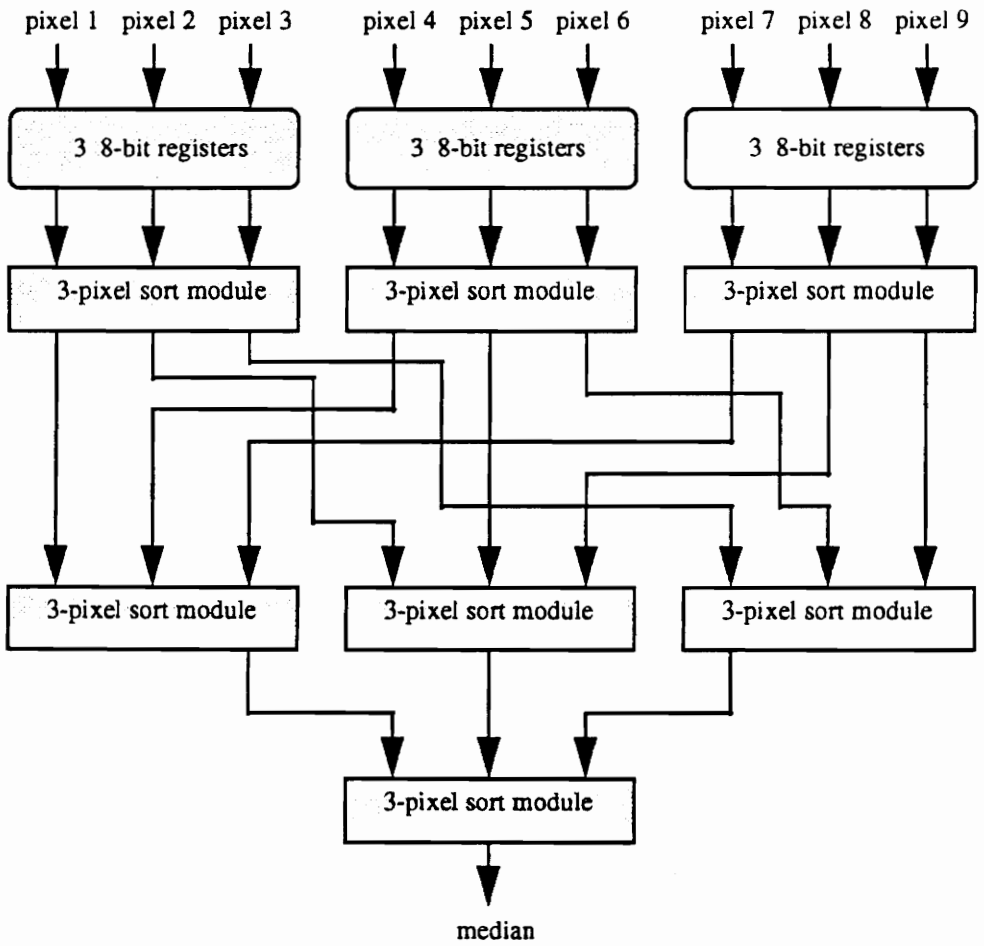


Figure 26. Logic for which the processing elements X9-X10 are programmed. The 3-pixel sort module sorts three 8-bit pixels and outputs them to the next stage. The leftmost output of each such module is the minimum value and the rightmost the maximum. All the registers in the design are synchronous with the system clock. The pixels (pixel 1 - pixel 9) are received from the crossbar. The median value is output to the crossbar.

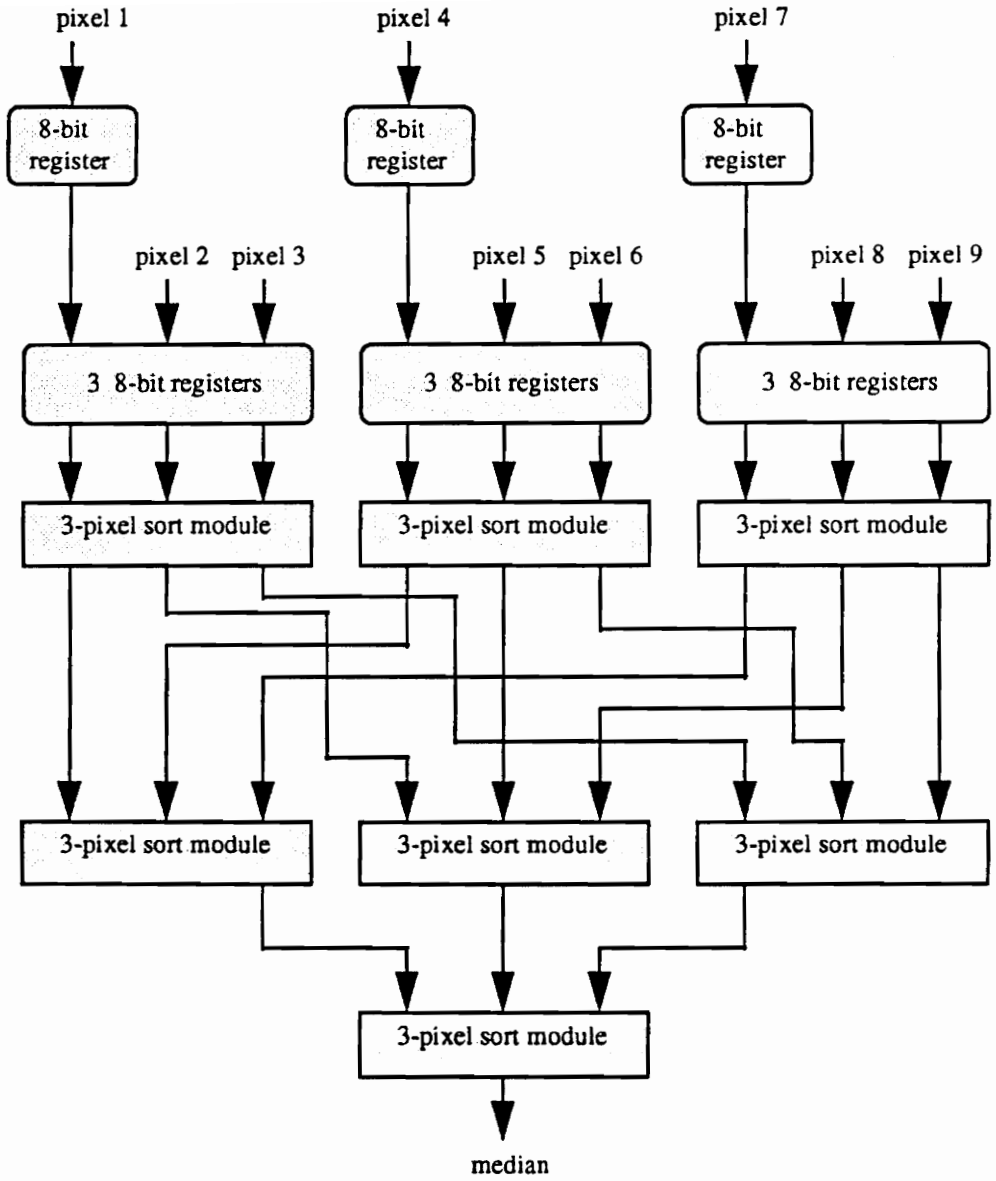


Figure 27. Logic for which the processing element X11 is programmed. The 3-pixel sort module sorts the three 8-bit pixels and outputs them to the next stage. The leftmost output of each such module is the minimum of the three pixels and the rightmost, the maximum. All the registers in the design are synchronous with the system clock. Note that three additional registers are needed to store a column of data from the previous 3×4 window.

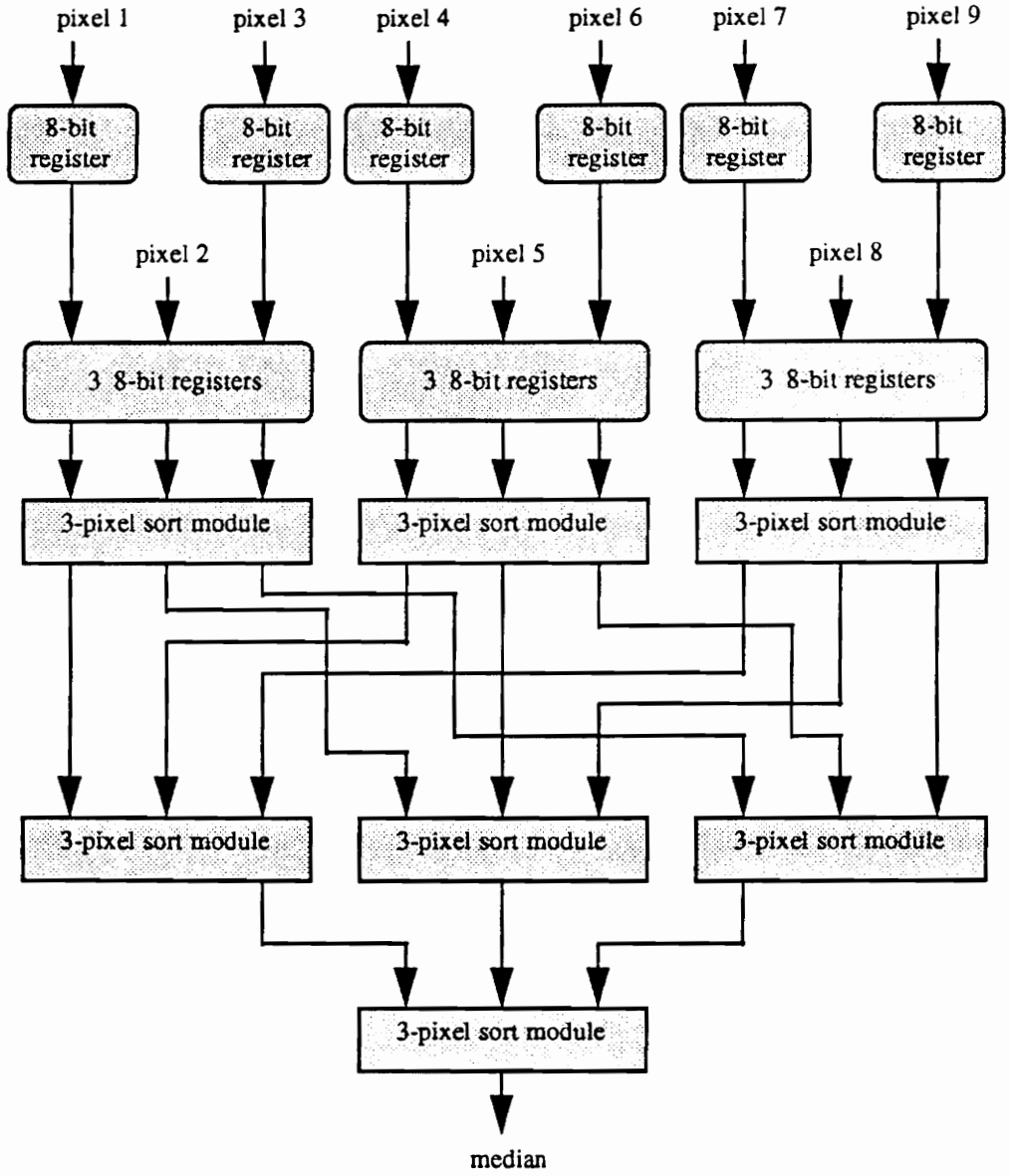


Figure 28. Logic for which the processing element X12 is programmed. The 3-pixel sort module is still the same as in Figures 24 and 25. All the registers in the design are synchronous with the system clock. Note that six additional registers are needed to store two columns of data from the previous 3×4 window.

Processing elements X9-X12 are designed to perform a combinational sort on the 8-bit pixels that are transferred to them. Each of these processing elements independently sorts its pixel set, and transfers the median value to the crossbar. X9-X12 are identical in their design, differing only in the number of registers used to store columns of pixels.

Each of X9-X12 receive nine pixels from the crossbar every four clock cycles. Three clock cycles are needed to receive these nine 8-bit pixel values. A combinational sort of the nine pixels is performed during the fourth clock cycle.

The circuit for sorting the nine 8-bit quantities is shown in Figures 26-28. The 3-pixel sort modules in each PE are circuits that sort the three pixels which they receive as input. By cascading these modules as illustrated in the Figures 26-28, all nine pixels can be sorted. The median value is then output to the crossbar.

X12 processes the left-most 3X3 window out of a set of four 3X3 windows (Refer to Figure 25) and hence has six 8-bit registers to store two columns of pixel data. X11 stores only one column of pixel data and has three 8-bit registers. X10 and X9 do not need to store any pixel values. Thus, in every four Splash-2 clock cycles, the system generates four pixels for the output which matches the input data rate. The design therefore performs in real time.

The median filter requires dense designs to be implemented within each processing element. The processing elements X3-X8, which perform the image storage and retrieval, have about 75 percent of their CLBs utilized for the task. Processing elements X9-X12, which perform the sort operations, utilize 75-80 percent of their total CLBs.

A substantial amount of data transfers are required between the processing elements on the array board, and this requires switching the crossbar configuration every clock cycle. This switching is controlled by the Xilinx element X0. X0 is programmed

such that in every clock cycle, it switches to one of the three possible crossbar configurations which are user specified.

The design does not require the entire image to be stored in memory. Only three rows are sufficient at any point of time. The latency between the input and output frames is approximately three rows. Normally, performing memory write after memory read on consecutive clock edges could lead to momentary bus contention [22]. This is avoided in this design by performing memory read/write operations on alternate clock edges only.

7.2 Results

The image shown in Figure 29 has been used to test the design on Splash-2. To demonstrate the noise cleaning ability of the median filter, impulsive noise was artificially introduced into the input image. The resulting median filtered image is seen in Figure 30. Most of the noise has been removed in the filtered image produced by the Splash-2. Careful observation reveals contours or regions of plateaus formed in the result image. This is a side effect of median filtering [4, 6].

The above results have been obtained with clock speeds much lower than the real-time rate of 10 MHz. This is because the system is not designed to output the processed images to a file on the Sun SPARC-2 workstation at real-time rates. A single image stored in a file on the Sun SPARC-2 is used to input the image. The design has, however been test run in real-time, with frame rates of 30 per second, and the processed frames viewed on the monitor.

The output image from Splash-2 hardware has been verified for correctness by comparing it with the output image obtained from Splash-2 simulations. The two images match, differing only in the pixel values at the frame edges. This difference arises because the border effect is ignored in the Splash-2 design.

Median filtering is performed in real time with a system clock speed of 10 MHz. The time to process a frame on the Splash-2 is only approximately 0.027 seconds[1] (in real-time) or 1.44 seconds (using the slower clock from the Sun), which is significantly faster than the time to perform a similar task on the Sun SPARC-2, which takes approximately 8 seconds, or on a Sun SPARC-10, which takes approximately 3.75 seconds[2].

[1]This time is computed based on the system clock speed and the processed image size. The pipeline latency and delay to configure the array board may be ignored if a sufficiently large number of frames are processed. Being a real-time system, this assumption is justified.
[2]These times do not include the times to write the output to file.

Figure 29. Input image for median filtering. The figure shows a 512×512 grayscale image which is input to the Splash-2. To demonstrate the noise cleaning effect of median filtering, noise is deliberately introduced in the input image. This is seen as black and white spots. Note that some of the contours seen in the image, especially on the face, are an artifact on the half-toning caused by the laser printer.

Figure 30. Median filtered image obtained from Splash-2. The noise that was introduced in the original image has been filtered out. This demonstrates the noise cleaning property of the median filter. Regions of flat or constant values are noticed in the image. These regions lead to boundaries which should not exist. Such artifacts are the side-effects of median filtering.

Chapter 8. Morphological filtering on Splash-2

Morphological erosion and dilation of grayscale images are discussed in Chapter 2. The implementation of these operators on Splash-2, using 3×3 structuring elements, is described here. Results of Splash-2 runs are available in the form of processed images.

8.1 Dilation

The morphological dilation implemented on Splash-2 uses 3×3 structuring elements. The problem partitioning for this task is therefore identical to that in the case of median filtering using 3×3 image neighborhoods. The arrangement for storage and retrieval of frames remains the same. Only the functionality of the processing elements X9-X12 is changed. Instead of performing a sort operation on the nine pixels received as input, these PEs compute the maximum value after adding the pixels within the 3×3 window with the structuring element. (Refer to chapter 2 for definition of dilation.)

Figure 31 shows how the problem is partitioned. Six PEs (X3-X8) are programmed to arrange and store the image pixels in a method identical to that used in the median filter. The first two processing elements (X1-X2) are also the same, used for the purpose of formatting the received input stream of pixels and transferring it to the crossbar. The control element X0 is also identical and is used for configuring the crossbar

as well as for frame requests. Four PEs (X9-X12) perform the window-based translation and compute the maximum value.

Figure 32 shows the functionality of PEs X9 and X10. Figures 33 and 34 show the same for PEs X11 and X12 respectively. Each of X9-X12 receives nine pixels (corresponding to a 3×3 window) every four clock cycles. They perform a vector addition of these pixels with the structuring element. The maximum value is then computed as shown in Figures 32-34. This maximum value is the required value for the new image and it is output to crossbar.

The CLB utilization for the processing elements X3-X8 is about 75 percent, which is the same as that in the median filter design. The processing elements X9-X12 utilize about 70-75 percent of the total CLBs in the Xilinx chips.

The structuring element is user-defined and can be programmed to be of any shape as long as the size is maintained within the maximum limit of 3×3. Each pixel within the structuring element is represented by an 8-bit quantity. The shape of the structuring element used to demonstrate this application is a disk of maximum size 3×3. One-point or two-point or any other shape structuring elements may be used.

Since the window-size is a rectangle of size 3×3, the maximum size of the structuring element is also a rectangle of size 3×3. However, this does not restrict the capability of the system to deal with effectively higher structuring element sizes. If the same design is loaded on more than one Splash-2 array board or if the frames are recirculated, morphological operations with effectively larger structuring elements may be performed [7, 8, 9, 10]. The structuring elements for morphological dilation are specified in the VHDL code. The designs need to be resynthesized and downloaded to the system for differing shapes or values of structuring elements.

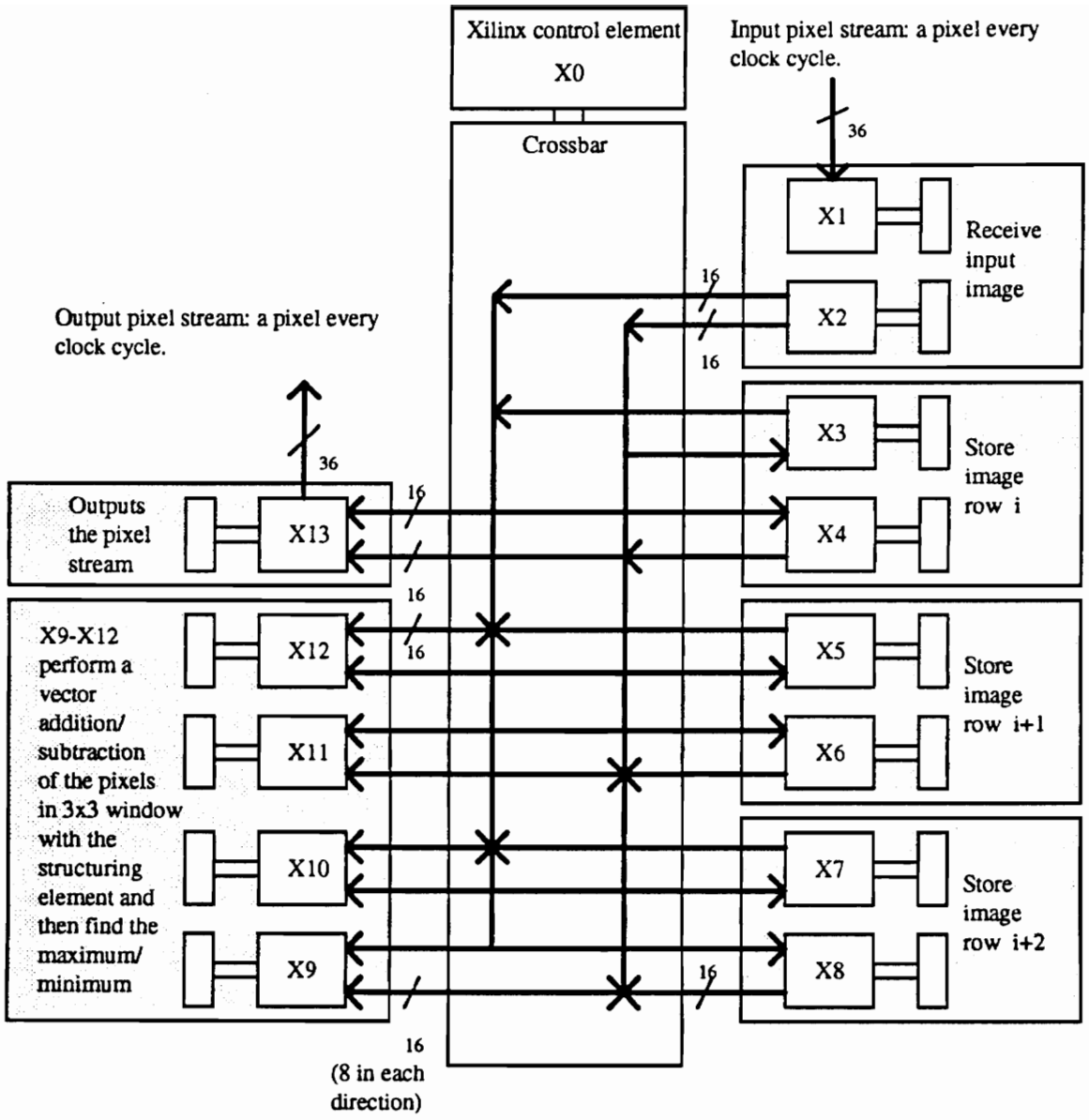


Figure 31. Problem partition for morphological dilation. The partition is identical to that of the median filter, differing only in the functionality of processing elements X9-X12. X9-X12 add the pixels in the 3X3 window with the structuring element, and the maximum value is then computed.

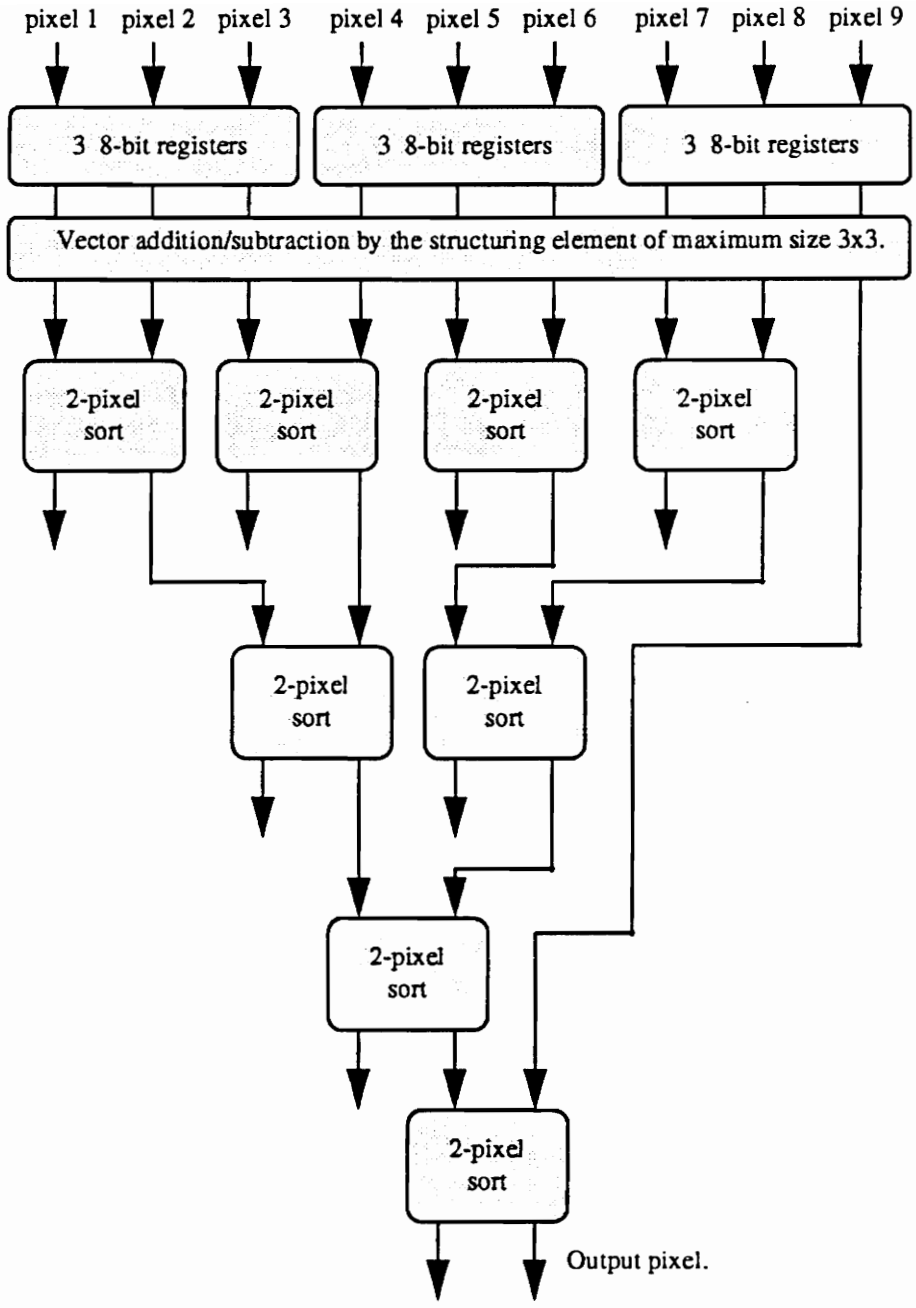


Figure 32. Logic for which X9-X10 are programmed. The pixels within the 3X3 window are vector added (dilation) or vector subtracted (erosion) with the 3X3 structuring element. The 2-pixel sort module sorts the two input pixels. The larger (dilation) or smaller (erosion) of the two values is transferred to the next stage. The other value is unused. The output pixel is the maximum (dilation) or minimum (erosion) of all the nine pixels within the 3X3 window.

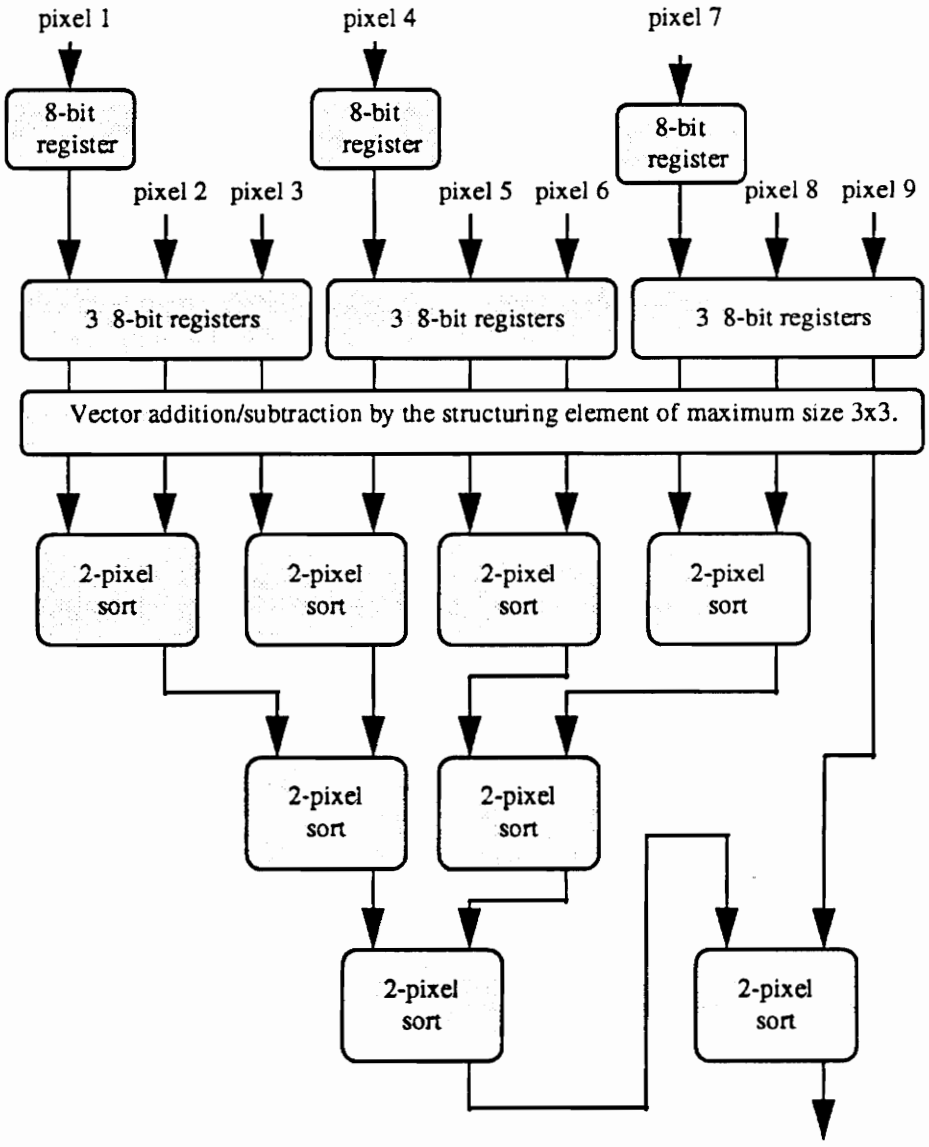


Figure 33. Logic for which X11 is programmed. The pixels within the 3×3 window are vector added (dilation) or vector subtracted (erosion) with the 3×3 structuring element. The 2-pixel sort module sorts the two input pixels. The larger (dilation) or smaller (erosion) of the two values is transferred to the next stage. The other value is unused. The output pixel is the maximum (dilation) or minimum (erosion) of all the nine pixels within the 3×3 window. Three additional registers are required to store a column of pixels from the previous 3×4 window.

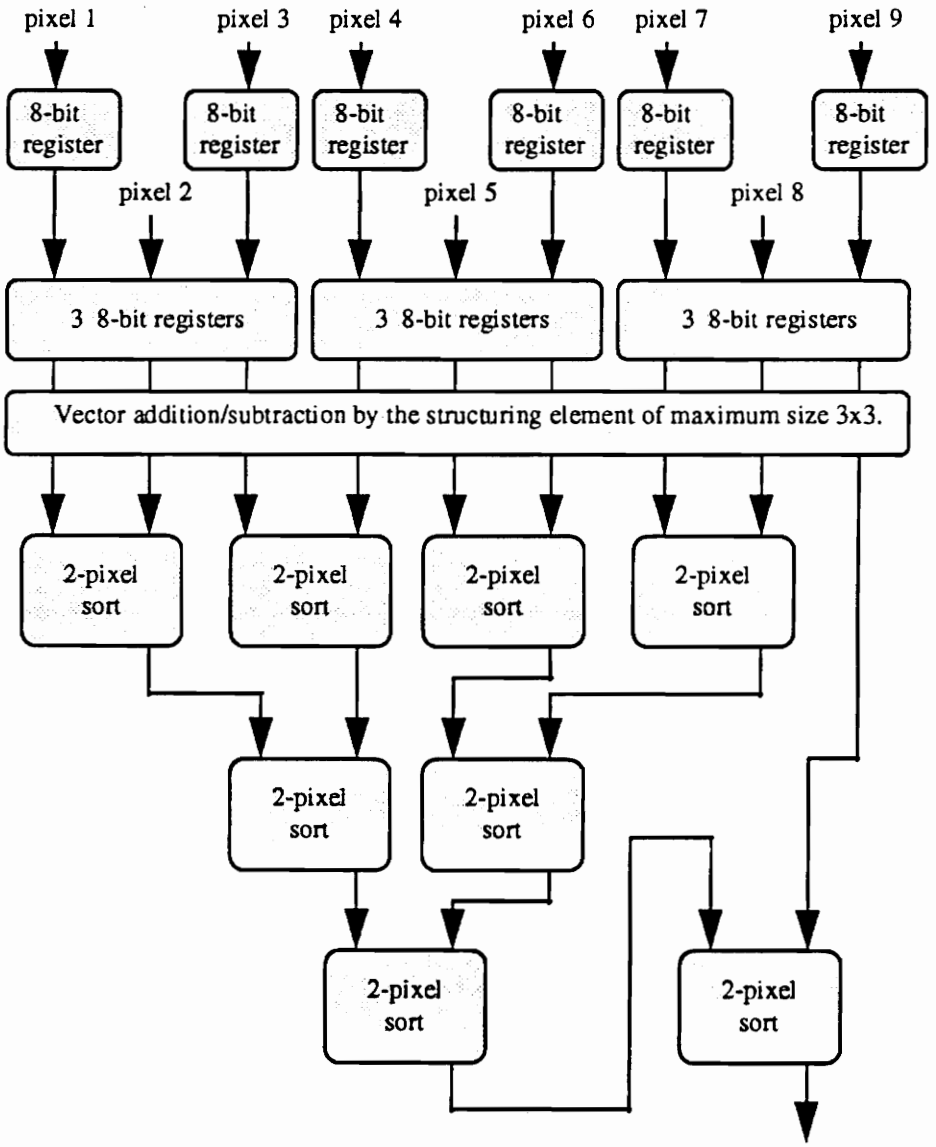


Figure 34. Logic for which X12 is programmed. The circuit implemented is the same as that in the PE X11. Instead of three, six additional registers are required to store two column of pixels from the previous 3X4 window.

8.2 Erosion

The Splash-2 implementation for morphological erosion is virtually identical to that of morphological dilation. The only difference is in the programming of PEs X9-X12. Instead of addition with the structuring element, as is done in dilation, subtraction of the structuring element is performed. Also, instead of computing the maximum value out of the nine pixel values in the 3×3 window, the minimum value is computed. The computational requirements of erosion are the same as for dilation. Figure 31 shows how the task is partitioned for implementation on the Splash-2 system. Figures 32-34 show the logic circuitry implemented within X9-X10, X11 and X12 respectively.

8.3 Results

Figure 35 shows the original grayscale image which is provided as input to the Splash-2 system. Figure 36 shows the dilated image obtained from Splash-2 using a structuring element of the form

1 1 1

1 3 1

1 1 1

This structuring element represents a disk with the center value being emphasized. Dilation by the disk structuring elements, similar to that used here, corresponds to isotropic swelling or expansion. (Refer chapter 2 for effects of morphological dilation) This may be observed in the output image shown in Figure 36.

The same input image is used to demonstrate the erosion operation. Also, the same structuring element is used. The eroded image from Splash-2 is shown in the Figure 37. The resulting image shown may be conceived as a shrinking of the original image. This property makes the transformation *antiextensive* [4].

As was done for median filtering, the morphological dilation and erosion operations are demonstrated with images processed on the Splash-2 at a much slower speed than the real-time rate of 10 MHz. Again, this is done because the Splash-2 system does not provide storing of processed images to the Sun SPARC-2 at real-time speeds. The designs have however been tested in real time with the processed images being displayed on a monitor. The frames are received, processed, and output to the monitor at a rate of 30 per second. The maximum clock rate at which the design is tested is 10 MHz.

The performance improvement for these operations, obtained from Splash-2, is significant over a Sun SPARC-2 or SPARC-10 workstation. These operations take approximately 7 seconds on a SPARC-2, 3.25 seconds on a SPARC-10 while they take only approximately 0.027 seconds[1] (in real-time), or 1.44 seconds (with slower clock from the Sun), on the Splash-2.

[1]These timings are computed based on the system clock speed and size of the image processed. It does not consider the pipeline delays and delay to configure the board. These approximations do not introduce much error if a large number of frames are processed. This is exactly what is done in a real-time vision system such as this.

Figure 35. Input image to demonstrate morphology. The 512×512 grayscale image is input to the Splash-2 to demonstrate morphological erosion and dilation.

Figure 36. Example of dilated image from Splash-2. The input image which is processed to produce this image is shown in Figure 33. The effects of "expansion" or "isotropic swelling" can be observed in the dilated image.

Figure 37. Example of eroded image from Splash-2. The input image used is shown in Figure 35. The effects of "shrinking" or "antiextension" can be observed in the eroded image.

Chapter 9. Conclusion

This thesis has described the design of high-speed image-processing tasks that have been implemented on a custom computing platform called Splash-2. The applications are capable of accepting and processing input images at 30 frames per second with a system clock rate of 10 MHz. The implementations have thereby demonstrated the effectiveness of Splash-2 for fast image processing.

The tasks have been found to run significantly faster on the Splash-2 as compared to on the Sun SPARC-2 and SPARC-10 workstations. The performance of the Splash-2 is not well characterized. The architecture is designed to have direct execution and does not have an instruction set. Therefore, it is hard to quantify the performance of Splash-2 for these tasks in terms of standard ratings such as MIPS. Significant speedup in the performance is obtained over one-processor designs due to concurrent processing of the images. Figure 38 shows the performance of the Splash-2 in terms of number of operations performed per second for each task implemented on it. These results have been obtained running the Splash-2 at a full system clock speed of 10 MHz. However, the designs could run faster than 10 MHz, thereby giving even higher performance figures those shown in Figure 38.

Though the performance levels of Splash-2 rival those of dedicated application-specific systems, the design process is not free of limitations. Many of these constraints

are those which are inherent to all such systems. Real-time image processing tasks demand a large bandwidth for data transfers and inter-processor communication. This makes the implementation of these problems more difficult than anticipated. The Splash-2 has sufficiently wide data-paths for the input/output ports, crossbar, and memory. But image-processing applications always seem to beg for slightly higher bandwidth than is available.

	Arithmetic operations per second	Memory operations per second	Boolean operations per second	Total operations per second
Histogram	2×10^7	2×10^7	10^8	1.4×10^8
Median filter	3×10^7	2×10^7	3.6×10^8	4.1×10^8
Morphological filters	1.2×10^8	2×10^7	3.6×10^8	5.0×10^8

Figure 38. Splash-2 operations per second. The numbers indicate the number of arithmetic, memory, and Boolean operations performed per second by the Splash-2 processor array for the real-time image processing tasks.

Also, the FPGA processing elements have limited hardware resources and computationally intensive problems need to be partitioned appropriately. This problem partition results in inter-processor communication overheads. The lack of adequate hardware resources to implement logic circuits can be a serious problem and often requires re-designing the problem partition. For the median filter application implemented in this research, the PEs X9-X12, which perform the combinational sort, have as much as 75-80 percent of the total CLBs utilized. For the morphological operations, this figure is approximately over 70 percent. This indicates that the above designs would be inadequate for larger computational needs, such as those required for larger image neighborhoods (such as 7×7 window sizes or higher).

The research presented here has illustrated the effectiveness of reconfigurable FPGA-based computing for high-speed image processing applications. The reconfigurable nature of Splash-2 provides the performance of application-specific hardware, while preserving the general-purpose nature of being able to accommodate a variety of tasks. Designs can be altered easily and two or more tasks can be pipelined to operate concurrently. These features demonstrate the potential of adaptive computing platforms for high performance applications.

Bibliography

[1] J. M. Arnold, D. A. Buell, and E. G. Davis, "Splash 2," in *Fourth Annual ACM Symposium on Parallel Algorithms and Architectures*, San Deigo, CA, pp. 316-322, 1992.

[2] S. Brown, R. Francis, J. Rose, and Z. Vizanesic, *Field-Programmable Gate Arrays*, Kluwer Academic Publishers, Boston, MA, 1992.

[3] Robert C. Vogt, *Automatic Generation of Morphological Set Recognition Algorithms*, Springer-Verlag, New York, 1989.

[4] Robert M. Haralick and Linda G. Shapiro, *Computer and Robot Vision*, vol. I, Addison-Wesley Publishing Company, June 1992.

[5] Charles R. Giardina and Edward R. Dougherty, *Morphological Methods in Image and Signal Processing*, Prentice Hall, New Jersey, 1988.

[6] N. C. Gallagher, Jr., and G. L. Wise, "A theoretical Analysis of the Properties of Median Filters," *IEEE Transactions on Acoustics, Speech, and Signal Processing*, vol. ASSP-29, pp. 1136-1141, 1981.

[7] J. Serra, *Image Analysis and Mathematical Morphology*, Academic Press, London, 1982.

[8] S. R. Sternberg, "Grayscale Morphology," *Computer Vision, Graphics, and Image Processing*, vol. 35, pp. 333-355, September 1986.

[9] X. Zhuang and R. M. Haralick, "Morphological Structuring Element Decomposition," *Computer Vision, Graphics, and Image Processing*, vol. 35, pp. 370-382, 1986.

[10] Jeffry M. Seaton and A. Lynn Abbott, "A Comparison of Multiresolution Morphological and Laplacian Techniques for Automated Inspection," SPIE, *Applications of Artificial Intelligence X: Machine Vision and Robotics*, vol. 1708, pp. 789-800, 1992.

[11] John P. Hayes, *Computer Architecture and Organization*, McGraw-Hill, New York, 1988.

[12] J. A. Trotter and W. R. Moore, "Mesh: an architecture for image processing," in *Parallel Processing for Computer Vision and Display*, P. M. Dew, Addison-Wesley Publishing Company, 1989.

[13] R. M. Lougheed, D. L. McCubbrey, S. R. Sternberg, "Cytocomputers: Architectures for Parallel Image Processing," *Proceedings, Workshop Picture Data Descr. and Management*, Pacific Grove, CA, pp. 281-286, August 27-28, 1980.

[14] A. L. Abbott, R. M. Haralick, and X. Zhuang, "Pipeline architectures for morphologic image analysis," *Machine Vision and Applications*, vol. 1, no. 1, pp. 23-40, 1988.

[15] Datacube Incorporated, "Maxvideo 20," Tech. Rep. DS599-1.0, January 1991.

[16] *VC-2 Computer Vision Chip*. Needham, Massachusetts, 1992.

[17] H. J. Seigel, "PASM: a reconfigurable multimicrocomputer system for image processing," in *Languages and Architectures for Image Processing*, M. J. B. Duff and S. Levialdi (Eds.), Academic Press, London, 1981.

[18] K. E. Batcher, "Design of a massively parallel processor," *IEEE Transactions on Computers*, vol. 29, pp. 836-840, 1980.

[19] M. J. B. Duff and T. J. Fountain, *Cellular Logic Image Processor (CLIP)*, Academic Press, 1986.

[20] C. C. Weems, S.P. Levitan, A. R. Hanson, E. M. Riseman, D. B. Shu, and J. G. Nash, "The Image Understanding Architecture," *Intl. Journal of Computer Vision*, vol. 2, pp. 251-282, Kluwer Academic Publishers, Boston, MA, 1989

[21] R. Cucchiara, L. Di Stefano, G. Neri and Salmon Cinotti, "A programmable image processing subsystem for real time applications," in *Proceedings, Eighth IASTED Int. Symp. on Applied Informatics*, Innsbruck, pp. 44-47, February 1990.

[22] J. M. Arnold and M. A. McGarry, "Splash 2 Programmer's Manual," Supercomputing Research Center, Tech. Rep. SRC-TR-93-107, Bowie, Maryland, 1993.

[23] Xilinx Inc., *The Programmable Gate Array Data Book*, CA, 1991.

[24] Randolph E. Harr and Alec G. Stanculescu, *Applications of VHDL to circuit design*, Kluwer Academic Publishers, Boston, 1991.

[25] IEEE, *IEEE Standard VHDL Language Reference Manual*, New York, March 31, 1988.

[26] James R. Armstrong and F. Gail Gray, *Structured Logic Design with VHDL*, PTR Prentice Hall, New Jersey, 1993.

[27] Synopsys Inc., *VHDL Compiler Reference Manual*, Version 3.0, November 1992.

[28] Synopsys Inc., *VHDL System Simulator Core Programs Manual*, Version 3.0, December 1992.

[29] Xilinx Inc., *XC4000 Design Implementation Reference Guide*, 1991.

Appendix A. VHDL entity declarations

The VHDL entity declarations for the user programmable components of the Splash-2 system are given below.

A.1. Xilinx_Processing_Part Entity Declaration

```
ENTITY Xilinx_Processing_Part IS
        Port (  XP_Left                 : inout DataPath;
                XP_Right                : inout DataPath;
                XP_Xbar                 : inout DataPath;
                XP_Xbar_EN_L            : out Bit_Vector(4 downto 0);
                XP_Clk          : in Bit;
                XP_Int                  : out Bit;
                XP_Mem_A                : inout MemAddr;
                XP_Mem_D                : inout MemData;
                XP_Mem_RD               : inout Bit;
                XP_Mem_WR               : inout Bit;
                XP_Mem_Disable          : in Bit;
                XP_Broadcast            : in Bit;
                XP_Reset                : in Bit;
                XP_HS1, XP_HS2          : inout RBit3;
                XP_GOR_Result           : inout RBit3;
                XP_GOR_Valid            : inout RBit3;
                XP_LED                  : out Bit);
```

```
END Xilinx_Processing_Part;
```

A.2. Xilinx_Control_Part Entity Declaration

```
ENTITY Xilinx_Control_Part IS
        Port (  X0_SIMD             : inout DataPath;
                X0_XB_Data          : inout DataPath;
                X0_Mem_A            : inout MemAddr;
                X0_Mem_D            : inout MemData;
                X0_Mem_RD           : inout Bit;
                X0_Mem_WR           : inout Bit;
                X0_Mem_Disable      : in Bit;
                X0_GOR_Result_In    : inout RBit3_Vector(1 to 16);
                X0_GOR_Valid_In     : inout RBit3_Vector(1 to 16);
                X0_GOR_Result       : out Bit;
                X0_GOR_Valid        : out Bit;
                X0_Clk              : in Bit;
                X0_XBar_Set         : out Bit_Vector(0 to 2);
                X0_X16_Disable      : out Bit;
                X0_XBar_Send        : out Bit;
                X0_Int              : out Bit;
                X0_Broadcast_In     : in Bit;
                X0_Broadcast_Out    : out Bit;
                X0_Reset            : in Bit;
                X0_HS1, X0_HS2      : inout RBit3;
                XP_LED              : out Bit);
END Xilinx_Control_Part;
```

Appendix B. Entity port descriptions

A detailed summary of each of the ports of the Xilinx Processing Element and Xilinx Control Element is given below.

B.1 Xilinx_Processing_Part

XP_Left : inout DataPath;

XP_Left is the 36-bit data path to the left (lower numbered) neighbor chip. XP_Left of chip X1 is either connected to the SIMD Bus if it's on board number 1, or to the previous board if it is on board 2 or greater. XP_Left and XP_Right together form the links in the linear data path.

XP_Right : inout DataPath;

XP_Right is the 36-bit data path to the right (higher numbered) neighbor chip. XP_Right of chip X16 is connected to the R-Bus if the board number matches the values of the Size bus. XP_Right of chip X16 is always connected to X1 of the next board.

XP_XBar : inout DataPath;

XP_XBar is the 36-bit data path between the Xilinx PE and the central crossbar.

XP_Xbar_EN_L : out Bit_Vector(4 downto 0);

XP_XBar_EN_L is the collection of output enables for each of the five crossbar ports. When an enable bit is low (logic '0'), the crossbar drives the corresponding byte of the data path. When a bit is high (logic '1'), the crossbar may receive data on the corresponding byte. The correspondence between enable bits and data paths is as follows:

XP_XBar_EN_L(0)	XP_XBar(7 downto 0)
XP_XBar_EN_L(1)	XP_XBar(15 downto 8)
XP_XBar_EN_L(2)	XP_XBar(23 downto 16)
XP_XBar_EN_L(3)	XP_XBar(31 downto 24)
XP_XBar_EN_L(4)	XP_XBar(35 downto 32)

XP_Clk : in Bit;

XP_Clk is the global Splash system clock. The duty of this clock is not guaranteed, so application programs should only use the rising edge.

XP_Int : out Bit;

XP_Int is the interrupt output signal. The interrupt signals from all of the 17 Xilinx PEs are combined with a 17 bit mask value to form the board interrupt signal to the Interface Board.

XP_Mem_A : inout MemAddr;

XP_Mem_A is the 18 bit address to the external memory.

XP_Mem_D : inout MemData;

XP_Mem_D is the 16 bit bi-directional data bus to and from the external memory.

XP_Mem_RD : inout Bit;

XP_Mem_RD is the external memory read signal. It is asserted by the Xilinx PE synchronous with the address to initiate a memory read operation. Valid data is available on the clock edge following the assertion of the XP_Mem_RD.

XP_Mem_WR : inout Bit;

XP_Mem_WR is the external memory write signal. It is asserted by the Xilinx PE synchronous with the address and data to perform a memory write operation.

XP_Mem_Disable : in Bit;

XP_Mem_Disable is a system level input to the Xilinx PE which, when asserted, causes the Xilinx chips to tristate all its I/O pads. This port should not be used by application programs.

XP_Broadcast : in Bit;

XP_Braodcast is a common input signal to all of the Xilinx Processing Parts. It is driven by chip X0.

XP_Reset : in Bit;

XP_Reset is a system level input to the Xilinx PE which, when asserted, causes the Xilinx chip to reset the state of all its internal flip flops. This port should not used by application programs.

XP_HS1, XP_HS2 : inout RBit3;

XP_HS1 and XP_HS2 are the signals of the two handshake registers. Signal XP_HS1 is common to all 17 Xilinx PEs, while XP_HS2 is an independent signal to each chip.

XP_GOR_Result : inout RBit3;

XP_GOR_Result is a bi-directional signal connected to chip X0 conventionally used to perform a "global OR" among all of the Xilinx PEs.

XP_GOR_Valid : inout RBit3;

XP_GOR_Valid is a bi-directional signal connected to chip X0 conventionally used to perform a "global OR" among all of the Xilinx PEs.

XP_LED : out bit;

XP_LED is an active high output which controls the corresponding LED on the back of the array board.

There are a number of pre-defined architectures for Xilinx_Processing_Part available in the Splash2 library, including a null design for applications which use less than full 16 PEs. The set of pre-defined architectures is as follows:

Blank

This is completely passive, and should be inserted for Xilinx PEs not used by an application.

Left_to_Right

This architecture moves 36 bits of data from the left side (XP_Left) to the right (XP_Right) with a two cycle pipeline delay. All other ports are passive.

Left_to_XBar

This architecture moves 36 bits of data from the left side (XP_Left) to the crossbar (XP_XBar) with a two cycle pipeline delay. All other ports are passive.

XBar_to_Right

This architecture moves 36 bits of data from the crossbar (XP_XBar) to the right side (XP_Right) with a two cycle pipeline delay. All other ports are passive.

B.2 Xilinx_Control_Part

X0_SIMD : inout DataPath;

X0_SIMD is the input from the common SIMD bus.

X0_XB_Data : inout DataPath;

X0_XB_Data is the output to the crossbar. This path is shared with the XP_XBar data path of chip X16.

X0_Mem_A : inout MemAddr;

X0_Mem_A is the 18 bit address to the external memory.

X0_Mem_D : inout MemData;

X0_Mem_D is the 16 bit bi-directional data bus to and from the external memory.

X0_Mem_RD : inout Bit;

X0_Mem_RD is the external memory read signal. It is asserted by the Xilinx PE synchronous with the address to initiate a memory read operation. Valid data is available on the clock edge following the assertion of X0_Mem_RD.

X0_Mem_WR : inout Bit;

X0_Mem_WR is the external memory write signal. It is asserted by the Xilinx PE synchronous with the address and data to perform a memory write operation.

X0_GOR_Result_In : inout RBit3_Vector(1 to 16);

X0_GOR_Result_In is the bi-directional vector of XP_GOR_Result signal from each of the Xilinx_Processing_Parts on the board.

X0_GOR_Valid_In : RBit3_Vector(1 to 16);

X0_GOR_Valid_In is the bi-directional vector of XP_GOR_Valid signal from each of the Xilinx_Processing_Parts on the board.

X0_GOR_Result : out Bit;

X0_GOR_Result is the output from X0 to the wire-ORed GOR Result signal of the backplane.

X0_GOR_Valid : out Bit;

X0_GOR_Valid is the output from X0 to the wire-ORed GOR Valid signal of the backplane.

X0_Clk : in Bit;

X0_Clk is the global Splash system clock.

X0_XBar_Set : out Bit_Vector(0 to 2);

X0_XBar_Set is the three bit crossbar configuration selector. When X0 drives a value of "000", the crossbar selects its pre-loaded configuration 0, "001" selects configuration 1, and so on.

X0_X16_Disable : out Bit;

When asserted, X0_X16_Disable causes Xilinx PE X16 to be isolated from the crossbar, and enables X0_XB_Data to drive into the crossbar.

X0_Int : out Bit;

X0_Int is the interrupt output signal. The interrupt signals from all of the 17 Xilinx PEs are combined with a 17 bit mask value to form the board interrupt signal to the Interface Board.

X0_Broadcast_In : in Bit;

X0_Broadcast_In is the broadcast signal input from the Interface Board.

X0_Broadcast_Out : out Bit;

X0_Broadcast_Out is the common broadcast signal to each of the 16 Xilinx PEs.

X0_Reset : in Bit;

X0_Reset is a system level input to the Xilinx which, when asserted, causes the Xilinx chip to reset the state all internal flip flops. This port should not be used by the application programs.

X0_HS1, X0_HS2 : inout RBit3;

X0_HS1 and X0_HS2 are the signals of the two handshake registers. Signal X0_HS1 is common to all 17 Xilinx PEs. X0_HS2 is an independent signal to each chip.

X0_LED : out Bit;

X0_LED is an active high output which controls the X0 LED on the back of the board.

There is one pre-defined architecture for Xilinx_Control_Part in the Splash2 library:

Blank

This architecture is completely passive, and should be inserted whenever chip X0 is not used by an application.

Appendix C. Splash-2 VHDL files

The sample VHDL files to execute a task on the Splash-2 are provided. Only the files modified as part of this research work are included. The complete set of files are available in the *Splash-2 Programmer's Manual, IDA Supercomputing Research Center, Bowie, MD, December 1992.*

Unless mentioned otherwise, the sample files included are the actual files used in the image histogram generator application discussed in Chapter 5.

C.1. Sample Configuration file

```
-- Sample configuration file for Splash-2 System (4/24/92)

library S2Board, Interface;

configuration TOP of Splash_System is
 for Structure
  for IFACE: Interface_Board
   use entity interface.Interface_Board(structure)

-- The following Generic Map specifies the path names for the input
-- and output data files, and the frequency of the Splash Clock (in MHz).
-- The null file names ("") are the default and need not be given
```

```
        Generic Map (input_file1  => "xinput",
--                   input_file2  => "",
--                   J3_file1     => "",
                     output_file1 => "xoutput.sim",
--                   output_file2 => "",
                     File_Type    => Hex,      -- one of (Bin, Binary, Hex)
                     Clock_Freq           => 20);
    for Structure
        for all: XL
-- Select the architecture for the Interface board XL Xilinx part here
          use entity interface.XL(Valid);
        end for;                          -- all: XL
        for all: XR
-- Select the architecture for the Interface board XR Xilinx part here
          use entity interface.XR(Valid);
        end for;                          -- all: XR
    end for;                              -- Structure of IFACE
  end for;                                -- IFACE: Interface_Board

  for Splash: Splash2_Boards
    use entity s2board.Splash2_Boards(Structure)

-- Set the number of Splash 2 boards here
    Generic Map (Number_Of_Boards => 1);

    for Structure
        for SBOARDS

-- If we had a single behavioral model (with architecture name "Behavior")
-- for the entire set of Splash boards, the configuration would be:
--
--          for BD : Splash2_Board
--                use entity work.Splash2_Board(Behavior);
```

```
--          end for;
--
-- To completely configure a one board system, the configuration is:

          for BD : Splash2_Board
         use entity S2Board.Splash2_Board(Structure);
           for Structure

-- To use the crossbar, we must specify a crossbar config file as follows
            for XBAR : Splash_Crossbar
                use entity S2Board.Splash_Crossbar(Behavior)
                 generic map (
             Config_File => "xcrossbar"
            );
            end for;                    -- XBAR: Splash_Crossbar

-- This design does not make use of X0, so just specify "Blank" architecture

            for all : Xilinx_Control_part
                use entity work.Xilinx_Control_Part(chipa0);
            end for;                    -- all: Xilinx_Control_Part


            for XPARTS (1)
                for all : Xilinx_Processing_part
--                use entity S2Board.Xilinx_Processing_Part(Left_to_Right)
                 use entity work.Xilinx_Processing_Part(chipa1);
                end for;                -- Xilinx_Processing_Part
                for all : Memory_Part
                 use entity S2Board.Memory_Part(Dynamic)
                 Generic Map(Load_File => "xmemory1");
                end for;                -- Memory_Part
            end for;                    -- XPARTS(1)
```

```
        for XPARTS (2)
            for all : Xilinx_Processing_part
--           use entity S2Board.Xilinx_Processing_Part(Left_to_Right)
             use entity work.Xilinx_Processing_Part(chipa2);
            end for;              -- Xilinx_Processing_Part
            for all : Memory_Part
             use entity S2Board.Memory_Part(Dynamic)
             Generic Map(Load_File => "xmemory1");
            end for;              -- Memory_Part
        end for;                  -- XPARTS(2)

        for XPARTS (3)
            for all : Xilinx_Processing_part
--           use entity S2Board.Xilinx_Processing_Part(Left_to_Right)
             use entity work.Xilinx_Processing_Part(chipa3);
            end for;              -- Xilinx_Processing_Part
            for all : Memory_Part
             use entity S2Board.Memory_Part(Dynamic)
             Generic Map(Load_File => "xmemory1");
            end for;              -- Memory_Part
        end for;                  -- XPARTS(3)

        for XPARTS (4)
            for all : Xilinx_Processing_part
--           use entity S2Board.Xilinx_Processing_Part(Left_to_Right)
             use entity work.Xilinx_Processing_Part(chipa4);
            end for;              -- Xilinx_Processing_Part
            for all : Memory_Part
             use entity S2Board.Memory_Part(Dynamic)
             Generic Map(Load_File => "xmemory1");
            end for;              -- Memory_Part
        end for;                  -- XPARTS(4)

        for XPARTS (5)
```

```
            for all : Xilinx_Processing_part
--           use entity S2Board.Xilinx_Processing_Part(Left_to_Right)
             use entity work.Xilinx_Processing_Part(chipa5);
            end for;                -- Xilinx_Processing_Part
            for all : Memory_Part
             use entity S2Board.Memory_Part(Dynamic)
             Generic Map(Load_File => "xmemory1");
            end for;                -- Memory_Part
          end for;                  -- XPARTS(5)

          for XPARTS (6)
            for all : Xilinx_Processing_part
--           use entity S2Board.Xilinx_Processing_Part(Left_to_Right)
             use entity work.Xilinx_Processing_Part(chipa6);
            end for;                -- Xilinx_Processing_Part
            for all : Memory_Part
             use entity S2Board.Memory_Part(Dynamic)
             Generic Map(Load_File => "xmemory1");
            end for;                -- Memory_Part
          end for;                  -- XPARTS(6)


-- The remaining PEs (7 to 16) are not used, so we specify the
-- "Blank" architectures for the Xilinx chips and the memories

          for XPARTS (7 to 16)
            for all : Xilinx_Processing_part
             use entity work.Xilinx_Processing_Part(Left_To_Right);
            end for;                -- XPARTS(7 to 16)
            for all : Memory_Part
             use entity S2Board.Memory_Part(Blank);
            end for;                -- Memory_Part
          end for;                  -- XPARTS (7 to 16)
        end for;                    -- Structure (of Splash2_Board)
```

```
        end for;                    -- BD: Splash2_Board
      end for;                          -- SBOARDS (0)
    end for;                        -- Structure (of SBoards)
  end for;                          -- Splash: Splash2_Boards
 end for;                           -- Structure (of Splash_System)
end TOP;
```

This configuration file, called "config.vhd", defines the configuration of the processing elements on the Splash-2. The design shown is configured as a single board design. It uses the first six Xilinx processing elements X1-X6. The VHDL descriptions of the designs for each of these six PEs are given in the files which are user specified. The remaining PEs X7-X16 are used in left_to_right configuration. The files which contain the input data to the Splash-2, the output data from the Splash-2, the crossbar, and the initial memory contents, are specified in this file.

C.2. Sample Crossbar Configuration File

```
configuration 0
1 1
2 1 2 2 1 1
3 1 1 1 3 3
4 1 4 4 1 1
5 1 1 1 5 5
6 1 2 2 3 3
7 7
8 8
9 9
10 10
11 11
12 12
13 13
14 14
```

```
15 15
16 16

configuration 7
1 1
2 1 2 2 1 1
3 1 1 1 3 3
4 1 4 4 1 1
5 1 1 1 5 5
6 1 4 4 5 5
7 7
8 8
9 9
10 10
11 11
12 12
13 13
14 14
15 15
16 16
```

The crossbar file defines from one to eight settings for the crossbar on the Splash-2 array board. The first line of each setting consists of the key word "configuration" followed by an integer from 0 to 7 (inclusive). The lines that follow are of the form

output-port-number input-port-specifier

where "output-port-number" ranges from 1 to 16, and corresponds to the Xilinx chip number. The "input-port specifier" is either a single integer (0 to 16), or 5 integers (0 to 16 each). If it is a single integer in the range (1 to 16), it specifies the source port for all 36 bits of the output. If it is zero, it disables (tristates) all 36 bits of "output_port_number". If "input-port-specifier" consists of 5 integers, each integer

specifies the source for one byte of the output, from most significant to least significant. If an output port is missing from the configuration, it is assumed to be set to zero.

Here, 2, 3, 4 and 5 receive the data from 1 while their output is transferred to 6. The rest of the PEs are not used. The output of 2 and 3 is transferred to 6 in configuration one while output of 4 and 5 is transferred to 6 in the configuration seven.

C.3. Sample Memory File

The memory model used by the Splash-2 simulator allows the user to specify a set of initial contents for the on board memory from an input file. The file format is as shown in the sample file. For each contiguous block of data, the base address of the block is given followed by one or more data values.

```
-- First block of data
address 0
1 2 3 4
5 6 7 8
-- Second block of data
address 256
1 3 5 7
2 4 6 8
-- End of file
```

For all the implementations in this thesis, including the histogram, the initial memory contents have to be zero. This is specified as

```
clear
-- End of file
```

C.4. Sample Input/Output File

The input/output file format used throughout this research is as shown in the sample file. The actual files used for simulation and actual Splash-2 runs are much larger.

```
00000008 8
02040608 8
00440055 8
00000000 8
44444444 8
00000000 0
00777700 0
```

The data is represented in hexadecimal notation such that every byte is seperately represented. The format is

D31-24 D23-16 D15-D8 D7-D0 D35-D32

Each data entry in the file is valid only if D35 is set to a '1'.

C.5. Sample VHDL file for Xilinx_Processing_Element

The sample VHDL file shown is the behavioral description of the Xilinx_Processing_Element entity used in the image histogram generator application. The description is shown for processing element X2.

```
library SPLASH2;
use SPLASH2.TYPES.all;
use SPLASH2.SPLASH2.all;
use SPLASH2.COMPONENTS.all;
```

```
architecture chipa2 of Xilinx_Processing_Part is

        signal PassThru: Bit_Vector(15 downto 0);
        signal xbar_out: Bit_Vector(DATAPATH_WIDTH-1 downto 0);
        signal xbar_in : Bit_Vector(DATAPATH_WIDTH-1 downto 0);
        signal xbar_enable : Bit_Vector(4 downto 0);

        signal mem_rd, mem_wr : Bit;
        signal mem_in, mem_out : Bit_Vector(15 downto 0);
        signal mem_addr : Bit_Vector(17 downto 0);
        signal mem_enable : Bit;

        signal swich : Bit_Vector(2 downto 0);
        signal main_sw : Bit_Vector(1 downto 0);
begin


        process
                variable addr1,addr2 : Bit_Vector(7 downto 0);
                variable data1,data2 : Bit_Vector(15 downto 0);
                variable pixel_cnt : Bit_Vector(16 downto 0);
                variable counter : Bit_Vector(8 downto 0);
        begin

-- wait for the rising edge of the clock.
                wait until XP_Clk'Event and XP_Clk = '1';

                Pad_Xbar(XP_Xbar,xbar_out,xbar_in,xbar_enable);

                Pad_InOut(XP_Mem_D, mem_out, mem_in, mem_enable) ;
                Pad_Output(XP_Mem_A, mem_addr) ;
                Pad_Output(XP_Mem_RD_L, mem_rd);
                Pad_Output(XP_Mem_WR_L, mem_wr);
```

```
-- start only if valid bit (35) is high and start in state 1.
                if main_sw = "00" then
                        if xbar_in(35) = '1' then
                                if swich = "000" then
                                        swich <= "001";

                                        mem_rd <= '0';
                                        mem_wr <= '1';
                                        mem_enable <= '0';
                                        mem_addr(7 downto 0) <= xbar_in(7 downto 0);
                                        mem_addr(17 downto 8) <= "0000000000";
                                        addr1 := xbar_in(7 downto 0);
                                        addr2 := xbar_in(15 downto 8);

                                elsif swich = "001" then
                                        swich <= "010";

                                        mem_rd <= '0';
                                        mem_wr <= '1';
                                        mem_enable <= '0';
                                        mem_addr(7 downto 0) <= addr2;

                                elsif swich = "010" then
                                        swich <= "011";

                                        mem_rd <= '1';
                                        mem_wr <= '1';
                                        mem_enable <= '0';


                                elsif swich = "011" then
                                        swich <= "100";
```

```
            mem_rd <= '1';
            mem_wr <= '1';
            mem_enable <= '0';

            data1 := mem_in;

for I in 0 to 15 loop
if data1(I) = '0' then
            data1(I) := '1' ;
            exit ;
else
            data1(I) := '0' ;
end if ;
end loop ;

elsif swich = "100" then
            swich <= "101";

            mem_rd <= '1';
            mem_wr <= '1';
            mem_enable <= '1';

            data2 := mem_in;

for I in 0 to 15 loop
if data2(I) = '0' then
            data2(I) := '1' ;
            exit ;
else
            data2(I) := '0' ;
end if ;
end loop ;

            if addr2 = addr1 then
```

```
            for I in 0 to 15 loop
            if data2(I) = '0' then
                    data2(I) := '1' ;
                    exit ;
            else
                    data2(I) := '0' ;
            end if ;
            end loop ;

            end if;


elsif swich = "101" then
            swich <= "110";

            mem_rd <= '1';
            mem_wr <= '0';
            mem_addr(7 downto 0) <= addr1;
            mem_enable <= '1';
            mem_out <= data1;

elsif swich = "110" then
            swich <= "111";

            mem_rd <= '1';
            mem_wr <= '0';
            mem_addr(7 downto 0) <= addr2;
            mem_enable <= '1';
            mem_out <= data2;

else
            swich <= "000";

            mem_rd <= '1';
```

```
                        mem_wr <= '1';
                        mem_enable <= '1';

                for I in 0 to 16 loop
                if pixel_cnt(I) = '0' then
                        pixel_cnt(I) := '1' ;
                        exit ;
                else
                        pixel_cnt(I) := '0' ;
                end if ;
                end loop ;

                        if pixel_cnt = "01000000000000000" then
--                      if pixel_cnt = "00000000000001000" then
                                main_sw <= "01";
                                pixel_cnt := "00000000000000000";
                        end if;
                end if;

        else
                mem_rd <= '1';
                mem_wr <= '1';
        end if;
-- This is the logic description for state 2.
        elsif main_sw = "01" then

        if xbar_in(35) = '1' then

                mem_rd <= '1';
                mem_wr <= '1';

        for I in 0 to 8 loop
        if counter(I) = '0' then
                counter(I) := '1' ;
```

```
                exit ;
        else
                counter(I) := '0' ;
        end if ;
        end loop ;

                if counter = "000000110" then
                        main_sw <= "10";
                        swich <= "000";
                        counter := "000000000";
                end if;

        end if;
-- This is the logic description for state 3.
        elsif main_sw = "10" then

                mem_rd <= '0';
                mem_wr <= '1';
                mem_enable <= '0';
                mem_addr(8 downto 0) <= counter;
                mem_addr(17 downto 9) <= "000000000";

        if swich = "000" then
                swich <= "001";

                xbar_out(31 downto 16) <= mem_in(15 downto 0);

        else
                swich <= "000";


        for I in 0 to 8 loop
        if counter(I) = '0' then
                counter(I) := '1' ;
```

```
                exit ;
        else
                counter(I) := '0' ;
        end if ;
        end loop ;

                if counter = "100000100" then
                        main_sw <= "11";
                        counter := "000000000";
                end if;

        end if;
else

                mem_rd <= '1';
                mem_wr <= '0';
                mem_enable <= '1';
                mem_addr(8 downto 0) <= counter;
                mem_addr(17 downto 9) <= "000000000";
                mem_out <= "0000000000000000";

        for I in 0 to 8 loop
        if counter(I) = '0' then
                counter(I) := '1' ;
                exit ;
        else
                counter(I) := '0' ;
        end if ;
        end loop ;

                if counter = "100000100" then
                        main_sw <= "00";
                        swich <= "000";
                        counter := "000000000";
```

```
                    end if;

                end if;

        end process;

-- The crossbar is enabled such that the D31-24 and D23-16 are in
-- in output mode while the rest of the bits are in input mode.
        xbar_enable <= "01100";
        XP_Xbar_EN_L <= "01100";

-- TriState unused ports.
        XP_Left <= TriState(XP_Left);
        XP_Right <= TriState(XP_Right);

end chipa2;
```

C.5. Sample VHDL file for Xilinx_Control_Element

The sample VHDL file shown is the behavioral description of the Xilinx_Control_Element entity used in the image histogram generator application. The description is shown for processing element X2.

```
library SPLASH2;
use SPLASH2.TYPES.all;
use SPLASH2.SPLASH2.all;
use SPLASH2.COMPONENTS.all;
```

```
architecture chipa0 of Xilinx_Control_Part is
        signal swich : Bit_Vector(1 downto 0);
begin

        process
        begin
-- wait until the rising clock edge.
                wait until X0_Clk'event and X0_Clk = '1';

-- State machine description.
-- Start in State 1, and configure crossbar to setting 0.
                if swich = "00" then
                        swich <= "01";
                        X0_XBar_Set          <= "000";

-- State 2.  Configure the crossbar to setting 7.
                else
                        swich <= "00";
                        X0_XBar_Set          <= "111";
                end if;
        end process;

-- Tristate or disable all unused signals and buses.
   X0_SIMD              <= TriState(X0_SIMD);
   X0_XB_Data           <= TriState(X0_XB_Data);
   X0_Mem_A             <= TriState(X0_Mem_A);
   X0_Mem_D             <= TriState(X0_Mem_D);
   X0_Mem_RD_L          <= '1';
   X0_Mem_WR_L          <= '1';
   X0_GOR_Result_In     <= "ZZZZZZZZZZZZZZZZ";
   X0_GOR_Valid_In      <= "ZZZZZZZZZZZZZZZZ";
   X0_GOR_Result        <= '1';
   X0_GOR_Valid         <= '0';
   X0_XBar_Send         <= '0';
```

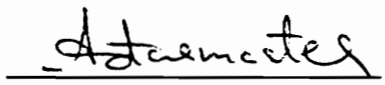
```
    X0_X16_Disable      <= '0';
    X0_Int              <= '0';
    X0_Broadcast_Out    <= '0';
    X0_HS0              <= 'Z';
    X0_LED              <= '0';

end chipa0;
```

Vita

Adit Tarmaster was born in Surat, India on December 17, 1970. He attended the College of Engineering, Pune (COEP), India, and obtained his B.S. in Electronics and Telecommunications in July, 1992. He decided to continue his education at Virginia Tech and is presently pursuing a M.S. in Electrical Engineering.

His research interests include digital and ASIC design, systems modeling, and he plans to work as a hardware design engineer.



Adit Tarmaster