Implementation of Coarse-to-Fine Visual Tracking on a Custom Computing Machine

by

Bharadwaj Pudipeddi

Thesis submitted to the faculty of the

Virginia Polytechnic Institute and State University

in partial fulfillment of the requirements for the degree of

Master of Science

in

Electrical Engineering

APPROVED

Dr. A. L. Abbott, chairman

Dr. P. M. Athanas

Dr. F. G. Gray

May 1996
Blacksburg, Virginia

Keywords: Tracking, Gaussian Pyramid, Custom Computing Machines, FPGAs, Computer Architecture

Implementation of Coarse-to-Fine Visual Tracking on a Custom Computing Machine

by

Bharadwaj Pudipeddi

Committee Chairman: Dr. A. Lynn Abbott

Electrical Engineering

(ABSTRACT)

"Smart" surveillance systems require a visual tracking system that is able to detect and follow a moving target in the field of view of a camera. Visual tracking systems have been traditionally developed either as application specific hardware or as software written for parallel architectures because of the large number of computations that have to be performed at very high speeds. This thesis describes the implementations of two visual tracking systems on a custom computing machine based on Field Programmable Gate Arrays (FPGAs). The implementations apply a coarse-to-fine search on Gaussian pyramids constructed from the images generated by a camera. One system tracks a target of size 16×16 in an image sequence with output images of size 256×256. This system is capable of operating at 30 pyramids per second. The second system tracks a target of size 16×16 in an image sequence with output images of size 512×512. This system is capable of operating at 15 pyramids per second. Both systems are designed with pipelined architectures and numerical computations are handled using a SIMD approach.

Acknowledgments

I would like to thank Dr. Abbott for helping me develop the idea for this thesis, and also for his encouragement and support in every stage of the work. I would like to express my gratitude to Dr. Athanas for helping me with many problems related to the Splash system and being a member of my committee. I would like to thank Dr. Gray for serving on my committee.

Special thanks go to Jim Peterson for helping me solve some irritating problems I encountered in my research. My gratitude and best wishes are extended to all the other members of VISC who made my stay at Virginia Tech an enjoyable experience. I would like to thank the system administrators Farooq and Chang for patiently helping me with software problems. We also had many an interesting conversation about everything under the Sun (SPARC).

My roommates and friends made me feel at home with a medley of stubborn arguments, philosophic discussions, late-night coffees and good-natured bantering. I would like to thank my brother, sisters and their families for moral and financial support. My parents have always played a special role in my life and I would not be here without their love and encouragement.

Table of Contents

List of Figures

List of Tables

CHAPTER 1

INTRODUCTION

1.1. Motivation

Visual tracking systems detect and follow a moving target in a sequence of images produced by a camera. Many applications of visual tracking exist, including autonomous vehicle navigation, missile tracking in military applications and autonomous surveillance systems. A problem faced in the design of a tracking system is the large number of computations that are needed. Common methods involve matching a part of one image with another image repeatedly. To achieve real-time operation, image matching needs to be performed at the rate that images are produced by the video camera, typically 30 images/second. To reduce the number of computations and to avoid false matches, some methods employ image pyramids as an efficient way to represent images [7]. But even these methods demand high computation rates that are not possible using conventional general purpose computers. The traditional solution for these problems is to use parallel computers [11] or application specific integrated circuits (ASICs) [9]. Applications that are implemented on parallel machines enjoy flexibility and low design costs but the high prices of these computers are not well suited for the users. ASICs, on the other hand, provide a relatively low-cost alternative when produced in large quantities, but specialized hardware solutions are rigid and they become obsolete in time when better solutions for the same application are discovered.

Recently, custom computing machines (CCMs) have emerged as a viable alternative to traditional methods of general-purpose computing. CCMs are designed with reconfigurable hardware that can be customized at a low level according to the needs of an application. The same hardware resources can be used for different applications, providing a general-purpose nature to these systems. Users who need hardware solutions to a wide range of problems can easily afford a CCM as compared with an ASIC for each

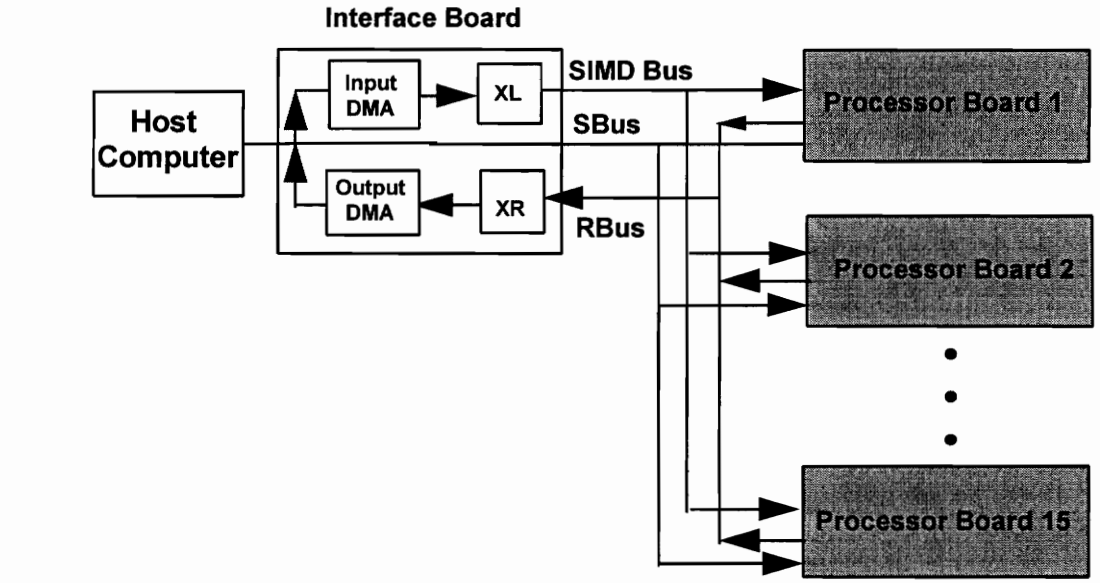
application. The major advantages of CCMs are low design costs, short design cycles (applications can often be developed within weeks), flexibility (applications can be redesigned and implemented on the same system) and a very high performance/price ratio over a wide range of applications.

This thesis describes the use of a CCM to implement a real-time visual tracking system. The system computes Gaussian image pyramids [7] at real-time rates and performs a coarse-to-fine search to track the movement of an object in the field of view. The CCM used for this purpose is Splash II [3], a system developed by the Center for Computing Sciences, formerly the Supercomputing Research Center in Bowie, Maryland.

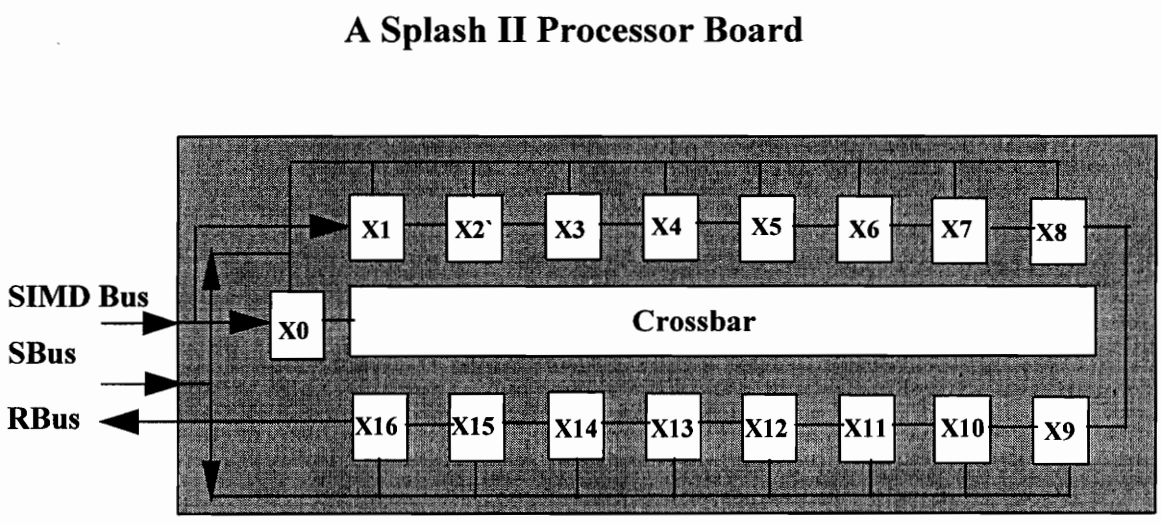
1.2. Splash II, A Custom Computing Machine

Splash II [3] is a custom computing machine based on field programmable gate arrays (FPGAs) that operates as an attached processor for a host workstation. Splash II is a general purpose system that can be tailored to deal with problems that would otherwise require custom hardware solutions. It is a multi-board system, using an array of Xilinx 4010 FPGAs [18] as the reconfigurable hardware devices.

Figure 1.1 shows the architecture of the Splash II system. It consists of up to 15 processor boards, each containing 17 FPGAs connected in a linear array. These FPGAs are denoted by X0 through X17 as shown in the figure. Each FPGA has a 256K by 16-bit (static RAM) local memory. A 16×16 crossbar allows 16 FPGAs at any time to communicate with each other directly. Two of the 17 FPGAs of a board share access to the crossbar of that board. One of them is X0, otherwise called the control processing element, and the other is X16, the last FPGA on the board. Each of the FPGAs, X2 to X15, have a 36-bit data path bus connected to the FPGAs on its left and right. An interface board allows a Sun SPARC 2 workstation to control the loading of the configurations into the FPGAs and to communicate with the processor boards.



(a)



(b)

Figure 1.1. The Splash II system. (a) System architecture, showing the fifteen Splash processor boards, the interface board and the host computer. (b) The structure of a Splash processor board showing the seventeen FPGAs connected as a linear array.

The Splash II system is attached to the host computer through the Sun SBus. Through this bus, the Sun workstation can directly access the memories of the each of FPGAs. The Splash II system also has a 36-bit input data path bus, called the SIMD bus, which is used for driving the input data stream and a 36-bit output data path bus, called the RBus, which is used for driving the output data stream. The interface board uses three bi-directional DMA channels to transfer data to and from the host computer through 256-word FIFOs. Two FPGAs, XL and XR, are used to handle incoming and outgoing data streams. XL is responsible for reading the data the input FIFOs and driving the SIMD bus with the input data stream. XR is placed between the output FIFOs and the RBus and it is responsible for reading data from the output FIFOs and driving the RBus with the output data stream.

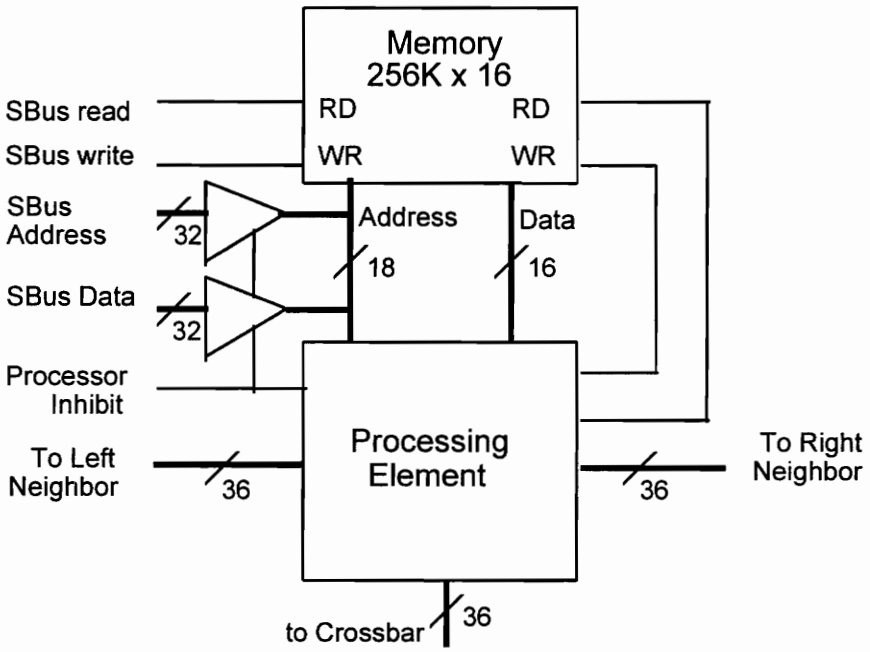


Figure 1.2. A processing element of the Splash II system. The SBus is an extension of the 64-bit Sun SPARC Sbus through a cable to the Splash II system. The processing element is a Xilinx 4010 FPGA.

Each of the FPGAs X2 to X15 has a 36-bit bi-directional data path bus connected on its left and right as shown in Figure 1.2. The other three FPGAs have a similar structure except that X0 and X1 have the SIMD bus as the input bus and X16 drives its output onto the RBus. Each FPGA has an 18-bit memory address path and a 16-bit bi-directional memory data path. The Sun workstation can access the local memories of the processing elements directly through the 64-bit SBus. The memories are not dual-ported so interlocks prevent simultaneous access by the host computer and the FPGAs.

1.3. The VTSplash System

The VTSplash system [4] uses the Splash II system and integrates it with a camera, a monitor and a video-interface component for implementation of real-time video applications. The system is shown in Figure 1.3.

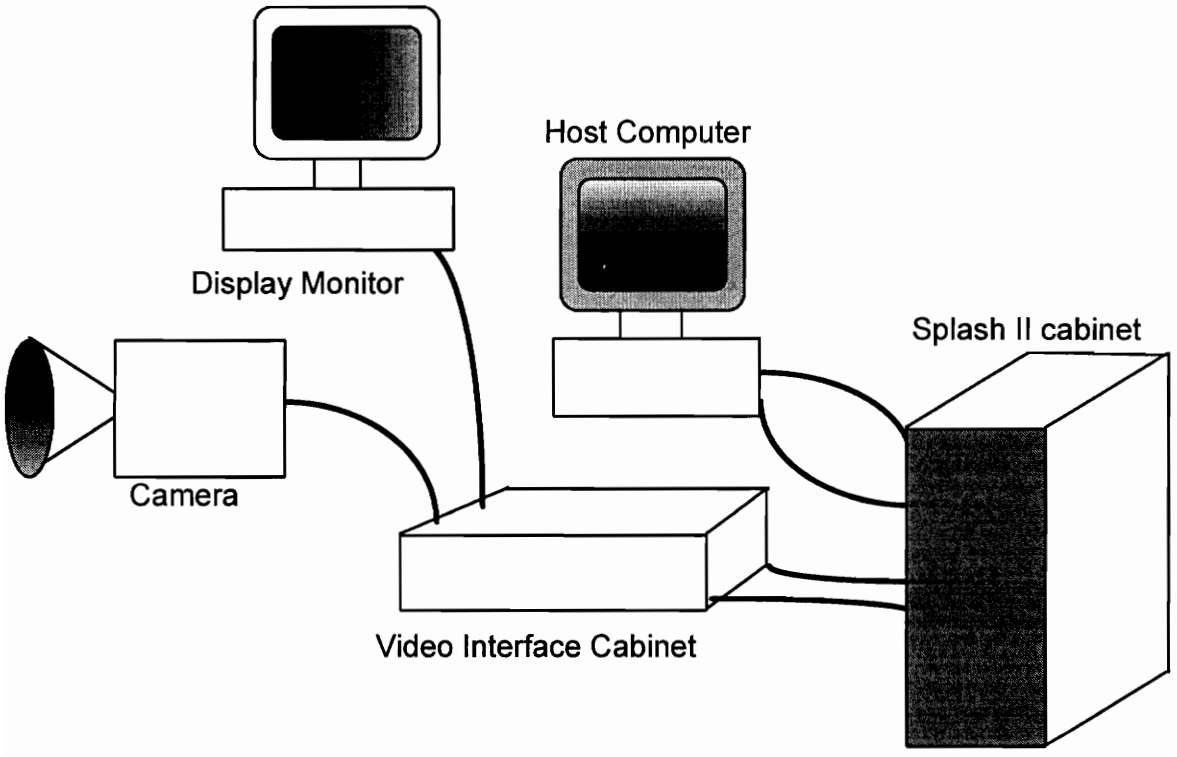


Figure 1.3. The VTSplash system with a camera, a video monitor, a video interface cabinet, a host computer and a Splash II system.

The video camera sends an RS-170 video signal to a frame-grabber card in the video-interface cabinet. The frame-grabber card uses an A/D chip to convert this to 8-bit gray scale pixel values with vertical and horizontal sync signals for control information. Images, typically of size 512×512, are transferred to the Splash II system in raster order at rates up to 30 frames per second. After appropriate configuration, Splash II processes this data and sends output images to a display card in the video-interface cabinet. The display card converts the output data stream to produce RS-170 signals for display on the video monitor.

1.4. Configuring a Xilinx 4010 FPGA

The 84-pin Xilinx 4010 FPGA [18] has 400 Configurable Logic Blocks (CLBs) and 160 Input/Output blocks (IOBs). Functional logic can be configured on the CLBs while the IOBs can be configured to provide an interface between the external world and the internal input/output signal lines. The CLBs and the IOBs can be interconnected with the help of the routing resources provided in the FPGA. The partitioning and the placement of logic among the CLBs and the routing between the blocks is automatically performed by a rich set of design tools provided by Xilinx Corporation.

The VHSIC hardware description language (VHDL) [2] can be used to model the design of the FPGA. VHDL is a powerful hardware description language that comes with a wide range of design constructs which can be used to describe complicated behavioral logic simply and accurately. VHDL supports both behavioral (finite state machine logic) and structural (component logic) modeling. The Synopsys VHDL tools [7] (VHDL analyzer, simulator, FPGA compiler) are used in this research to model the designs of the FPGAs. A VHDL model is written and synthesized using the Synopsys FPGA compiler. The synthesis produces a netlist file that can be used by the Xilinx tools, after a format conversion, for partitioning, placement and routing of the logic on the CLBs and the

IOBs of the FPGAs. The Xilinx tools provide a bitstream file that can be used to configure the FPGA from the host computer.

1.5. Contributions of this Research

This research achieves the following goals:

- Implementation of a real-time visual tracking system for 256×256 images at 30 frames/second on a custom computing machine with an input image size of 512×512 and a target window size of 16×16.
- Implementation of a real-time visual tracking system for 512×512 images at 15 frames/second on a custom computing machine with an input image size of 512×512 and a target window size of 16×16.
- Design of a SIMD architecture for the tracking systems that is synchronous and easily scaleable.
- Demonstration that a custom computing machine like Splash II is capable of supporting computationally intensive and speed-critical applications like object tracking.

1.6. Organization of this Thesis

This thesis has been organized with six chapters, a bibliography and appendices. A brief overview of the remaining five chapters is given here.

Chapter 2 introduces a tracking algorithm that will be used for the two tracking systems developed in this thesis. Results of a C implementation of the tracking algorithm will be shown in this chapter.

Chapter 3 describes the architecture of the 256×256 tracking system and its design on Splash II. A theoretical validation for the system performance is also given in this chapter.

Chapter 4 gives a brief overview of the changes that are made to the 256×256 tracking system to yield the 512×512 tracking system. A theoretical validation for the 512×512 system performance is also provided in this chapter.

Chapter 5 looks at the steps taken during the implementation of the system, specifically in modeling in Synopsys VHDL, simulation, synthesis, testing and configuration on Splash II.

Chapter 6 provides the results of the two tracking systems, configured on Splash II. Performance of the two systems is presented in this chapter.

Finally, Chapter 7 describes conclusions and possible future work based on this research.

CHAPTER 2

A TRACKING ALGORITHM USING COARSE-TO-FINE SEARCH

2.1 Introduction

A major problem in the design of a "smart" surveillance system is to determine an effective and efficient algorithm for automatic tracking of a moving object in a sequence of images. One approach employs a "coarse-to-fine search" [14] using image pyramids. An image pyramid is constructed from an image by generating a set of lower resolutions of the image. The lowest level of the pyramid corresponds to the highest resolution of the pyramid. Conversely, the highest level of the pyramid corresponds to the lowest resolution of the image.

In a coarse-to-fine search, motion is initially estimated in the highest level of the pyramid and this estimate is successively refined at each lower level of the pyramid. By restricting the search area in the lower levels of the pyramid (i.e., at higher resolution), a great deal of processing in the original image, as well as a tendency to fall victim to false matches, is avoided in this method. This method is akin to the human eye trying to locate a distant object in a scene. It first scans the scene cursorily, trying to detect a region that coarsely relates to the object. Then it uses higher resolution vision to examine that region more closely. This process is repeated until the object is fixated at the highest resolution available.

In the next section, one type of image pyramid, known as the Gaussian pyramid, is discussed briefly. Section 2.3 looks at a coarse-to-fine tracking algorithm which uses Gaussian pyramids. This chapter ends with a brief overview of some research conducted in the area of multiresolutional motion analysis.

2.2 The Gaussian pyramid

Burt and Adelson [7] introduced the Gaussian pyramid as an efficient data structure for multiresolution and multirate image coding. A Gaussian pyramid is constructed by computing a weighted sum of neighboring pixels in an image, and simultaneously down-sampling. By applying the same procedure to every new level, a complete pyramid can be constructed.

The weighted sum of neighboring pixels is calculated to obtain a single pixel at the next coarser pyramid level. Common weighting functions resemble the Gaussian function and they are half-band, low-pass filters applied over a 5×5 neighborhood of pixels by the use of a two-dimensional 5×5 convolution operator.

To simplify computations, the 5×5 convolution operator can be broken into two one-dimensional convolutions by operators w_x and w_y. The operator w_x is applied along the X-direction (columns) and the operator w_y is applied along the Y-direction (rows) of the image. The mathematical equations used are as follows:

$$P^{k,x}(i,j) = \sum_{m=-2}^{2} w_x(m)\, P^{k-1}(i,\ 2j-m) \tag{2.1a}$$

$$P^{k}(i,j) = \sum_{n=-2}^{2} w_y(n)\, P^{k,x}(2i-n,\ j) \tag{2.1b}$$

where the symbol P^k represents level k of the pyramid and $P^{k,x}$ represents an intermediate image obtained by a one-dimensional convolution of P^{k-1} in the X-direction.

Figure 2.1 illustrates convolution of the Gaussian pyramid generation in one-dimension. Because of down-sampling, every level of the Gaussian pyramid has half the number of rows and half the number of columns of the preceding lower level.

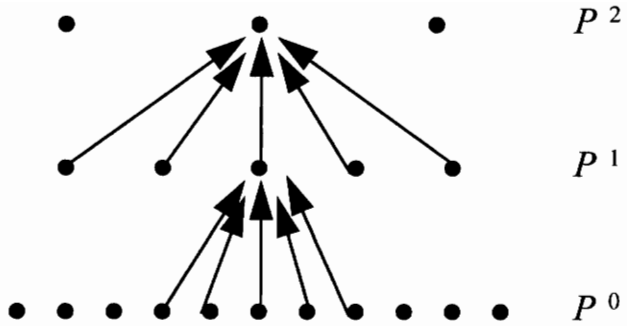


Figure 2.1. Representation of Gaussian pyramid generation in one-dimension. Five pixels of a pyramid level determine one pixel of the next pyramid level.

Figure 2.2 illustrates a Gaussian pyramid of four levels, 256×256, 128×128, 64×64 and 32×32. The pyramid is constructed from a standard image of a taxi turning around the corner of a street, using a C implementation of Eq. 2.1 on a Sun SPARC 2 workstation. The algorithm used for the C implementation uses two functions: one function convolves an image in the *X*-direction (using Eq. 2.1(a)) while the other function convolves an image in the *Y*-direction (using Eq. 2.1(b)). The two one-dimensional convolution masks used for the implementation are,

$$w_x = w_y^{\mathrm{T}} = \left[\frac{1}{16}, \frac{1}{4}, \frac{3}{8}, \frac{1}{4}, \frac{1}{16}\right] \tag{2.2}$$

A description of the algorithm is given in Figure 2.3.

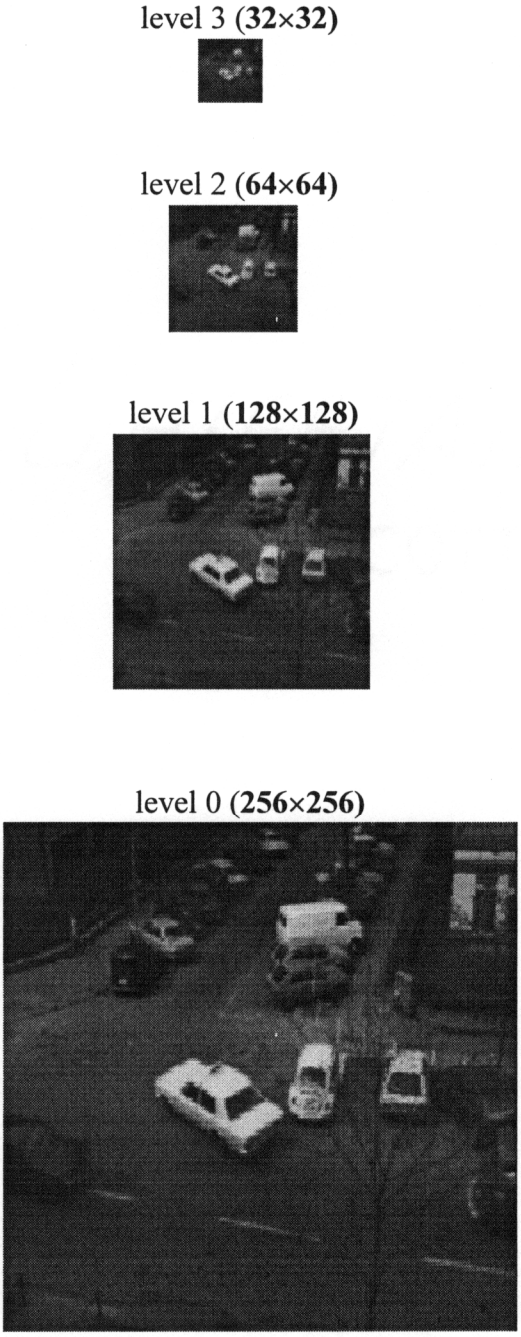


Figure 2.2. A Gaussian pyramid of four levels. The pyramid is constructed from the original 256×256 image which is the lowest level of the pyramid. This image is the first frame of an image sequence of a white taxi turning around the corner of a street [20].

Algorithm Generate_Gaussian_Pyramid

Input: A square image I of size $N \times N$, where N is a power of 2.

l is the desired number of pyramid levels.

1. Set P^0 equal to I /* this is the lowest level of the Gaussian pyramid */
2. For $k = 1$ to $l - 1$ /* from level 1 to level $l - 1$ of the Gaussian pyramid */

   /* apply horizontal operator w_x */
3. For $i = 0$ to $\frac{N}{2^{k-1}} - 1$ /* each row of level $(k - 1)$ level of the pyramid */
4. For $j = 1$ to $\frac{N}{2^k} - 2$ /* each interior column of level k of the pyramid */
5. Compute $P^{k,x}(i, j) = \sum_{m=-2}^{2} w_x(m)\, P^{k-1}(i, 2j - m)$
6. End for j.
7. $P^{k,x}(i, 0) = P^{k-1}(i, 0)$; $P^{k,x}(i, \frac{N}{2^k} - 1) = P^{k-1}(i, \frac{N}{2^{k-1}} - 1)$ /* edge control */
8. End for i.

   /* apply vertical operator w_y */
9. For $j = 0$ to $\frac{N}{2^k} - 1$ /* each row of level k of the pyramid */
10. For $i = 1$ to $\frac{N}{2^k} - 2$ /* each interior column of level k of the pyramid */
11. Compute $P^k(i, j) = \sum_{n=-2}^{2} w_y(n)\, P^{k,x}(2i - n, j)$
12. End for i.
13. $P^k(0, j) = P^{k,x}(0, j)$; $P^k(\frac{N}{2^k} - 1, j) = P^{k,x}(\frac{N}{2^{k-1}} - 1, j)$ /* edge control */
14. End for j.
15. End for k.
15. Return pyramid levels $P^0, P^1, \ldots, P^{l-1}$.
16. End algorithm.

Figure 2.3. An algorithm for generating a Gaussian pyramid of l levels from an input image of size $N \times N$.

Chen [10] implemented this algorithm with the same two masks (Eq. 2.2) on Splash II. Her system takes as input a sequence of 512×512 images. For a given image, it generates pyramid levels 256×256, 128×128, 64×64 and 32×32. She has developed two configurations for the pyramid generator. In one configuration, four chips of a Splash II board are used to generate pyramids for every alternate frame coming from a camera. In the other configuration, nine chips are used to produce a complete pyramid for every frame coming from the camera. Her implementation has been used to generate the input for the two tracking systems developed in this thesis.

2.3 Description of a coarse-to-fine tracking algorithm

Common approaches for tracking in an image sequence involve extracting a window from one image (this window represents the target) to be used as a reference to match with the next image in the sequence. Matching is a computationally intensive task because of the large number of blocks of the size of the reference window in the new image. By applying a coarse-to-fine search on a pyramid constructed from the image, the number of computations required for tracking is reduced significantly.

In a coarse-to-fine search, a reference window is extracted from every pyramid level of one image such that the reference window of the lowest level of the pyramid is the object that is being tracked while the other reference windows contain the target as well as some of its neighborhood. As illustrated in Figure 2.4, this is achieved by choosing a constant size for the reference windows. The size of the reference window increases at higher pyramid levels relative to the image size. Search is initially conducted on the highest pyramid level. This search will cover a large portion of the image because of the small size of this pyramid level. The result of this search is used to estimate the region of search (henceforth called the search window) for the next pyramid level. The reference window of the next pyramid level is matched with every block of the search window of this level and the position of the best match is again used to find the search window for the next

lower pyramid level. This process is repeated for all levels of the pyramid until the best match for the reference window of the lowest pyramid level is found in that level. As seen from the figure, the size of the search window of a level decreases with respect to the resolution of that level as the levels increase.

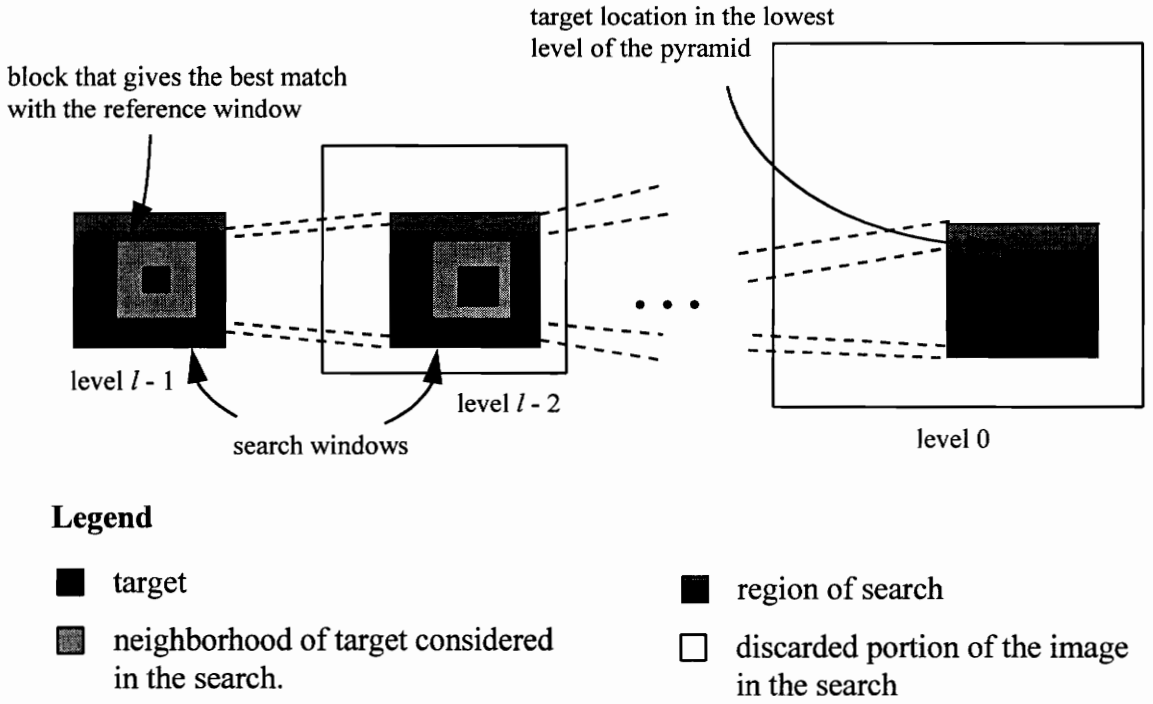


Figure 2.4. A coarse to fine search process. The pyramid levels decrease from left to right.

The following notation is used in the description of the algorithm. An image sequence is denoted by I. The nth image in this image sequence is denoted by I_n. A pyramid constructed from the image I_n is denoted by the symbol P_n. The kth level of the pyramid P_n is denoted by P_n^k. The reference window for the pyramid level P_n^k is denoted by R_n^k. The set of all reference windows for the pyramid P_n is denoted by R_n. The search window of the pyramid level P_n^k is denoted by S_n^k. In the implementation described here, the highest level search window is the entire highest level of the pyramid, and therefore S_n^{l-1} is the same as P_n^{l-1}. The upper left hand coordinates of search window in a level are

denoted by (x, y) where x represents the row coordinate and y represents the column coordinate.

A block is defined as a section in a search window of the size of a reference window. The upper left-hand coordinates of a block (or a reference window) in a search window are denoted by (r, c).

The description of the algorithm is given in Figure 2.5. For simplicity, the initial target has been chosen as the central 16×16 window of the first image of the image sequence I. Hence the initial set of reference windows will be the central 16×16 windows of each level of the Gaussian pyramid of the first image. Since the highest level search window is the entire highest level, its size will be 32×32. For the all the other levels, the search windows have twice as many rows and twice as many columns as a block of the previous level search window. Since the block size is 16×16, all the search windows of a pyramid have the same size of 32×32.

The results of the algorithm for one example, as implemented in the C programming language and executed a Sun SPARC 2 workstation, are shown in Figure 2.6. The chosen image sequence is a taxi turning around the corner of a street. The images are of size 256×256 and the Gaussian pyramid has four levels of resolution (l = 4), 256×256, 128×128, 64×64 and 32×32. Pyramids, constructed for each image in an image sequence of five images, are given as inputs to the C program. The last four images are generated as outputs with the target highlighted in each image by superimposing a white rectangle of side 16 on the original image. The pyramid generated from each image is used by the program to determine the set of reference windows for each successive image.

Algorithm Coarse_to_Fine_Search

Input: Gaussian pyramids, with l levels, constructed from each image of an image sequence I of p images.

1. Extract the initial set of reference windows, R_0, from the first pyramid, P_0, as the central 16×16 windows of each level of the pyramid.
2. For $n = 1$ to $p - 1$
   /* from the second pyramid to the last pyramid of the sequence */
3. Set x and y to zero.
   /* row and column of the upper left corner of the search window */
4. For $k = l\text{-}1$ to 0
   /* from the highest level to the lowest level of the current pyramid */
5. Extract the search window S_n^k from location (x, y) within P_n^k.
6. Perform correlation on each block of S_n^k with $R_{n\text{-}1}^k$ and determine the location (r, c) within $R_{n\text{-}1}^k$ that represents the best match at this level.
7. Extract the block from the location (r, c) within $R_{n\text{-}1}^k$ and store it as the reference window R_n^k for the next pyramid.
   /* the upper left corner coordinates of the block in $R_{n\text{-}1}^k$ are (r, c) */
8.. Set x to $2x + 2r$ and y to $2y + 2c$. These are the coordinates of the search window of the next level of the pyramid.
9. End for k.
10. The coordinates of the target in the lowest level of the pyramid, P_n^0, are $(x + r, y + c)$. Highlight the target in P_n^0 and display that as the output image.
11. End for n.
12. Return the lowest levels of the Gaussian pyramids for the last $(p - 1)$ images of the image sequence I, displayed as output images with the target highlighted.
13. End algorithm.

Figure 2.5. A coarse-to-fine tracking algorithm using Gaussian pyramids of l levels.

(a)

(b)

(c)

(d)

Figure 2.6. These four frames are the results of the coarse-to-fine search process applied to a sequence of five images (images 0 to 5) of a taxi turning a corner [19]. The first frame (not shown) has been used to generate the initial set of reference windows. White rectangles of size 16×16 in images (a) to (d) represent the best match in images 1 through 4, respectively, to the target found in the previous image.

The first step of the algorithm extracts the reference windows for the second pyramid from the first pyramid. No other processing is done on the first pyramid. Steps three to ten of the algorithm are applied over each pyramid starting from the second pyramid in the pyramid sequence, as specified by the second step (the variable n indicates the position of the current pyramid in the input pyramid stream). In the third step, two variables x and y are introduced which specify the position of the search window of the current level of the current pyramid. The current level is the highest level of the pyramid at this step and hence x and y are set to zero. The fifth step to the eighth step of the algorithm are executed from the highest level to the lowest level of the current pyramid. The variable k, introduced in the fourth step, indicates the current level of the pyramid being processed.

The fifth step requires a comparison on each block of the search window with the reference window. For a 32×32 search window and a 16×16 reference window, there will be 17 blocks in each row and 17 blocks in each column of the search window i.e., a total of 289 blocks. The sum of absolute differences is used to measure the dissimilarity between a search block and the reference window. This is loosely referred to as correlation and results in 289 sums for all the blocks of the search window. The block that gives the minimum of these sums or errors is the new reference window and its upper left-hand coordinates will determine the next search window.

The equation for this is as follows:

$$E_n^k(x,y) = \sum_{r=0}^{15}\sum_{c=0}^{15} \left| S_n^k(x+r, y+c) - R_{n-1}^k(r,c) \right| \tag{2.2a}$$

The best match is chosen by selecting (x, y) to minimize $E_n^k(x, y)$ as follows:

$$E_{n(\min)}^k = \min_{\substack{0 \le x \le 16 \\ 0 \le y \le 16}} E_n^k(x,y) \tag{2.2b}$$

$E_n^k(x, y)$ denotes the error obtained by correlation of a block with upper left-hand coordinates (x, y) in the search window S_n^k and $E_{n\,(\min)}^k$ denotes the minimum of the errors obtained for each block in the search window. The minimum error position is the position of the block whose minimum error is $E_{n\,(\min)}^k$. This block is defined as the reference window of the current level, k, for the next pyramid. In this way, both translation as well as rotation of the target are taken into consideration.

Since every level has twice as many rows and twice as many columns as its preceding higher level, the seventh step of the algorithm defines the upper left-hand coordinates of the new search window by doubling each of the upper left-hand coordinates of the absolute minimum error position in the preceding higher level. The absolute minimum error position is the sum of the search window position of that level and the minimum error position in the search window obtained through correlation.

Finally the ninth step of the algorithm highlights the target by drawing a white rectangle around the target in the lowest level of the pyramid. The target position is the absolute minimum error position in this level.

The execution time for the algorithm as measured on a Sun SPARC 2 workstation for generating the results shown in Figure 2.3 is 13.59 seconds of CPU time. The CPU time required to process one pyramid of the input sequence is 3.25 sec. These numbers do not include the time required to construct the pyramids. Since input pyramids arrive at the rate of 30 pyramids/sec, the workstation-based implementation is not capable of real-time operation. The next two chapters explain how the algorithm is partitioned and implemented on Splash II for the two real-time tracking systems developed in this thesis.

2.4 Previous work on multiresolutional object tracking

A good deal of research has been conducted on motion detection and object tracking. Song et al. [14] developed a motion vision system in which motion is detected and estimated using a multiresolutional search process, not unlike the approach used in this thesis. They obtain a primary mask using temporal gradient (which is the gradient difference between successive images to distinguish stationary objects and a moving object) and dynamic thresholding for detection of motion. Then they use a region-growing algorithm which improves the search area after each iteration. Finally they use a hierarchical search to identify the position of the moving target.

Burt and van der Waal [8] designed a segmented pipeline architecture for multiresolutional focal processing. They present an example in which motion analysis is performed on Gaussian pyramids in real-time. The pipeline they use is not a standard "*lattice*" pipeline in which the data flow rate is uniform. The segmented pipeline is formed by breaking a standard pipeline into segments and introducing buffers between those segments so that for a certain period of time, data flow rate in different segments of the pipeline can have different data rates. This structure allows faster processing of down-sampling, up-sampling and windowing operations. Motion analysis is partly performed on two image frames A and B by the segmented pipeline using a prior motion estimate vector V. Initially the earlier image frame A is warped towards the image frame B by using the vector V so that some of the displacement computation is reduced. Then the second levels of the Gaussian pyramids of both the frames are taken and cross-correlation is performed on them. An external microprocessor accesses these values and estimates the motion.

Cremonesi et al. [11] provide a case study on motion detection and tracking using pyramidal algorithms on MIMD architectures. They describe a coarse-to-fine search for tracking and suggest possible approaches for parallel implementation of a pyramid

generator and a motion detector and tracking system. They make interesting conclusions on the best strategy for parallelization. They also discuss the implementation of one parallel approach for motion detection and tracking on a 32-node Meiko Computing surface, a Transputer-based parallel machine, using C code for a 512×512 image.

But all these tracking systems have been developed either as special purpose hardware or as applications that run on parallel machines. This research demonstrates that the same systems can be implemented on a custom computing machine, thereby making the systems less expensive.

CHAPTER 3

DESIGN OF A TRACKING SYSTEM FOR 256×256 IMAGE ON SPLASH II

3.1. System Overview

This chapter describes the design of a tracking system using the tracking algorithm given in Figure 2.5 of Chapter 2 on Splash II for a 256×256 image with a target window size of 16×16 and a search window size of 32×32. The system accepts pyramids that are produced by the Gaussian pyramid generator developed by Chen [10] as input. A pyramid of four levels, 256×256 (level 0), 128×128 (level 1), 64×64 (level 2) and 32×32 (level 3) is generated for each 512×512 image coming from a camera. The system performs tracking and displays the 256×256 level with the target highlighted on a video monitor as the output image. For this tracking system, the 256×256 level is considered as the lowest level of the pyramid since the process of searching for the target ends at this level. The initial set of reference windows are the central 16×16 sections of the each level of the first pyramid.

The algorithm developed in Chapter 2 cannot be implemented on one Xilinx device due to the requirements of real-time performance. In order to achieve real-time performance, the hardware design is divided into four basic parts as illustrated in Figure 3.1.

Part 1 of the design receives each incoming pyramid and simultaneously forwards it to Part 2 and at the same time, forwards the lowest level of each pyramid (which is the 256×256 image) to Part 4. Part 2 stores the pyramids in its memory and furnishes the reference windows and the search windows to Part 3 which performs correlation. Part 4 stores each 256×256 image coming from the first part in its memory, computes the final

minimum error position from the correlation results, highlights the target in the 256×256 image, and finally sends that image as output.

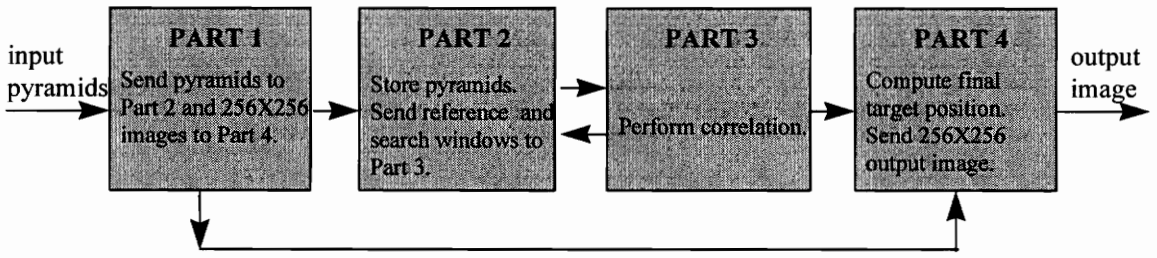


Figure 3.1. Block diagram of the tracking system. This four-part decomposition of the tracking algorithm, given in Figure 2.5 of Chapter 2, is capable of real-time operation for 256×256 image.

Part 1 is relatively simple and has been designed using one Xilinx chip. Since pyramids are generated continuously and they need to be stored in memory for processing, information will be lost if only one chip is used for Part 2. Hence Part 2 has been designed using two Xilinx chips. Using the convention that image frames are numbered beginning with zero since system initialization, the first chip stores all odd-numbered pyramids while the second one stores all the even-numbered pyramids. Part 4 has been designed using two Xilinx chips as it too depends on the completion of the operations of Part 2.

The correlation, performed in Part 3, is a time-consuming task and it requires two chips for real-time performance. This is explained in greater detail later on in this chapter. The first chip correlates all 16×16 blocks of the search window of each level that begin on even columns with the reference window of that level. The second chip correlates all 16×16 sections of the search window of each level that begin on odd columns with the reference window. The best of the two results is then evaluated in the second chip and broadcast to Part 2 and Part 4.

A total of seven chips are required for this system. This system has SIMD nature because the same correlation operation is cast on two chips over different sets of data.

3.2. Architecture of the tracking system on SPLASH II

The seven-chip tracking system architecture is shown in Figure 3.2. It is preceded either with the five-chip Gaussian pyramid generator which works at 15 frames/sec or the nine-chip Gaussian pyramid generator which works at 30 frames/sec [10]. This system is capable of processing 30 pyramids/sec. As there are 16 chips on each Splash II board, the entire system can be implemented on one processor board .

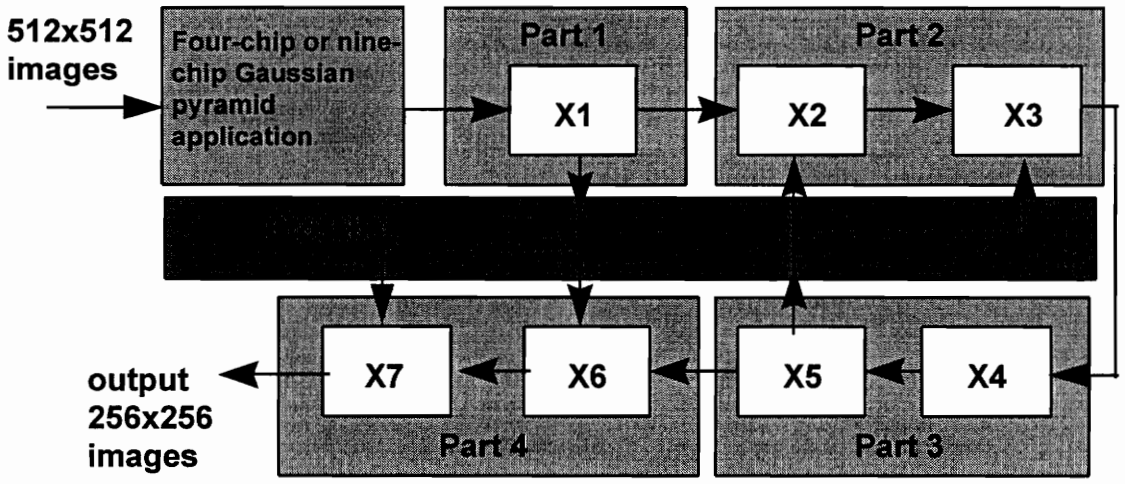


Figure 3.2. Seven-chip architecture of the object tracking system on Splash II. The input pyramid is produced by a Gaussian pyramid application on Splash II [5]. The output image is the 256×256 level of the input pyramid with a white rectangle superimposed over the target window.

Part 1 is implemented with an extra operation on chip X1. The initial set of reference windows extracted from the first pyramid (pyramid 0) is also taken care of by this chip. After that, it sends all odd-numbered pyramids to chip X2, all even-numbered pyramids to chip X3, all odd-numbered 256×256 frames to chip X6 and all even-numbered

256×256 frames to chip X7. It uses the 36-bit data bus to the crossbar to send data to chips X6 and X7 and uses the 36-bit data bus to its right neighbor to send data to the other four chips.

Chips X2 and X3 implement Part 2 of the system. Chip X2 stores all odd-numbered pyramids and chip X3 stores all even-numbered pyramids coming from chip X1. While one chip is storing a pyramid, the other chip is transmitting search windows or reference windows to chips X4 and X5 in response to requests from chip X5 which arrive through the crossbar.

Initially chip X5 requests the highest level of the Gaussian pyramid. Because of its small size (32×32), it represents a complete search window as discussed in Section 2.3. Therefore no other information is required for Part 2 at this level. For all other search windows, chip X5 sends the appropriate request and the minimum error position in the search window of the previous higher level. The request is in the format of a four bit tag, in which one bit indicates valid data, two bits indicate the level of the search window requested and the last bit identifies the recipient chip (chip X2 or X3). The recipient chip of Part 2 computes the absolute search window position within the image at the current pyramid level and then sends the data to Part 3. This chip automatically sends the new set of reference windows after receiving the minimum error position of the lowest level from chip X5.

Part 3 is implemented on chips X4 and X5. The equations for correlation, Eq. 2.1a and Eq. 2.1b, can be partitioned in the following way. The correlation of all 16×16 blocks of the search window that begin on even columns can be represented by the equations (using the notation introduced in Chapter 2),

$$E_n^k(x,y) = \sum_{r=0}^{15}\sum_{c=0}^{15}\left|S_n^k(x+r,2y+c) - R_{n-1}^k(r,c)\right| \qquad (3.1a)$$

Select (x, y) to minimize $E_n^k(x, y)$ as follows:

$$E_{n\,(min\_even)}^{k} = \min_{\substack{0\le x\le 16\\ 0\le y\le 8}} E_n^k(x,y) \qquad (3.1b)$$

The correlation of all 16×16 blocks of the search window that begin on odd columns can be represented by the equations,

$$E_n^k(x,y) = \sum_{r=0}^{15}\sum_{c=0}^{15}\left|S_n^k(x+r,2y+1+c) - R_{n-1}^k(r,c)\right| \qquad (3.2a)$$

Select (x, y) to minimize $E_n^k(x, y)$ as follows:

$$E_{n\,(min\_odd)}^{k} = \min_{\substack{0\le x\le 16\\ 0\le y\le 7}} E_n^k(x,y) \qquad (3.2b)$$

Finally the best of the two errors is obtained in the following way,

$$E_{n\,(min)}^{k} = \min\left(E_{n\,(min\_even)}^{k}, E_{n\,(min\_odd)}^{k}\right) \qquad (3.3)$$

Eq. 3.1 is implemented on chip X4. The other two equations are implemented on chip X5. Upon receiving a search window, both chips process it simultaneously and find their respective minimum error positions, each using a different portion of the search window. Chip X5 finishes processing first as there are fewer odd-position search blocks (17×8) compared to the number of even-position search blocks (17×9). Chip X5 waits for chip X4 to send its minimum error and minimum error position which are transmitted by the latter as soon as it finishes processing. Then chip X5 determines the final minimum error position (using Eq. 3.3) and sends it through the crossbar for chips X2 and X3 and through the 36-bit data bus to chip X6. Chip X6 forwards this data to chip X7.

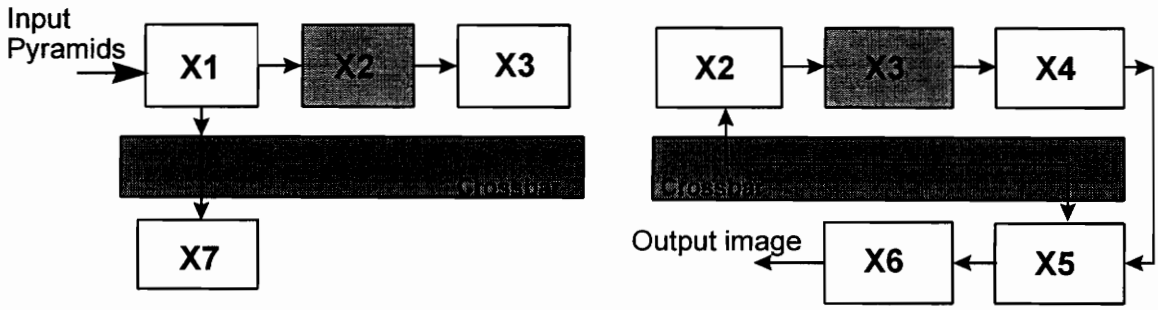
Chips X6 and X7 constitute Part 4 of the system. Chip X6 has previously stored the corresponding odd-numbered 256×256 image and chip X7 has previously stored the corresponding even-numbered 256×256 image. While one chip is storing an image coming from chip X1, the other chip calculates the absolute minimum error position in the 256×256 frame by accumulating the successive minimum error positions of each level of the pyramid received from chip X5. This final minimum error position is the upper left-hand coordinate of the detected target in the image stored in the memory of the chip.

The chip then draws a white rectangle around the target and sends the resulting image through the 36-bit bus to its right neighbor as the output.

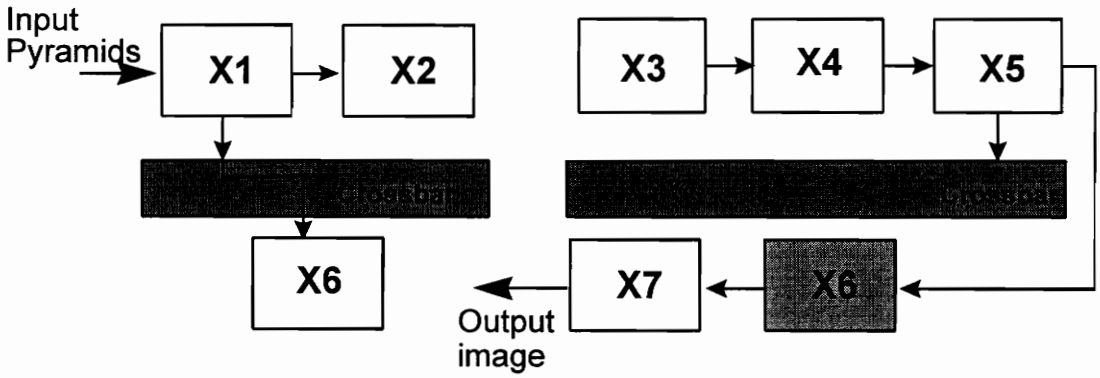
For all data transfers between the chips, a four-bit tag is used to send control information. For all tags, one bit is used to indicate whether the data is valid or not. In the case of chip X5, two bits of the tag are used to indicate the level of the search window requested from chip X2 or X3. The last bit of the tag is used to distinguish the recipient (chip X2 or X3). For all other chips, three bits of the tag are unused.

The design involves three distinct configurations. One is strictly for initialization and lasts only for the duration of the arrival of the first pyramid. In this configuration, chip X1 extracts the initial set of reference windows and sends them to chips X4 and X5. The other two are steady-state configurations as illustrated in Figure 3.3 on the next page. The first of these two, shown in Figure 3.3(a), is during the arrival of odd-numbered pyramids while the other, shown in Figure 3.3(b), is during the arrival of even-numbered pyramids.

To illustrate communication details, Table 3.1 gives the sequence of events for the processing of an odd-numbered pyramid stored in chip X2. The column "Time" indicates the order in which the events occur. The column "Event" describes the main events that are responsible for communication between the chips. While this processing is taking place, chip X3 is receiving the current incoming pyramid from Part 1 of the system.



(a)



(b)

Figure 3.3. The two steady-state configurations of the tracking system. The chips which are represented by shaded rectangles forward the data from their left neighbor to their right neighbor.
(a) Configuration for odd-numbered pyramids. Chip X3 stores the incoming odd-numbered pyramid and chip X7 stores the incoming odd-numbered 256×256 image. Concurrently, chips X4 and X5 process the previous pyramid that is stored in chip X2, sending the correlation results to chip X6 which sends the output image using the previous 256×256 image stored in its memory.
(b) Configuration for even-numbered pyramids. Chip X2 stores the incoming even-numbered pyramid and chip X6 stores the incoming even-numbered 256×256 image. Concurrently, chips X4 and X5 process the previous pyramid that is stored in chip X2, sending the correlation results to chip X7 which sends the output image using the previous 256×256 image stored in its memory.

Table 3.1.

Communication between the chips for the processing of an odd-numbered pyramid.

TIME	EVENT
1	Chip X5 sends request "1110" through the crossbar for highest level search window (level "11") to chip X2.
2	Chip X2 sends the 32×32 pyramid level from its memory through the 36-bit bus to chip X3 and chip X3 forwards this data to chips X4 and X5.
3	Chips X4 and X5 perform correlation of this search window with the highest level reference window. Chip X5 sends the minimum error position and the request "1100" (for the level "10" search window) through the crossbar to chip X2 and through the 36-bit bus to chip X6 (for using these positions to evaluate the final target position).
4	Chip X2 evaluates the absolute search window position and extracts the 32×32 search window from the 64×64 level and sends it to chips X4 and X5 through chip X3.
5	Chip X5 sends the level "10" minimum error position and the request "1010" (for the level "01" search window) to chip X2 and chip X6.
6	Chip X2 extracts the 32×32 search window from the 128×128 level and sends it to chips X4 and X5 through chip X3.
7	Chip X5 sends the level "01" minimum error position and the request "1000" (for the level "00" search window) to chip X2 and chip X6.
8	Chip X2 extracts the 32×32 search window from the 256×256 level and sends it to chips X4 and X5 through chip X3 and through the 36-bit bus to chip X6.
9	Chip X5 sends the level "00" minimum error position and the tag "1000" (for the reference windows) to chip X2 and chip X6.
10	Chip X2 uses this minimum error position and sends the reference windows of each of the four levels of the pyramid to chips X4 and X5. Simultaneously, chip X6 evaluates the final target position and sends the output image.

3.3 Architecture of the Xilinx chips

This section briefly describes the architecture of the seven chips. Every chip is implemented as a finite state machine, where the transitions between states are controlled not only by the system clock but also by the inputs. All state transitions occur only at the rising edge of the clock and hence the system is synchronous.

3.3.1 Structure of chip X1

Figure 3.4 shows the three main states of chip X1. States S0 and S1 are used to send the initial set of reference windows to chips X4 and X5. State S2 is used for forwarding the pyramids to chips X2 and X3 and the 256×256 images to chips X6 and X7.

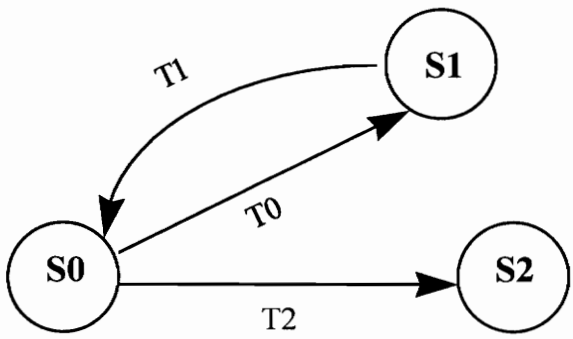


Figure 3.4 State diagram of chip X1.

In state S0, chip X1 ignores incoming pixels until it detects the first reference window pixel which occurs in the 256×256 level of the first pyramid at coordinates (r, c) = (120, 120). Then it goes to state S1 (transition T0) where the reference window pixels are sent to chips X4 and X5. After 16 pixels are sent, it returns to state S0 (transition T1). On the arrival of the next reference window pixel, which is at (121, 120), a transition occurs

again to state S1. This continues until the entire reference window for the 256×256 level is transmitted (i.e., after all 16 rows of the reference window are transmitted).

The same state machine is used for each of the remaining three levels until all the reference windows are transmitted. Then control returns to state S0 where chip X1 waits until the last pixel of the first pyramid arrives. Then a transition to state S2 occurs (transition T2) where all pyramids are sent to chips X2 and X3 while all 256×256 images are simultaneously sent to chips X6 and X7. No more state transitions will occur after this point as the initial set of reference windows has been transmitted.

The logic complexity of this chip lies primarily in the detection of the initial set of reference windows. For this purpose, the following hardware resources are used.

A 2-bit "level counter" is used to retain the current level of the arriving pyramid. A 16-bit "pixel counter" is used to infer the coordinates of the current pixel in the current level. An 8-bit "row counter" is used which is loaded with the row coordinate of first reference window pixel of a level before the arrival of that level. The logic for state S0 requires the detection of the first pixel of each row of the reference window of each level because when such a pixel arrives, control is transferred to state S2 where the entire row of the reference window is transmitted. Since the column coordinate of such pixels is a constant for a level, the row counter is concatenated with this constant column coordinate of the current level (the concatenation is converted to 16 bits for higher levels by padding with zeros) and compared with the pixel counter. If a match occurs, the row counter is incremented so that the next comparison in this state is for the first pixel of the next row of the reference window and control is transferred to state S1. In state S1, this pixel and the next fifteen arriving pixels are transmitted, completing the transmission of the first row of the reference window. Control returns to state S0 where the first pixel of the next row is awaited. In this manner, all the reference windows of the first pyramid are sent and control is transferred to state S2. In state S2, all incoming pyramids are transmitted to

chips X2 and X3 and the 256×256 level of the pyramids are transmitted to chips X6 and X7. The combination of the level counter and the pixel counter is used to detect the beginning and end of each 256×256 level.

3.3.2 Structure of chips X2 and X3

Chips X2 and X3 have almost identical architectures. They both have four main states, S0, S1, S2 and S3 as shown in Figure 3.5. In state S0, pyramid pixels coming from chip X1 are accepted and stored in the off-chip memory. As soon as a complete pyramid is stored, a transition to state S1 occurs (transition T0). State S1 is a waiting state for requests from chip X5. The first four requests will be for 32×32 search windows from the four levels of the stored pyramid. The first request is just a tag sent from chip X5 for the highest level search window. With all other requests, the minimum error position in the previous higher level search window is sent by chip X5. This is used to calculate the search window position (the upper left-hand coordinates of the search window) in the next level and the absolute minimum error position of the previous level for the next pyramid. Then control is transferred to state S2 (transition T1).

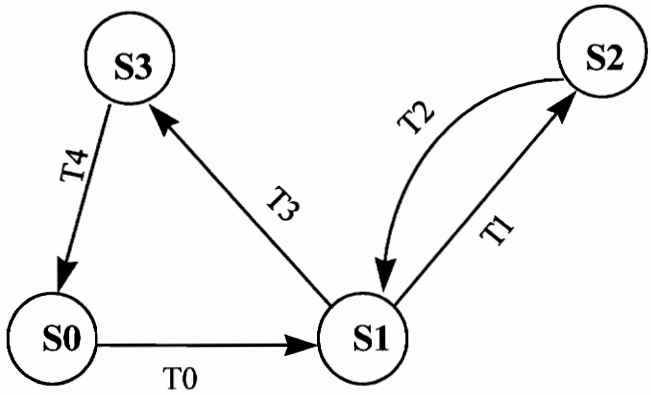


Figure 3.5. State diagram of chips X2 and X3.

In state S2, the search window is transmitted to chips X4 and X5. Then control returns to state S1 (transition T2) where the next request for a search window is awaited. After the

last search window is transmitted, a request for reference windows is awaited in state S1. When this request arrives from chip X5 with the minimum error position in the lowest-level search window, the lowest-level minimum error position is calculated and a transition to state S3 occurs (transition T3). In state S3, the set of reference windows are sent to chips X4 and X5 and control returns to state S0 (transition T4) where the next pyramid is awaited.

All the pixels of a pyramid are stored in contiguous memory locations and base pointers for each level are preserved in separate registers (they are constants since the starting address for each level is unchanged). An 18-bit memory address register is used for memory accesses. An 18-bit index register works with the base pointers to determine the memory addresses of the pixels. The calculation of the search window positions in state S1 is graphically depicted in Figure 3.6.

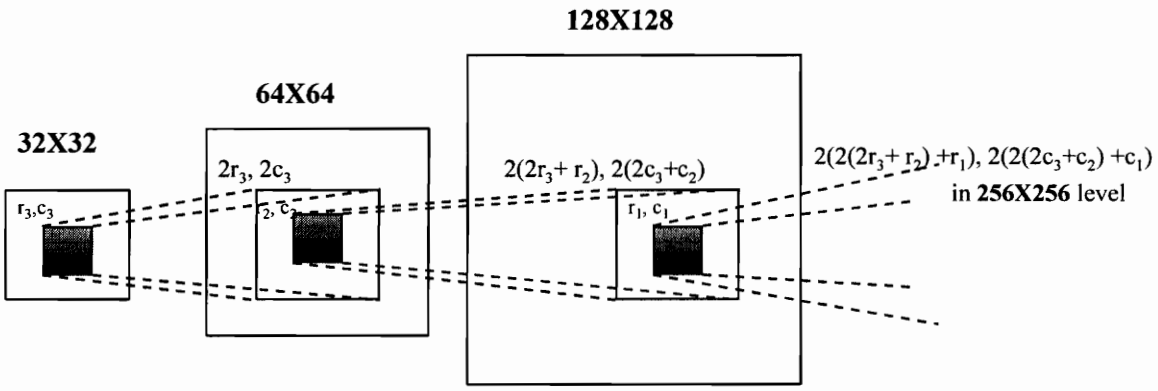


Figure 3.6 Graphical representation of search window position in different levels of the pyramid.

In Figure 3.6, minimum error positions in a search window are denoted by (r_i, c_i) where the subscript i indicates the level of the search window. As observed from the figure, the absolute search window position of the current level i can be calculated using the following equation,

$$x_{i-1} = 2(x_i + r_i), \qquad y_{i-1} = 2(y_i + c_i) \tag{3.4}$$

where x_i and y_i are the row-coordinate and column-coordinate of the search window in level i.

The equation for absolute reference window position in level i can be found according to the equation,

$$rx_i = x_i + r_i, \qquad ry_i = y_i + c_i \tag{3.5}$$

where rx_i and ry_i are the row-coordinate and column-coordinate of the reference window in level *i*.

Unfortunately, the relationship between the two equations is not as trivial as it seems to be. When a minimum error position of a level arrives from chip X5, the position of the search window of *the next lower level* is calculated by Eq. 3.4 whereas Eq. 3.5 determines the position of the reference window of *the current level*. But the total coordinate space occupied by each level differs and therefore the number of bits used to represent the coordinate space is different for each level. For example, y_3, the column coordinate of the position of the search window in the 32×32 level can only be represented by five bits whereas y_0, the column coordinate of the search window in the 256×256 level can only be represented by eight bits. To use the same resources for evaluating both equations, a mapping controller should be used which maps the resultant positions to the appropriate level.

A block diagram of the implementation of the two equations is shown in Figure 3.7. A 16-bit adder is used to calculate the reference window position of each level. The minimum error position of a level, arriving from chip X5, passes through an initial mapping controller which enhances the number of bits in the column coordinate according to the level before feeding it to the adder. The other input to the adder comes from the search window position register. For the 64×64 level, it adds one zero, for the 128×128 level, it adds two zeros and for the 256×256 level, it adds three zeros. The

32×32 level minimum error position is passed to the adder unchanged. This has to be done because chip X5 sends a minimum error position consisting of a 5-bit row coordinate and 5-bit column coordinate relative to a 32×32 search window. For this row coordinate to correspond to the row coordinate of a lower level window, the column coordinate has to be enhanced appropriately.

The output of the adder is stored in one of the four reference window position registers, selected according to the level. The contents of the adder are also fed to a second mapping controller which calculates the next lower level search window position. Since the adder contents correspond to the current level, the column coordinate should be enhanced by one zero bit for the next lower level. But from Eqs. 3.4 and 3.5, the search window position of the next level should be twice the reference window position of the current level. Hence the zero bit is added as the least significant bit (doubling the column coordinate) and the row coordinate is shifted left with the padding to yield the final search window position.

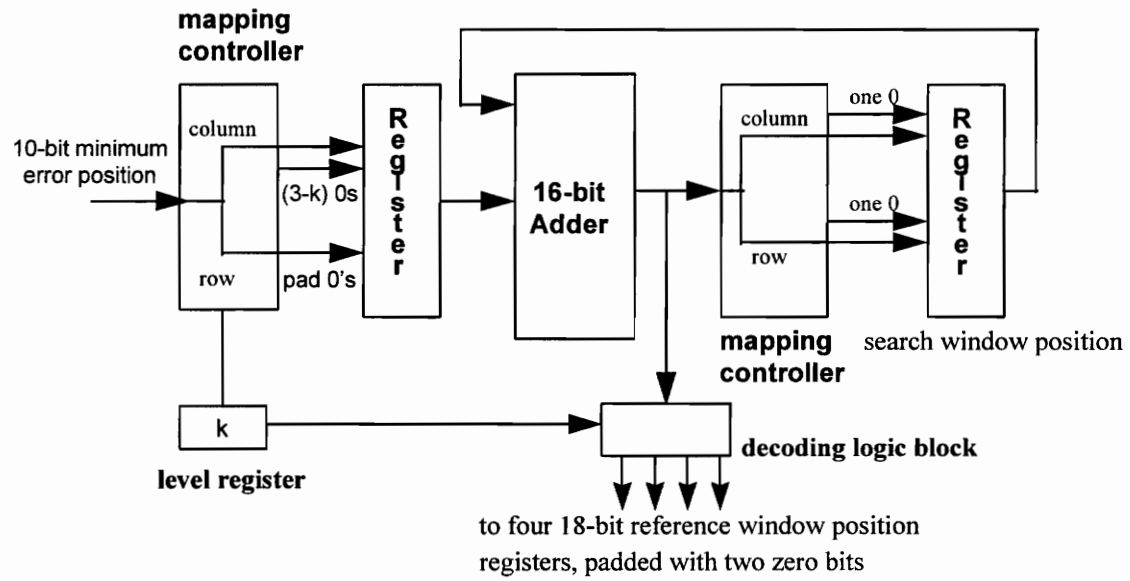


Figure 3.7 Block diagram of the logic for calculating the search window position of the next lower level (level *l-1*) and the reference window position of the current level (level *l*).

3.3.3 Structure of chips X4 and X5

Chips X4 and X5 constitute the processing hub of the system. Together, they find the minimum error correlation matching of a 16×16 reference window over all the 16×16 search blocks of a 32×32 search window. They have very similar architectures though chip X5 is slightly more complicated. As discussed in Chapter 2, correlation is a computationally intensive task requiring 256 subtractions and 256 additions for each block.

If pixels are stored serially in memory, this would require a minimum of 512 clock cycles since each memory access takes one clock cycle. Since this would prevent real-time operation (Section 3.5 discusses real-time operation in greater detail), two pixels are stored in each memory location (the memory data width is 16 bits) so that only 256 memory accesses are required to read 256 reference window pixels and 256 pixels of a block of the search window. But this means two subtractions and two additions should be performed in sequence. Since this could limit the clock frequency, the four operations are pipelined.

At any point of time during the actual correlation process, each chip has all the reference windows of the pyramid that is being processed and the search window of the current level. The memory is organized so that all the search window pixels are in one part of the memory and the four reference windows are in the other part of the memory as shown in Figure 3.8.

To avoid 18-bit address manipulations, the memory address register is divided into two parts, a base (the most significant part) and an offset. The base can take five values, four for each of the reference windows and one for the search window. The bases for the reference windows are 11 bits and the base for the search window is 9 bits. The offset can

either be the contents of 7-bit counter called the reference pointer or the contents of a 9-bit counter called the search pointer.

This organization is graphically depicted in Figure. 3.8. The base registers for the reference windows for each level are denoted by the word "base" followed by the resolution of the level. The reference pointer is denoted by "ref ptr". A two-bit level register is used to select the appropriate base register for the each level.

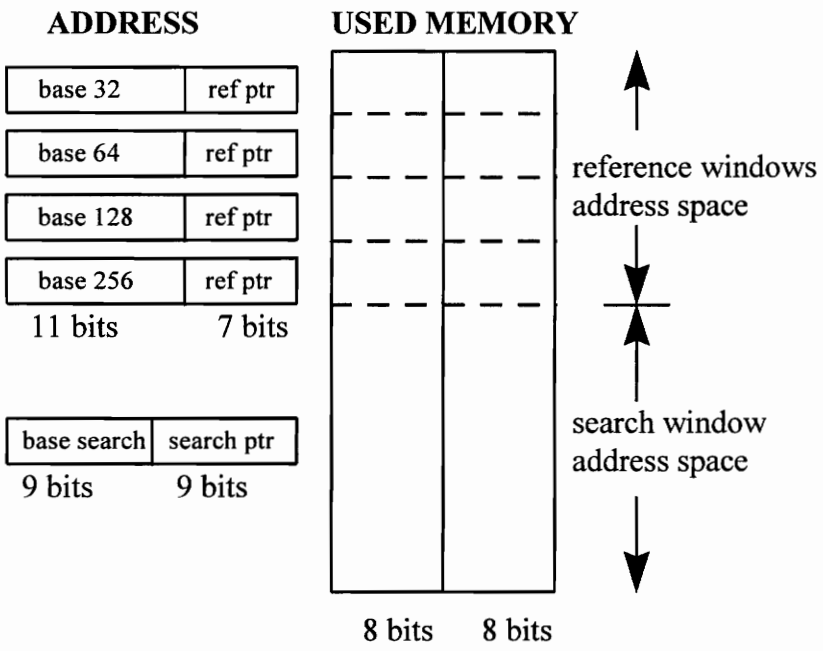


Figure 3.8 A block diagram of the division of the address space used by chips X4 and X5. The unused memory is not shown here. "ref ptr" stands for the reference pointer register.

Two pixels are stored in each memory address. For each reference window, all odd-numbered pixels are stored in the most significant 8 bits of the memory and all even-numbered pixels are stored in the other 8 bits in both the chips. Search windows are organized in the memory of chip X4 in the same way as the reference windows but for chip X5, all the odd-numbered pixels are stored in the least significant 8 bits and all the

even-numbered pixels are stored in the other 8 bits. Chip X5 also drops the first (pixel 0) and last (pixel 1023) pixels of the search window since it does not use them. Figure 3.9 shows how the windows are organized in the memory.

This organization makes the access of corresponding reference window pixels and search window pixels very simple. For instance, in chip X4, since blocks begin on even columns, the first pixel of a block of a search window of a level is always in the least significant eight bits of the memory location. This means that the corresponding pixel of the reference window of that level is also in the least significant eight bits. Hence only 128 accesses are necessary to read the 256 pixels of a block from the memory instead of 129 accesses if the corresponding pixels of the reference window and a block beginning on an even column of the search window are not stored in the same part of a memory location. The same is the case for chip X5 too.

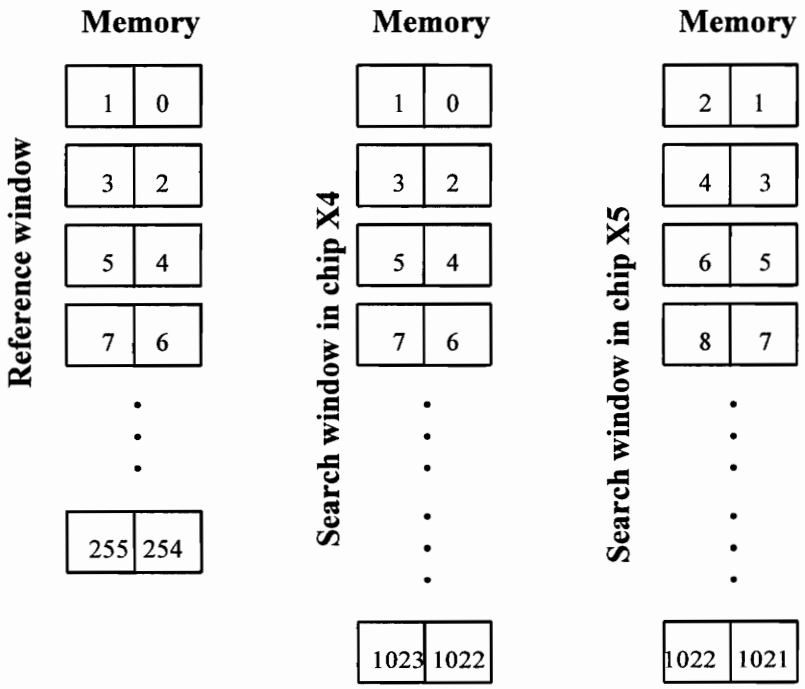


Figure 3.9. Organization of a reference window and the search window in the occupied memory space for chips X4 and X5.

The five main states for the correlation process is shown in the state transition diagram in Figure 3.10. In state S0, reference window pixels are accepted and stored in memory. Initially they come from chip X1. After all the reference windows are received, a transition to state S1 occurs (transition T0).

In state S1, a request for the lowest level search window is sent through the crossbar to chips X2 and X3 by chip X5. The request is a four bit tag. One bit is for valid data, one bit is for distinguishing chips X2 and X3 and two bits indicate the level of the search window requested. The first request is for the highest level search window. When the search window pixels arrive, they are stored in the search window address space. After the search window is completely received, a transition to state S2 occurs (transition T1).

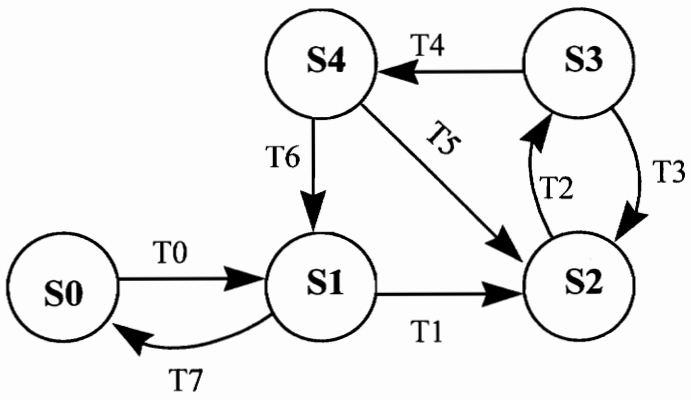


Figure 3.10 State diagram of chips X4 and X5. The transitions are numbered in the order they occur.

In state S2, the first two reference window pixels are read from the memory. Then a transition to state S3 occurs (transition T2) where two search window pixels are read from the memory. Then the four pixels are sent to an subtractor-adder-accumulator block which computes the absolute differences of the corresponding pixels, adds the two results and accumulates the net result. This block will be described later in this section. Control returns to state S2 (transition T3) and the next two reference window pixels are read from

the memory. This continues until an entire block is read from the search window memory. Then a transition to state S4 occurs (transition T4).

In state S4, a minimum generator is used to calculate the minimum error and the minimum error position. Initially, a 16-bit minimum register is loaded with all ones. In state S4, the minimum register is compared with the 16-bit accumulator from the subtractor-adder-accumulator block and the lower of the two values is loaded into the minimum register. A ten-bit block pointer register is used to indicate the position of the current block of the search window that is being processed. Initially this block pointer is 0 in chip X4 and 1 in chip X5 (since correlation in chip X4 begins with block 0, the first even-numbered column, and correlation in chip X5 begins with block 1, the first odd-numbered column).

The minimum generator is modeled using behavioral VHDL code. Figure 3.11 shows a section of the pseudo-code used to model the minimum generator. A 10-bit minimum error position register is used to record the minimum error position. This register is loaded with the block pointer when the accumulator contents are less than the minimum register contents. The block pointer is incremented by two if there are still unprocessed blocks left in the current row of the search window. If the last five bits of the block pointer equal 16 in chip X4 and 15 in chip X5, the last block in the current row of the search window has been processed for each chip. In such cases, the block pointer will be incremented by 16 in chip X4 and 18 in chip X5 to point to the first block in the next row of the search window. and the search pointer is loaded with the most significant nine bits of the block pointer (since two pixels are stored in every memory location).

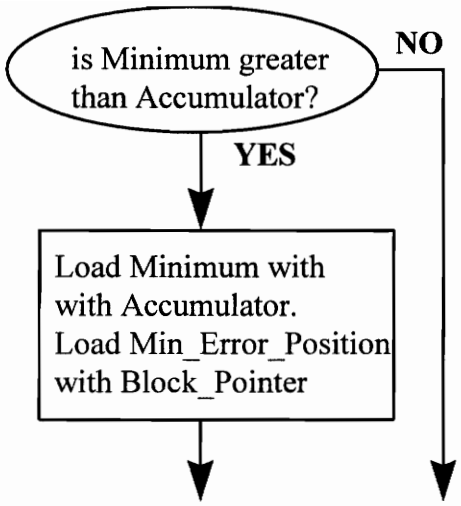


Figure 3.11 Behavioral description of the minimum generator. Minimum is a 16-bit register that contains the minimum error. Min_Error_Position is a 10-bit register that contains the position of the block that has the minimum error with respect to the current reference window. Block_Pointer is a 10-bit register which holds the position of the current block being processed.

A transition to state S2 occurs (transition T5) if there are still unprocessed blocks in the search window and the next block is processed. If all the blocks of the search window are exhausted, control is transferred to state S1 (transition T6) where the minimum error position contents is sent with a request for the next lower level search window to chips X2 and X3 and the base for the reference window address access is changed to the next lower level reference window base pointer. Once the lowest level search window is processed, a request for reference windows is transmitted to chips X2 and X3 with the minimum error position of the lowest level search window from state S1. The end of all search windows is detected by a two-bit level counter. Then control is transferred to state S0 (transition T7) where the new set of reference windows are awaited.

The subtractor-adder-accumulator block is shown in Figure 3.12. The absolute subtractors give the absolute difference of the two inputs. An absolute subtraction is formed by comparing the two inputs and sending them in an order to the subtractor so that the

smaller input is always subtracted from the larger input. The accumulator is 16 bits so that 256 8-bit values can be accumulated in it without overflow. Since the adder output is 9 bits, it is padded with 7 zeros before going to the accumulator. Pipeline registers are used to preserve intermediate results for use in the next clock cycle. The latency of the pipeline is three clock cycles as seen from the figure.

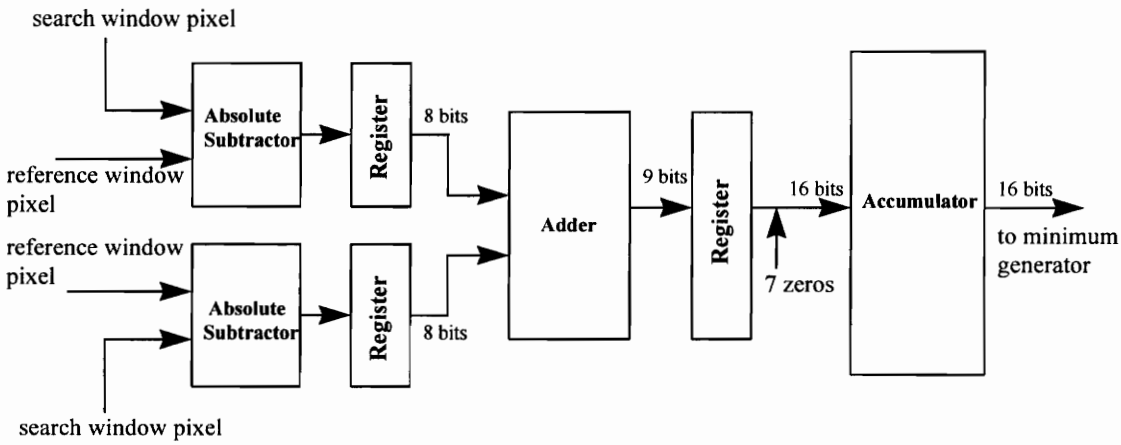


Figure 3.12 Block diagram of the subtractor-adder-accumulator block.

After chip X5 has finished processing its part of the search window, it goes into a wait state where it waits for chip X4 to finish its processing. After chip X4 finishes, it sends its minimum error and minimum error position to chip X5. Chip X5 loads the minimum error of chip X4 in the accumulator and the minimum error position of chip X4 in the block pointer. Then it compares the two minimums and selects the best of the two positions using the same minimum generator block. This will be transmitted to chips X2, X3, X6 and X7 with the appropriate request.

3.3.4 Structure of chips X6 and X7

These two chips have to store 256×256 images coming from chip X1, compute the final target locations using information received from chip X5, highlight the detected target

and send the resulting image as output. Highlighting is done by drawing a white rectangle of size 16×16, enclosing the target area, on the original frame. The method that is used for highlighting the target is similar to the method used in chip X1 for determining the initial set of reference windows from the input pyramid stream. The computation of the final minimum error position is similar to the method used to determine the search window positions in chips X2 and X3.

There are four main states for chips X6 and X7 as depicted in the state diagram in Figure 3.13. Initially, in state S0, incoming pixels are accepted until a complete frame is received. Then a transition to state S1 occurs (transition T0). In state S1, the relative minimum error positions of each level are received from chip X5 and they are accumulated and the final minimum error position is computed very similar to the method used in chips X2 and X3. In fact, the only change is the slight modification of Eq. (3.5) using Eq. (3.4) to obtain the new equation,

$$rx_i = 2rx_{i+1} + r_i \qquad (3.6a)$$

$$ry_i = 2ry_{i+1} + c_i \qquad (3.6b)$$

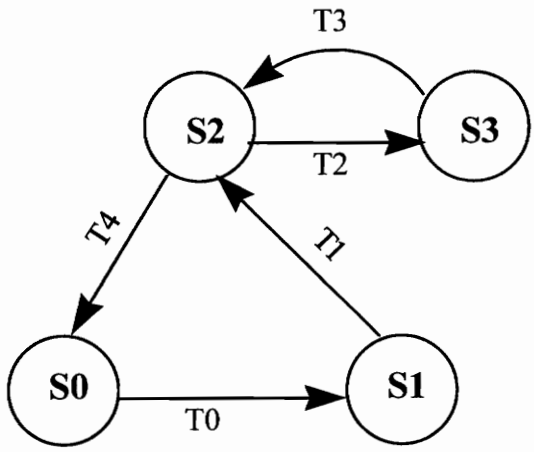


Figure 3.13. State diagram of chips X6 and X7.

When the highest-level minimum error position arrives, it is stored in a 16-bit "target register". This register represents the absolute minimum error position of the current level (which is (rx_i, ry_i) in Eq. 3.6). When the minimum error position of the next level arrives, the contents of the target register are mapped to the coordinate space of that level using a mapping controller block, similar to the one used in chips X2 and X3. The new row and column coordinates are each shifted left once by the mapping controller and the result is added to the minimum error position (to account for the multiplication by two of the previous absolute minimum error position (rx_{i+1}, ry_{i+1}) in Eq. 3.6). The result of the addition is stored back in the target register. This continues until the lowest-level minimum error position, arriving from chip X5, is added to the shifted contents of the target register. The target register will hold the final target position now. A transition to state S2 occurs at this point (transition T1).

In state S2, the pixels in the memory are read and transmitted until the first pixel of the target is encountered (the position of this pixel is the final target position). Then a transition to state S3 occurs (transition T2) where sixteen white pixels are transmitted. These pixels form the upper horizontal side of the white box. Then the system returns to state S2 (transition T3) and in this state, the two white pixels of the next fourteen rows are transmitted, completing the vertical sides of the white box. When the first pixel of the last row of the target arrives, state S3 is again entered where sixteen white pixels (completing the white box) are transmitted and the system is returned to state S2. The remainder of the image is transmitted in state S2 and control returns to state S0 (transition T4) where the arrival of a new frame is awaited.

A similar mechanism to the one used in chip X1 for detecting reference window pixels is used here to detect the location to place a white box pixel. A 16-bit pixel counter is used to infer the coordinates of the pixel currently being read from the memory. An 8-bit row counter, two 8-bit row registers and two 8-bit column registers work with the pixel counter to detect a white box pixel. The first row register and the first column register are

loaded the row coordinate and the column coordinate of the upper left-hand position of the white box (which is the target position, held by the target register) while the second row register and the second column register are loaded with the coordinates of the lower right-hand column coordinate of the white box (this value is obtained by adding 15 to each of the coordinates of the target position). The row counter is initially loaded with the row coordinate of the target position.

The first row register is concatenated with the first column register and compared with the pixel counter. When a match occurs, the row counter is incremented and control is transferred to state S2 where the top side of the white box is transmitted. The row counter is now setup for detecting the next row of the target. For the next fourteen rows, it is concatenated first with the first column register and if a match with the pixel counter occurs, a white pixel is transmitted, and then it is concatenated with the second column register and if a match with the pixel counter occurs, another white pixel is transmitted. After the second white pixel in a row is transmitted, the row counter is incremented. After the row counter is incremented fourteen times, it is reset to zero. Finally, for the last row of the target, the second row register is concatenated with the first column register (this will detect the lower left-hand coordinate of the white box) and if a match with the pixel counter occurs, control is transferred to state S2 where the bottom side of the white box is transmitted.

In the behavioral VHDL implementation, all the comparisons mentioned above are made simultaneously but the actions are prioritized such that a control transfer to state S2 has the highest priority and a no-action comparison result (i.e., not sending a white pixel) has the lowest priority.

3.4 Theoretical validation of the architecture for real-time performance

The system has been designed, but it does it really work for real-time data? Before actual implementation, this system has been validated for real-time performance. The following three factors are considered for the analysis,

1. Data transfers to chips X4 and X5,
2. Arithmetic operations and memory accesses for correlation, and
3. Sending output image by chips X6 and X7.

At any point during steady-state operation, one chip has an entire pyramid stored in its memory, which is being processed, and another chip is receiving a new pyramid. To validate the architecture, it should be proved that the entire processing time for the stored pyramid is less than time needed for the arrival of the current incoming pyramid. If this constraint is violated, the next incoming pyramid cannot be stored in the chip which has the pyramid that is being currently processed. Pyramids arrive at the rate of one pixel per clock cycle.

Because there are four pyramid levels, the number of the data transfers required for correlation is equivalent to four reference windows and four search windows of pixels. This is equal to $4\times(16\times16 + 32\times32)$ because the size of the reference window is 16×16 and the size of the search window is 32×32. In chip X4, (17×9) blocks of a search window are correlated with a reference window in each level. Correlation of each block requires 256 memory accesses (128 each for the block of the search window and the corresponding reference window) and 256 arithmetic operations (the 256 subtractions and 256 additions happen simultaneously because of pipelining). Therefore chip X4 requires approximately $4\times(256\times160)$ clock cycles including overhead. This is greater than the processing time of chip X5 (only 17×8 blocks) and since they happen almost simultaneously, processing time for correlation in chip X5 is not considered.

Because images have to be transmitted at the rate of one pixel/second, sending the output image requires at least 256×256 cycles. Until then, the chip which is sending the output image is not ready to accept a new image. Therefore the total number of processing cycles is approximately 4×(256 + 1024) + 4×(256×160) + (256×256) which is less than 460×512. However, taking overhead into account, a ceiling of 480×512 cycles can be taken as a conservative estimate of the total processing time.

But a pyramid generation requires at least 512×512 cycles because the original input to the pyramid generation system is a 512×512 image. Therefore this system can work under real-time conditions.

CHAPTER 4

DESIGN OF A TRACKING SYSTEM FOR 512×512 IMAGES ON SPLASH II

4.1. System Overview

The four-chip Gaussian pyramid application on Splash II [5] generates a pyramid of four levels, which are 256×256, 128×128, 64×64 and 32×32, for every alternate image of size 512×512 coming from a video camera. In the tracking system described in Chapter 2, these four levels are designated as levels 0, 1, 2 and 3 respectively. The 256×256 tracking system applies the tracking algorithm, described in Chapter 2, on these four levels to detect the location of the moving target in the lowest level of the pyramid. However, for the detection of the location of the target in the 512×512 original image coming from the camera, the search has to be extended to the 512×512 original image. This can be achieved by storing the 512×512 image in the memory as the lowest level of the pyramid, level 0, and the four levels of the pyramid generator as levels 1, 2, 3 and 4, so that the same tracking algorithm can be applied on the resulting five levels of the pyramid to detect the location of the target in the lowest level of the pyramid.

The four part tracking system, described in Section 3.1, has been modified to accommodate the processing of a five-level pyramid as shown in Figure 4.1. Part 1 extracts the initial set of reference windows from the highest four pyramid levels (coming from the Gaussian pyramid application) and sends them to Part 3 and forwards every other pyramid to Part 2. Part 2 stores the highest four pyramid levels in its memory and sends the reference windows and search windows of each level to Part 3. Part 3 performs correlation of a reference window over a search window and sends the minimum error position to Parts 2 and 4 for computing the locations of the reference windows, search windows and the target. Part 4 stores alternate 512×512 images coming from a camera, sends the reference window and search window of each image to Part 3 for correlation,

computes the location of the target in the 512×512 image and displays image with the target highlighted by drawing a white rectangle of size 16×16 enclosing the target. Part 4 also provides the initial level 0 reference window for Part 3 by extracting the central 16×16 block from the first image.

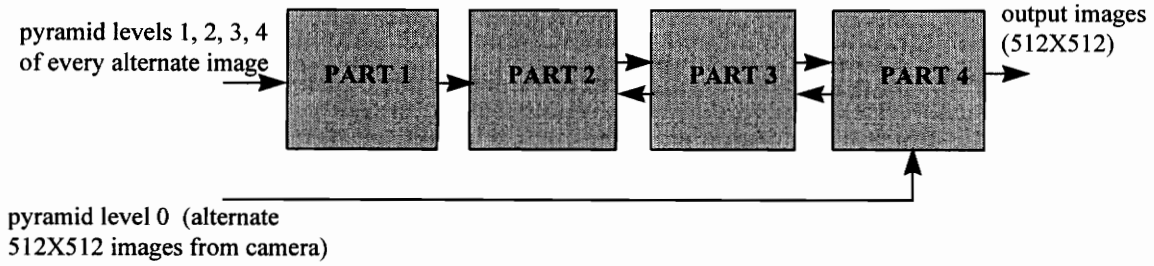


Figure 4.1. Four-part tracking system for 512×512 output image. The input to Part 1 is the four levels of the pyramid, 256×256, 128×128, 64×64 and 32×32, for every alternate 512×512 image from a camera, generated by a Gaussian pyramid application on Splash II. The original 512×512 image from the camera is directly stored by Part 4. The output image is the original image stored in Part 4 with the target highlighted.

The first three parts of this system are very similar to the first three parts of the 256×256 tracking system. One chip is required for Part 1 and two chips each are required for Parts 2 and 3 as explained in Section 3.1. Part 4 requires two chips for processing, the first chip processes all even-numbered images and the second chip processes all odd-numbered images, and one chip for receiving data from Part 3 and from the camera and forwarding the data to the other two chips. Hence, a total number of eight chips are required for this system. The control chip of the Splash II system is used to generate alternate images for the four-chip Gaussian pyramid generator and the tracking system.

This system is incapable of processing every image coming from the camera. This is because 512×512 images are transmitted continuously from the camera and each image has to be stored, processed and transmitted out. Since an output image can be transmitted only at the rate of one pixel per clock cycle, for the system to work for every image,

processing time becomes zero. If simultaneous memory reads and writes were possible or a large FIFO becomes available, this system can be modified to work for every image coming from the camera.

The next section describes the architecture of the overall system on Splash II. Since there are no major changes in Parts 1 and 2 from the 256×256 system, the next Section 4.3 briefly describes the modifications made in Parts 3 and 4.

4.2. Architecture of the 512×512 tracking system on Splash II

The eight-chip design of the tracking system is shown in Figure 4.2. The control chip receives 512×512 images coming from a camera and forwards alternate images to the four-chip Gaussian pyramid generator [10] and the four-part tracking system. Since only twelve chips are required for the entire system, it can fit on one Splash II processor board.

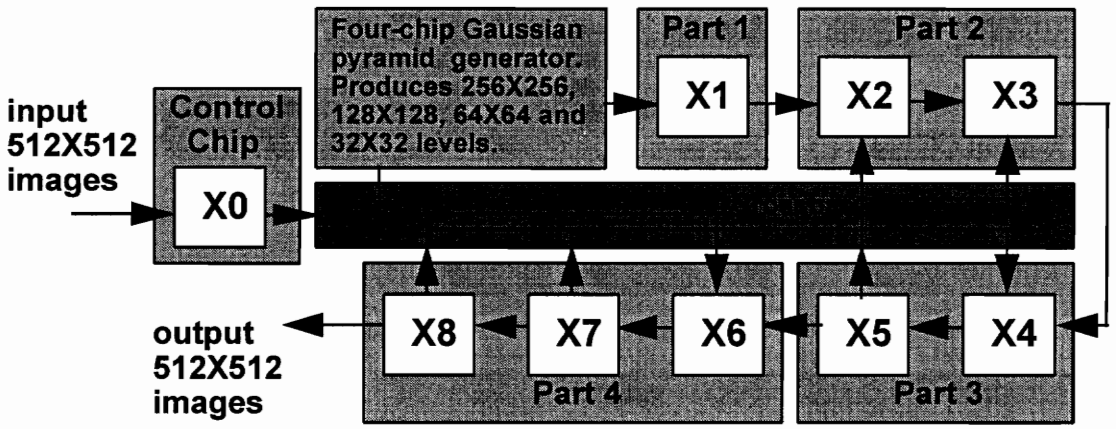


Figure 4.2. Architecture of the eight-chip tracking system on Splash II for a 512×512 output image. The control chip sends every alternate 512×512 image coming from the camera to the pyramid generator and chip X6 of Part 4 of the tracking system.

The Gaussian pyramid generator produces the four levels 256×256, 128×128, 64×64 and 32×32 for every image and these levels are stored as levels 1, 2, 3 and 4 in Part 2 of the

tracking system. Simultaneously, the original 512×512 image is stored as level 0 in Part 4 of the tracking system.

Chip X1 constitutes Part 1 of the tracking system. It extracts the initial set of reference windows from the four levels produced by the Gaussian pyramid generator for the first image received from the control chip. Chip X1 forwards the four levels produced by the Gaussian pyramid generator for every other image to Part 2 of the system.

Chips X2 and X3 constitute Part 2 of the tracking system. Chip X2 stores the highest four pyramid levels (produced by the Gaussian pyramid generator) for every odd-numbered image that the pyramid generator receives from the control chip and chip X3 stores the highest four pyramid levels of every even-numbered image. While one chip is storing data, the other chip extracts the search windows and reference windows of the four pyramid levels in its memory in response to requests from chip X5 and sends them to chips X4 and X5. The request is in the format of a five-bit tag. The first bit is the valid bit, the next three bits indicate the level of the search window requested and the last bit is identifies the recipient chip.

The third part is implemented on chips X4 and X5. Initially the reference windows of the highest four pyramid levels are received from Part 2. The lowest level reference window is received from Part 4. The correlation of all 16×16 blocks of the search window of a level that begin on even columns with the reference window of that level is implemented on chip X4 (using Eq. 3.1) and the correlation of all 16×16 blocks that begin on odd columns of the search window is implemented on chip X5 (using Eq. 3.2), just like in the 256×256 tracking system. However, in this case, correlation is performed on five pyramid levels for every image. After the minimum error position in the 256×256 level search window is evaluated, chip X5 requests for the search window in the 512×512 level stored in Part 4. Every block of this search window is correlated with the lowest level reference window and the result is transmitted to Part 4 for generating the output.

Chips X6, X7 and X8 constitute Part 4 of the system. Chip X6 receives incoming images from the control chip on its crossbar bus and data from chip X5. Chip X6 combines the two sets of data in a proper order and transmits them to chips X7 and X8. Chip X7 stores all even-numbered images received from chip X6 and chip X8 stores all odd-numbered images received from chip X6. While one chip is storing image data, the other chip accumulates the minimum error positions of the highest four levels received from chip X5 and evaluates the search window position in the lowest level of the pyramid stored in its memory. It then transmits the lowest level search window to chips X4 and X5 through the crossbar. Finally, it receives the lowest level minimum error position from chip X5. This is used for evaluating the absolute position of the target in the stored image, for transmitting the new reference window of this level and for transmitting the output image. The new reference window of this level is the 16×16 block beginning at the target location. The output image is formed by drawing a white rectangle of size 16×16 enclosing the target in the original 512×512 image (the stored image).

Table 4.1 shows the communication between the chips for the processing of an odd-numbered pyramid stored in chips X2 and X7. During the processing of this pyramid, the incoming pyramid is being stored in chips X3 and X8.

The system is designed to be completely synchronous. The structures of chips X1, X2 and X3 are the same as the corresponding chips in the 256×256 tracking system. The next section describes briefly the changes made to the structures of the chips of Parts 3 and 4 of this system as compared to Parts 3 and 4 of the 256×256 tracking system.

Table 4.1.
Communication between the chips of the tracking system for the processing of an odd-numbered pyramid.

TIME	EVENT
1	Chip X5 sends request "11000" through the crossbar for highest level search window (level "100") to chip X2.
2	Chip X2 sends the 32×32 pyramid level from its memory through the 36-bit bus to chip X3 and chip X3 forwards this data to chips X4 and X5.
3	Chips X4 and X5 perform correlation of this search window with the highest level reference window. Chip X5 sends the minimum error position and the request "10110" (for the level "011" search window) through the crossbar to chip X2 and to chip X7 through chip X6 (for using these positions to evaluate the lowest level search window position and the final target position).
4	Chip X2 evaluates the absolute search window position and extracts the 32×32 search window from the 64×64 level and sends it to chips X4 and X5 through chip X3.
5	Chip X5 sends the level "011" minimum error position and the request "10100" (for the level "010" search window) to chip X2 and chip X7.
6	Chip X2 extracts the 32×32 search window from the 128×128 level and sends it to chips X4 and X5 through chip X3.
7	Chip X5 sends the level "010" minimum error position and the request "10010" (for the level "001" search window) to chip X2 and chip X6.
8	Chip X2 extracts the 32×32 search window from the 256×256 level and sends it to chips X4 and X5 through chip X3.
9	Chip X5 sends the level "001" minimum error position and the request "10000" (for the level "000" search window) to chip X2 (for evaluating reference window positions) and chip X7 (for level "000" search window).
10	Chip X7 extracts the 32×32 search window from the 512×512 level and sends it to chip X4 through the crossbar. Chip X4 forwards it to chip X5.
11	Chip X5 sends the request "10000" (for the reference windows) to chip X2.
12	Chip X2 sends the reference windows of the highest four levels of the pyramid to chips X4 and X5.
13	Chip X5 sends the level "000" minimum error position and the request "10000" for the lowest level reference window to chip X7.
14	Chip X7 sends the lowest level reference window to chips X4 and X5. After this, chip X7 sends the output image through the 36-bit bus to its right neighbor.

4.3.1. Structure of Chips X4 and X5

The state machine used for chips X4 and X5 in Part 3 of the 256×256 tracking system is modified with two additional states. Figure 4.3 shows the state machine for the two chips. Notice that this figure shows two additional states, $S0_0$ and $S1_0$ as compared with Figure 3.10, the state diagram for chips X4 and X5 of the 256×256 tracking system. Initially, chip X4 receives the reference windows of the highest four pyramid levels from chip X3 (these windows are extracted by chip X1) in state S0. Then a request for the reference window of the lowest level is transmitted by chip X5 to Part 4 of the system and control is transferred to state $S0_0$ (transition $T0_0$). In this state, the reference window of the lowest pyramid level is received from Part 4 by chip X4 on its crossbar bus. Chip X4 forwards this data to chip X5. Control is then transferred to state S1 (transition $T0_1$) where the highest level search window is requested, as in the 256×256 tracking system. This search window is received by chip X4 from Part 2 of the system and chip X4 forwards it to chip X5.

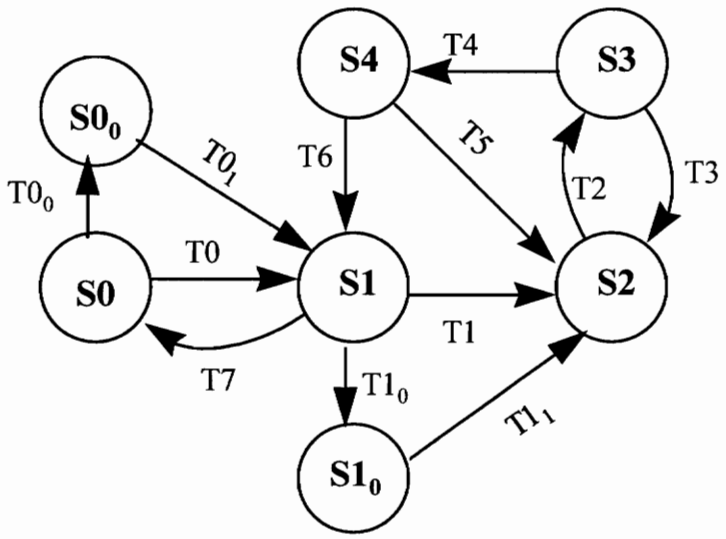


Figure 4.3. State diagram of chips X4 and X5.

The system then steps through the states S2, S3, S4 and S1 for processing each of the search windows of the highest four levels as it does in the 256×256 tracking system. When control returns to state S1 after processing the search window of the 256×256 pyramid level (transition T6), a request for the 512×512 level search window is transmitted by chip X5 to Part 4 of the system and control is transferred to state $S1_0$ (transition $T1_0$).. In this state, the lowest level search window is received by chip X4 from Part 4 through the crossbar and the system processes this search window by stepping through the states S2, S3 and S4. Finally, control returns to S0 (transition $T1_1$) where the reference windows for processing the next pyramid are awaited.

4.3.2. Structure of Chips X7 and X8

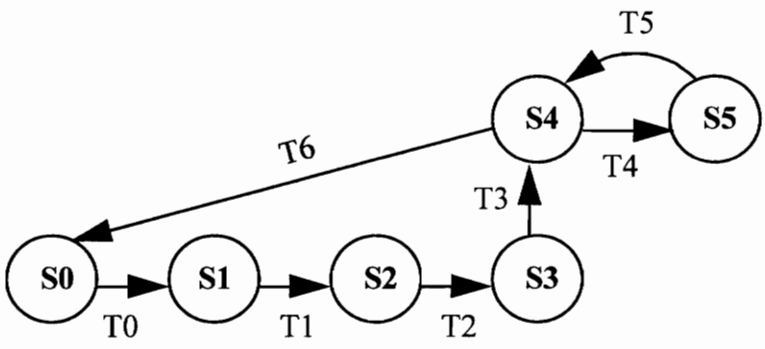


Figure 4.4. State diagram of chips X7 and X8.

Figure 4.4 shows the state diagram of chips X7 and X8. In state S0, a 512×512 image is received from chip X6 and control is transferred to state S1 (transition T0). In state S1, minimum error positions of the highest four levels are received one at a time from chip X6 (originally transmitted by chip X5 and forwarded by chip X6). These minimum error positions are accumulated using Eq. 3.4 to yield the search window position in the lowest level of the pyramid. The search window is extracted from the stored image (which corresponds to the lowest level of the pyramid) and transmitted to chip X4 through the

crossbar. A transition to state S2 occurs at this point (transition T1). In state S2, the minimum error position of the lowest level search window is received from chip X5 and this position is used together with the lowest level search window position to evaluate the location of the target in the 512×512 image using Eq 3.5. Control is transferred to state S3 (transition T2) where a request for the lowest level reference window is received from chip X5 (this reference window is the 16×16 block beginning at the target position in the stored image). This window is transmitted and a transition to state S4 occurs (transition T3). The system steps through the states S4 and S5 to send the output image (these states correspond to states S2 and S3 for chips X6 and X7 in the 256×256 tracking system). Finally, after the last pixel of the output image is transferred, control returns to state S0 (transition T6) where the next 512×512 image is awaited.

4.4. Theoretical Validation of the Architecture for Real-Time Performance

This following factors are considered for analysis of the architecture,

1. Data transfers to Part 3 (chips X4 and X5 which perform correlation),
2. Arithmetic operations and memory accesses for correlation, and
3. Sending output image by Part 4 of the system.

At any point during steady-state operation, one chip has the highest four levels of the pyramid stored in its memory (Part 2 of the system) and another chip has the lowest level of the pyramid stored in its memory (Part 4 of the system). While this pyramid is being processed by the system, the current incoming pyramid is being stored in two other chips of the system. To validate the architecture, it has to be proved that the entire processing time for the stored pyramid is less than the time needed for the arrival of the incoming pyramid.

Since the 512×512 image, which is the lowest level of the pyramid, arrives simultaneously with the other four levels of the pyramid, the time needed for arrival of a pyramid can be conservatively estimated as 512×512 clock cycles (images arrive at the rate of one pixel per clock cycle). Since the system is designed to process only alternate pyramids, the total time for the processing of a pyramid has to be less than 2×512×512.

Because there are five pyramid levels, the number of data transfers to chips X4 and X5 is equivalent to five reference windows and five search windows. This is equal to 5×(16×16 + 32×32). The time required for correlation in chip X4 is 5×(17×9×256) (see Section 3.4).

Because images have to be transmitted at the rate of one pixel per clock cycle, sending the output image by Part 4 of the system requires 512×512 clock cycles.

Adding all these figures and accounting for overhead, the total number of processing cycles required is approximately 950×512 clock cycles which is less than 2×512×512 clock cycles. Therefore the system is capable of processing every alternate image generated by a camera.

CHAPTER 5
IMPLEMENTATION ON SPLASH II

5.1. Implementation Procedure

The visual tracking systems developed in this thesis are designed in VHDL behavioral code. A VHDL model for each chip of the two tracking systems is designed using special libraries for Splash II developed by the Center for Computing Sciences, formerly the Supercomputing Research Center in Bowie, Maryland. The Synopsys VHDL design tools are used for compiling, simulating and synthesizing the VHDL code. The synthesis produces a netlist file in EDIF format. This is converted into the Xilinx Netlist Format (XNF) using a shell script. The XNF file is used as input by the Xilinx tool *ppr* for partitioning, placement and routing of the design on an FPGA and generates a Logic Cell Array (LCA) file as the output file. The timing extraction is performed by the Xilinx tool *xdelay* using the LCA file. The software *makebits* converts the LCA file into a bitstream file that can be used to configure the Xilinx chips on Splash II.

Simulation is performed for functional verification. Since the VHDL models requires a large amount of data, a "test" VHDL model is developed for each chip. This VHDL model is designed to work for small images (of the size 32×32). This test VHDL model is simulated for functional testing. If any bugs are discovered, the test VHDL model is refined to remove the bugs. This process is repeated until the test VHDL model simulation produces expected results. Then this VHDL model is modified to work for the real input data (for images of the size 512×512) and synthesized with the Synopsys and Xilinx tools.

Figure 5.1 shows a flow diagram of the implementation process. The design is described in a natural language like English and then converted to a VHDL behavioral model which works for test inputs. This model is simulated for functional verification. After the simulation shows that the model is logically correct, the next step is synthesis of the

VHDL model which works for real inputs. A timing extraction is performed after synthesis to check if the design can function at the desired clock frequency. The VHDL design is modified and synthesized until it is able to function at the desired frequency of operation.

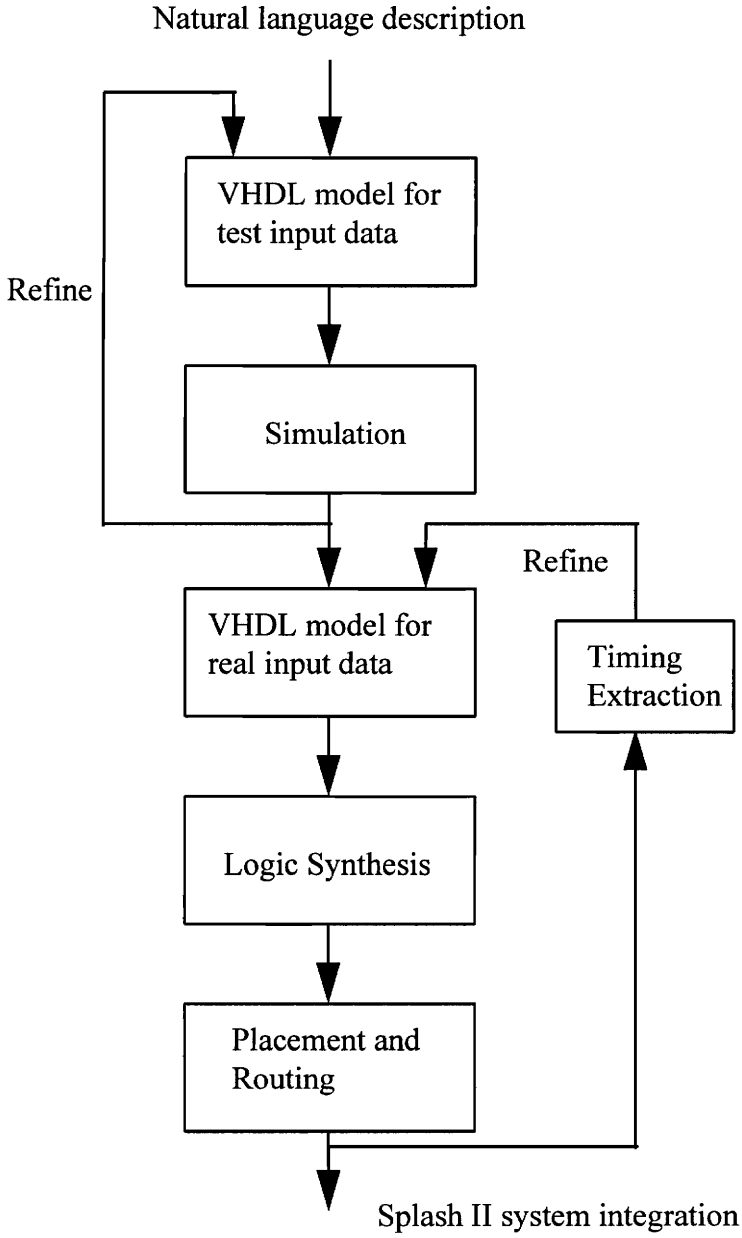


Figure 5.1. Flow diagram of the process of implementation of a chip design on a Xilinx chip of Splash II. The design is originally described in natural language and the following steps are taken to realize its implementation on Splash II.

5.2. Integration of the Tracking Systems on Splash II

After all the chip designs are synthesized successfully, the bitstream files of each chip are configured on the Splash II FPGAs using a C control program. Highly versatile C libraries and macros are available with the Splash II system. These can be used for loading the configurations of the Splash II FPGAs, for direct access to the memories of the chips, for running the system with a software clock (single-stepping), for using data files to provide the inputs and store the outputs of the system, and for many other purposes.

A crossbar configuration file is used to set the configuration of the crossbar. The crossbar can have eight different settings during the course of operation of the system. Each setting specifies the input ports and output ports of the crossbar. The 36-bit data path bus to the crossbar of each chip can be divided into five sections: a four-bit tag and four bytes. Each of these sections can be connected to a different port. For instance, in the 512×512 tracking system, chip X4 receives 16 bits of data on the crossbar from chip X7 and 16 bits of data on the crossbar from chip X8. In the crossbar configuration file, the input port for the most significant two bytes of the crossbar bus of chip X4 has been specified as chip X7 while the input port for the least significant two bytes of the crossbar bus has been specified as chip X8.

A C control program is used for configuring the Splash II system and running it with a software clock to generate the results for this thesis. For real-time operation, the system is run with the hardware clock for the data coming from the VTSplash camera.

CHAPTER 6
RESULTS

6.1. Results of the 256×256 Tracking System

The 256×256 tracking system (as described in Chapter 3) has been tested with a sequence of five images of a taxi turning a corner. The same sequence has been used for the C based software model explained in Chapter 2. The results generated by the 256×256 tracking system are displayed in Figure 6.1 (on the next page). The first image in the sequence has been used to produce the initial set of reference windows. A comparison of Figure 6.1 and Figure 2.6 (the results of the C based software model) shows that the resulting image sequences produced by the tracking system and the software model are identical.

The upper-left corner coordinates of the target image are given under each image. The positions indicate that the target has been both horizontally and vertically displaced in the later images of the sequence. A slight rotation of the target can also be observed from the images.

6.2. Timing Analysis of the 256×256 tracking system

The system has been designed to work at 30 frames (of size 512×512) per second. Therefore the tracking system needs to function at a minimum rate of 7.87 MHz. The speed of the system has been measured by finding the individual speeds of the FPGA chips that are used in the tracking system. The Xilinx *xdelay* software has been used to analyze the critical path delays of each chip of the tracking system. The Xilinx *XACT* schematic design editor can also used to perform a timing analysis of a chip design. Table 6.1 summarizes the critical speeds of the seven chips of the tracking system.

(a) (120, 120)

(b) (117, 119)

(c) (115, 113)

(d) (109, 108)

Figure 6.1. Five images of taxi sequence [20] have been processed to produce these four images. The upper left corners of the target in each image has been shown under the image. The same input sequence has been used to produce the software results given in Figure 2.6. A comparison between the two figures shows a perfect match. The target has been displaced by approximately 10 units each in the horizontal and vertical directions from the first image to the fourth image.

Table 6.1.

Maximum operating clock frequencies of the seven chips of the 256×256 tracking system

Chip X1	Chip X2	Chip X3	Chip X4	Chip X5	Chip X6	Chip X7
12.0 MHz	10.2 MHz	10.0 MHz	8.7 MHz	8.8 MHz	13.2 MHz	12.7 MHz

Since the speed of the tracking system is only as good as the speed of the slowest chip of the system, the maximum operating clock frequency of the tracking system is 8.70 MHz. But this speed is above the calculated minimum rate of 7.87 MHz required for processing 30 pyramids per second. Therefore the system can operate at 30 pyramids per second.

The A/D board in the video interface cabinet of the VTSplash system transmits images at the pixel rate of 10.0 Mhz. Therefore the system cannot be guaranteed to work with this camera. However, the design of the seven chips of the system can be improved, as explained later in the next chapter, to enable the system to operate at 10.0 MHz.

6.3. Results of the 512×512 tracking system

The 512×512 tracking system has been tested with a sequence of four images of a coat on a chair. The four images are actually a stereo image pair that is repeated as follows: Left, Right, Left, Right. The results generated by the tracking system are displayed in Figure 6.2 (on the next page). The first image in the original sequence has been used to produce the initial set of reference windows. These images are identical to the results produced by a software model for the same input image sequence.

Figure 6.2. Results of 512×512 tracking system. **(a)** Image orginally captured by the left camera of a stereo pair. The original image has been used as the first and third images of the sequence. The first image has been used to produce the initial set of reference windows. The third image has been processed to generate the above image. The upper-left corner of the target image is (248, 248).

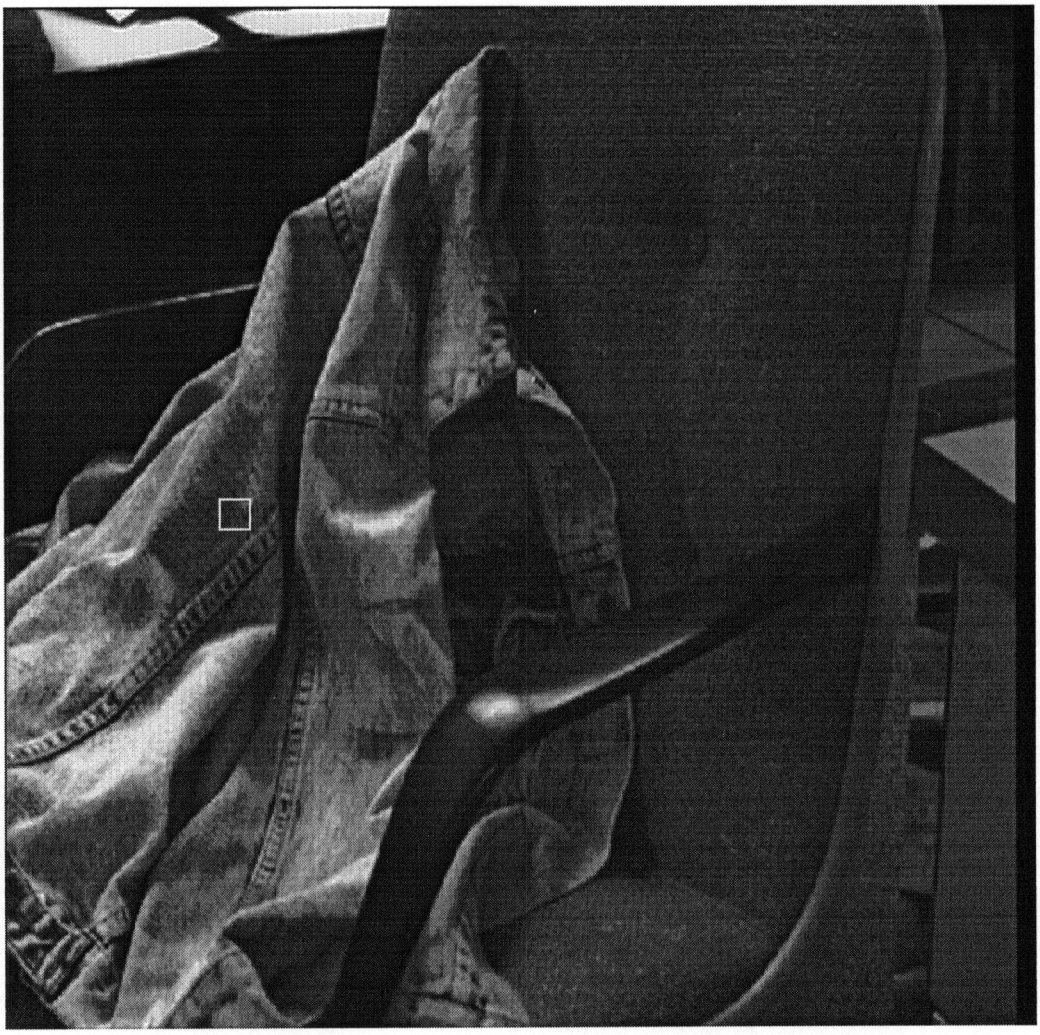


Figure 6.2. (continued) **(b)** Image orginally captured by the right camera of a stereo pair. The original image has been used as the second and fourth images of the sequence. The second image has been processed to produce the above image. The fourth image has been discarded. The upper-left corner of the target image is (245, 106). A comparison between images (a) and (b) indicates a lot of translation in the horizontal direction but very little in the vertical direction.

Table 6.2.

Maximum operating clock frequencies of the eight chips of the 512×512 tracking system.

Chip X1	Chip X2	Chip X3	Chip X4	Chip X5	Chip X6	Chip X7	Chip X8
12.0 MHz	10.2 MHz	10.0 MHz	8.6 MHz	8.7 MHz	20.0 MHz	8.9 MHz	8.9 MHz

The maximum operating clock frequency of the tracking system is 8.60 MHz. This speed is above the minimum rate of 6.6 MHz. But the system can not be guaranteed to work with the VTSplash camera which operates at 10 MHz. As explained in the next chapter, slight improvements in the designs of the chips can enable this system to operate at 10 MHz.

CHAPTER 7
FUTURE WORK AND CONCLUSIONS

7.1. Enhancements to the Tracking Systems and Suggested Future Work

Counters, adders and other arithmetic hardware resources have been modeled in behavioral VHDL code in the design of the tracking systems. In certain cases, these resources lie on a critical path. By using hard macros for these resources, the routing of the CLBs in an FPGA can be improved and thereby the operational speed can be increased as long delay lines will be minimized. Hard macros represent structures that are already partitioned, placed and pre-routed on the FPGAs.

There are many other ways to improve the speed of the system. Critical paths in the design can be broken into smaller paths by using buffers and wait states. The Xilinx tool *XACT* is a schematic design editor. The *lca* files produced by the Xilinx tool *ppr* is used by *XACT* to produce a schematic of the routing of the CLBs in the FPGA. By using this tool, small timing problems can be cleared by editing the schematic to obtain a slightly better routing of the CLBs and IOBs of the FPGAs.

Without great difficulty, the tracking system can be scaled to work for pyramids constructed from images of very large sizes. For larger images, the number of chips used for correlation can be increased to reduce the number of processing cycles for this operation. For instance, correlation has been partitioned such that all search blocks beginning on even columns of the search window are processed by one chip while simultaneously, all the search blocks beginning on odd columns are processed by another pyramid. This can be extended to four chips by letting a pair of chips process all search blocks that begin on even rows and another pair of chips process all search blocks that begin on odd rows so that the time taken for performing correlation will be reduced by half.

A motion detection system can be used to produce the target images for the tracking system. This motion detection system can detect motion by constructing a difference image from two successive image frames. Thresholding is applied to this difference image to detect a moving target in the image. This target image is extracted and transmitted to the tracking system as an initial reference window. The combination of these two systems will form a complete automatic surveillance system.

A velocity-estimation system can be developed which can run on top of the tracking system. The vertical and horizontal positions of the target in successive images is sent by the tracking system to the velocity-estimator. The velocity-estimator uses appropriate scaling of these positions and the time difference between these images to find the horizontal and vertical speeds of the target. By implementing mathematical functions, the magnitude and the direction of the tangential velocity can also be evaluated.

7.2. Conclusions

This thesis has described the implementation of two real-time visual tracking systems on Splash II. The tracking systems have a pipelined architecture with a SIMD approach to numerical computations. The systems have been designed for a custom computing machine, demonstrating the abilities of CCMs to handle speed-critical applications. It has also been shown that the algorithm successfully used for these tracking systems on Splash II is incapable of real-time operation when implemented on a Sun SPARC workstation.

The advantages of the Splash II architecture are clearly demonstrated by these tracking systems. The Splash II architecture allows easy implementation of parallel and pipelined structures making it ideal for applications like visual tracking systems. A wide range of applications on Splash II have explored the versatility of the architecture. A real-time Gaussian pyramid and Laplacian pyramid generation system has been implemented on Splash II [10]. The Hough transform has also been implemented on this system [1] as well as floating point arithmetic [13]. All this work has proven that custom computing

machines are capable of handling high-speed tasks and therefore they can play in important role in the world of computer vision.

Bibliography

[1] A. L. Abbott, P. M. Athanas, L. Chen and R. Elliott, "Finding Lines and Building Pyramids with Splash II," *Proceedings of the IEEE Workshop on FPGAs for Custom Computing Machines*, Napa, CA, April, 1994.

[2] J. R. Armstrong and F. G. Gray, "Structured Logic Design with VHDL," Prentice Hall, Englewood Cliffs, NJ, 1993.

[3] J. M. Arnold, D. A. Buell and E. G. Davis, "Splash II," *Proceedings: ACM Symposium on Parallel Algorithms and Architectures*, pp. 316-322, 1992.

[4] P. M. Athanas and A. L. Abbott, "Real-Time Image Processing on a Custom Computing Platform," *IEEE Computer*, Vol. 28, No. 2, February, 1995, pp.16-24.

[5] D. A. Buell, J. M. Arnold, W. J. Kleinfelder, "Splash II," published by the IEEE Computer Society Press, 1996.

[6] P. J. Burt, "Multiresolution Techniques for Image Representation, Analysis and Smart Transmission", *Proceedings of the SPIE*, 1989, v. 1199, pt. 1., pp. 2-15.

[7] P. J. Burt and E. H. Adelson, "The Laplacian Pyramid as a Compact Image Code", *IEEE Transactions on Communications*, vol. COM-31, pp. 532-540, 1993.

[8] P. J. Burt and G. S. van der Waal, "An Architecture for Multiresolution, Focal, Image Analysis," *Proceedings of the 10th International Conference on Pattern Recognition*, 1990.

[9] P. J. Burt and G. S. van der Waal, "VLSI Pyramid Chip for Multiresolution Image Analysis," *International Journal of Computer Vision*, 1992, vol. 8, no. 3, pp. 177-189.

[10] L. Chen, "Fast Generation of Gaussian and Laplacian Image Pyramids Using an FPGA-based Custom Computing Platform," Master's Thesis in Electrical Engineering, Virginia Tech, September, 1994.

[11] P. Cremonesi, M. Pugarti, N. Scarabottolo, "Motion Detection on Distributed-Memory Machines: A Case Study," *Proceedings of the First IEEE Conference on Algorithms and Architectures for Parallel Processing (ICA^{3}PP)*, 1995.

[12] M. Pyeron et al., "A Robust Hierarchical Probabilistic Framework for Visual Target Tracking," *Proceedings of the IEEE International Conference on Acoustics, Speech and Signal Processing*, Vol. 4, !991, pp. 2457-2460.

[13] N. D. Shirazi, "Implementation of a 2-D Fast Fourier Transform on an FPGA-based Custom Computing Platform," Master's Thesis in Electrical Engineering, Virginia Tech, May 1995.

[14] Z. Zhang and B. Yuan, "Multiresolution Target Detection and Tracking through a Parallel Coarse-to-Fine Search Approach," *Proceedings of the IEEE Conference on Computer, Communication, Control and Power Engineering (TENCON)*, Part 2, 1993.

[15] "IEEE Standard VHDL Language Reference Manual," IEEE, New York, 1988.

[16] Synopsys Inc., "VHDL System Simulator - Command Reference Manual, Version 3.0," December 1992.

[17] Synopsys Inc., "Design Compiler Reference Manual, Version 3.0," December, 1992.

[18] Xilinx Inc., "The Programmable Logic Data Book," San Jose, CA, 1995.

[19] Xilinx Inc. "The XACT 4000, Design Implementation", San Jose, CA, 1991.

[20] Taxi Images, Internet Web Site, "http://www.images.com/data/taxi".

APPENDIX A

Xilinx Processing Part Entity

```
library IEEE,SPLASH2;
use IEEE.std_logic_1164.all;
use SPLASH2.TYPES.all;
use SPLASH2.SPLASH2.all;
use SPLASH2.COMPONENTS.all;
use SPLASH2.ARITHMETIC.all;
use SPLASH2.HMacros.all;

-- Begin Entity Comment
--------------------------------------------------------------------------
-- 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;
-- End Entity Comment
```

APPENDIX B

CODE LISTING FOR 256×256 TRACKING SYSTEM

1. Code for Chip X2 of Part 2 of the System

2. Code for Chip X4 of Part 3 of the System

3. Code for Chip X6 of Part 4 of the System

Chip X2: Stores pyramids

```
architecture Store_Even_Pyr of Xilinx_Processing_Part is

type  STATE_TYPE is (invalid, store, get_con_inf, idle0, idle1, idle2, idle3, idle4, send_data, send_srch,
idle5, srch_st);
 SIGNAL Right                    : Bit_Vector(DATAPATH_WIDTH-1 downto 0);
 SIGNAL Left                     : Bit_Vector(DATAPATH_WIDTH-1 downto 0);
 SIGNAL XBAR_in                  : Bit_Vector(DATAPATH_WIDTH-1 downto 0);
 SIGNAL maddr, newmemaddr        : UNSIGNED(17 downto 0);
 SIGNAL incmemaddr, offset       : UNSIGNED(17 downto 0);
 SIGNAL R, C1, C2, Reg           : UNSIGNED(15 downto 0);
 SIGNAL t1, Trg                  : UNSIGNED(15 downto 0);
 SIGNAL trg128                   : UNSIGNED(13 downto 0);
 SIGNAL trg64                    : UNSIGNED(11 downto 0);
 SIGNAL trg32                    : UNSIGNED(9 downto 0);
 SIGNAL mdata_out, dmout         : Bit_Vector(MEM_WIDTH-1 downto 0);
 SIGNAL mdata_in, dmin           : Bit_Vector(MEM_WIDTH-1 downto 0);
 SIGNAL Rout                     : Bit_Vector(7 downto 0);
 SIGNAL cnt, inccnt              : UNSIGNED(9 downto 0);
 SIGNAL skip                     : UNSIGNED(7 downto 0);
 SIGNAL RValid                   : UNSIGNED(3 downto 0);
 SIGNAL Rtemp                    : Bit_Vector(11 downto 0);
 SIGNAL lev                      : Bit_Vector(1 downto 0);
 SIGNAL mwrite, mread            : BIT;
 SIGNAL get_pyr, snd_srch, i0    : BIT;
 SIGNAL cur_st                   : STATE_TYPE;
constant one_18          : UNSIGNED(17 downto 0) := "000000000000000001";
constant one_10          : UNSIGNED(9 downto 0) := "0000000001";
constant one_8           : UNSIGNED(7 downto 0) := "00000001";

begin
        XP_Left <= Tristate(XP_Left);
        XP_Right <= Tristate(XP_Right);
        XP_XBAR <= Tristate(XP_XBAR);
        XP_Xbar_EN_L <= "00000";
        Pad_InOut  (XP_Mem_D, dmout, dmin, not(mwrite));
        Right <= RValid&Rout&"000000000000"&Rtemp;
process

        constant pt256    : UNSIGNED(17 downto 0) := "000000000000000000";
        constant pt128    : UNSIGNED(17 downto 0) := "010000000000000000";
        constant pt64     : UNSIGNED(17 downto 0) := "010100000000000000";
        constant pt32     : UNSIGNED(17 downto 0) := "010101000000000000";
        constant set_18   : UNSIGNED(17 downto 0) := "111111111111111111";
        constant pyrend   : UNSIGNED(17 downto 0) := "010101001111111110";
        constant zero_16  : UNSIGNED(15 downto 0) := "0000000000000000";

        constant lev32    : BIT_VECTOR(3 downto 0):= "1000";
        constant lev64    : BIT_VECTOR(3 downto 0):= "1001";
```

```
    constant lev128   : BIT_VECTOR(3 downto 0):= "1010";
    constant lev256   : BIT_VECTOR(3 downto 0):= "1011";

    constant zero_10 : UNSIGNED(9 downto 0) := "0000000000";
    constant thirone  : UNSIGNED(4 downto 0) := "11111";
    constant fift     : UNSIGNED(3 downto 0) := "1111";

    variable LValid   : BIT_VECTOR(3 downto 0);
    variable XValid   : BIT_VECTOR(3 downto 0);

begin

    wait until XP_Clk'Event and XP_Clk = '1';

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

    Pad_Output (XP_Mem_A, maddr);
    Pad_Output (XP_Mem_RD_L, mread);
    Pad_Output (XP_Mem_WR_L, mwrite);

    LValid := Left(11 downto 8);
    XValid := XBAR_In(13 downto 10);

    Rtemp <= Left(11 downto 0);
    Rout <= Bit_Vector'("00000000");
    RValid <= Bit_Vector'("0000");

    dmout <= mdata_out;
    mdata_in <= dmin;

    Reg <= R;

    case cur_st is

     WHEN invalid =>
            mwrite <= '1';
            mread <= '1';
            maddr <= set_18;
            snd_srch <= '0';
            lev <= B"00";
            cur_st <= store;

     WHEN store =>
            mwrite <= '0';
            mread <= '1';
            if (LValid = BIT_VECTOR'("1000")) then
              maddr <= incmemaddr;
              mdata_out <= "00000000"&Left(7 downto 0);
            end if;
```

```
        if (maddr = pyrend) then
          get_pyr <= '0';
          cur_st <= get_con_inf;
        end if;

WHEN get_con_inf =>
        mread <= '1';
        mwrite <= '1';

    if (XValid = lev32) then
          skip <= UNSIGNED'("00000001");
          C1 <= zero_16;            --setting it to all zeros
          C2 <= zero_16;            --setting it to all zeros
          maddr <= pt32;
          cur_st <= idle1;
        end if;

        if (XValid = lev64) then
          skip <= UNSIGNED'("00100001");
          C1 <= zero_16;            --setting C1 to all zeros
          C2 <= "0000"&XBar_In(9 downto 5)&'0'&XBar_In(4 downto 0)&'0';
          trg32 <= XBar_In(9 downto 0);
          maddr <= pt64;
          cur_st <= idle1;
        end if;

        if (XValid = lev128) then
          skip <= UNSIGNED'("01100001");
          C1 <= "00"&Reg(11 downto 6)&'0'&Reg(5 downto 0)&'0';
          C2 <= "000"&XBar_In(9 downto 5)&"00"&XBar_In(4 downto 0)&'0';
          t1 <= "00000"&XBar_In(9 downto 5)&'0'&XBar_In(4 downto 0);
          maddr <= pt128;
          cur_st <= idle0;
        end if;

        if (XValid = lev256) then
          skip <= UNSIGNED'("11100001");
          C1 <= Reg(13 downto 7)&'0'& Reg(6 downto 0) & '0';
          C2 <= "00"&XBar_In(9 downto 5)&"000"&XBar_In(4 downto 0)&'0';
          t1 <= "0000"&XBar_In(9 downto 5)&"00"&XBar_In(4 downto 0);
          maddr <= pt256;
          cur_st <= idle0;
        end if;

WHEN idle0 =>
        if (XValid = lev128) then
          trg64 <= Trg(11 downto 0);
          cur_st <= idle1;
        end if;
```

```
          if (XValid = lev256) then
            trg128 <= Trg(13 downto 0);
            i0 <= '1';
            cur_st <= idle1;
          end if;

          if (XValid = lev32) then
            t1 <= "000"&XBar_In(9 downto 5)&"000"&XBar_In(4 downto 0);
            cur_st <= srch_st;
          end if;

WHEN idle1 =>
          offset <= "00"&R;
          cur_st <= idle2;

WHEN idle2 =>
          mwrite <= '1';
          mread <= '0';
          cnt <= zero_10;
          maddr <= newmemaddr;
          cur_st <= idle3;

WHEN idle3 =>
          cnt <= inccnt;
      maddr <= incmemaddr;
          offset <= UNSIGNED'("0000000000"&skip);
          cur_st <= idle4;

WHEN idle4 =>
          cnt <= inccnt;
      maddr <= incmemaddr;
          if (snd_srch = '1') then
           cur_st <= send_srch;
          else
           cur_st <= send_data;
          end if;

WHEN send_data =>
          cnt <= inccnt;
          RValid <= BIT_VECTOR'("1000");
          Rout <= mdata_in(7 downto 0);
          if (cnt(4 downto 0) = thirone) then
            maddr <= newmemaddr;
          else
            maddr <= incmemaddr;
          end if;
          if (cnt = one_10) then
           cur_st <= idle5;
          end if;

WHEN send_srch =>
```

```
            cnt <= inccnt;
            RValid <= BIT_VECTOR'("1111");
            Rout <= mdata_in(7 downto 0);
            if (cnt(3 downto 0) = fift) then
              maddr <= newmemaddr;
            else
              maddr <= incmemaddr;
            end if;
            if (cnt(7 downto 0) = one_8) then
             cur_st <= idle5;
            end if;

WHEN idle5 =>
            offset <= pt256;   --setting it to all zeros
            mwrite <= '1';
            mread <= '1';
            if (i0 = '1') then
             i0 <= '0';
             cur_st <= idle0;
            elsif (snd_srch = '1') then
             cur_st <= srch_st;
            else
             cur_st <= get_con_inf;
            end if;

WHEN srch_st =>
            snd_srch <= '1';
            if (lev = Bit_Vector'("00")) then
             offset <= "00"&Trg;
             maddr <= pt256;
  skip <= UNSIGNED'("11110001");
             lev <= "01";
            elsif (lev = Bit_Vector'("01")) then
             offset <= "0000"&trg128;
             maddr <= pt128;
 skip <= UNSIGNED'("01110001");
             lev <= "11";
            elsif (lev = Bit_Vector'("11")) then
             offset <= "000000"&trg64;
             maddr <= pt64;
  skip <= UNSIGNED'("00110001");
             lev <= "10";
            else
             offset <= "00000000"&trg32;
             maddr <= pt32;
  skip <= UNSIGNED'("00010001");
             lev <= "00";
             get_pyr <= '1';
            end if;
            if (get_pyr = '1') then
             RValid <= "1110";
```

```
                cur_st <= invalid;
               else
                cur_st <= idle2;
        end if;

    end case;

 end process;

        incmemaddr <= maddr + one_18;
        newmemaddr <= maddr + offset;
        R <= C1 + C2;
        Trg <= t1 + Reg;
        inccnt <= cnt + one_10;

end Store_Even_Pyr;
```

Chip X4: Correlation

```
architecture Correlate_Even of Xilinx_Processing_Part is

 type   STATE_TYPE is (invalid, srch_st, get_data, idle0, idle1, idle2, stsrch, sttrg, idle, idle3, idle4, idle5,
idle6, set_lev);
  SIGNAL Right                  : Bit_Vector(DATAPATH_WIDTH-1 downto 0);
  SIGNAL Left                   : Bit_Vector(DATAPATH_WIDTH-1 downto 0);
  SIGNAL maddr                  : Bit_Vector(17 downto 0);
  SIGNAL mdata_out, dmout       : Bit_Vector(MEM_WIDTH-1 downto 0);
  SIGNAL mdata_in, dmin         : Bit_Vector(MEM_WIDTH-1 downto 0);
  SIGNAL Awl, Bwl, Cwl          : UNSIGNED(9 downto 0);
  SIGNAL tA1, tB1, tC1          : UNSIGNED(8 downto 0);
  SIGNAL tA16, tB16, tC16       : UNSIGNED(8 downto 0);
  SIGNAL trgpos                 : Bit_Vector(9 downto 0);
  SIGNAL temp, reg1, sreg       : Bit_Vector(7 downto 0);
  SIGNAL sA1, sB1, sC1          : UNSIGNED(6 downto 0);
  SIGNAL mwrite, mread, RS      : Bit ;
  SIGNAL sw, pix, odd           : Bit;
  SIGNAL addv, add1             : Bit ;
  SIGNAL cur_st                 : STATE_TYPE;
  SIGNAL cnt                    : UNSIGNED(3 downto 0);
  SIGNAL trg_pt                 : UNSIGNED(8 downto 0);
  SIGNAL srch_pt                : UNSIGNED(6 downto 0);
  SIGNAL diff1, diff2           : UNSIGNED(7 downto 0);
  SIGNAL diff                   : UNSIGNED(8 downto 0);
  SIGNAL min, sum               : UNSIGNED(15 downto 0);
  SIGNAL win1                   : UNSIGNED(9 downto 0);
  SIGNAL lev                    : UNSIGNED(2 downto 0);
  SIGNAL RValid                 : Bit_Vector(3 downto 0);
  SIGNAL Rout                   : Bit_Vector(25 downto 0);
  SIGNAL Rtemp                  : Bit_Vector(5 downto 0);
  SIGNAL srch                   : Bit_Vector(10 downto 0);
-- SIGNAL cs1, cs2, ct1, ct2     : UNSIGNED(7 downto 0);
begin
        Pad_InOut  (XP_Mem_D, dmout, dmin, not(mwrite));

  process
        constant trg            : Bit_Vector(8 downto 0) :="000000100";
        constant two            : UNSIGNED(9 downto 0) :="0000000010";
        constant zero           : UNSIGNED(8 downto 0) :="000000000";
        constant sixt           : UNSIGNED(9 downto 0) :="0000010000";
        constant max            : UNSIGNED(15 downto 0) :="1111111111111111";
        constant srczer         : UNSIGNED(6 downto 0) := "0000000";
        constant szero          : UNSIGNED(15 downto 0) := "0000000000000000";
        constant srchend        : UNSIGNED(6 downto 0) := "1111111";
        constant winend         : UNSIGNED(9 downto 0) := "1000100000";
        constant endlin         : UNSIGNED(4 downto 0) := "10000";
        constant frt            : UNSIGNED(3 downto 0) := "1110";
        constant trgend         : UNSIGNED(8 downto 0) := "111111111";
```

```
        variable LValid, LSrch      : Bit_Vector(3 downto 0);
        variable st1, ss1           : UNSIGNED(7 downto 0);
        variable st2, ss2           : UNSIGNED(7 downto 0);
        variable temp1, temp2       : UNSIGNED(8 downto 0);
        variable temp3              : UNSIGNED(15 downto 0);

begin
        wait until XP_Clk'Event and XP_Clk = '1';
        dmout <= mdata_out;
        mdata_in <= dmin;
        Pad_Input (XP_Left, Left);
        Pad_Output (XP_Right, Right);

        Pad_Output (XP_Mem_A, maddr);
        Pad_Output (XP_Mem_RD_L, mread);
        Pad_Output (XP_Mem_WR_L, mwrite);


        LValid := Left(35 downto 32);
        LSrch := Left(11 downto 8);

        Rtemp <= Left(35 downto 30);
        RValid <= Left(29 downto 26);
        Rout <= Left(25 downto 0);

        if (sw = '0') then
                maddr <= srch&srch_pt;
        else
                maddr <= trg&trg_pt;
        end if;

        case cur_st is
            WHEN invalid =>
                mwrite <= '1';
                mread <= '1';
                lev <= "010";
                sw <= '0';
                srch_pt <= srczer;
                srch <= "00000000011";
                Bwl <= two;
                cur_st <= srch_st;

            WHEN srch_st =>
                mwrite <= '0';
                mread <= '1';
                if (LSrch = BIT_VECTOR'("1111")) then
                  if (pix = '0') then
                    reg1 <= Left(7 downto 0);
                    pix <= '1';
                  else
```

```
          mdata_out <= Left(7 downto 0)&reg1;
         srch_pt <= sC1;
          pix <= '0';
          if (srch_pt = srchend) then
           srch <= "00000000"&lev;
           lev <= lev - UNSIGNED'("001");
         end if;
        end if;
       end if;
      if (Lsrch = Bit_Vector'("1110")) then
       lev <= "000";
       trg_pt <= zero;
       sw <= '1';
       cur_st <= get_data;
      end if;

WHEN get_data =>
      srch <= "00000000"&lev;
      if (LValid = BIT_VECTOR'("1000")) then
       if (pix = '0') then
        reg1 <= Left(31 downto 24);
        pix <= '1';
       else
        mwrite <= '0';
        mread <= '1';
        mdata_out <= Left(31 downto 24)&reg1;
        trg_pt <= tC1;
        if (trg_pt = trgend) then
         trg_pt <= zero;
         winl <= zero&'0';
         trgpos <= zero&'0';
         min <= max;
         cur_st <= idle0;
        end if;
        pix <= '0';
       end if;
      end if;

WHEN idle0 =>
      mwrite <= '1';
      mread <= '0';
      sum <= szero;
      srch_pt <= srczer;
      sw <= '0';
      cur_st <= idle1;

WHEN idle1 =>
      cnt <= UNSIGNED'("0010");
      trg_pt <= tC1;
      sw <= '1';
      cur_st <= idle2;
```

```
  WHEN idle2 =>
        srch_pt <= sC1;
        sw <= '0';
        add1 <= '0';
        cur_st <= stsrch;

  WHEN stsrch =>
        st1 := mdata_in(7 downto 0);
        st2 := mdata_in(15 downto 8);
        addv <= add1;
        if (cnt = frt) then
         trg_pt <= tC16;
        else
trg_pt <= tC1;
        end if;
        cnt <= cnt + UNSIGNED'("10");
        sw <= '1';
        cur_st <= sttrg;

  WHEN sttrg =>
        ss1 := mdata_in(7 downto 0);
        ss2 := mdata_in(15 downto 8);
        srch_pt <= sC1;
        addv <= '0';
        add1 <= '1';
        sw <= '0';
        if (srch_pt = UNSIGNED'("0000000")) then
         cur_st <= idle;
        else
         cur_st <= stsrch;
        end if;

  WHEN idle =>
        addv <= '1';
        cur_st <= idle3;

  WHEN idle3 =>
        addv <= '0';
        cur_st <= idle4;

  WHEN idle4 =>
        mread <= '1';
        mwrite <= '1';
        if (sum < min) then
          min <= sum;
          trgpos <= winl;
        end if;
        srch_pt <= srczer;
        winl <= Cwl;
        cur_st <= idle5;
```

```
                WHEN idle5 =>
                    if (winl = winend) then
                      RValid <= "1111";
                      Rout <= min&trgpos;
                      lev <= lev + UNSIGNED'("001");
                      cur_st <= set_lev;
                    else
                      cur_st <= idle6;
                    end if;

                WHEN idle6 =>
                    trg_pt <= winl(9 downto 1);
                    if (winl(4 downto 0) = endlin) then
                      Bwl <= sixt;
                    else
                      Bwl <= two;
                    end if;
                    sw <= '1';
                    cur_st <= idle0;

                WHEN set_lev =>
                    if (lev = UNSIGNED'("100")) then
                      sw <= '0';
                      cur_st <= invalid;
                    else
                      sw <= '1';
                      cur_st <= get_data;
                    end if;
                  trg_pt <= zero;

            end case;

--cs1 <= ss1;
--cs2 <= ss2;
--ct1 <= st1;
--ct2 <= st2;
            if (ss1 > st1) then
              diff1 <= ss1 - st1;
            else
              diff1 <= st1 - ss1;
            end if;
            if (ss2 > st2) then
              diff2 <= ss2 - st2;
            else
              diff2 <= st2 - ss2;
            end if;

            temp1 := UNSIGNED'('0'&diff1);
            temp2 := UNSIGNED'('0'&diff2);
            diff <= temp1 + temp2;
```

```
        temp3 := UNSIGNED'("0000000"&diff);
        if (addv = '1') then
          sum <= sum + temp3;
        end if;

  end process;

        XP_XBAR <= Tristate(XP_XBAR);
        Right <= Rtemp&RValid&Rout;
        tA16 <= "000001001";
        tA1 <= "000000001";
        sA1 <= "0000001";
        sB1 <= srch_pt;
        tB1 <= trg_pt;
        tB16 <= trg_pt;
        Awl <= winl;
        tC1 <= tA1 + tB1;
        tC16 <= tA16 + tB16;
        Cwl <= Awl + Bwl;
        sC1 <= sA1 + sB1;

end Correlate_Even;
```

Chip X6: Displays output image

```
architecture Display_Trg_Even of Xilinx_Processing_Part is
  type   STATE_TYPE is (invalid, store, get_pos, idle1, idle2, idle3, out_pix, display);

  SIGNAL Right                    : Bit_Vector(DATAPATH_WIDTH-1 downto 0);
  SIGNAL Left                     : Bit_Vector(DATAPATH_WIDTH-1 downto 0);
  SIGNAL XBar_In                  : Bit_Vector(DATAPATH_WIDTH-1 downto 0);
  SIGNAL Rout                     : Bit_Vector(7 downto 0);
  SIGNAL maddr, incmaddr          : UNSIGNED(17 downto 0);
  SIGNAL mdata_out, dmout         : Bit_Vector(MEM_WIDTH-1 downto 0);
  SIGNAL mdata_in, dmin           : Bit_Vector(MEM_WIDTH-1 downto 0);
  SIGNAL mwrite, mread            : Bit  ;
  SIGNAL RValid                   : Bit_Vector(3 downto 0);
  SIGNAL Rtemp                    : Bit_Vector(13 downto 0);
  SIGNAL whitecnt, incwhcnt       : UNSIGNED(3 downto 0);
  SIGNAL Lx, Ly, Rx, Ry           : UNSIGNED(7 downto 0);
  SIGNAL BRx, BRy, BLx            : UNSIGNED(7 downto 0);
  SIGNAL inLx, incLx              : UNSIGNED(7 downto 0);
  SIGNAL cnt, inccnt              : UNSIGNED(15 downto 0);
  SIGNAL cur_st                   : STATE_TYPE;
  SIGNAL R, C1, C2, Reg           : UNSIGNED(15 downto 0);
 constant one_18                  : UNSIGNED(17 downto 0) := "000000000000000001";
 constant one_16                  : UNSIGNED(15 downto 0) := "0000000000000001";
 constant one_8                   : UNSIGNED(7 downto 0) := "00000001";
 constant one_4                   : UNSIGNED(3 downto 0) := "0001";
 constant fift                    : UNSIGNED(7 downto 0) := "00001111";

begin

 Pad_InOut  (XP_Mem_D, dmout, dmin, not(mwrite));
 Right <= RValid&"0000000000"&Rtemp&Rout;
 XP_XBar_EN_L <= "00000";

 process
   variable XValid, LValid : Bit_Vector(3 downto 0);
   constant frend          : UNSIGNED(15 downto 0) :="1111111111111111";
   constant zero_16        : UNSIGNED(15 downto 0) :="0000000000000000";
   constant zero_8         : UNSIGNED( 7 downto 0) :="00000000";
   constant set_18         : UNSIGNED(17 downto 0) :="11111111111111111";
   constant white          : Bit_Vector(7 downto 0) := "11111111";
 begin
        wait until XP_Clk'Event and XP_Clk = '1';

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

        Pad_Output (XP_Mem_RD_L, mread);
        Pad_Output (XP_Mem_WR_L, mwrite);
```

```
Pad_Output (XP_Mem_A, maddr);

RValid <= Bit_Vector'("0000");
Rtemp <= Left(21 downto 8);
Reg <= R;
Rx <= BRx;
Ry <= BRy;
incLx <= BLx;

XValid := XBAR_in(11 downto 8);
LValid := Left(21 downto 18);

dmout <= mdata_out;
mdata_in <= dmin;

case cur_st is

 WHEN invalid =>
   mwrite <= '1';
   mread <= '1';
   cnt <= zero_16;
   maddr <= set_18;
   cur_st <= store;

 WHEN store =>
   if (XValid = BIT_VECTOR'("1000")) then
         cnt <= inccnt;
         mwrite <= '0';
         mread <= '1';
         mdata_out <= "00000000"&XBar_In(7 downto 0);
         maddr <= incmaddr;
   end if;
   if (cnt = frend) then
         cur_st <= get_pos;
   end if;

 WHEN get_pos =>
   if (LValid = Bit_Vector'("1000")) then
         C1 <= "000000"&Left(17 downto 8);
         C2 <= zero_16;
   end if;

   if (LValid = Bit_Vector'("1001")) then
         C1 <= "00000"&Left(17 downto 13)&'0'&Left(12 downto 8);
         C2 <= "0000"&Reg(9 downto 5)&'0'&Reg(4 downto 0)&'0';
   end if;

   if (LValid = Bit_Vector'("1010")) then
         C1 <= "0000"&Left(17 downto 13)&"00"&Left(12 downto 8);
         C2 <= "00"&Reg(11 downto 6)&'0'&Reg(5 downto 0)&'0';
   end if;
```

```
  if (LValid = Bit_Vector'("1011")) then
        C1 <= "000"&Left(17 downto 13)&"000"&Left(12 downto 8);
        C2 <=  Reg(13 downto 7)&'0'&Reg(6 downto 0)&'0';
        cur_st <= idle1;
  end if;

WHEN idle1 =>
  mwrite <= '1';
  mread <= '0';
  maddr <= "00"&zero_16;
  Lx <= R(15 downto 8);
  Ly <= R(7 downto 0);
  cur_st <= idle2;

WHEN idle2 =>
  inLx <= Lx;
  maddr <= incmaddr;
  cur_st <= idle3;

WHEN idle3 =>
  inLx <= BLx;
  whitecnt <= "0000";
  maddr <= incmaddr;
  cnt <= zero_16;
  cur_st <= out_pix;

WHEN out_pix =>
  RValid <= BIT_VECTOR'("1000");
  Rout <= mdata_in(7 downto 0);
  maddr <= incmaddr;
  cnt <= inccnt;

  if (cnt = Rx&Ly) then
        Rout <= white;
        cur_st <= display;
  end if;
  if (cnt = inLx&Ly) then
        Rout <= white;
  end if;
  if (cnt = inLx&Ry) then
        inLx <= incLx;
        Rout <= white;
  end if;
  if (cnt = Lx&Ly) then
        Rout <= white;
        cur_st <= display;
  end if;
  if (cnt = frend) then
        cur_st <= invalid;
  end if;
```

```
        WHEN display =>
            RValid <= BIT_VECTOR'("1000");
            maddr <= incmaddr;
            Rout <= white;
            cnt <= inccnt;
            whitecnt <= incwhcnt;
            if (whitecnt = 14) then
                  whitecnt <= "0000";
                  cur_st <= out_pix;
            end if;
           if (cnt = frend) then
                  cur_st <= invalid;
           end if;

    end case;

     if (inLx > Rx) then
                  inLx <= zero_8;
    end if;

 end process;

 R <= C1 + C2;
 BRx <= Lx + fift;
 BRy <= Ly + fift;
 incmaddr <= maddr + one_18;
 inccnt <= cnt + one_16;
 incwhcnt <= whitecnt + one_4;
 BLx <= inLx + one_8;

end Display_Trg_Even;
```

Vita

Bharadwaj Pudipeddi was born on the first of January, 1973 in Vizag, India. He graduated with a Bachelor of Engineering degree in Electrical and Electronics Engineering from the College of Engineering, Andhra University, Vizag, India in June 1994. He enrolled for graduate study at Virginia Tech in August, 1994 and received a Master of Science degree in June, 1996. He is currently employed with Intel Corp., Santa Clara, California as a design engineer.