Region Detection and Labeling of Images in Real-time using an FPGA-based Custom Computing Platform

By

Ramana V. Rachakonda

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:

Peter Athanas

Dr. Peter M. Athanas, Chairman

James R. Armstrong — A L Abbott

Dr. James R. Armstrong — **Dr. A. Lynn Abbott**

January 1995
Blacksburg, Virginia

Region Detection and Labeling of Images in Real-time using an FPGA-based Custom Computing Platform

By

Ramana V. Rachakonda

Dr. Peter M. Athanas, Chairman

Electrical Engineering

(Abstract)

Region detection can be defined as identifying connected components in an image. Connected component labelling is an important part of performing feature extraction. Industrial applications performing online image analysis may require a real-time implementation of a region labelling algorithm (typically processing 30 frames per second). Using application specific hardware or VLSI implementations to solve the problem sacrifices the ability to alter the design tradeoffs dynamically. General purpose software platforms are considerably slower for the task. While the massively parallel machines can accomplish the job without sacrificing the general purpose nature, they are highly un-economical. This research discusses an algorithm which was implemented on a FPGA-based reprogrammable hardware platform and demonstrates the effectiveness of custom computing platforms for high performance real-time image processing. The input to the system is VHDL, so the design can be modified very easily. Also, a variety of applications can be run on the system unlike application specific hardware or VLSI implementations.

Acknowledgments

I would like to thank my advisor, Dr. P. M. Athanas, for having so untiringly followed my research and the preparation of this document. His help, advice and encouragement were invaluable.

I would also like to thank Dr. A.L. Abbott for aiding me with technical suggestions that proved most helpful for the progress of my own work.

Finally, I am grateful to Dr. J.R. Armstrong who was on my committee, my friends Brad, Nabeel, Adit, Al, Gayatri, Kalpana, Gummadi, Suri and Sumanth for their help in completing my long-distance thesis.

Table of Contents

List of Figures

CHAPTER 1 Introduction

1.1 Motivation

Most image processing applications can be characterized as computationally intensive. In many cases the complexity of the job increases with the size of the image. Real-time image processing applications are more challenging because the processing needs to be done at real-time speeds, typically at 30 frames per second. Such applications are usually data parallel, i.e. several simple operations are performed on a large number of picture elements (pixels). The sequential general purpose machines are inferior to their parallel counterparts on such tasks. Considering the expense of the massively parallel machines, and the inflexible nature of the application specific implementations, the custom computing platforms prove to be a viable and cost effective solution for computationally intensive applications.

The goal of this research is to demonstrate that the *Splash-2* custom computing platform is suited for performing real-time image processing tasks. The implementation demonstrates the effectiveness and potential of adaptive computing platforms for one real-time image processing application.

1.2 Contribution of this research

An efficient algorithm for component labeling has been developed and successfully implemented during the course of this research. State diagrams and VHDL models for the various building blocks have been developed. An adaptable image thresholder has been developed to convert the incoming grayscale image to a binary image. Changes have been made to the architecture and tough trade-offs were considered and used to derive an efficient implementation on the system. A novel implementation of the look-up table for implementing an equivalence table has been developed.

The tasks are performed in real-time with the data coming in from a camera at the rate of 30 frames per second. The various objects identified in an image are mapped to different colors using a random color map. Thus, the monitor displaying the images shows various colored objects in the image, changing in real-time. The design also produces images to a workstation monitor in the application debug mode.

1.3 Organization of thesis

Relevant concepts of image processing pertaining to this project are discussed in Chapter 2. A literature survey of related work is included in Chapter 3. The architecture of *Splash-2* is described in Chapter 4. Chapter 5 explains the design process for programming a *Splash-2* attached processor system and is an overview on the *Splash-2* software environment, the design process, and various tools used for programming. Chapter 6 explains the implementation of the component labeling algorithm in detail. Data flow diagrams, state diagrams, logical pinouts of chips and inter-processor communication diagrams are provided to aid the process. A summary of conclusions from the thesis is in Chapter 7.

CHAPTER 2

Image Processing Background

2.1 Introduction

Image processing is the problem of extracting useful information from an image or sequence of images. Digital image processing examines images that are represented as a 2D-arrays of *pixels*. A quantized digital grayscale image is typically composed of a two-dimensional array of picture elements called *pixels*. Each pixel takes a value from *0* to *(K*-1*)* (usually K=2^N). This value is called the *grayscale* value of the pixel. The image array may be represented by *I(r,c)*, where *r* and *c* are the row and column location of the pixel respectively. Figure 2.1 shows a typical 512 x 512 digital quantized image with 2^8 levels of quantization. A camera generates pixels in the raster-scan order, in which the pixels in the image are scanned from left to right and from top to bottom.

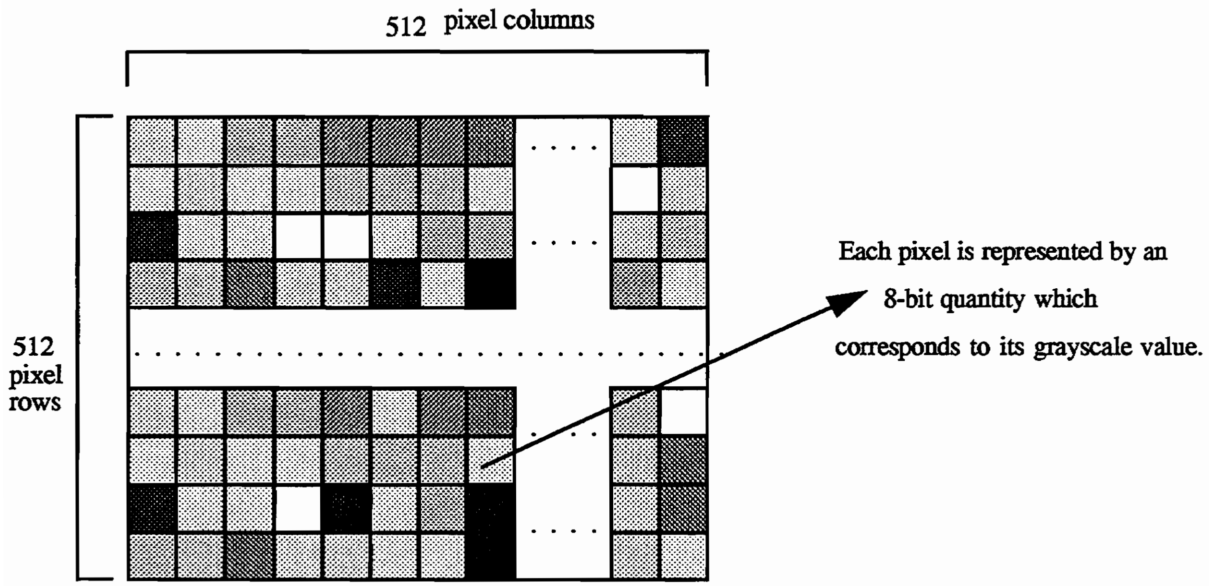

512 pixel columns
512 pixel rows
Each pixel is represented by an 8-bit quantity which corresponds to its grayscale value.

FIGURE 2.1. **Example Grayscale Image.**

Image operations may be broadly classified into various generic classes. If two images of a given type are combined to form an image of the same type, the operation is said to belong to the *combination* class. The operations of *generation* class generate an image from scratch. No input image is needed for such type of operations. The operations belonging to the *transformation* class transform an input image of a given type to another image of the same type. The operations of the *measurement* class are used to calculate various norms of an image. The output is mostly a small number, or a distribution that summarizes various properties of an image. The operations of the *conversion* class convert an image of a given type to a completely different type [22].

A foreground pixel, *P*, is said to be *eight-connected* to another foreground pixel, *Q*, if there exists a path of zero or more foreground pixels between *P* and *Q*, traversing in any direction. They are said to be *four-connected,* if the path does not traverse diagonally. Assume that the pixels with alphabets belong to the foreground and the others belong to the background. Pixel A is eight-connected to pixels B, C, D and E, but is four-connected only to D and E. This is because Pixel A and Pixel B are connected to each other through a diagonal arc.

			C	
			B	
	D	*A*		
	E			

FIGURE 2.2. Illustration of *eight-connectedness.*

Image thresholding and region detection may be classified as operations of the transformation class. One can look at this as changing the number of quantized levels a pixel in an image can take, according to a set of rules. The former converts eight-bit pixel values of a grayscale image (2^8quantization levels) to one-bit quantities (two quantization levels). The latter assigns one of *R* labels or colors (*R* quantization levels), where *R* equals the total number of objects in the image, to each foreground pixel in a binary image (two quantization levels). Two pixels belong to the same quantization level $r < R$, if and only if they are *eight-connected* [22].

2.2 Image Thresholding

Thresholding is a transformation class operation which reduces the number of quantized levels a pixel of an image can assume. If the output image were to contain two quantization levels, one threshold is chosen for a simple thresholding scheme and the set of all pixels with grayscale values below the chosen threshold are said to belong to the foreground and the rest belong to the background. Hence an 8-bit grayscale image is converted to a 1-bit binary image after thresholding. The segment of the program in Figure 2.3 illustrates the process. Generally, *N* thresholds can be used to generate an output with $(N+1)$ quantization levels. Note that information is lost by thresholding an image [24].

```
#include <stdio.h>
#define FOREGROUND 1
#define BACKGROUND 0
#define THRESHOLD 100

void threshold(int* output, int* input, int Max_cols, int Max_rows)
{
   int i,j;

   for (i=0; i < Max_rows; i++)
      for (j=0; j < Max_cols; j++)
         output[i*Max_cols + j] = (input[i*Max_cols + j] < THRESHOLD)?
                                          FOREGROUND   : BACKGROUND;
}
```

FIGURE 2.3. **Function for thresholding.**

The above function processes the incoming image in *input* and returns a thresholded image in *output*. The input image may contain any number of quantization levels but the output image contains only two values, FOREGROUND or BACKGROUND (1 or 0 respectively). The function can be generalized to handle *N* thresholds. The output of such a function can take *(N+1)* values (*0* to *N*). If *T(1)* to *T(N)* represent the thresholds, a pixel can take an output value *i* if its input grayscale value is in-between *T(i)* and *T(i+1)* (*T(0)* = 0 and *T(N+1)* = Max{input[i][j]: for all i,j}). Normally, single threshold operations are used for all practical purposes. Example input image and output image after thresholding is in Figure 2.4.

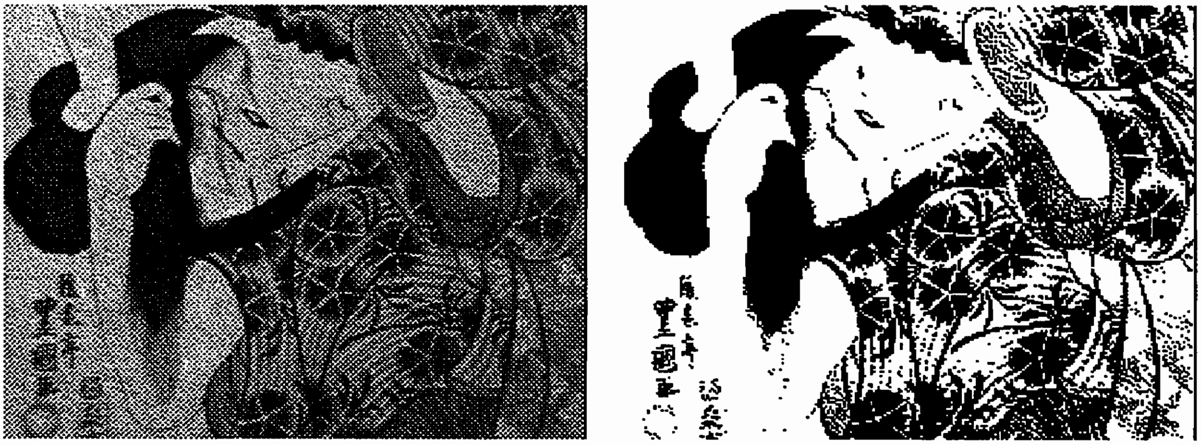


FIGURE 2.4. **Example Input and Thresholded Output Images.**

2.3 Region Detection and Labeling

The problem of region detection and labeling is the process of identifying connected components in an image. Assuming a binary image in which each pixel is represented by 0 or 1, a label R can be assigned to a foreground pixel if and only if it meets the following conditions:

1. All the foreground pixels that are 8-connected (or 4-connected) by a path to the pixel in discussion are also labeled R.
2. All the foreground pixels that are not 8-connected (or not 4-connected) by a path to the pixel in discussion have a label other than R.

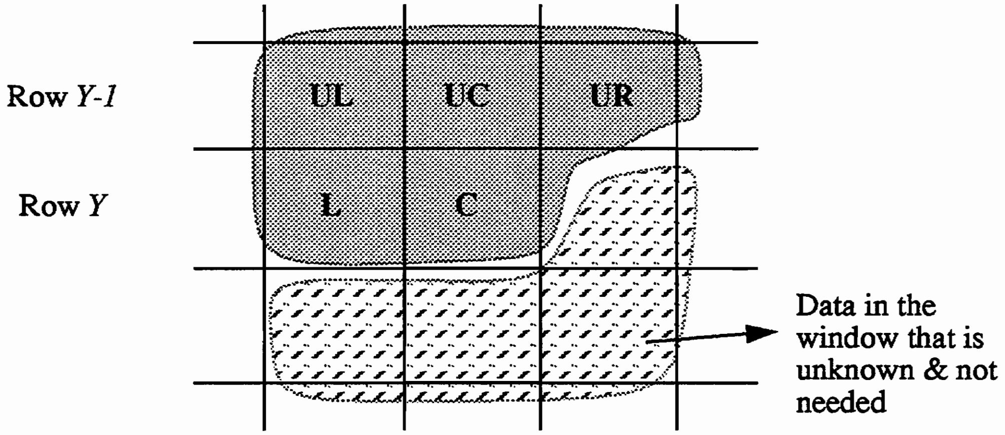


FIGURE 2.5. Window to be considered for labeling the current pixel.

For images processed in the raster-scan order, algorithms for connected component labeling consider labels of row *Y-1* while labeling row *Y*. Such algorithms do not use the labels of a row *m* > *Y*. Figure 2.5 shows the four neighbors considered for labeling a current pixel C. The neighbors are named L, UL, UC, and UR to represent the left, upper-left, upper-center, and upper-right pixels with respect to the current pixel C, respectively. The current pixel C is given a label *R* if and only if one of its neighbors has the label *R* and C belongs to the foreground. If the current pixel C encounters two different labels, *P* and *Q*, given to its neighbors, it acquires one of the labels and an *equivalent pair* is generated indicating *P* and *Q* belong to the same region in the image. The equivalence pairs generated are stored in an *equivalence table*. The window then advances by one pixel and similar processing occurs.

An equivalence table can be implemented as a lookup table in memory. The table is initialized by writing the index i to a memory location M(i) (0 =< i < K, where K is the maximum number of labels the algorithm can handle). If an equivalence pair of P and Q is generated, the value in the memory location P, M(P) and the value in the memory location Q, M(Q) are fetched. All the memory locations containing the value M(Q) are replaced with the value M(P). The process is repeated for all equivalence pairs generated for that particular frame. The equivalence table hence generated contains an identical value in the locations pointed by labels of the same equivalence class. Thus various labels given to the same object are mapped to one label. The image is processed again to merge all the inconsistent labels and the output contains one consistent label per equivalence class. This is accomplished by replacing all the first-pass labels, f, by the values they point to, M(f), from the lookup table. The labels generated by second pass are consistent.

The process described above takes more than 3K (K is the maximum number of labels the design can handle) clock cycles for a lookup table in an external memory. This is due to the fact that a read and write to an external memory can be accomplished in three clock cycles and each memory location needs to be read and conditionally written to, by the above process. The process becomes very inefficient for designs handling large number of labels. As the generation of equivalence pairs is sporadic and uncommon, the algorithm works better for objects with lesser number of labels. A simulated input and expected region labelled output image is shown in Figure 2.6. If the output were to be displayed on the monitor, one can see the different colors given to the ellipse, triangle, circle, rectangle, and the polygon.

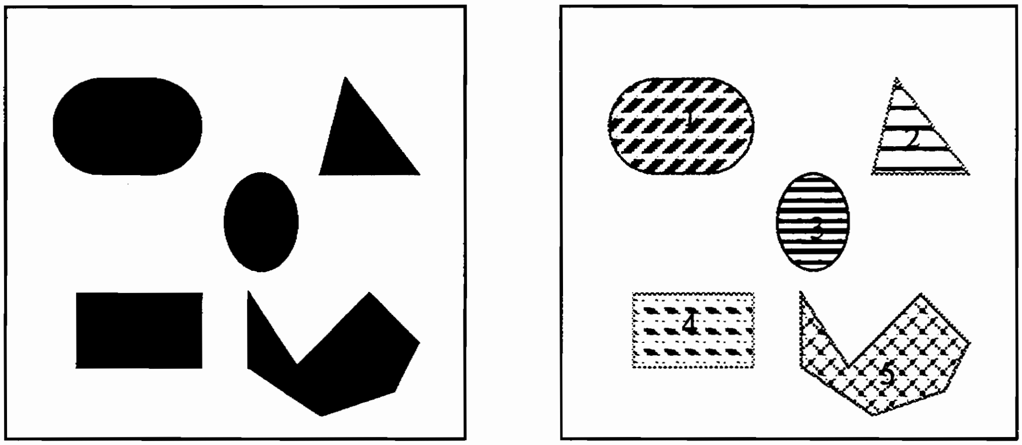


FIGURE 2.6. **Example Input and Region Labelled Output Image.**

CHAPTER 3

Existing Algorithms for Component Labeling

3.1 Introduction

The problem of component labeling is computationally intensive. It is quantified in [5], which states that a typical image of size 2500 x 2048 pixels may have over 80,000 regions. Degenerative case images of this resolution can be constrained to contain more than a million distinct regions. Many algorithms have been proposed for different computing platforms and are examined in this chapter. They can be broadly divided into two categories: sequential and parallel. The sequential algorithms process one pixel at a time while the parallel algorithms usually, have an *NxN* array of processors analyzing all the pixels at one time. While software implementations on sequential general-purpose machines cannot handle real-time data, their parallel counterparts are extremely expensive and hence, are not cost effective.

This thesis describes an algorithm based on hardware/software co-design implemented on a reconfigurable hardware platform called *Splash-2*. *Splash-2* is an inexpensive FPGA-based experimental general-purpose hardware platform. The real-time design on *Splash-2* is an excellent cost effective solution for the problem of component labelling.

3.2 Sequential Algorithms

3.2.1 The two-pass method

One of the first sequential algorithms was proposed by Rosenfeld and Pfaltz [3] in 1966, which uses two row-wise top-to-bottom passes through the image. In the first pass, the input image is processed to compute the equivalence of labels (Refer to Section 2.3). Two arrays are created, one containing the currently assigned component labels (first pass labels) and the other containing the corresponding equivalent minimum labels. During the second pass, the first pass labels are replaced by their corresponding equivalent minimum labels. The equivalence table is stored in a memory, usually an on-chip RAM. The size of the table increases with the size of the image. Hence the dynamic storage requirements for this algorithm are high and therefore not practical for larger images. The algorithm is best suited for hardware implementations and for real-time applications [12]. The algorithm needs exactly two passes through the image and hence the processing time is independent of the number of inconsistencies encountered in the image. The algorithm implemented as a part of this thesis resembles the above-mentioned algorithm.

3.2.2 The iterative method

An iterative method was proposed by Haralick in [4]. This algorithm overcame the high storage requirements of the two-pass method described above. An initial pass is made through the image which assigns a unique label to all the foreground pixels in the image. Then, the image is processed iteratively. Each iteration consists of two passes. The first of which is a top-to-bottom pass that assigns the minimum of the labels of the foreground pixels in its neighborhood. The second pass performs a similar assignment of labels, but in a bottom-to-top fashion. These two-pass iterations are repeated until no label changes occur during an iteration. This algorithm trades off high storage requirements with the time complexity. The number of iterations are not fixed in the algorithm, and is dependent on the nature of the input image. There is an upper bound on time given for the algorithm which depends on the size of the image. But it is large and usually far from the typical time taken. Because the algorithm does not give a fixed bound on time, it is not ideally suited for real-time applications.

3.2.3 Other sequential methods

Another algorithm involving only two passes through the image was proposed by Lumia et. al. [5]. This approach is a compromise in terms of dynamic storage and time of the above-mentioned algorithms. It employs an equivalence table for every row. Refer to [2] for further details. Another two-pass algorithm, using a technique called bracket matching, was proposed by Schwartz et al in [6]. This algorithm stores relatively low information in an axillary bracket table and derives its computational efficiency by utilizing stacks. Algorithms have also been proposed that not only detect regions in an image but also obtain topological and geometrical information about the labels [7]. Lochovsky proposed an algorithm [1] that derives its efficiency from reusability of labels. A label is said to be *extinct* if it does not exist in a previous row, and it can be reused to save internal storage. The reused label is assigned to another external label at a later time. The algorithm saves not only on the internal storage but also on communication and processing of labels. Hence is faster than its counterparts if the overhead of label remapping is offset by the gain in speed.

3.3 Parallel Algorithms

Several parallel algorithms were proposed, most of which consist of a 2-D array of processors in mesh configuration. An *O(N)* parallel algorithm that uses a *NxN* mesh for an image of size NxN is discussed in [10]. Several parallel algorithms, such as the $O(log4N^2)$ algorithm discussed in [9] and the $O(log\ N^2)$ algorithm discussed in [8], use pointers to keep track of their connected neighbors or shrink their components to a single pixel before labeling them. Though the sequential algorithms are ineffective in terms of space/time requirements, the parallel algorithms are based on expensive general purpose parallel machines. To implement one such algorithm on *Splash-2,* one ideally requires *N* processing boards (where N=512, in this case). Also, the communication resources between successive *Splash-2* boards need to be larger. Due to the architecture of *Splash-2,* it is not feasible to implement any of these algorithms on the *VTSplash* real-time system. This thesis aims to obtain a higher cost to speed ratio for the particular application than both the sequential and parallel algorithms can achieve, while retaining the advantages of a software design. The algorithm has been implemented on a reconfigurable hardware platform which has the speed of a hardware based design and the flexibility of a software based design.

3.4 Hardware Implementations

Board level implementations of the problem were proposed by Yang [12] and Schwartz et al [6]. MSI and LSI chips were used in their wire-wrapped multi-board designs. The architecture for those designs was complex and required several components for various stages of processing. A systolic algorithm for VLSI implementation was proposed by Ranganathan et. al. [2]. It involved two passes: a top-to-bottom and a bottom-to-top pass through the image. The standard cell based design included muxes and registers. Being a VLSI, design it lacked the re-programmability of the software-based designs.

CHAPTER 4

Splash-2: an adaptive computing platform

4.1 The VTSplash system

A real-time image processing custom computing platform has been developed at Virginia Tech using the *Splash-2* adaptive computing platform [17]. The goal of the project is to demonstrate the advantages of using re-programmable custom computing platforms for computationally intensive image processing applications. The laboratory setup of this system, called *VTSplash*, uses a *Splash-2* attached processor with a SUN SPARCstation-2 serving as a host and an IBM-PC as a host for the custom frame grabber cards to interface the camera and the monitor.

Figure 4.1 shows the setup of the *VTSplash* system. The system consists of a video camera, a digitizer for converting the analog signal from the camera to digital data, a *Splash-2* attached processor that forms the crux of the system, a frame buffer to grab data from *Splash-2* and send it to a monitor for display, a color video monitor for displaying the processed image from *Splash-2* and a SUN SPARCstation-2 to program and to send control signals to *Splash-2* attached processor.

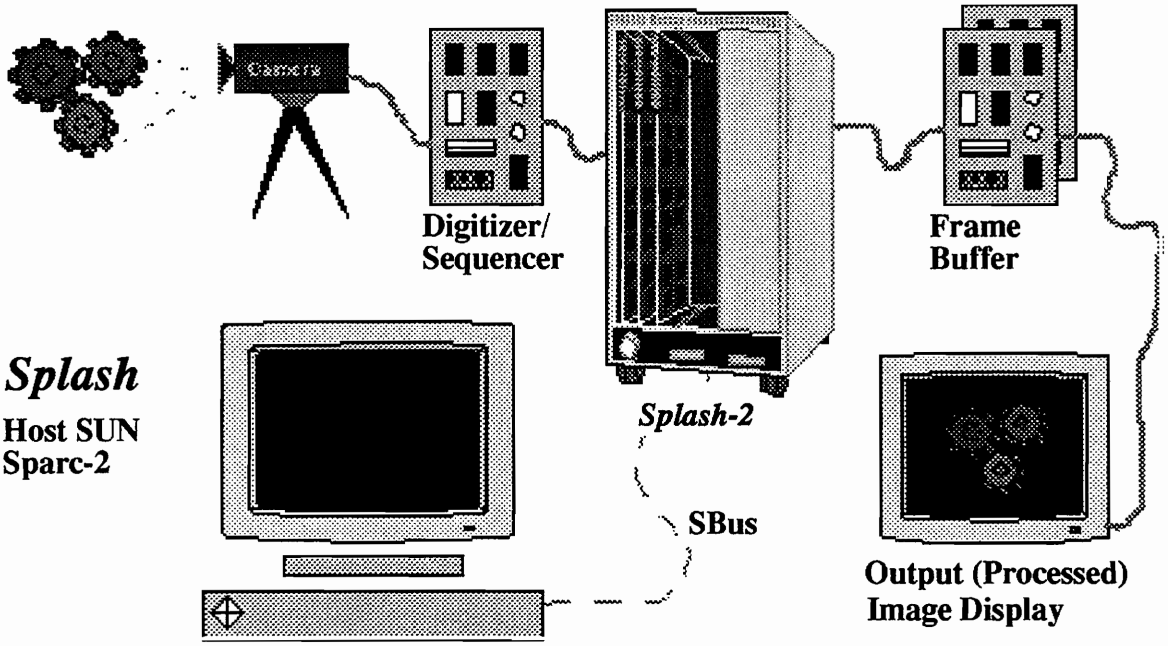


FIGURE 4.1. The *VTSplash* System.

The camera produces an RS-170 analog video signal. The A/D converter digitizes the signal at 9.8 MHz. A custom digitizer/sequencer was developed to reformat the digitized stream of data and pass it to the *Splash-2* attached processor at a rate of 10 Million 8-bit pixels per second. The *Splash-2* processes the incoming data and presents it to another custom board, which formats the data. The output of this board is presented to a commercial frame grabber card (DT2867LC) which displays the processed images to a video monitor [16]. The SUN SPARCstation-2 provides an interface to the *Splash-2* attached processor. It is used to program the *Splash-2* array boards to perform specific image processing tasks. It is also used to send control signals to the *Splash-2* attached processor, once it is programmed and running. It may also be used to send non-real-time simulated images to the *Splash-2* in the software clocked mode.

4.2 Introduction to *Splash-2*

Splash is a reconfigurable systolic array processor developed by the Supercomputing Research Center (SRC) in Bowie, Maryland [15]. *Splash-2* is a second generation custom computing platform developed at SRC [13]. The architecture of the *Splash-2* system consists of two different types of boards, namely, the interface board and the array board. A typical *Splash-2* system consists of one interface board and from 1 to 15 array boards.

The interface board acts as the front end of *Splash-2* to the external world. It performs the tasks such as 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 maps the 32-bit data width of the host to 36-bits used by *Splash-2*.

The *Splash-2* array board, also called the *Splash-2* processing board, consists of a 1-D systolic array of Xilinx XC4010 FPGA's [19], and a full crossbar connecting all the chips [14]. The name *processing element* (PE) is used to refer to a combination of a Xilinx chip and its local memory (256K x 16 static RAM). Each array board has sixteen such processing elements (X1 to X16) and a control chip (X0). The control chip is used for configuring the crossbar to one of eight predetermined crossbar configurations. It is also used to do global broadcasting and handshaking. Processing element X1 receives its data from its left port and the output is taken from the right port of the processing element X16. Technically, *N* such boards can be connected with the processing element X16 of a previous board feeding data to the processing element X1 of the current board. Thus, the data enters the system through X1 of the first board and leaves through X16 of the last board. Figure 4.2 shows the block diagram of *Splash-2* attached processor system.

In Figure 4.2, we can see the SUN SBus is connected to the interface board. This is used to program the Xilinx FPGAs, to provide simulated image data input and control signals to the *Splash-2* array boards. The image data is also fed to the interface board through the camera via the digitizer board. Similarly, the output from the *Splash-2* can either be directed to the SUN SBus or to a frame grabber card for display onto a monitor. The SIMD bus is used for broadcasting the image to all the array boards.

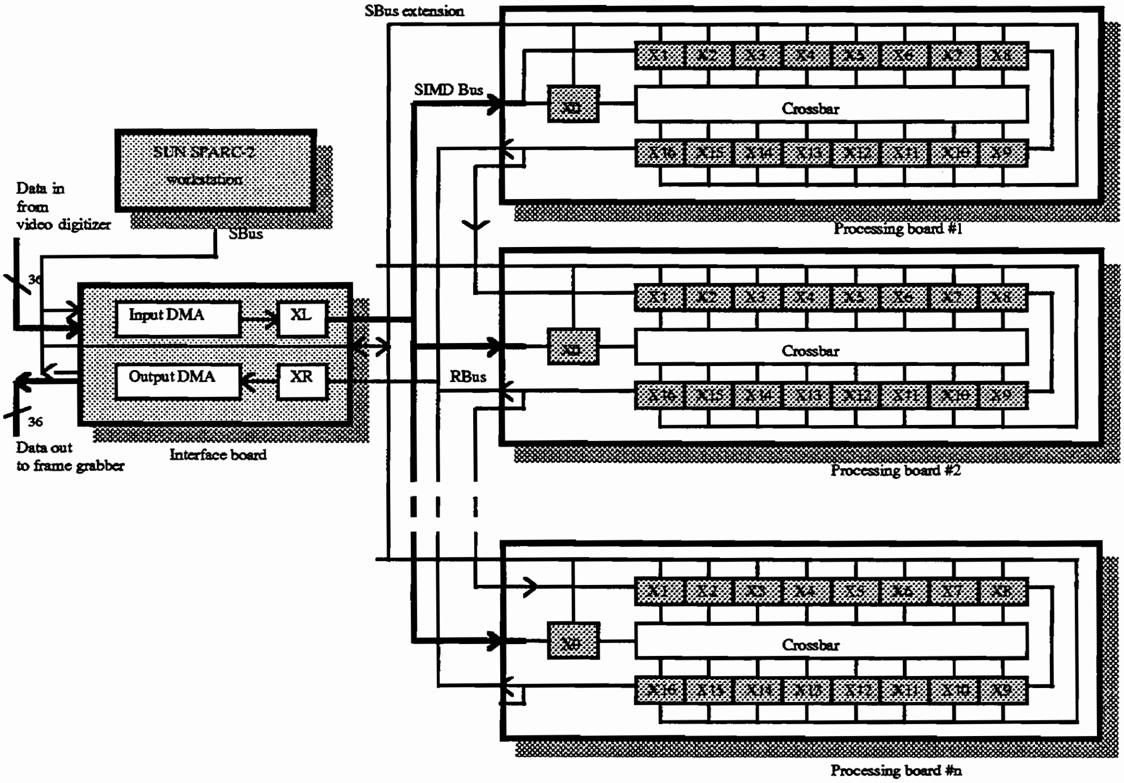


FIGURE 4.2. The *Splash-2* attached processor system from [14].

Figure 4.3 shows the structure of a processing element. The local memory is addressed by an eighteen bit address from the corresponding processing element. It also receives a read and a write signal from the Xilinx FPGA, both of which are active low, indicating whether a read or a write has to be performed, respectively. As the memory is 16-bits wide, the Xilinx FPGA and the local memory have a 16-bit bi-directional data bus connecting each other. Each of the local memories can be initialized from the SUN host. Thus, the SUN SBus is connected through tristate buffers to the data and address buses of the local memory. The SUN host uses these buses to write to the individual local memories and read back from them. The local memory uses the same port to the SUN and the Xilinx FPGA, which means that only one of them can access the memory at a time.

Each Xilinx FPGA also has two 36-bit bi-directional bus connecting it to the processing element to the right and left to another the processing element. 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 processing board and to the left port of X1 of the first processing board. The right port of X16 connects to the RBus and to the left port of X1 on the next board.

A third bus is connected to the *crossbar* through which it can connect to any other chip on the board. The 36 bits going to the crossbar can be divided into five different segments: four octets and a nibble, each of which can be programmed to connect to a different processing element. The central crossbar is built from nine Texas Instruments SN74ACT8841 16 x 16 4-bit crossbar chips. These can store up to eight dynamically selectable configurations which, as mentioned before, are selected using the control chip. The crossbar allows point to point, multicast and broadcast communication between all of the processing elements on each array board.

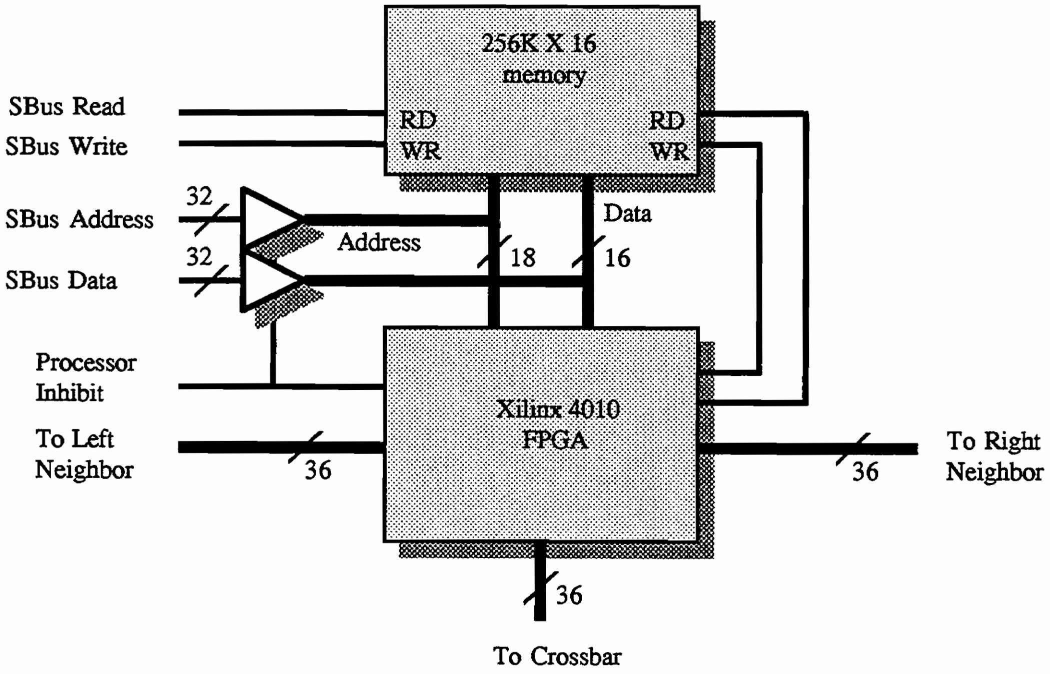


FIGURE 4.3. The *Splash-2* Processing Element.

The control paths between the SUN SPARCstation-2 host and the application program running on *Splash-2* consist 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. The two handshake registers on each *Splash-2* board may be used to perform asynchronous communication between the computing elements and the host. Handshake register 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 HS2 contains a single bit connected to all 17 processing elements and the direction of HS2 is input to the computing elements only.

A single bit on the interface board writable by the host and connected to the control chip (X0) of every board is used as a broadcast bit. X0 in turn outputs a signal to all the processing elements. One bit broadcasts can be performed from the host in two steps. First the host writes to the broadcast register on the interface board, and then each control chip (X0) broadcasts the signal to all the processing elements on its array board.

The *Splash-2* provides a status register, readable by the host, which is used for sending interrupt signals from each of the processing elements to the host. All the interrupt signals from each of the 17 processing elements are collected and fed to the status register after ANDing with a 17-bit mask. The system interrupt signal is formed by ORing all the status signals from each of the array boards.

4.3 Field Programmable Gate Arrays (FPGAs)

Field programmable gate arrays or FPGAs are EEPROM-like devices which are functionally equivalent to PLAs. They contain programmable blocks called Configurable Logic Blocks (CLBs), which are interconnected via programmable interconnect resources.

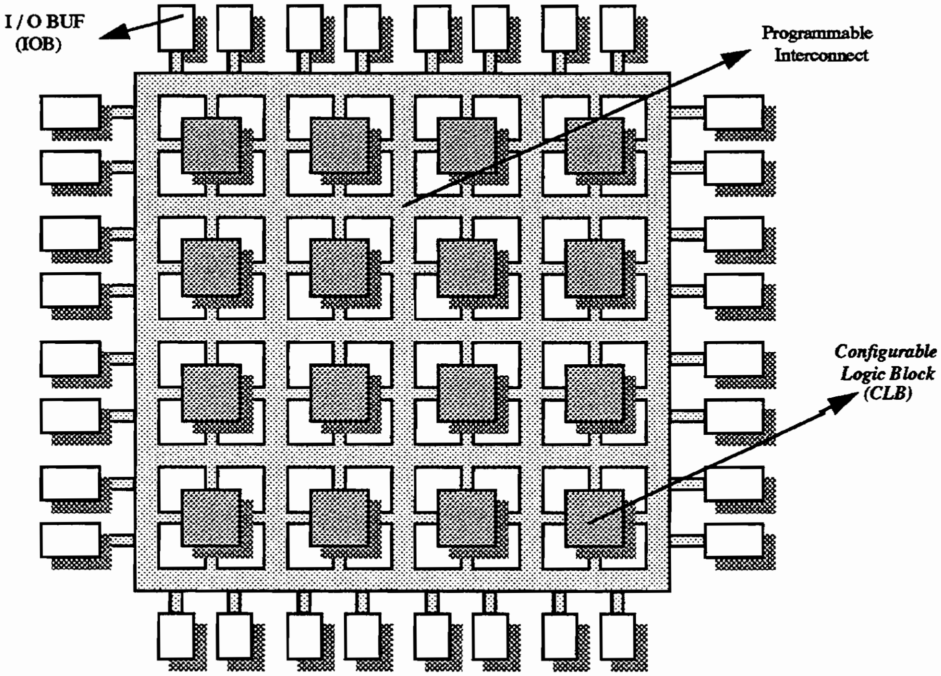


FIGURE 4.4. **Conceptual FPGA showing CLBs and interconnect.**

Figure 4.4 depicts a typical FPGA. It shows several configurable blocks and the interconnect. Programmable blocks of a third kind, called Input/Output Buffers (IOBs) act as pads to the chip. They can be programmed to behave like fast, medium, and slow buffers for the data, so that the data may be programmed to be latched or unlatched, pad delays can be set along with other properties.

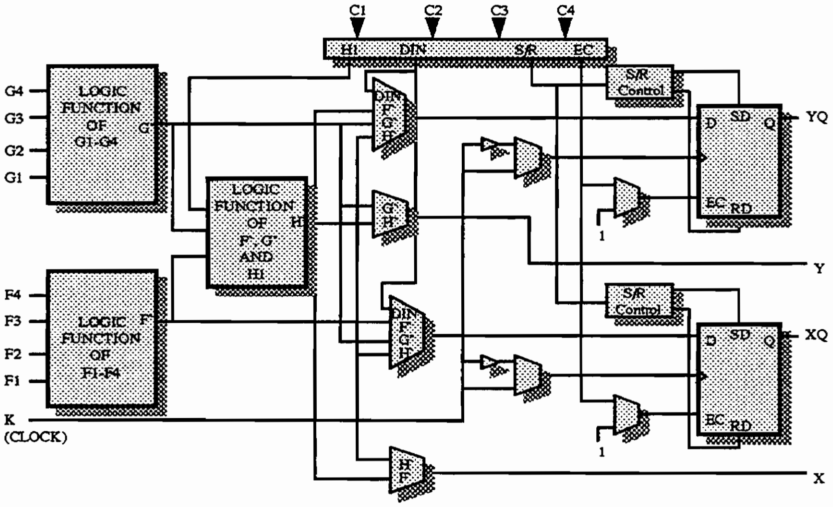


FIGURE 4.5. Structure of a CLB in an XC4010 FPGA from [20].

The configurable logic blocks are implemented using the look-up-table architecture. The RAM bits are programmed to implement various functions. In addition to this, they may have some flip-flops and other resources. Hence, most CLBs can provide registered and unregistered outputs. The programmable interconnect is used to interconnect various CLBs. Complex functions may be implemented by dividing the logic among different CLBs and using the programmable interconnect to connect the blocks. Figure 4.5 shows the structure of a CLB in an XC4010 FPGA used in the *Splash-2* system.

CHAPTER 5

Splash-2 Design Methodology

5.1 Introduction

The architecture of *Splash-2* is based on the concept of reprogrammability. Almost every resource in the machine can be programmed to suit the needs of the particular application. More specifically, the functional units and the interconnection between the functional units are alterable. This provides the programmer with the flexibility to modify the architecture not only statically but also dynamically. This chapter describes, in detail, the design methodology used for programming *Splash-2*. The programmer starts with a higher level description of the circuit in a hardware description language such as VHDL. This model is then simulated to check for correctness. Finally, the model is synthesized to a format which can be downloaded on to the processing elements, using various synthesis tools at different levels. This chapter also describes various debugging tools available for a *Splash-2* programmer.

5.2 *Splash-2* design process

Programming *Splash-2* is a well-defined process that has proven to be fully functional [18]. Figure 5.1 shows the basic steps taken in programming an application on *Splash-2.*

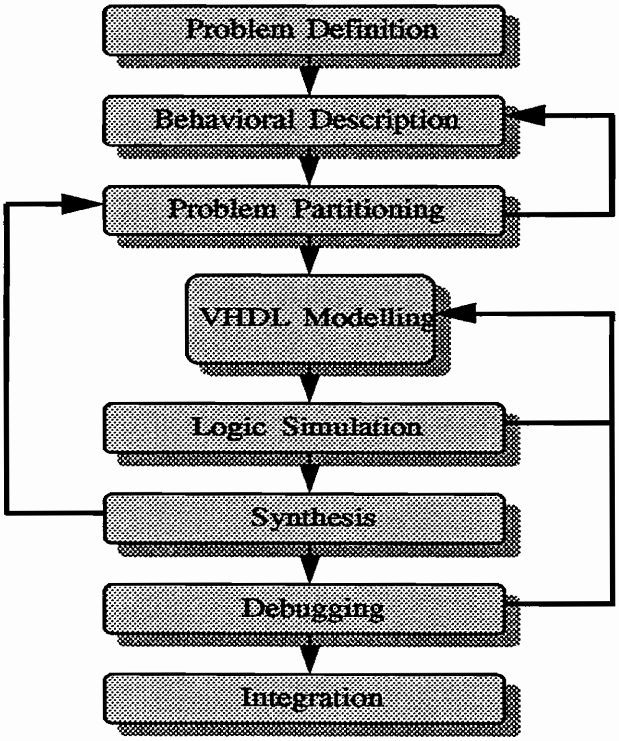


FIGURE 5.1. Block Diagram showing the *Splash-2* Design Process.

The first step in the design process is the *problem definition.* The algorithm to be implemented on *Splash-2* must be well defined and understood for an easier flow through the design process. The formats for the input and output images should be defined along with other required formats for intermediate signals. For example, in algorithms using floating point operations, one might define the sizes of the mantissa and exponent for a fixed point representation of the operand.

The second step is the *algorithm verification.* In this stage, a programmer develops a high-level program for the algorithm, most commonly in C, to check the correctness of the proposed algorithm. This gives the programmer a deeper insight into the problem. This step also gives statistical information regarding the speed of the algorithm on other machines and other information such as sample outputs for some sample inputs, which can be used to validate the outputs produced at later stages.

The next step in the design process is *problem partitioning.* After the correct algorithm has been obtained, a floor plan to partition the C-program into several different PEs has to be laid down. This is a difficult process because a programmer most often uses heuristics and needs experience to overcome the problems at this stage. Partitioning is generally done by identifying mutually independent tasks in the algorithm and pipelining them into different chips. Sometimes, two or more pipelines can be implemented in one chip. Similarly, identifying the tasks that can be done in parallel and implementing them on different chips also gives the programmer another heuristic for problem partitioning. After identifying parallel and pipelineable tasks in the algorithm, the programmer then proceeds to fit different tasks to the processing elements, considering various factors such as *time, area,* and *communication complexity* [18].

The design, thus obtained needs to work at a particular clock speed for real-time operation. This is typically 10 MHz, the pixel rate, for the image processing applications run on *Splash-2*. The speed at which a PE can run is determined by the delay of the maximum length path through the circuit. The Xilinx tool, called *Xdelay* is used to get an estimate of the maximum delay through the FPGA chip [20]. It also gives the maximum frequency at which one can run the chip. Even though this parameter is from the vendor of the chip, laboratory tests have proved that the chips can be run safely at a much higher speed than the prescribed. Thus, all the PEs in the design have to meet all the constraints to be able to handle real-time data coming into the system.

Area is usually the most critical constraint. The total amount of logic an FPGA can implement is fixed. Thus, the synthesized model of the tasks assigned to a particular processing element should fit in a Xilinx FPGA. The design has to be repartitioned whenever the estimate on the area exceeds the capacity of the FPGA. Before a design is synthesized for the first time, the area it occupies has to be estimated because a VHDL model of the design does not contain information about area requirements. The estimation is typically done by listing all the multiplexers, registers and other functional elements in the processing element and approximating the number of CLBs necessary to implement them. This always give a loose lower bound on the area required for the design. The actual count can be much higher than this estimate because it is difficult to estimate the number of CLBs required for control logic and routing resources. This estimate of the processing element is used at an early stage to determine if a design has to be repartitioned.

If the design is partitioned, the communication overhead increases with the number of PEs. The communication resources on *Splash-2* are limited. Hence, the programmer has to keep in mind the resources available to him and make effective use of them. Some of the resources include two 36-bit data paths connecting to the neighbors on the right and the left, a 36-bit data path connecting a processing element to the crossbar which can be programmed to connect to any other processing element on the same board, and a 16-bit data bus connecting to the local memory of a processing element.

Though the available resources are adequate for most applications, some may require the programmer to make tough design trade-offs. Also, as the programmer proceeds from this stage with estimates rather than accurate results, he might have to wait till the end of the synthesis step before he can confirm the feasibility of the particular design.

Referring back to Figure 5.1, the next step in the design process after design partitioning is the *VHDL Modeling* of the proposed design. The *VHDL Modeling* can be done at different levels such as algorithmic or logic level in the behavioral domain or RTL in the structural domain. Normally, the designs are not modeled in the physical domain because of the complexity at that level. Xilinx provides a software tool with its package, called *xact*, for programming in the physical domain. The design may not totally belong to a specific domain as the programmer may choose to model some parts of the program in a different domain for simplicity. From experience, logic level description for top-level designs and structural domain programming for individual modules such as counters is found to be advantageous.

The designer then proceeds to simulate the design written in VHDL in the *Splash-2* software environment. The *Splash-2* programming environment consists of a set of VHDL programs written to emulate accurately the *Splash-2* hardware environment. The designer can insert his design into the software environment, simulate the whole system and get the results that an actual *Splash-2* hardware would produce for a fully synthesized design. The input to the SIMD Bus is taken from and the output is written back to a file using VHDL ASCII file handling routines. One can use any of the commercially available VHDL simulators. The Synopsys VHDL simulator [29] was used for simulation for the implementation described in this thesis. Synopsys also provides a GUI (Graphical User Interface) to its simulator along with a waveform viewer called *waves* [29]. Using the above mentioned tools the programmer debugs his design tracing the critical signals for accuracy of his design. Finally, he checks the output file for the accuracy of the produced output. Note that the design at this stage does not have delays associated with it and hence, the simulator assumes a Unit Delay Model. In a Unit Delay Model, each gate in a gate-level design or each signal assignment statement in a behavioral design has a small delta delay attached to it. Hence each signal in the design is updated unit time after its immediate source is updated.

The designer then proceeds to synthesize the correctly simulated design. The synthesized model is downloaded onto the *Splash-2* hardware and output is obtained after running is compared to that obtained from simulation. The designer repartitions his design if any of the chips were oversized. The designer might change some statements in the design or remodel a part of the design such that it passes the synthesis stage successfully.

The synthesis stage can be a very crucial one because the actual hardware can cause many problems unforeseen at the modeling stage. Some of these problems may be user independent and totally dependent on the placement and routing for the design. The placement and routing algorithms do not produce the same output everytime they are run so that some problems may be present in a synthesized model and may then magically disappear in a model synthesized at a later time. Some other problems that a hardware can pose are from delay dependent paths in the design. As the simulation is done using the Unit Delay Model and the designer cannot have accurate delays until the end of the synthesis step, the outputs from the hardware may not match those of the simulation. To debug the design at this level, a debugger called *T2 interactive debugger* has been developed at SRC [14]. It has features that allow monitoring internal state variables and tracing internal signals. Thus, the designer can single-step in the hardware and evaluate the actual values of the signals from hardware to debug his design. If the outputs from the actual runs tally with the outputs from the simulation the design is pronounced working. Finally, the design is run at real-time to verify timing. If the design fails to meet the required timing, the designer repartitions the design to meet the required timing.

The final stage is integrating two or more working models to do a more complex job.

This can be done very easily on *Splash-2* because of its systolic array-like architecture. The programmer needs to feed the output of the first design to the input of the second. The final output is taken from the second stage.

5.3 The *Splash-2* simulation environment

5.3.1 VHDL

The VHSIC Hardware Description Language (VHDL) is an industry standard high-level language for the design and description of hardware systems [26]. VHDL was chosen as the programming language because it is hardware specific, simulatable and can be input to synthesis tools for automated design generation and placement. In addition to the aforementioned advantages, it is also a high-level language that can be used to model algorithms. VHDL is a powerful medium to represent designs in both the structural and behavioral domains.

5.3.2 *Splash-2* processing element model

The processing part entity in the *Splash-2* environment is fixed at a priori. The programmer has to model an architecture for this predefined entity whose inputs and outputs are already defined to accomplish his aspirations. The entity declaration for the Xilinx processing part is shown in Figure 5.2.

```
Library SPLASH2;
Use SPLASH2.TYPES.all;
Use SPLASH2.SPLASH2.all;
Use SPLASH2.COMPONENTS.all;
Use SPLASH2.HMacros.all;

--------------------------------------------------------------------------
-- Splash 2 Simulator v1.5 Xilinx_Processing_Part Entity Declaration
--------------------------------------------------------------------------
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;           -- Left Data Bus
        XP_Right          : inout DataPath;           -- Right Data Bus
        XP_Xbar           : inout DataPath;           -- Crossbar Data Bus
        XP_Xbar_EN_L      : out Bit_Vector(4 downto 0); -- Crossbar Enable (low-true)
        XP_Clk            : in Bit;                   -- Splash System Clock
        XP_Int            : out Bit;                  -- Interrupt Signal
        XP_Mem_A          : inout MemAddr;            -- Splash Memory Address Bus
        XP_Mem_D          : inout MemData;            -- Splash Memory Data Bus
        XP_Mem_RD_L       : inout RBit3;              -- Splash Memory Read Signal (low-true)
        XP_Mem_WR_L       : inout RBit3;              -- Splash Memory Write Signal (low-true)
        XP_Mem_Disable    : in Bit;                   -- Splash Memory Disable Signal
        XP_Broadcast      : in Bit;                   -- Broadcast Signal
        XP_Reset          : in Bit;                   -- Reset Signal
        XP_HS0            : inout RBit3;              -- Handshake Signal Zero
        XP_HS1            : in    Bit;                -- Handshake Signal One
        XP_GOR_Result     : inout RBit3;              -- Global OR Result Signal
        XP_GOR_Valid      : inout RBit3;              -- Global OR Valid Signal
        XP_LED            : out   Bit                 -- LED Signal
   );
end Xilinx_Processing_Part;
--------------------------------------------------------------------------
-- Splash 2 Simulator v1.5 Xilinx_Processing_Part Entity Declaration
--------------------------------------------------------------------------
```

FIGURE 5.2. Xilinx Processing Part Entity Declaration.

The generics board ID (BD_ID) and processing element ID (PE_ID) are used for identification of different processing elements on various boards. These are set to zero by default and the programmer need not worry about them. The declaration also shows the ports (input, output, and inout) that are used for communication. The *XP_Mem_A*, *XP_Mem_D*, *XP_Mem_RD_L*, and *XP_Mem_WR_L* are the memory address, memory data, memory read enable, and the memory write enable signals which are used to access the external memory of each PE. *XP_Left* and *XP_Right* are the left and right data ports. *XP_Xbar* and *XP_Xbar_EN_L* are the crossbar data and enable ports. *XP_Clk* is the clock to the system. *XP_LED* is a signal driving the LEDs on the array board. Refer to Section B.1 in Appendix B for further details.

A typical model of a Xilinx processing part is shown by the architectural body in Figure 5.3. This models a *left-to-right* architecture, in which the data coming into the chip from the left port of the processing element is transferred onto the right port.

The programmer first needs to instantiate pads to the ports he is using. This is done using the *Pad_Input and Pad_output* statements. Then he writes a VHDL model for the application. The memory and the crossbar have to be configured if they are used. Unused active low control ports are pulled up, active high control ports are pulled down, and others are tristated. The internal signals (*Left* and *Right* in the above example) can not be accessed from a different chip because the Xilinx processing part entity is predefined. Thus, the programmer has to use the available resources to communicate between chips. Refer to [14] for further details.

Synchronous designs are less erratic on hardware than their asynchronous counterparts. Hence, the *wait* statement is used to synchronize all signal assignments in the design. In the design shown in Figure 5.3, the value of signal *Left* is assigned to signal *Passthru* and the signal *Passthru* to signal *Right* only on the 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. Refer to [26] for more information about VHDL and its syntax.

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

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

   P0: process
   begin
    wait until XP_Clk'Event and XP_Clk = '1';
    Right                 <= PassThru after 2 ns;
    PassThru              <= Left after 2 ns;
   end process P0;

   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";
   XP_LED                 <= '1';

end Left_to_Right;
```

FIGURE 5.3. Example Architecture: Left To Right.

CHAPTER 6

Splash-2 Implementation of Region Detection

6.1 Introduction

Detecting regions, as described in Chapter 2, is the process of identifying the connected components in an image. The various regions determined are used to identify different objects in the image. The implementation on *Splash-2* identifies the regions in a given image and assigns various colors to them for displaying on the monitor. The design utilizes nine of the sixteen available processing elements on the *Splash-2* Array Board. In this chapter, a general description of the algorithm and the partitioning of the design have been discussed in Section 6.2 and Section 6.3, respectively. A detailed description of the processing elements is in Section 6.4. The results obtained for the above experiment are quantified in Section 6.5.

6.2 Splash-2 Implementation of Region Detection

Referring to Figure 2.1, the input image pixel from the frame grabber card is an 8-bit gray-scale quantity. Each pixel in the input image, *I(P)*, is thresholded to generate a 1-bit black/white image, *T(P)*. A first-pass labelling process operates on the thresholded image (*T(P)*) to produce a inconsistently labelled image, *F(P)*, and a set of equivalent pairs, *{EQ(m1, m2)}*. Each label in the image *F(P)* is obtained by observing the labels assigned to its neighbors. Every foreground pixel inherits an existing label should one of its neighbors belong to the foreground, otherwise a new label is assigned to the it. The equivalence pair is also obtained from the labels assigned to the neighbors of the current pixel. An equivalent pair (*EQ(m1,m2)*) is generated every time two neighbors of a pixel have different labels, *m1* and *m2*. These are stored in a look-up table which is updated for every equivalent pair generated. At the end of the first pass processing, each pixel belonging to *F(P)* is mapped to an equivalent consistent label from the look-up table to produce a component labelled image, *R(P)*.

6.3 Problem Partitioning

The region detection and labelling has been implemented as a single board design on *Splash-2*. The inter-chip communication and partitioning are shown in Figure 6.2. The size of the input image is 512x512 pixels and the system clock frequency is 10MHz. The design accepts the eight bit gray-scale valued pixels from the camera in raster scan order. The region detected and labelled image is sent to the camera in real-time. As stated in Chapter 2, the design is a two-pass process. Hence the image is processed twice before it is sent to the monitor for display. This causes a pipeline latency of one frame, which means that a frame entering in the N^{th} frame slice exits the pipeline in the $(N+2)^{th}$ frame slice.

The algorithm for the implementation was described in Chapter 2. The task of problem partitioning can be approached by first identifying the various independent tasks in the algorithm. The tasks can then be assigned to different processing elements. Each processing element may execute one or more tasks depending on their complexity and communication requirements. If the hardware required to implement each task is smaller than the capacity of a processing element, the inter-chip communication resources, which are limited in *Splash-2*, play a major role in problem partitioning. Some of the independent tasks identified from the algorithm are:

1. Image Thresholding,
2. Labelling (First Pass Processing),
3. Reporting inconsistent labels,
4. Data Storage for second pass processing,
5. Date Retrieval for second pass processing,
6. Processing and maintaining equivalence table for mergers, and

7. Merging (Second Pass Processing).

A data-flow model of the above algorithm is shown in Figure 6.1. It highlights the tasks that can be done in parallel. We recall that the any incoming pixel is thresholded, labelled, and stored in the first pass processing. Also, any inconsistencies observed in the labelling process are noted in the form of an equivalence table. Hence, Tasks 1, 2,3,4 and 6 are done in the first pass, while Tasks 5, 6 are accomplished in the second pass. Also, 2 and 3 are done in parallel, and so are Tasks 4 and 6.

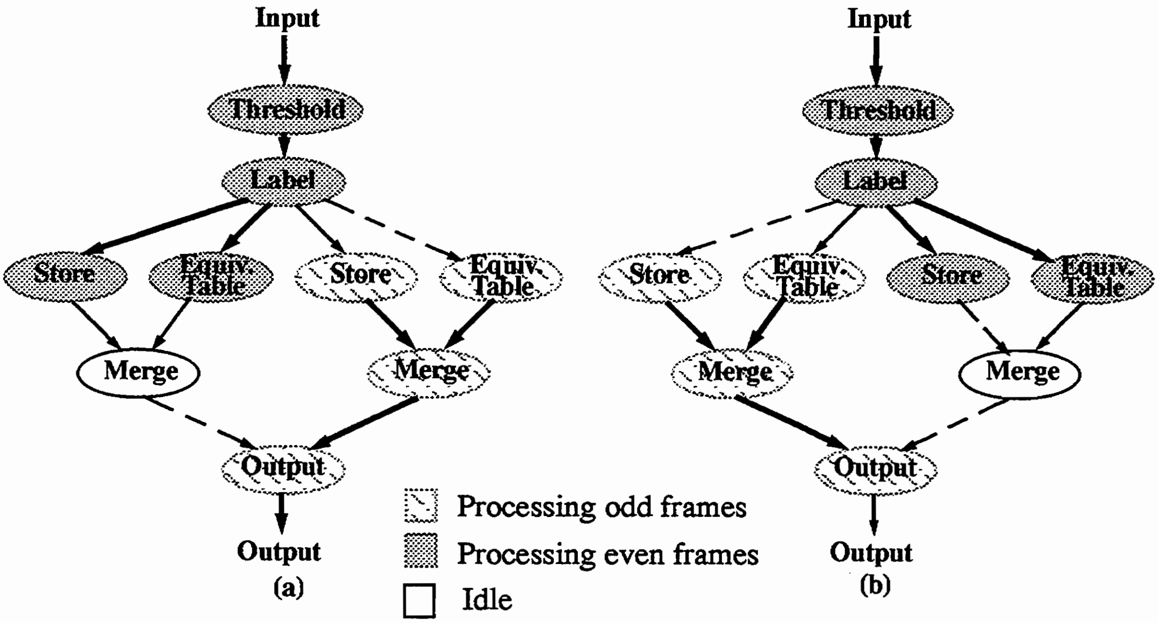


FIGURE 6.1. Data flow through the algorithm.

With the aforementioned information about the various tasks to be accomplished from data flow model in Figure 6.1, one can visualize the pipeline of tasks the data passes through. The pipeline has five stages, the first of which takes care of thresholding the incoming image. The second stage consists of Tasks 2 and 3, and the third consists of Tasks 4 and 6. Further processing of pixels from the current frame is stopped and the pixels are stored until all the pixels in the frame have been processed through the third stage of the pipeline. The pixels are retrieved in the fourth stage and sent for merging in the fifth. An equivalence table created in the previous frame slice is used to give fresh and consistent labels to the regions. Hence stages three, four, and five in the pipeline need to be duplicated and used alternately.

The next step is to determine the required communication between various stages. This is the process of establishing the protocol needed for the correct functioning of the pipeline. The tasks are then assigned to various processing elements such that the hardware required for the tasks assigned to each processing element is less than the capacity of the processing element and the communication resources available satisfy the requirement. The signals to and from the processing elements are assigned to the available resources. A block diagram of the design representing the communication is shown in Figure 6.2.

The design consists of nine processing elements. From Figure 6.2, one can see that there exists a correspondence between the aforementioned tasks and the processing elements. Task 1 is performed by PE-0. Processing Elements PE-1 and PE-2 work in unison to accomplish Tasks 2 and 3, PE-4 takes care of Tasks 4 and 5 for odd numbered frames, while PE-6 performs the same for even numbered frames. Tasks 6 and 7 are done by PE-5 for odd numbered frames and by PE-7 for even numbered frames.

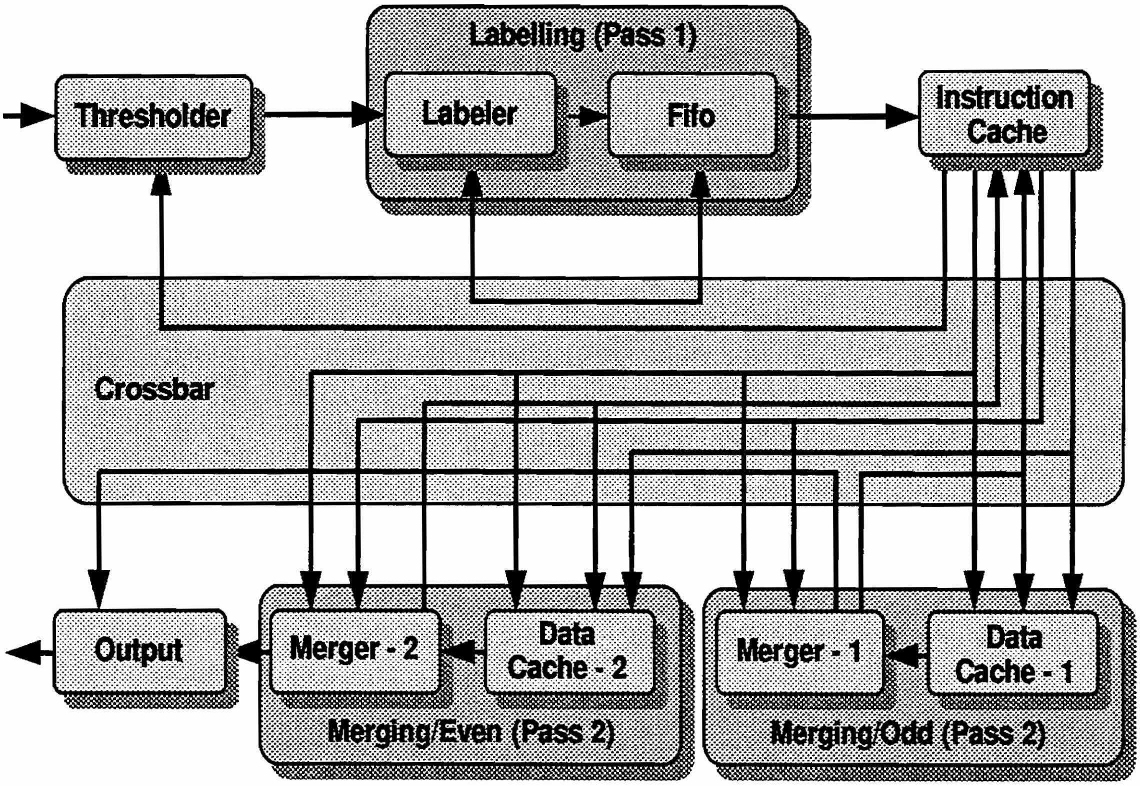


FIGURE 6.2. **Block Diagram for the implementation on *Splash-2*.**

The processing elements PE-3, and PE-8 perform tasks not mentioned in the above list of tasks. PE-3 places the generated data on the cross-bar for other processing elements to access. It also handles the task of providing the merge information to the respective mergers (M1 for odd frames and M2 for even frames), if and when they are ready to accept them. If a merger inconsistency occurs and the Merger (M1/M2) is not ready, PE-3 buffers the information in its external memory and retrieves it at a later time when the corresponding Merger PE is ready. PE8 collects the region detected frames arriving from M1 and M2 and sends them to PE16, which in turn sends them to the monitor for display.

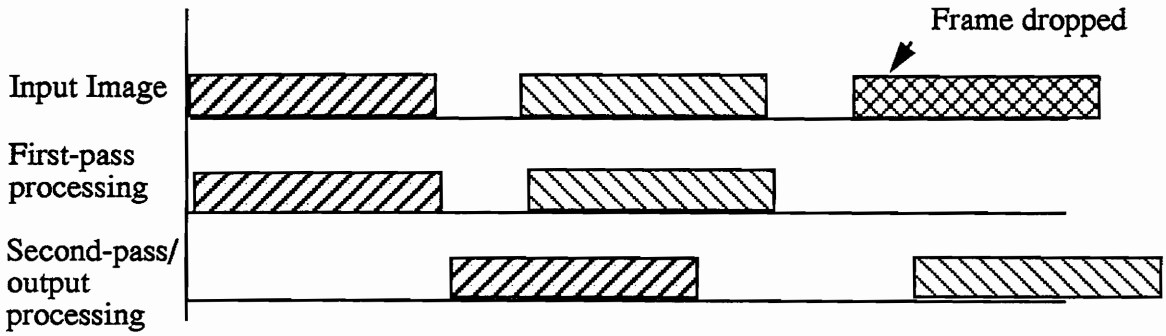


FIGURE 6.3. **Time-line illustration of the pipeline**

A time-line illustration of the pipeline is shown in Figure 6.3. The first-pass processing commences with the advent of the frame and is completed when the last merger instruction updates the equivalence table. The second-pass processing does not start until the first-pass is complete and this is dependent on the number of mergers in the image. The earliest time the second-pass processing may start is one clock cycle after the end of the frame. As the number of regions the design can handle are limited to thirty two, a loose worst case time for the start of second-pass processing is the beginning of the next frame. This may be attributed to the fact that there exists around 40 lines of invalid data between two valid frames and due to the fact that the algorithm takes 100 clock cycles to process a merger instruction that effects the look-up table while it only takes 7 clock cycles to process a merger instruction that does not update the look-up table. If the first-pass processing of a frame is not complete when a new frame arrives, it is dropped.

The various handshakes among the above mentioned processing elements are shown in Figure 6.2. The individual functions of each processing element and their input/output block diagrams will be discussed in detail in Section 6.4.

6.4 Processing Element Architectures

6.4.1 Introduction

This section describes the functionality of the processing elements with the help of state diagrams and logical input/output block diagrams. The state diagram explains the response of the processing element to changes in inputs. The logical input/output block diagram describes the inputs and outputs to the processing element. Figure 6.2 and the following section help describe the design implemented on *Splash-2* for Region Detection and Labelling.

6.4.2 Thresholder

6.4.2.1 Introduction

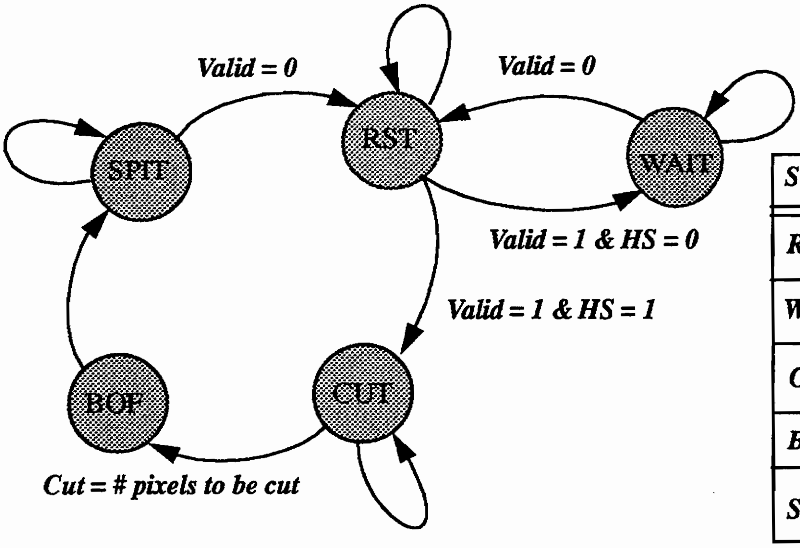
The basic function of the processing element is image thresholding. Refer to Chapter 2 for further details about thresholding. We recall that the image enters *Splash-2* from the left port of PE-0 in the raster scan order and each pixel is represented by an eight-bit grey scale value. This eight-bit value is converted to a one-bit value by using a predetermined threshold. The threshold is hard-wired into the design. If this value needs to be changed, one has to edit the VHDL file and re-synthesize the design.

In addition to image thresholding it also preforms such tasks as optionally omitting specified number of pixels from the beginning of every frame, and skipping frames when the PE-3 (or Icache) has not finished sending out all the instructions it received during the previous frame. It also performs the function of generating various control signals such as **BOFR** (Beginning of Frame), **BOLN** (Beginning of Line), **EOFR** (End of Frame) and **EOLN** (End of Line) using the **Valid** signal coming into the PE from the frame grabber card. Thus the design requires no other signals from the camera other than **Valid** and the Data itself. It assumes that **Valid** is never negated during the frame transmission.

6.4.2.2 State Diagram

The state-diagram of the processing element PE-0 shown in Figure 6.4 illustrates the various states for the design. **RST** is the reset state. The PE enters the **RST** state on POR (Power on Reset), and remains there until the first frame arrives. A protocol has been implemented that informs the PE if PE-3 is ready and waiting. If the handshake is a *1* (i.e. PE-3 is ready and waiting), the PE advances into the **CUT** state. Otherwise (i.e. PE-3 is not ready to accept the frame) it proceeds into the **WAIT** state and the current frame is skipped.

The **CUT** state has been added to cut some unwanted pixels from the top of the frame. Due to some hardware problems, the first four pixels coming from the camera were erroneous. To correct this problem a variable called **CUT** has been hardwired which determines the number of pixels that have to be cut from the beginning of the frame. The PE stays in the **CUT** state until the number of pixels to be cut arrive at the input. It then proceeds to the **BOF** (beginning of frame) state.



State	*Description*
RST	*Restart/Idle State* *All Outputs Negated*
WAIT	*Wait State* *No operation*
CUT	*Cut State* *Pixels are dropped*
BOF	*BOF State* *BOFR is Asserted*
SPIT	*Spit State* *Thresholding &Control* *Signal Generation*

FIGURE 6.4. Thresholder: State Machine.

The PE stays in the current state only for one clock cycle. The output control signal **BOFR** (beginning of frame) gets asserted in this state. The input is passed to the output after thresholding. The PE then moves on to the **SPIT** state.

In this state the PE keeps *spitting* the thresholded output while generating signals such as **EOLN** (end of line), and **BOLN** (beginning of line). When the end of frame is reached (we conclude that we reached the end-of-frame when the **Valid** goes back to *Zero*), it generates a **EOFR** (end of frame) signal and goes back to **RST** state, where it waits for the next frame to arrive.

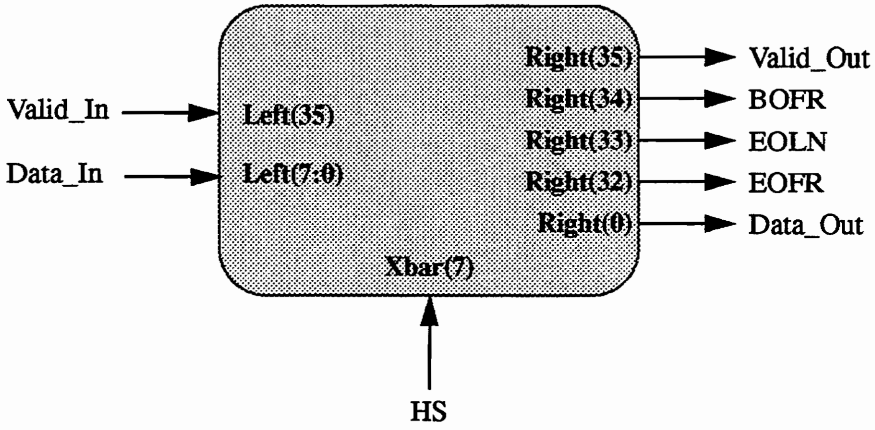


FIGURE 6.5. **Thresholder: Logical input/output block diagram.**

6.4.2.3 Logical input/output block diagram

Figure 6.5 shows the signals flowing in and out of the processing element PE-0. The eight-bit **Data_In** (Left(7:0)) and the one-bit **Valid_In** (Left(35)) are the data and valid signals coming into the *Splash-2* system from the frame grabber card. The handshake bit **HS** (Xbar(7)) is driven by PE-3. This, when high, informs the PE that PE-3 is ready and waiting and that PE-0 can pass the next newly arriving frame for processing. PE-0 only checks this handshake when a new frame arrives. It skips the frame if PE-3 is not ready for the present frame. **Data_Out** (Right(0)) is the one-bit thresholded form of the input and **Valid_Out** (Right(35)), derived from **Valid_In**, indicates to PE-1 that it process the data on its left port.

BOFR, **EOLN**, and **EOFR** are the control signals generated by the PE which are useful to other PEs.

6.4.3 Labeller

6.4.3.1 Introduction

The processing element accepts the one-bit thresholded image and performs first pass image labelling on it. Refer to Chapter 2 for further details on image labelling. The procedure consists of utilizing the labels of the immediate neighbors and giving a color to the current pixel. Only the neighbors already covered in the raster-scan order are considered for labelling as they are the ones that have a label attached to them by the time the current pixel is processed.

Hence the processing element has to store only the row above the current for using at a later time. This is one of the other functions PE-1 accomplishes with the help of PE-2. Pairs of the five-bit labels are alternately stored in the external memories of the processing elements PE-1 and PE-2. They are retrieved at a later time and fed back to PE-1 (or Labeller) for processing.

6.4.3.2 State Machine

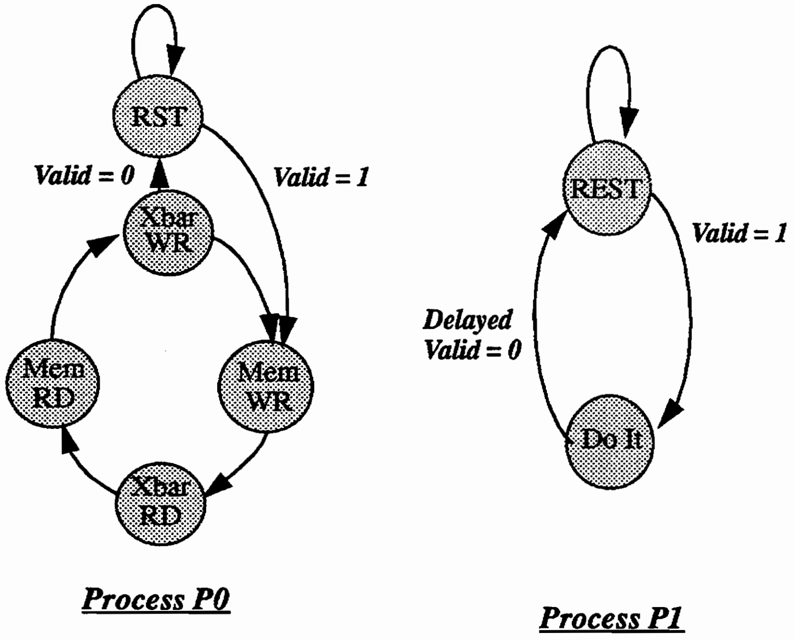
The processing element has two different processes, one for taking care of storing and retrieving the labels and the other for generating the labels depending on the labels given to its neighbors. The state machine for this is shown in Figure 6.6. Process P0 is responsible for the former while Process P1 is for the latter.

Process P0 starts and remains in the reset state **(RST)** in between frames. The four states **MemWR, XbarRD, MemRD,** and **XbarWR** (the keyword **Mem** refers to the external memory attached to PE-1 and **Xbar** refers to the crossbar. The keywords **WR** and **RD** refer to the type of operation being performed) are the active states and the process rotates in these four states while a valid frame is being input. The process stores the incoming data in two different external memories. One is that of the PE-1 (Labeller), and the other is that of PE-2 (FIFO). Process P0 communicates with PE-2 (FIFO) over the crossbar and it writes to and reads from its external memory through the interface provided in the Processing Element (Refer to Section 4.2 and Figure 4.3). The process combines the incoming pixels into pairs and organizes them such that each memory receives a pair alternately. If the first pair of pixels were sent to the external memory of PE-1, the second pair would be routed to that of PE-2, the third would be dispatched to PE-1, and so on. The data that is being written into the memories is required by the PE for processing the next row. Hence, the process retrieves the same from the external memories and arranges it into registers **In_Int1**, **In_Int2**, **In_Ext1**, and **In_Ext2** (the first part of the name states that the pixels are coming *in* from the memory, while the second part determines which memory, internal or external, the pixel has come from. The number refers to the order of the pixels). These are reordered and fed to registers **UL**, **UR**, and **UC** (the acronyms stand for Upper-Left, Upper-Right, and Upper-Center pixels with respect to a Current pixel, **Curr**), depending on the state of the Process P0.

Hence the process has four basic operations to perform.

1. Write to the external memory of PE-1 via the local external memory interface.
2. Read from the external memory of PE-1 via the local external memory interface.
3. Write to the crossbar.
4. Read from the crossbar.

Tasks 1 to 4 are recursively executed by the process in the states **MemWR**, **MemRD**, **XbarWR**, and **XbarRD** respectively. While the external local memory is directly controlled by the process, the transactions through the crossbar are synchronized by the **BOFR** (Beginning of Frame) signal generated by PE-0. Thus the synchronization of the state machines for process P0 and that for PE-2 is crucial for the correct working of the algorithm.



State	*Description*
RST	*Restart/Idle State* *All Outputs Negated*
Xbar WR	*Crossbar Write State*
Mem WR	*Memory Write State* *Mem_WR Asserted*
Xbar RD	*Crossbar Read State*
Mem RD	*Memory Read State* *Mem_RD Asserted*
REST	*Restart/Idle State* *All Outputs Negated*
DoIt	*Label State* *1st Pass Labels Assigned*

FIGURE 6.6. Labeller: State Machine.

Process P1 has two different states. In the **REST** state the process resets all its signals and waits for a valid frame arrive. Upon arrival of a valid frame, the process moves on in the **DOIT** state in which it generates labels for the pixels coming in from the left port, using the labels assigned to the neighbors of that pixel.

Process P1 processes the labels in the registers **UL**, **UR**,**UC**, and **L** (all the labelled pixels in the active window adjacent to the currant pixel) to generate a label for the current pixel (**Curr**). It also checks for discrepancies in the assigned labels and reports the labels that are to be merged. The process first checks whether the current pixel belongs to the foreground. For every foreground pixel encountered, it checks to see if any of its neighbors belong to the foreground. If only one of its neighbors belongs to the foreground, the current pixel gets the label of that neighbor. If more than one neighbor belongs to the foreground, the process first checks to see if there is an inconsistency in the labels. That is, it checks to see if any two neighbors of the current pixel which belong to the foreground have different labels. A flag is raised if an inconsistency is discovered and the inconsistent labels are passed to PE-3 for further processing. The process then assigns the current pixel one of the inconsistent labels. If none of the neighbors belongs to the foreground, the process assigns a fresh and distinct label to the current pixel.

6.4.3.3 Logical input/output block diagram

The logical input/output block diagram of PE-1 is shown in Figure 6.7. The primary inputs to the PE are the **Control_In** and the one bit **Data_In**. The **Control_In** is a *frame valid* signal from PE-0, while the **Data_In** is a one bit thresholded value for the incoming pixel. **Fifo_In** and **Fifo_Out** are signals coming from and going to the crossbar, carrying data from and to PE-2, respectively. The crossbar configuration for the transaction is static (which means it does not change dynamically). **Xbar(15:0)** are configured to be inputs, while **Xbar(31:0)** are configured to be outputs from PE-1.

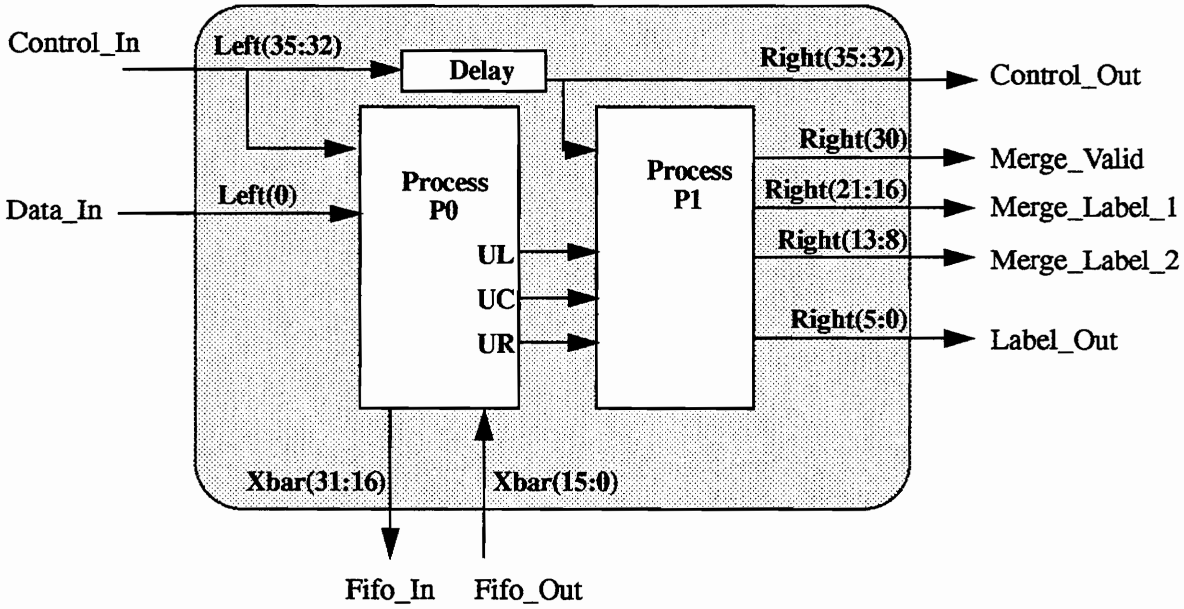


FIGURE 6.7. **Labeller: Logical input/output block diagram.**

The output **Control_Out** is a delayed version of **Control_In,** which when asserted signifies the validity of the label output by Process P1. **Label_Out** is the first-pass label of the current pixel output by process P1. When an inconsistency is observed in the labels, the process asserts **Merge_Valid**. It then places the inconsistent labels on **Merge_Label_1** and **Merge_Label_2**.

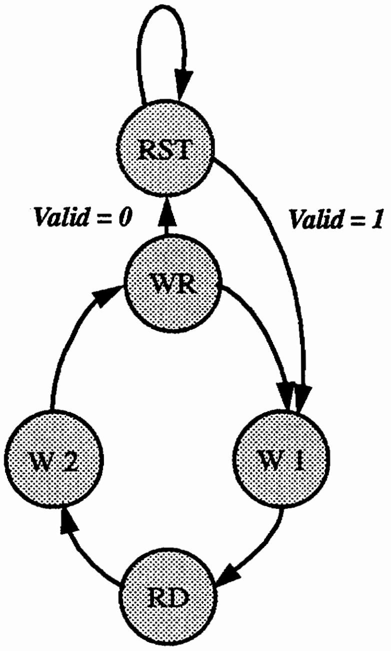
6.4.4 Fifo

6.4.4.1 Introduction

The primary function of the PE is to assist PE-1 (Labeller) in first-pass labelling. Recall from Chapter 2 that row *(n-1)* is required for processing row *n*. Hence, the PE functions as a FIFO (First In First Out) and delays the input stream by the length of a row.

6.4.4.2 State Machine

The PE starts in a reset state called **RST**. When a **BOFR** (Beginning Of Frame) is encountered, the process enters a transition state named **W1**. From this point the process rotates in the four states **MemRD, W2, MemWR,** and **W1** until an **EOFR** (End of Frame) is encountered. When an **EOFR** is encountered, the PE returns into the **RST** state. The states **W1** and **W2** are transition states, and the process is idle in these states. **MemRD** and **MemWR** are the active states, wherein the process reads and writes to the external memory through the MemBus. In the **MemWR** state the PE takes the data off the crossbar and stores it in the external memory. In **MemRD** state the PE reads from the memory and writes the data back to the crossbar. Any pixel written into the memory during n^{th} clock-cycle is used again in the $(n+r)^{th}$ clock-cycle, where *r* is the number of pixels in a row of the incoming image. This is accomplished by writing to a location *r/4* words away from the read counter. The read counter is incremented every four clock-cycles, and hence the location $(n+r/4)$ is read exactly after r clock-cycles. In reality, the offset from the read counter is less than $r/4$ due to the pipeline delays encountered in storing and fetching the data from the memory.



State	*Description*
RST	*Restart/Idle state.* *All outputs negated*
WR	*Write State* *Memory Write Line Asserted*
RD	*Read State* *Memory Read Line Asserted*
Wx	*Transition States* *No Operation*

FIGURE 6.8. FIFO: State Machine.

6.4.4.3 Logical input/output block diagram

The logical input/output block diagram for the PE has a **Valid_In** signal coming in to the PE from the left. It starts and synchronizes the state machine with the one in PE-1. There is a 16-bit label pair **(Fifo_In)** coming into and a 16-bit label pair **(Fifo_Out)** leaving the PE via a crossbar. A label pair entering the system from **Fifo_In** at a time *n* exits the system through **Fifo_Out** at a time *(n+r)* (*r* is the number of rows). The PE also passes all the bits on the left port coming from PE-1 to the right port to PE-3 for further processing.

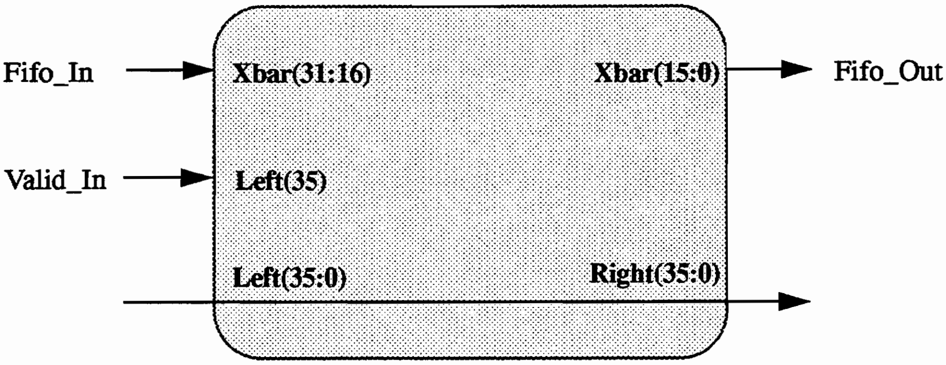


FIGURE 6.9. **FIFO: Logical input/output block diagram.**

6.4.5 Icache

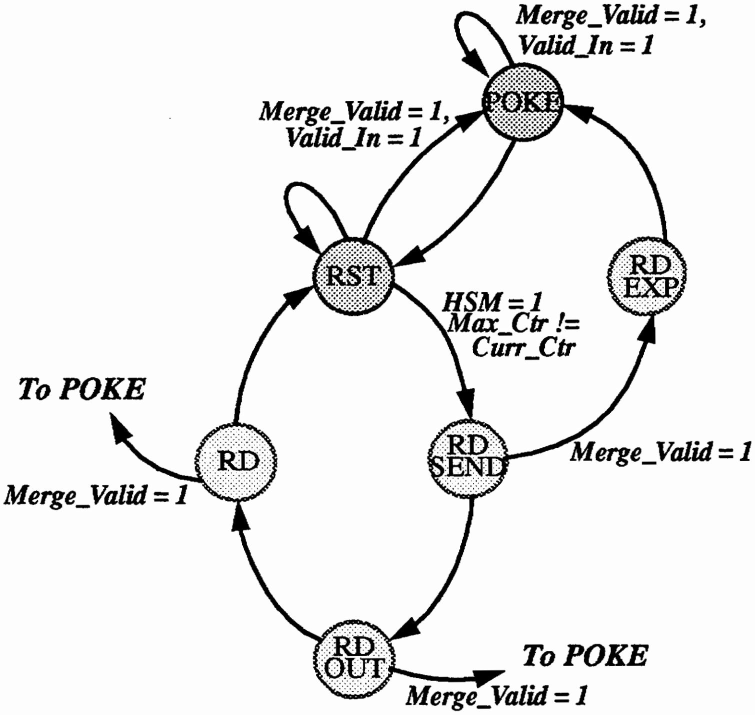
6.4.5.1 Introduction

The inconsistencies generated by PE-1 are stored in the PE temporarily. The inconsistencies, also called the merger instructions, are fed to the crossbar when PE-5 (for odd numbered frames) or PE-7 (for even numbered frames) is ready for further processing. The PE also places the data and control signals such as **BOFR**, and **EOFR** onto the crossbar. The PE generates and distributes a signal called **Frame_no**, which is used to distinguish between odd and even numbered frames, via the crossbar.

The PE processes the merger instructions coming in via the left port. The assumption behind the algorithm is that the inconsistencies are spread-out in time and are infrequent. They may occur back to back but by queuing the merger instructions until the Merger PE (PE-5 for odd numbered frames and PE-7 for even numbered frames) is ready makes optimal use of not only the frame valid time but also the dead time between frames. Hence, the Merger PE utilizes the time until the start of the next frame to complete the processing of the merger instructions. The PE accepts an incoming merger instruction and writes it to the memory. It remains in the memory until the Merger PE is ready to accept a new merger instruction. Once the Merger PE is ready (**HS_M1** or **HS_M2** asserted), the PE fetches the next unprocessed merger instruction from the local memory and places it on the crossbar. The PE aborts the attempt to fetch the merger instruction if it encounters a new merger before it completes the present transaction. It writes the new merger into the memory and fetches the aborted location again. The PE has two counters called the **Max_Ctr** and **Curr_Ctr** to assist with the procedure. Initially, the **Max_Ctr** and the **Curr_Ctr** have the same value. Whenever a new merger instructions comes from PE-1, the PE writes it into the memory to a location pointed by **Max_Ctr** and increments the same. When the Merger PE is ready, it reads the instruction out of the location pointed by **Curr_Ctr** and increments it upon successful completion of the transaction. Thus, a queue of merger instructions is built in the memory, with **Max_Ctr** pointing to the tail of the queue and **Curr_Ctr** pointing to its head. Merger PE requires one hundred clock-cycles to process an instruction. The next fetch occurs when the Merger PE finishes processing the current instruction. The fetching continues till **Curr_Ctr** equals the **Max_Ctr**. The queue is empty when **Curr_Ctr** equals **Max_Ctr**. The PE then waits for a fresh merger to arrive.

6.4.5.2 State Machine

The PE starts in the reset state called **RST**. It proceeds into the **POKE** state when it encounters an inconsistency, where it does a *mem_write* to the local memory to write the incoming pair of inconsistent labels into it. It remains in the state if another inconsistency occurs back-to-back or it returns to the **RST** state. If the PE is in the **RST** state and no inconsistency has occurred, it checks if the memory queue is empty (**Max_Ctr** not equal to **Curr_Ctr**). If the queue is not empty, it attempts to fetch a instruction from the local memory and proceeds to **RDSEND**. Recall that a memory fetch takes three clock cycles. The process then proceeds to the **RDOUT** state if it does not encounter a new merger from PE-1. When it reaches the **RD** state from **RDOUT** state, it puts the data from the **Mem_D** (MemBus Data) onto the crossbar, increments the **Curr_Ctr**, and goes back to the **RST** state. If a new merger arrives while the PE is in any of the **RDSEND, RDOUT,** or **RD** states, the PE aborts the process, executes a memory write to store the new merger to the external memory, and jumps to the **POKE** state.



State	*Description*
RST	*Restart/Idle State* *All Outputs Negated*
POKE	*Insert State* *Lookup Table Updated*
RD SEND	*RD SEND State* *Issue a Memory Read*
RD OUT	*Read Transition State* *Abort for new instruction*
RD	*Instruction Output State* *Issue an Inst. to Crossbar*
RD EXP	*Special Wait State* *Avoid a WAR exception*

FIGURE 6.10. **Icache: State Machine.**

A memory write cannot be issued in a clock cycle succeeding a memory read. In order to take care of this special case, the PE enters a state called **RDEXP** where it buffers the incoming merger instruction. It issues a memory write in the next clock cycle and moves to the **POKE** state. It resumes fetching new instructions after it returns to the **RST** state. The **Curr_Ctr** is not incremented unless the crossbar write is successful, so the next fetch after an unsuccessful write defaults to the same memory location.

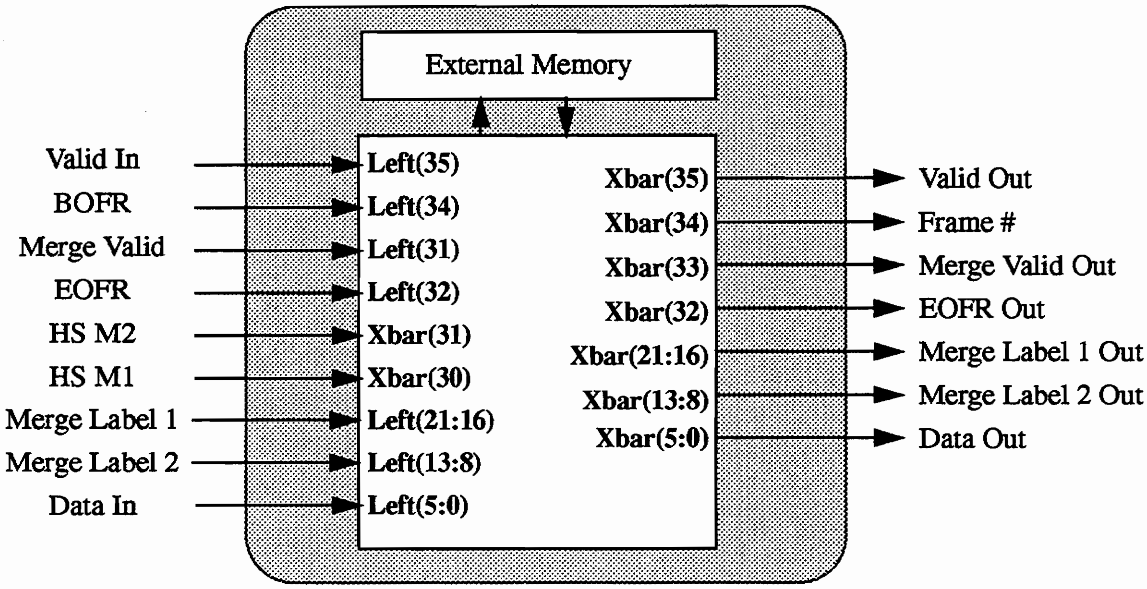


FIGURE 6.11. Icache: Logical input/output block diagram.

6.4.5.3 Logical input/output block diagram

The inputs to PE-3 include the control signals **Valid_In, BOFR,** and **EOFR**. The 5-bit **Data_In** is put onto the crossbar after buffering. The **Merge_Valid** signal, when active, informs the PE that a new merger instruction has arrived. The PE then stores **Merge_Label_1** and **Merge_Label_2** into the memory. The PE looks at **HS_M1** (for odd numbered frames) or **HS_M2** (for even numbered frames) to see it the Merger PE (PE-5 for odd numbered frames and PE-7 for even numbered frames) is available.

The outputs **Valid_Out** and **Data_Out** are buffered versions of **Valid_In** and **Data_In**. **Frame_#** is a bit generated in the PE and toggled at the beginning of every frame. It is used to distinguish between even and odd numbered frames. **Merge_Valid_Out** is asserted when **Merge_Label_1_Out** and **Merge_Label_2_Out** are written out onto the crossbar by the PE from the local memory. Finally, **EOFR_Out** is a delayed version of the input control signal EOFR. It is used to signify end of frame for later stages of the pipeline.

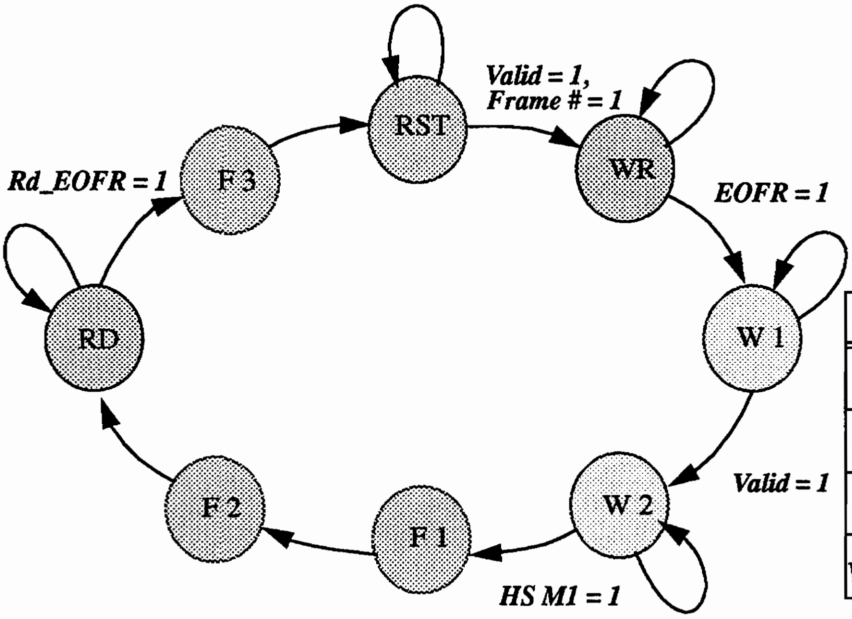
6.4.6 Dcache

6.4.6.1 Introduction

Recall that the algorithm iterates twice through the image. The labels generated by PE-1 during the first pass have to be stored and retrieved after PE-1 is finished with the first-pass processing. This task is accomplished by PE-4 (or **Dcache-1**) for odd-numbered frames and PE-6 (or **Dcache-2**) for even-numbered frames. PE-4 receives the generated labels from the crossbar, (we recall that PE-3 puts the labels onto the crossbar) and stores the labels in its local memory while PE-1 places the processed labels belonging to an odd numbered frame on the crossbar. It then waits for the next frame to arrive and produces the first-pass processed data to PE-5 (or **Merger-1**) for second-pass processing.

6.4.6.2 State Machine

The PE starts in the reset state called **RST**. It then waits for PE-1 to produce the first-pass processed labels. When PE-1 starts to produce the labels, the PE stores them in its local memory. When an end-of-frame **(EOFR)** is encountered, the PE enters a wait state called **W1**, where it waits for the next frame to arrive. Upon arrival of the next frame, the PE enters another wait state called **W2** and proceeds from this state to output only when the corresponding Merger PE (PE-5 for PE-4 and PE-7 for PE-6) is available. When the Merger PE is available, the PE starts reading the first-pass processed labels which were written into the local memory while the preceding frame was processed. These are then sent to the corresponding Merger PE for further processing. The states **F1** and **F2** are the pipeline stages for a memory read, wherein the PE has to wait for the requested data from the memory. After the PE is finally in the **RD** state, it proceeds to the state **F3** after it has finished reading all the labels that were written into the local memory. It then returns to the **RST** state after resetting the *valid* and *memory* counter.



State	*Description*
RST	***Restart/Idle State*** ***All Outputs Negated***
RD	***Read State*** ***Mem_RD Asserted***
WR	***Write State*** ***Mem_WR Asserted***
W*/F*	***Wait/Flush States*** ***No Operation***

FIGURE 6.12. Dcache: State Machine.

6.4.6.3 Logical input/output block diagram

The primary input to the PE is **Data_In** which it receives from the crossbar, and the primary output is **Data_Out** which it puts on the right port for the corresponding Merger PE. The input **Valid_In** is active when the **Data_In** on the crossbar is valid. The signal **Frame_#** assumes a value of *0* when the data on the crossbar corresponds to an odd numbered frame and *1* for an even numbered frame. The PE stops writing to the memory when the input signal **EOFR** is asserted. The PE assumes that its corresponding Merger PE is available when handshake signal **HS_M1** (or **HS_M2** for **Dcache-2**) is active.

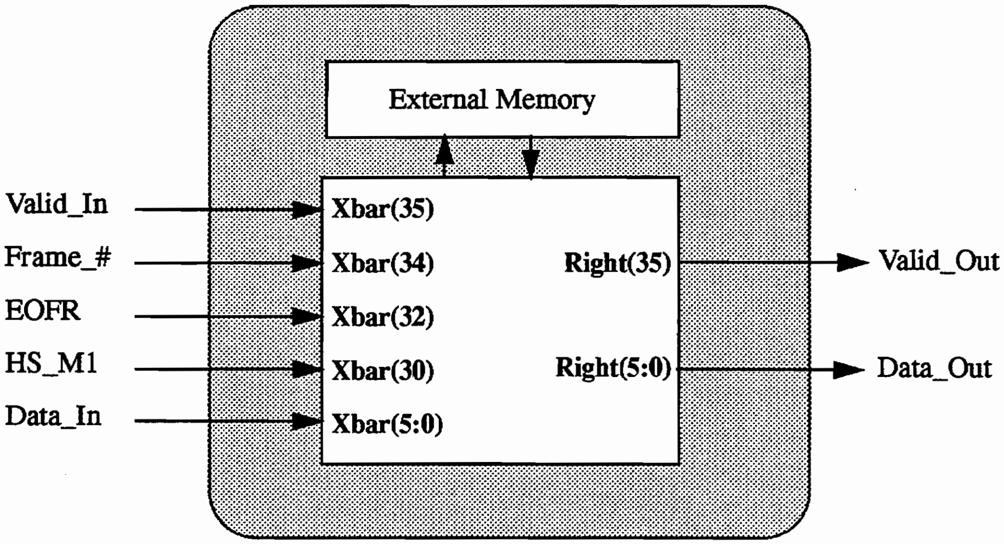


FIGURE 6.13. **Dcache: Logical input/output block diagram.**

6.4.7 Merger

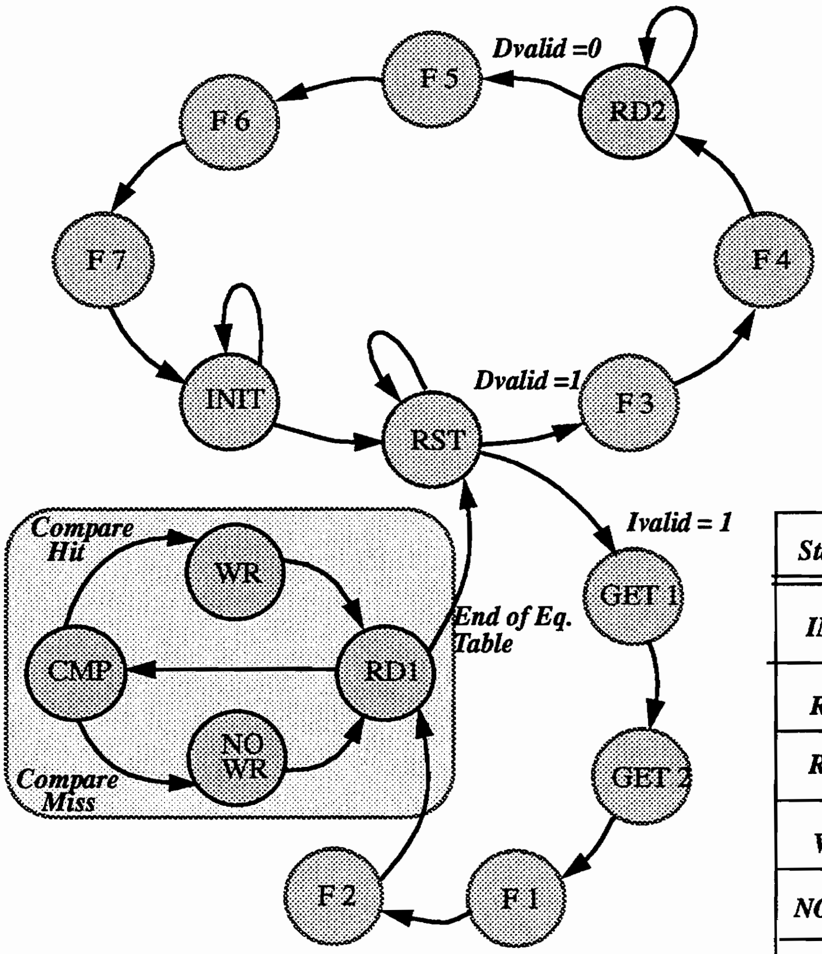
6.4.7.1 Introduction

Merger is the largest chip in area in the whole design since it manages the equivalence table. The table is updated during one frame slice and is made use of during the next. The system contains two such PEs (M1/M2), and when one of them is doing the former task, the other does the latter. The equivalence table is implemented as a look-up table in the external memory. Refer Section 2.3 in Chapter 2 for more information on the look-up table implementation.

6.4.7.2 State Machine

The state diagram for the PE is illustrated in Figure 6.14. The PE starts in the **INIT** state, and transitions into the **RST** state, where it waits for the instructions. The instruction contains two labels, *P* and *Q*, that are equivalent. When an instruction or an inconsistency arrives the PE enters the **GET1** state, where it fetches *M(P)*, the value at the memory location pointed by *P*. It then proceeds to the **GET2** state, where it fetches *M(Q)*, the value at the memory location pointed by *Q*. The states **F1** and **F2** are transition states. Recall that a memory read access requires three clock cycles and that if an address is latched at the rising edge of the N^{th} clock cycle, the corresponding data is available at the rising edge of the $(N+3)^{rd}$ clock cycle. Hence, the data *M(P)* is latched in state **F1** and *M(Q)* in state **F2**. The memory address counter is reset in the **F2** state. The PE issues a read to all the memory locations, one by one, in the **RD1** state. On the next entry to the **CMP** state, it checks if this matches *M(P)* and if so, it proceeds into the **WR** state. Otherwise it proceeds to the **NOWR** state. In the **WR** state, the PE replaces the contents of the memory location *P*, *M(P)*, with *M(Q)*. In the **NOWR** state the PE stays idle. The PE returns to the **RD1** state from both **WR** and **NOWR** states and a read to the next memory location is issued. This continues until the end of the table is reached and the PE returns to the **RST** state where it waits for the next instruction. The handshake signal **HS_M1 (HS_M2)** is asserted while **Merger-1 (Merger-2)** is performing the above process.

In the second phase, the updated equivalence table is used to re-map the labels from the first pass, which are then stored in the corresponding Dcache. The Dcache assures that its corresponding Merger is in the **RST** state before it starts to output the first pass labels. The Merger enters into the transition state **F3** as soon as the first datum arrives from the Dcache. When received it issues a memory read to the location pointed by the datum. It enters the **RD2** state through the next transition state **F4**. Reads are issued in consecutive cycles to the memory locations indexed by the incoming data. In the **RD2** state, the PE passes the label read from the memory to its output port. This continues until the end of the frame is reached. At the end of frame, the PE goes through the transition states **F5**, **F6**, and **F7** to flush the data from the read pipeline, and enters the **INIT** state. In the **INIT** state, the PE initializes the memory look-up table and returns to the **RST** state where awaits another frame.



State	*Description*
INIT	*Reset state* *Initialize Memory*
RST	*Reset state* *All outputs negated*
*RD**	*Read state* *Mem_RD asserted*
WR	*Write state* *Mem_WR asserted*
NOWR	*Idle state* *No operation*
CMP	*Compare state* *(M(Mem)==M(P)) ?*
*GET**	*Read state* *Look up Eq. table*
*F**	*Wait States* *No operation*

FIGURE 6.14. Merger: State Machine.

6.4.7.3 Logical input/output block diagram

The PE inputs the one-bit **Valid_In** and the five-bit **Data_In** from the corresponding Dcache. The control signals **Frame#, EOFR, Merge_Valid,** and the equivalent labels **Merge_Valid_1** and **Merge_Valid_2** are fetched from the crossbar. The primary outputs **Valid_Out** and **Data_Out** are placed on to the right port or to the crossbar. The control signal **HS_M1 (HS_M2)** is placed by M1 (M2) on to the crossbar. When asserted it indicates that the PE is busy.

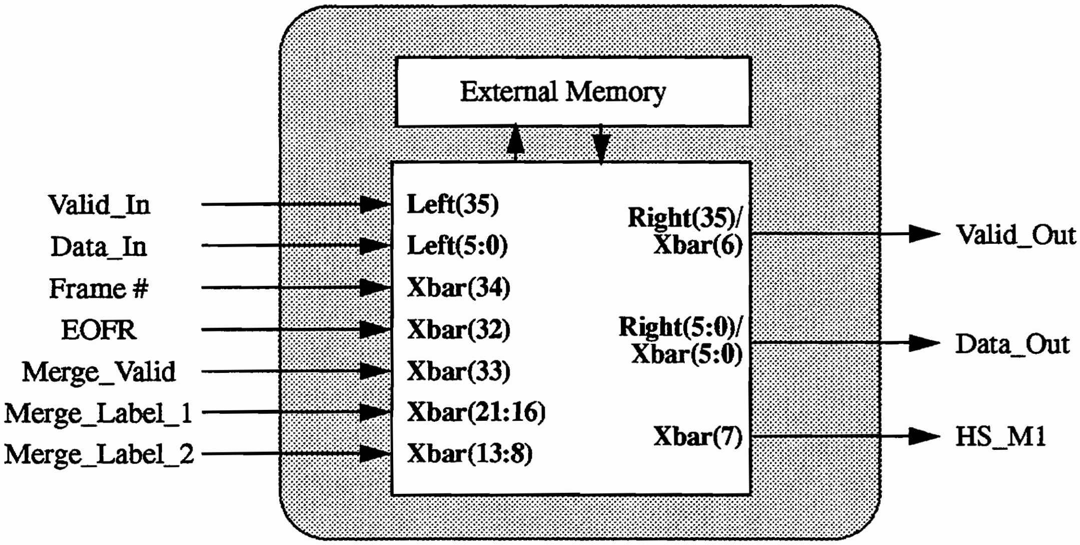


FIGURE 6.15. **Merger: Logical input/output block diagram.**

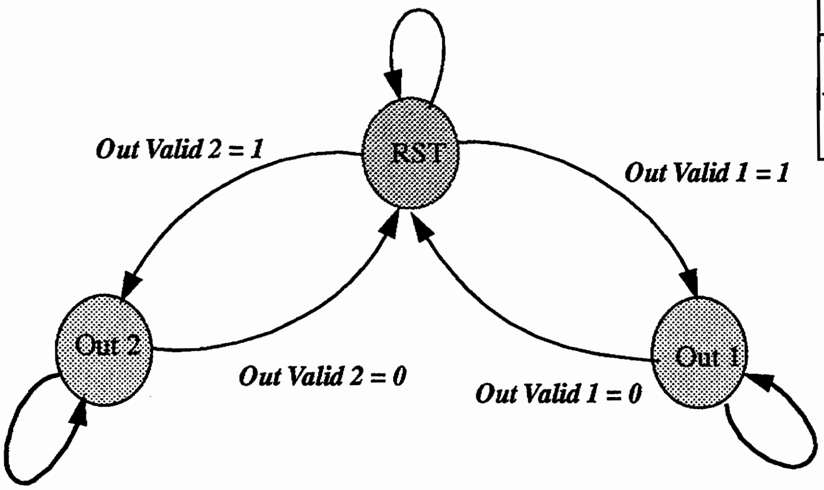
6.4.8 Output

6.4.8.1 Introduction

PE-5 and PE-7 place the second-pass processed labels on the crossbar and the on the right port, respectively. The primary function of PE-8 (or Output) is to accept the processed labels while they are valid and passes them to its right port for displaying on the monitor.

6.4.8.2 State Machine

The PE starts in the **RST** state and waits for PE-5 (**Merger-1**) or PE-7 (**Merger-2**) to produce valid second-pass processed labels. Recall that **Merger-1** and **Merger-2** output alternate frames (i.e. at any given time only one of them is actively sending the output). The PE takes labels from the crossbar and places them on its right port when **Merger-1** is sending valid labels and places the labels from its left port onto its right port when **Merger-2** is outputting. The PE jumps to and stays in the **Out1** state while **Merger-1** produces valid labels. The PE comes back to the **RST** state when **Merger-1** no longer produces valid labels. Similarly, it enters the **Out2** state when **Merger-2** is outputting valid labels and comes back to the **RST** state when **Merger-2** no longer produces valid labels.



State	*Description*
RST	*Restart/Idle State* *All Outputs Negated*
OUT1	*Odd Frame State* *Output From Xbar*
OUT2	*Even Frame State* *Output From Left*

FIGURE 6.16. **Output: State Machine.**

6.4.8.3 Logical input/output block diagram

Label_In_M1 and **Label_In_M2** are the labels coming in from **Merger-1** and **Merger-2** respectively. The control signals **Valid_In_M1** and **Valid_In_M2** inform the PE whether the incoming labels from **Merger-1** and **Merger-2** are valid. **Data_Out** is the label sent out to be displayed on the monitor and **Valid_Out** informs the monitor interface whether an incoming label is valid. **HS_M1** and **HS_M2** come to the PE before being broadcast on the crossbar. The signals are placed on the crossbar after buffering.

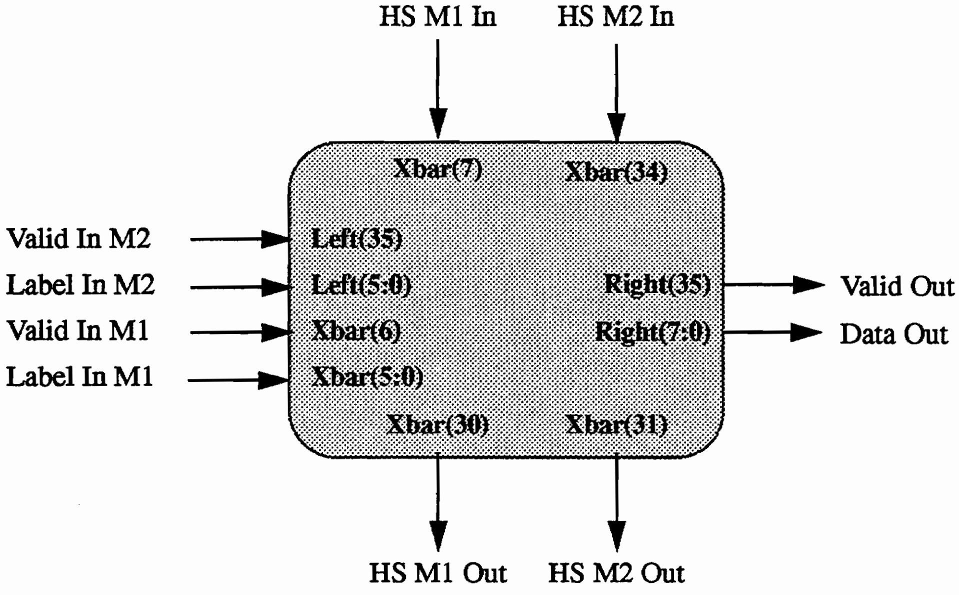


FIGURE 6.17. **Output: Logical input/output block diagram.**

6.5 Results

The implementation of the above design on the *VTSplash* real-time image processing platform has been a success. The design was executed at the required clock speed and was able to display the labeled regions on the monitor in real-time. The required clock speed for real-time operation is 10 MHz, and it corresponds to a 512 x 512 image at 30 frames per second. The design was also executed in the software mode, where a sample data file was input to the system and the output was captured into another file. Some examples are shown in Figure 6.20 and Figure 6.21.

The performance improvement for the algorithm, obtained from *Splash-2*, is significant over a Sun SPARCstation-2 or SPARCstation-10. A C-model of the algorithm was run on the SPARC stations on an input image in the memory and the output was written back to the memory. The time required for the memory transfers on the SUNs have been excluded from the following calculations in order for the results to be more comparable. Similar operation takes 0.99 seconds on a SPARCstation-2, 0.44 seconds on a SPARCstation-10 while it takes 0.03 seconds (in real-time), or 1.01 seconds in the software clock mode (the input image to *Splash-2* is via the SUN SBUS interface). The speed-up for the *Splash-2* implementation of the region labelling algorithm is 14.67 over a SPARCstation-10. This corresponds to 0.9×10^9 equivalent operations per second, including 70×10^6 memory operations per second. Equivalent operations could refer to simple operations such as comparison, multiplexing, incrementing and so on. A memory operation refers to a read or write to the local external memory of a processing element.

TABLE 1. Execution time required to process one 512 by 512 frame.

Platform	Time taken (sec)
SS-2	0.99
SS-10	0.44
Splash-2 (debug mode)	1.01
Splash-2 (Real-time mode)	0.03

The algorithm for the implementation is not innovative. It has been adapted from various algorithms to suit the design specifications of an adaptive computing platform. The speed-up obtained from the above design is due to the inherent hardware-like nature of the programmable logic and innovative implementation of various operations on the adaptive computing platform. Several tough trade-offs have been considered. In order to satisfy the design specification for area, the look-up table was transferred to the external memory. Redundant processing elements were incorporated into the design to operate in different frame slices such that the second pass processing continues without intervention from a first pass processing of a succeeding frame. The time required to update the external look-up table increases with the size of the look-up table, which is directly dependent on the maximum number of regions handled by the algorithm. Because of the aforementioned reason and due to the unavailability of sufficient crossbar communication resources the maximum number regions has been limited to 32.

A real merger updates the equivalence table. If the two labels from a merger instruction already belong to the same equivalence class or is repeated, it is not a real merger and it takes eight clock cycles for processing. A worst case analysis of the time taken to complete the processing was calculated. The example images shown below illustrate the process. This case exemplifies the process when the total number of real mergers are the maximum. For the image shown in Figure 6.18, the total number of mergers are 65,536 and hence the number of clock cycles taken for completion is 64 x 10^5.

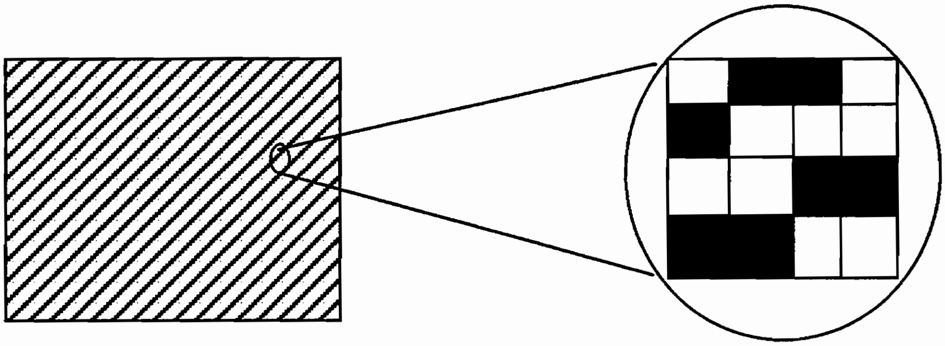


FIGURE 6.18. **Worst case analysis: Case 1.**

The second case is illustrated in Figure 6.19. In this case there are N real mergers, (N is the number of rows) and $\left(4\sum_{n=1}^{\frac{N}{2}} n - N\right)$ other mergers. Hence, this example considers the case when the total number of false mergers are the maximum. Recall that a real merger updates the look-up table and takes ($2K$+36) clock cycles to update the look-up table, which is typically 100 clock cycles for an equivalence table of size thirty two. The total processing time for such an image is given in the equation below. δ is the time taken to update the equivalence table

$$TimeTaken\,(clock-cycles) \;=\; \sum_{n=1}^{\frac{N}{2}} 4\,(n-N) \times 8 + N \times \delta$$

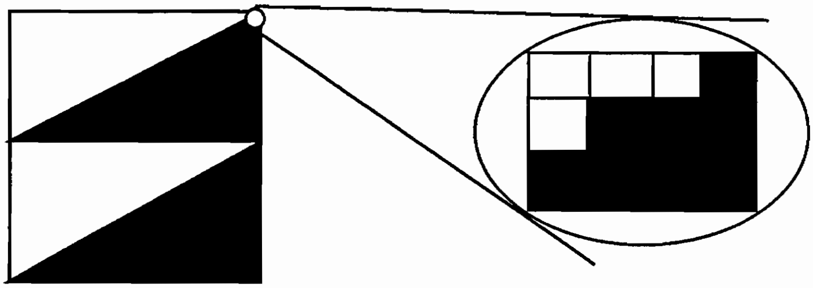


FIGURE 6.19. Worst case analysis: Case 2.

Some problems were encountered due to real-time processing of images. The image displayed on the monitor after detecting regions is observed to flicker when the input was taken from the camera. This was non-existent for a constant image from the SUN. The problem was finally attributed to the thresholder and the inability of the camera to provide exactly the same image from frame-to-frame. A hysteresis was implemented in 1-D to eliminate the flicker. A range of hysteresis values including (90, 100), (75, 125) and (50, 200) were implemented. But the results were did not show any improvement. As the variation of threshold values in the non-temporal hysteresis did not prove to be effective in reducing the noise introduced by the thresholder, it was concluded that the approach was incorrect. A temporal approach to the problem may be more effective. A 1-D filter was implemented to eliminate the salt and pepper noise generated by the thresholder. Several other problems such as a wrap-around of one pixel were observed but all of them were eventually fixed.

The improvements to the above algorithm may include incorporating temporal hysteresis into the thresholder to eliminate the flicker, implementing a small cache to nullify recurring instructions, investigating a feasibility of a single-board parallel design and quantifying a trade-off of number of regions with speed.

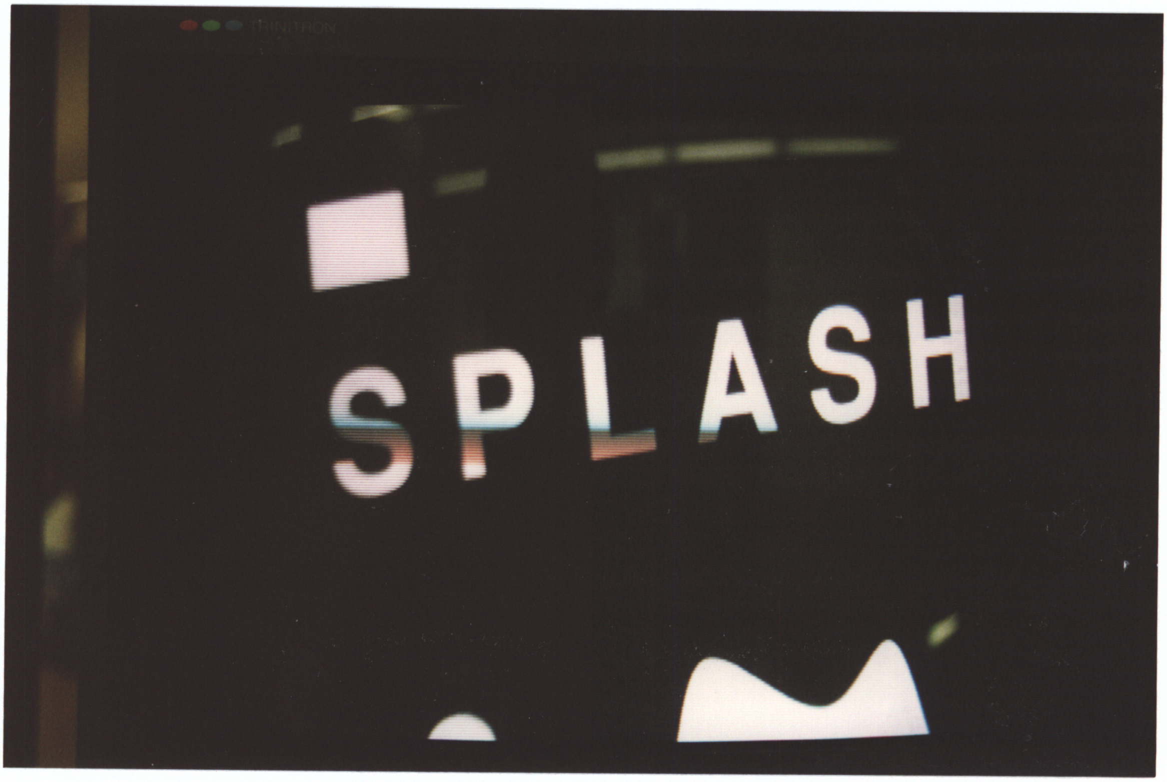


FIGURE 6.20. **Example input image from the camera**

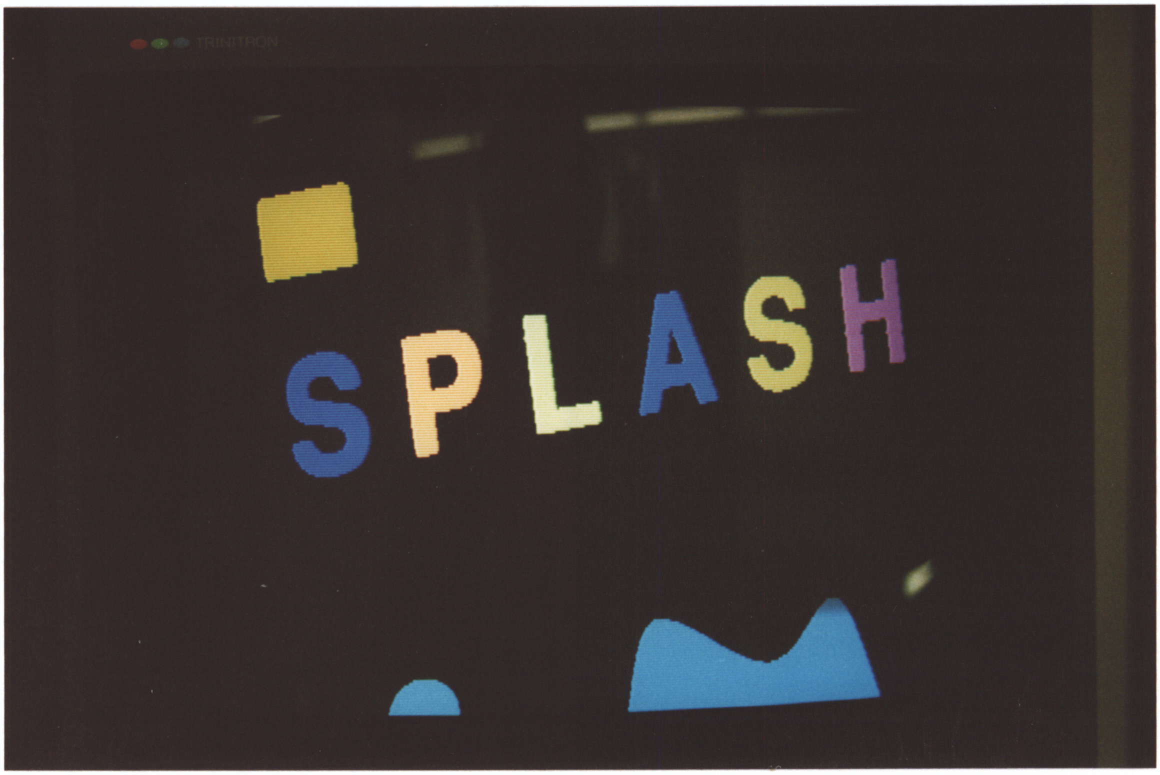


FIGURE 6.21. Generated output for the image in Figure 6.20

CHAPTER 7 Conclusion

This thesis has described the design of high-speed image-processing application that has been successfully implemented on a reconfigurable custom computing platform called *Splash-2*. The application accepts data in real-time, at the rate of 30 frames per second, and processes the image twice before displaying it on the monitor.

The design has been found to run significantly faster on the *Splash-2* system as compared to on the Sun SPARCstation-2 and SPARCstation-10 workstations. Considering the alternatives for the real-time implementation of the application, the *VTsplash* real-time platform is an efficient and cost effective solution when compared to other general purpose platforms or hardware implementations. General purpose sequential architectures such as the Sun

SPARCstation-2 and SPARCstation-10 can not handle real-time data. Application specific hardware implementations are the fastest for the task, but they lack the general purpose nature of a workstation. The massively parallel general purpose machines, such as the Cray, are capable of handling real-time image processing tasks, but they are too expensive to be cost effective. The *Splash-2* system is a very good trade-off as it cheaper than a SPARCstation and has the ability to handle real-time applications. Application specific hardware to solve this problem may be cheaper, but to run a variety of tasks the accrued cost of building many such hardware is large.

Though the performance levels of *Splash-2* excel those of other general purpose machines, it has some limitations. Many of these constraints are those which are inherent to all such systems. Most of the resources on the *Splash-2* system are limited. The communication resources such as the crossbar, left and right ports, and memory resources such as the on-chip RAM, and the off-chip local memory are limited. The total amount of hardware a PE can handle and even, the number of PEs are limited. Though, in theory, the number of array boards can be infinite, they are constrained in reality by hardware limitations. Though the architecture of *Splash-2* is programmable, multi-board solutions to massively parallel algorithms requiring large communication resources are limited by the 32-bit bus used for data transfer between two array boards. The system is limited by the number of pins on the XC4010 FPGA for communication and the capacity of the XC4010 FPGA for the size of the design. If the number of PEs on a single board is increased, it would be very difficult to implement a full-crossbar for communication. Considering the above limitations, *Splash-2* has been fine tuned for best throughput.

Real-time image processing applications need a high bandwidth of storage and communication resources. Though *Splash-2* has sufficient resources, due to the inherent nature of such applications, there is always enough room for trade-off with speed. Hence, faster than real-time applications on *Splash-2* are limited by the available resources on the system.

The research presented in this thesis has illustrated the effectiveness of reconfigurable hardware based custom computing platforms 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. Designs can be changed easily. Multiple designs can be pipelined to run at the same time. These features demonstrate the potential of adaptive computing platforms for computationally intensive applications.

Bibliography

[1] **A. F. Lochovsky**, “Algorithms for Real-time Component Labelling of Images,” *Image and Vision Computing,* vol. 6, February 1988, pp. 21-27.

[2] **N. Ranganathan, R. Mehrotra, and S. Subramanian**, “A High Speed Systolic Architecture for Labeling Connected Components in an Image,” *IEEE Transactions on Parallel Processing*, 1991, pp. 110-117.

[3] **A.Rosenfeld and P. Pfaltz**, “Sequential Operations in Digital Picture Processing,” *J. Assoc. Compute. Mach.*, vol. 12, 1966, pp. 471-494.

[4] **R.M. Haralick**, “Some neighborhood operations”, in *Real Time/Parallel Computing Image Analysis* (M. Onoe, K. Preston, and A. Rosenfeld, Eds.), Plenum Press, New York, 1981.

[5] **R. Lumia, L. Shapiro, and O. Zuniga**, “A new connected components algorithm for virtual memory computers,” *Computer Vision, Graphics, and Image Processing*, 1983, pp. 287-300.

[6] **J. T. Schwartz, M. Sharir, and A. Siegel**, “An Efficient Algorithm for finding connected components in a binary image,” Tech. Report 156, Courant Institute, NYU, 1985.

[7] **C. Ronse, and P. A. Devijver**, “Connected Components in binary Images: The Detection Problem,” *Research Studies Press Ltd.*, John Wiley and Sons Inc., 1984.

[8] **A. Agarwal, L. Nekludova, and W. Lim,** "A Parallel O(log N) algorithm for finding Connected Components in Planar Images," P*roc. 1987 International Conference on Parallel Processing*, pp. 783-786.

[9] **V. K. P. Kumar, and M. M. Eshaghian,** "Parallel Geometric Algorithms for Digitized pictures on Mesh of Trees", *Proc. 1986 International Conference on Parallel Processing (ICPP)*, pp. 270-273.

[10] **D. Nassimi and S. Sahni,** "Finding Connected Components and Connected Ones on a Mesh connected Parallel Computer," *SIAM Journal of Computing*, vol. 9, No. 4, pp. 744-757, November 1980.

[11] **M. Manohar, and H. K. Ramapriyan,** "Connected Component Labeling of Binary Images on a Massively Parallel Processor," *CVGIP*, 45, 1989, pp. 133-149.

[12] **X. D. Yang,** "Design of Fast Connected Components Hardware," *Proc. Computer vision and Patt. Recognition Conf.*, 1988, pp. 937-944.

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

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

[15] **Maya Gokhale et. al.,** "Building and using a highly parallel programmable logic array", *Computer* V 24 n 1, Jan 1991, pp. 81-89.

[16] **Jeff Nevits, and Brad Fross,** "VTSplash hardware", Virginia Tech, Blacksburg, VA.

[17] **P. M. Athanas, and A. L. Abbott,** "VTSplash", Virginia Tech, Blacksburg, VA, 1994.

[18] **A. Tarmaster,** "Median and morphological filtering of images in real-time using an FPGA-based custom computing platform", *Graduate thesis*, Virginia Tech, VA, 1994.

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

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

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

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

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

[24] **Edward R. Dougherty and Charles R. Giardina,** *Image Processing - Continuous to Discrete*, Vol. 1, Prentice Hall, Inc., New Jersey, 1987.

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

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

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

[28] **Synopsys Inc.,** *VHDL Design Compiler Family Reference Manual*, Version 3.1a, March 1994.

[29] **Synopsys Inc.,** *VHDL System Simulator Core Programs Manua*l, Version 3.1a March 1994.

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 Declarations for Splash 2 Simulator v1.5 (bkf 6/8/93)
--
-- Copyright (C) 1992 United States Government as represented
-- by the Director, National Security Agency.
-- All rights reserved.
--
-- Do not distribute further without the concurrence of the authors
--

-----------------------------------------------------------------------
--  Splash 2 Simulator v1.5 Xilinx_Processing_Part Entity Declaration
-----------------------------------------------------------------------
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;                    -- Left Data Bus
  XP_Right      : inout DataPath;--  Right Data Bus
  XP_Xbar: inout DataPath;                    -- Crossbar Data Bus
  XP_Xbar_EN_L: outBit_Vector(4 downto 0); -- Crossbar Enable (low-true)
  XP_Clk: in                                  Bit;          -- Splash System Clock
  XP_Int: out                                 Bit;          -- Interrupt Signal
  XP_Mem_A      : inout MemAddr;-- Splash Memory Address Bus
  XP_Mem_D      : inout MemData;-- Splash Memory Data Bus
  XP_Mem_RD_L       : inout RBit3;        -- Splash Memory Read Signal (low-true)
  XP_Mem_WR_L       : inout RBit3;        -- Splash Memory Write Signal (low-true)
  XP_Mem_Disable: inBit;                                    -- Splash Memory Disable Signal
  XP_Broadcast: inBit;                                      -- Broadcast Signal
  XP_Reset      : inBit;                                    -- Reset Signal
  XP_HS0            : inout RBit3;-- Handshake Signal Zero
  XP_HS1        : in    Bit;                  -- Handshake Signal One
  XP_GOR_Result: inoutRBit3;                                -- Global OR Result Signal
  XP_GOR_Valid: inoutRBit3;                                 -- Global OR Valid Signal
  XP_LED: out   Bit                                         -- LED Signal
 );
end Xilinx_Processing_Part;
```

A.2. Xilinx_Control_Part Entity Declaration

```
------------------------------------------------------------------------
-- Splash 2 Simulator v1.5 Xilinx_Control_Part Entity Declaration
------------------------------------------------------------------------
entity Xilinx_Control_Part is
 Generic(
  BD_ID             : Integer := 0;-- Splash Board ID
  PE_ID         : Integer := 0                -- Processing Element ID
 );
 Port (
  X0_SIMD: inout DataPath;                    -- SIMD Data Bus
  X0_XB_Data        : inout DataPath;-- Crossbar Data Bus
  X0_Mem_A      : inout MemAddr;-- Splash Memory Address Bus
  X0_Mem_D      : inout MemData;-- Splash Memory Data Bus
  X0_Mem_RD_L       : inout RBit3;        -- Splash Memory Read Signal (low-true)
  X0_Mem_WR_L       : inout RBit3;        -- Splash Memory Write Signal (low-true)
  X0_Mem_Disable: inBit;                                    -- Splash Memory Disable Signal
  X0_GOR_Result_In   : inoutRBit3_Vector(1 to XILINX_PER_BOARD);-- Global OR Result Bus
  X0_GOR_Valid_In: inoutRBit3_Vector(1 to XILINX_PER_BOARD);-- Global OR Valid Bus
  X0_GOR_Result: outBit;                                    -- Global OR Result Signal
  X0_GOR_Valid: outBit;                                     -- Global OR Valid Signal
  X0_Clk: in                                  Bit;          -- Splash System Clock
  X0_XBar_Set       : outBit_Vector(0 to 2);-- Crossbar Set Signals
  X0_XBar_Send        : outBit;               -- Crossbar Send Signal
  X0_X16_Disable: out   Bit;                  -- X16 Disable Signal
  X0_Int: out                                 Bit;          -- Interrupt Signal
  X0_Broadcast_In: inBit;                                   -- Broadcast Input Signal
  X0_Broadcast_Out    : outBit;               -- Broadcast Output Signal
  X0_Reset      : inBit;                                    -- Reset Signal
```

```
   X0_HS0          : inout RBit3;-- Handshake Signal Zero
   X0_HS1      : in    Bit;                  -- Handshake Signal One
   X0_LED: out   Bit                                      -- LED Signal
  );
end Xilinx_Control_Part;
```

A.3. XL Entity Declaration

```
-----------------------------------------------------------------
-- Splash 2 Simulator v1.5 XL Entity Declaration
-----------------------------------------------------------------
entity XL is
  port(
    XL_FIFO_IN: inout DataPath;

    XL_FIFO_Sel: inout Rbit3;
    XL_FIFO_En_L: inout Rbit3;
    XL_FIFO3_OE: inout Rbit3;

    XL_FIFO1_RE: inout Rbit3;
    XL_FIFO2_RE: inout Rbit3;
    XL_FIFO3_RE: inout Rbit3;

    XL_FIFO1_EF: inout Rbit3;
    XL_FIFO2_EF: inout Rbit3;
    XL_FIFO3_EF: inout Rbit3;

    XL_Buf_CLK: in Bit;
    XL_EN_CLK: out Bit;
    XL_SYS_CLK: in Bit;

    XL_GOR_Result: in Bit;
    XL_GOR_Valid: in Bit;

    XL_TO_XR: inout DataPath;
    XL_SIMD                          : inout DataPath;
    XL_INT                           : inout RBit3);
end XL;
```

A.4. XR Entity Declaration

```
-----------------------------------------------------------------
-- Splash 2 Simulator v1.5 XR Entity Declaration
-----------------------------------------------------------------
entity XR is
   port(
     XR_FIFO_OUT: inout DataPath;
```

```
    XR_FIFO1_WE: inout RBit3;
    XR_FIFO2_WE: inout RBit3;

    XR_FIFO1_FF: inout RBit3 := 'Z';
    XR_FIFO2_FF: inout RBit3 := 'Z';

    XR_SYS_CLK: in Bit;
    XR_TO_XL: inout DataPath;

    XR_R                                 : inout DataPath;
     XR_RDir: out Bit := '0';
    XR_INT                               : inout Rbit3 := 'Z');
end XR;
```

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 in TABLE 2.

TABLE 2.

Enable	Crossbar Ports
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

VHDL configuration and source files required for the simulation and execution of a design on Splash-2 system are provided. Only files modified as a part of this thesis are included. Refer to [14] for the complete set of files.

C.1. Soft binding configuration file : Config.vhd

```
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",
             output_file1 => "xoutput",
             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 work.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 => 2);

   for Structure
   for SBOARDS (0)

--
-- 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 S2Board.Xilinx_Control_Part(Blank);
        end for;                                   -- all: Xilinx_Control_Part

-- X1 to X9 are loaded with the various designs
-- The memories are cleared from a file "xmemory1"

        for XPARTS (1)
          for all : Xilinx_Processing_part
            use entity work.Xilinx_Processing_Part(Thresh);
          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 work.Xilinx_Processing_Part(labeler);
  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 work.Xilinx_Processing_Part(fifo);
  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 work.Xilinx_Processing_Part(Icache);
  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 work.Xilinx_Processing_Part(Dcache1);
  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 work.Xilinx_Processing_Part(M1);
  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)

for XPARTS (7)
  for all : Xilinx_Processing_part
```

```
        use entity work.Xilinx_Processing_Part(Dcache2);
       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(7)

      for XPARTS (8)
       for all : Xilinx_Processing_part
        use entity work.Xilinx_Processing_Part(M2);
       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(8)
      for XPARTS (9)
       for all : Xilinx_Processing_part
        use entity work.Xilinx_Processing_Part(Output);
       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(9)

-- X10 to X16 gets the "Left_To_Right" architecture for the Xilinx chip,
-- and the "Blank" architecture for the memory model

      for XPARTS (10 to 16)
       for all : Xilinx_Processing_part
        use entity S2Board.Xilinx_Processing_Part(Left_to_Right);
       end for;
       for all : Memory_Part
        use entity S2Board.Memory_Part(Blank);
       end for;
      end for;                                         -- XPARTS(10 to 16)

     end for;                                          -- Structure (of Splash2_Board)
    end for;                                           -- BD: Splash2_Board
   end for;                                            -- SBOARDS (0)
   for SBOARDS (1)
--
-- 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 S2Board.Xilinx_Control_Part(Blank);
        end for;                                        -- all: Xilinx_Control_Part

-- X1 to X16 get the "Left_To_Right" architecture for the Xilinx chip,
-- and the "Blank" architecture for the memory model

        for XPARTS (1 to 16)
          for all : Xilinx_Processing_part
            use entity S2Board.Xilinx_Processing_Part(Left_to_Right);
          end for;
          for all : Memory_Part
            use entity S2Board.Memory_Part(Blank);
          end for;
        end for;                                        -- XPARTS(1 to 16)

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

C.2. Crossbar configuration file

```
configuration 0
1 4
2 2 2 2 3 3
3 2 2 2 3 3
4 4 9 4 4 4
5 4 9 5 5 4
6 4 6 4 4 6
7 4 9 7 7 4
8 4 8 4 4 8
9 9 9 9 9 6
16 4
configuration 7
1 4
2 2 2 2 3 3
```

```
3 2 2 2 3 3
4 4 9 4 4 4
5 4 9 5 5 4
6 4 6 4 4 6
7 4 9 7 7 4
8 4 8 4 4 8
9 9 9 9 9 6
16 4
```

C.3. Memory Initialization file

```
-- Clears all the locations in the memory
clear
-- End of File

-- Image To download into a Memory
address 0
12 15 87 222
18 28 67 193
...........
...........
13 12 32 213
-- End of File
```

C.4. Sample Input/Output file : .sbm format

```
-- Start of file
00000000 0
00000000 0
00000098 8
92093400 a
9a98b3c8 f
-- End of File
```

C.5. Example VHDL source file : threshold.vhd

```
-- This program has been changed to let alternate frames
-- It has been changed on May 25th to accomodate Frame request
-- The simple 1D filter has been added to this design
-- On D-Day
Library SPLASH2;
Use SPLASH2.TYPES.all;
Use SPLASH2.SPLASH2.all;
Use SPLASH2.COMPONENTS.all;
Use SPLASH2.ARITHMETIC.all;
Use SPLASH2.HMacros.all;
```

```
Architecture THRESH Of Xilinx_Processing_Part Is
 Signal Left1  : Bit_Vector(35 Downto 0);
 Signal Right1, Xbar_in, Xbar_Out : Bit_Vector(35 Downto 0);
 Signal CNT : Unsigned ( 8 downto 0);
 Signal Tmp : Unsigned(7 downto 0);
 type states is ( RST, CUT1, W1, BOF, SPIT);
 Signal State  : states;
 CONSTANT ZERO_9 : Unsigned(8 downto 0) := "000000000";
 SIGNAL drive_xbar : bit_vector(4 downto 0);
 SIGNAL BOFR,EOLN,EOFR,Valid,Valid_Out : bit;
 SIGNAL Data_In : bit_vector(7 downto 0);
 SIGNAL Data_Out, HS : bit;
 CONSTANT Threshold : Unsigned(7 downto 0) := "01100100";
 CONSTANT ONE_9 : Unsigned(8 downto 0) := "111111111";
 SIGNAL cut : Unsigned(12 downto 0);
 CONSTANT ZERO_13 : Unsigned(12 downto 0) := "0000000000000";
 SIGNAL Data_Out_Fil : bit;
 SIGNAL St1, St2 : bit;
Begin

 drive_xbar <= "00000";
 XP_Xbar_EN_L <= "00000";

 -- I will remap my signals here
 valid      <= Left1(35);
 Right1(35) <= Valid_out;
 Right1(34) <= BOFR;
 Right1(33) <= EOLN;
 Right1(32) <= EOFR;
 Data_In    <= Left1(7 Downto 0);
-- Right1(0)  <= Data_Out;
 Right1(0)  <= Data_Out_Fil;
 HS         <= Xbar_in(7);

 XP_LED     <= Valid_out;

 -- This is where the 1D filter code goes
 P0 : PROCESS
 BEGIN -- PROCESS P0
  WAIT Until XP_Clk'event and XP_Clk = '1';

  St1 <= Data_Out;
  St2 <= St1;

  IF (St1 = '1') THEN
   IF (Data_Out = '0' and St2 = '0') THEN
    Data_Out_Fil <= '0';
   ELSE
    Data_Out_Fil <= '1';
   END IF;
 ELSE
   Data_Out_Fil <= '0';
 END IF;
END PROCESS P0;

P1: Process
Begin

 Wait Until XP_Clk'event And XP_Clk = '1';

 Pad_Input (XP_Left, Left1);          --type Conversion From
 Pad_Output (XP_Right, Right1);        --off-chip To On-chip Signal
 Pad_xbar(XP_xbar, xbar_out, xbar_in, drive_xbar);

 case State is
  when RST =>
   IF Valid = '1' THEN
    IF (HS = '0') THEN
     State <= W1;
    ELSE
     State <= CUT1;
     Tmp <= Data_In;
    END IF;
   ELSE
    State <= RST;
   END IF;

   Valid_out <= '0';
   BOFR     <= '0';
   EOLN     <= '0';
   EOFR     <= '0';
   CNT     <= ZERO_9;
   CUT     <= ZERO_13;

  WHEN  CUT1 =>
   IF (Cut(12) = '1') THEN
    Cut <= ZERO_13;
    State <= BOF;
   ELSE
    Cut <= Cut + 1;
    State <= CUT1;
   END IF;

   Tmp <= Data_In;

  WHEN  W1 =>
   IF (Valid = '0') THEN
    State <= RST;
   END IF;

  WHEN  BOF =>
```

```
    -- caluculating the control signals
    Valid_Out <= '1';
    BOFR <= '1';
    -- Latching the input data
    Tmp <= Data_In;
    -- next state caluculations
    State <= SPIT;
    -- Incrementing counter
    CNT <= CNT + 1;

    -- Thresholding operation
    if (Tmp < Threshold)  then
     Data_Out <= '0';
    else
     Data_Out <= '1';
    end if;

   when SPIT =>
    -- next state caluculations
    if Valid = '0' then
     State <= RST;
     EOFR <= '1';
    ELSE
      State <= SPIT;
    end if;

    -- caluculating the control signals
    Valid_Out <= '1';
    BOFR <= '0';

    -- Thresholding operation
    if (Tmp < Threshold) then
     Data_Out <= '0';
    else
     Data_Out <= '1';
    end if;

    Tmp <= Data_In;
    CNT <= CNT + 1;

    -- Caluculating the EOLN signal
    if CNT = ONE_9 then
     EOLN <= '1';
     CNT <= ZERO_9;
    else
     EOLN <= '0';
    end if;

   end case;
  End Process;
End THRESH;
```