Fast Generation of Gaussian and Laplacian Image Pyramids Using an FPGA–based Custom Computing Platform

by

Luna Chen

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:

[illegible]

Dr. A. Lynn Abbott, Chairman

[illegible]

Dr. Peter M. Athanas

[illegible]

Dr. Walling R. Cyre

September, 1994

Blacksburg, Virginia

C.2

LD
5655
V855
1994
C5465
c.2

Fast Generation of Gaussian and Laplacian Image Pyramids Using an FPGA–based Custom Computing Platform

by

Luna Chen

Dr. A. Lynn Abbott, Chairman

Electrical Engineering

(ABSTRACT)

This thesis describes the implementation of a system that can generate two types of image pyramids: the Gaussian pyramid and the Laplacian pyramid. These have been developed using the SPLASH II attached processor, which is a reconfigurable platform based on Field Programmable Gate Arrays (FPGA). The design was first modeled in VHDL, and was then simulated and synthesized to a gate list using a SPLASH II simulator and the Synopsys synthesis tool. The gate list was then mapped onto Xilinx XC4010 FPGA architectures.

Three complete designs have been developed to generate pyramids on SPLASH II: two for generating the Gaussian pyramid, and one for generating the Laplacian pyramid. One of the designs produces a complete image pyramid within one image frame time of 33 ms. The other two designs produce complete pyramids within two frame times. All three designs can be used as pipeline stages within a larger image processing system.

Acknowledgements:

I wish to express my deep appreciation for the guidance and support given by my advisor, Dr. A. Lynn Abbott. I would also like to thank other members of the committee—Dr. Peter M. Athanas and Dr. Walling R. Cyre—for their help and constructive criticism during the course of this project.

Also, I would like to thank all members in the VTsplash team for their assistance and making the project so much fun.

Finally, my special appreciation and love go to my father and mother for their understanding and love during the period of graduate work.

Table of Contents

List of Figures

1. Introduction

1.1. High performance pyramid generation

General–purpose workstations can provide acceptable performance for many image processing tasks when speed is not an essential factor. However, many applications such as visual tracking and robotic control require fast image processing. For such applications, it is difficult to achieve the needed computation rates because of the large amount of data associated with images. Because real–time image analysis involves massive computations and requires high input/output (I/O) capacity, most conventional uniprocessor and parallel computer architectures are not well suited for fast image processing.

The traditional approach to this problem has been to develop application–specific hardware. Such hardware is usually able to accomplish the desired image analysis application efficiently, and can provide relatively satisfactory performance. The drawbacks are lengthy design cycles and high redesign costs when an error is detected in the original design, or when a new algorithm is developed. An additional drawback of application–specific hardware is that several components may be needed to accomplish a single, complex image analysis task. Special care must be taken to provide appropriate data flow between these components and with external interfacing devices.

An attractive alternative to these traditional approaches is to use reconfigurable FPGA (Field Programmable Gate Array) based platforms that can be tailored to different user applications without sacrificing performance. The SPLASH II attached processor, developed by the Supercomputing Research Center (SRC) in Bowie, Maryland, is such a system [1]. It was designed as a general–purpose system, and can implement many image processing tasks with high performance. A brief overview of SPLASH II system is given in the next chapter.

This thesis describes the use of the SPLASH II attached processor for *image pyramid* generation. Image pyramid generation is the process of transforming a single video image into a hierarchy of filtered images [2]. This set of filtered images contains information at different levels of spatial resolution of the image. Multiresolution image processing techniques have been recognized as important components of advanced image analysis systems. They are well suited for many image analysis tasks, including image feature encoding, stereo matching, motion tracking, and image compression [3, 4, 5].

Several image pyramid types have been developed over the past decade. Two of the most popular ones are the Gaussian and Laplacian pyramids. The Gaussian pyramid is a set of smoothed images, in which each pixel is computed as a local weighted average, using a unimodal Gaussian–like weighting function centered on the pixel itself. The Laplacian pyramid consists of a set of band–passed image subsets, each of which is formed as the differences between successive Gaussian levels, so that each Laplacian pyramid level represents intensity edges at a particular resolution. The detailed data structures of these two types of pyramid and example applications are given in Chapter 3.

1.2. The VTSplash Project

VTSplash is a system that consists of four components: a Sun SPARCstation 2, an image collection system, a SPLASH II attached processor, and an image display system. Figure 1.1 shows the relationship of the components. Image frames are sampled by a video camera at the standard rate of 30 frames per second, and are processed within the SPLASH II system at the same speed. The original or processed images may be displayed on the monitor, or stored on a mass–storage device through the Sun host. The Sun station is also responsible for controlling the application programs that run on SPLASH II.

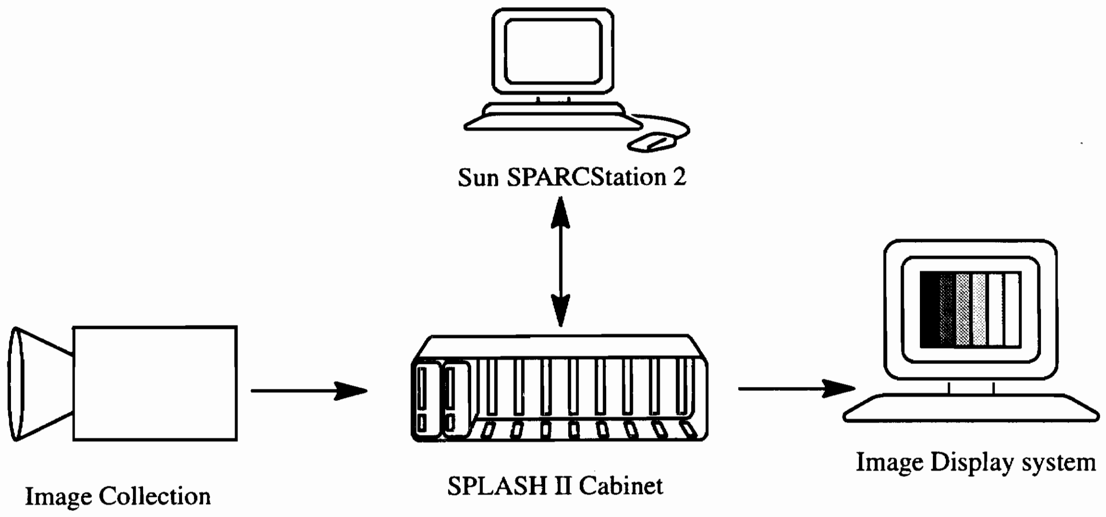


Figure 1.1. VTSplash image processing system. Raw image data from a camera is processed on SPLASH II, and displayed on the monitor. A Sun workstation communicates with SPLASH and configures the system.

This research is a part of VTSplash group project in the Bradley Department of Electrical Engineering at Virginia Tech. The goal of this project is to develop a set of useful image processing operations on the SPLASH II system so that complex image analysis tasks can be carried out at high speed.

In addition to pyramid generation, the image processing operations that have been implemented on SPLASH II include image histogram generation, median filtering [6], region detection, and the Hough transform [7]. These applications represent a diverse set of general image analysis tasks. Since all of these applications are developed for the same

FPGA–based SPLASH board, many of them can be pipelined into a multiboard SPLASH II system.

1.3. Contributions of this research

The contributions of this research can be briefly summarized as follows:

1. Three complete designs have been developed to generate Gaussian and Laplacian pyramid on SPLASH II: two for generating the Gaussian pyramid, and one for generating the Laplacian pyramid.
2. One of the designs produces a complete image pyramid within one frame time. The other two designs produce complete pyramids within two frame times.
3. All three designs are completely synchronous.
4. Each design can be used as a pipeline stage, and can be easily coupled with other components in a large system.
5. The SPLASH II implementations are compared with other pyramid generation architecture.
6. The thesis provides suggestions for improving the SPLASH II system for image processing applications.

1.4. Organization of thesis

This thesis describes the implementation of a system which can generate both Gaussian and Laplacian pyramids in real time. The system has been designed, built, and tested utilizing the SPLASH II attached processor. More details of the SPLASH II system are given in Chapter 2. Chapter 3 presents the Gaussian and Laplacian pyramids, and briefly summarizes previous architectures that can generate these two types of image pyramid. Chapter 3 also describes other useful pyramid–generation architectures and discusses applications of image pyramids. Chapters 4 and 5 describe the implementation of Gaussian and Laplacian pyramid generation on SPLASH II. Chapter 6 describes the development environment of SPLASH II. Chapter 7 presents the results, and proposes future enhancements to SPLASH II. Chapter 8 summarizes the thesis.

2. SPLASH II System Overview

SPLASH II is a general–purpose parallel processor in which the programmer has the freedom to design the architecture of each of the processing elements (PE). As discussed in Chapter 1, a SPLASH II system consists of a Sun SPARCstation and a SPLASH II cabinet. The cabinet contains an interface board and from one to sixteen SPLASH boards [8]. Each SPLASH II board has 17 PEs, with each linearly pipelined and fully connected through a crossbar. The SPLASH II subsystem is connected via a cable to an SBus adapter board in the host. The system architecture is shown in Figure 2.1. The reprogrammable and reconfigurable features of the crossbar support a wide variety of applications.

2.1 SPLASH II array board

Up to 16 SPLASH II array boards are supported within a single system, and each contains 16 Xilinx 4010 FPGA processing elements, referred as X1 to X16, and a seventeenth Xilinx FPGA control element, X0. The sixteen processing elements are arranged in a linear array with each connected to its left and right neighbors via 36–bit wide data bus. Each PE is also connected through a separate 36–bit bus to a 16×16 bi–directional crossbar. The control element X0 is connected to the SIMD (Single Instruction Multiple Data) bus, and may drive the input data into crossbar, and is capable of broadcasting 36–bit data to all 16 processing elements simultaneously.

The central crossbar can store up to 8 dynamically selectable configurations. The configuration in use can be changed as frequently as once per clock cycle. A set of port connections is stored in the crossbar configuration file, where each port can be either an input

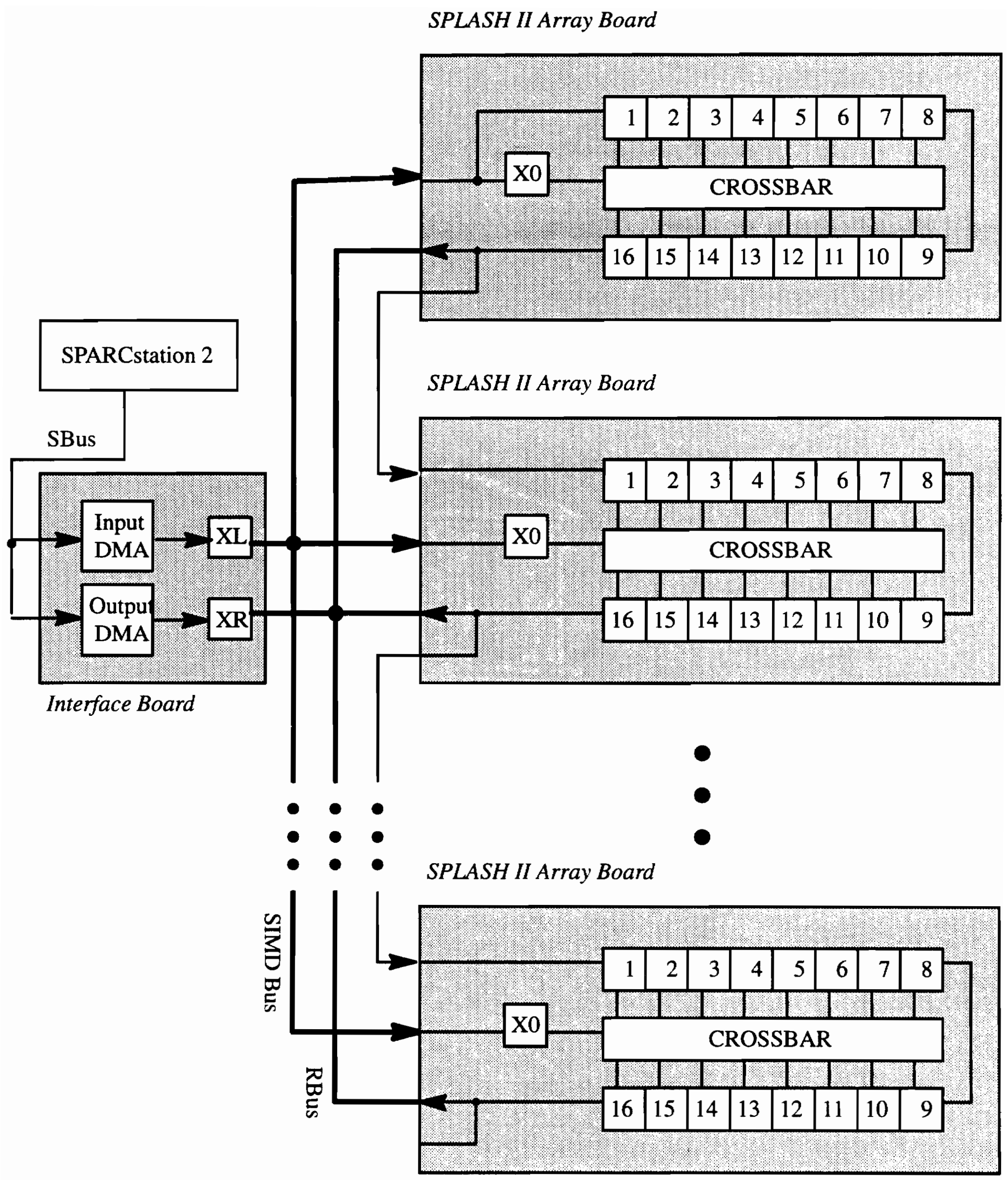


Figure 2.1. The SPLASH II architecture. Up to 16 array boards can be placed in a system. Each board contains 17 Xilinx FPGAs, designated as X0 and 1 to 16.

or an output. In addition to the 36–bit data path, a 5–bit unidirectional control path connects the crossbar to each PE. This allows the application program to control each byte of the 36–bit data bus independently of the others within a configuration.

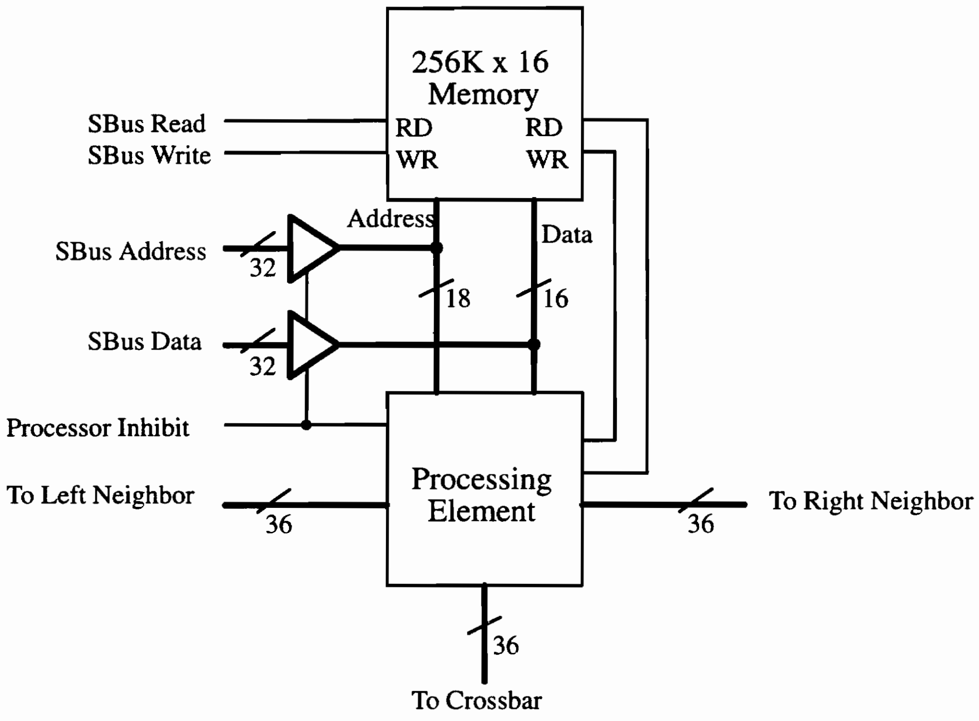


Figure 2.2. SPLASH II Processing Element Organization.

Each of the 16 FPGAs is associated with 512 KB external fast static RAM, which is capable of performing a read or write operation every clock cycle. Figure 2.2 shows the organization of the processor. The data/control path between each PE and its memory is a 16–bit wide data bus, an 18–bit address bus, one read signal, and one write signal. There are two constraints due to the single–ported memory data path. First, the read control signal and the write control signal cannot be asserted simultaneously, and the result is unpredictable if such access occurs. Second, a read operation cannot be followed immediately by a write operation. Therefore, the read control signal must be de–asserted at least one clock cycle before the write signal is asserted. The host SBus is extended to each of the SPLASH array boards, allowing the SUN host to have direct access to the external memory. However, the host and the processing element cannot access the memory simultaneously.

2.2 SPARCstation host

SPLASH II is attached to the host through the SPARC SBus. The extended SBus supports words–aligned reads and writes from the host, and burst mode DMA transfers between the SPLASH system and the host memory. The SPARCstation host is responsible for the control of SPLASH II application programs including reading and writing the configuration state of FPGA elements and the on–board memory, controlling the SPLASH clock, and servicing interrupt or Direct Memory Access requests from SPLASH, etc.

2.3 Interface board

The SPLASH II interface board contains three bi–directional DMA channels for transferring data to and from the host, and two user programmable FPGA devices, known as XL and XR. Each of the DMA channels is decoupled from the SPLASH system by a 256 word FIFO. Both FPGA devices are situated between the FIFOs and the SPLASH array, and responsible for reading and writing data and control signals to and from the SPLASH board. Although the two devices are symmetric, XL is usually programmed to read the input FIFO data, and drive the SIMD data path to the SPLASH array board, and XR writes the data from the last board to the output FIFOs. XL and XR can communicate with each other through a 36–bit wide bi–directional data path.

The system clock is controlled from the host by writing to a set of registers on the interface board. The clock frequency is selectable from 100 Hz to 30 MHz in increments of approximately 50 Hz. The control chips XL and XR also have the ability to start and stop the clock distributed to the SPLASH arrays.

2.4 Xilinx 4010 FPGA

Xilinx FPGAs are user programmable and reprogrammable devices, and are capable of implementing variety of user–defined logic. The Xilinx XC4000 family provides a flexible architecture of Configurable Logic Blocks (CLBs). Each CLB has a pair of flip–flops and three function generators. Figure 2.3 is a simplified block diagram of the XC4000 family CLB cell. Two function generators FG1 and FG2 have four independent inputs (a–d and e–h). The two function generators are implemented as memory look–up tables, each of which is able to implement any 4–input Boolean function at high speed. The propagation delay is independent of the function being implemented. The third function generator FG3 implements any function of the two outputs from FG1 and FG2: O1 and O2 and a third input from outside of the block. The output signals from CLB include signals directly from function generators or registered signals from two D flip–flops. More detailed description about the cell can be found in [9].

Each CLB is surrounded by a hierarchy of routing resources and a perimeter of programmable Input/Output Blocks (IOBs). The IOB provides interfacing between the package pins and internal signal lines such as the boundary scan circuit. The routing resources provide routing paths to connect the inputs and outputs of the CLBs and IOBs onto the appropriate networks. These types of connections are made by metal lines with programmable

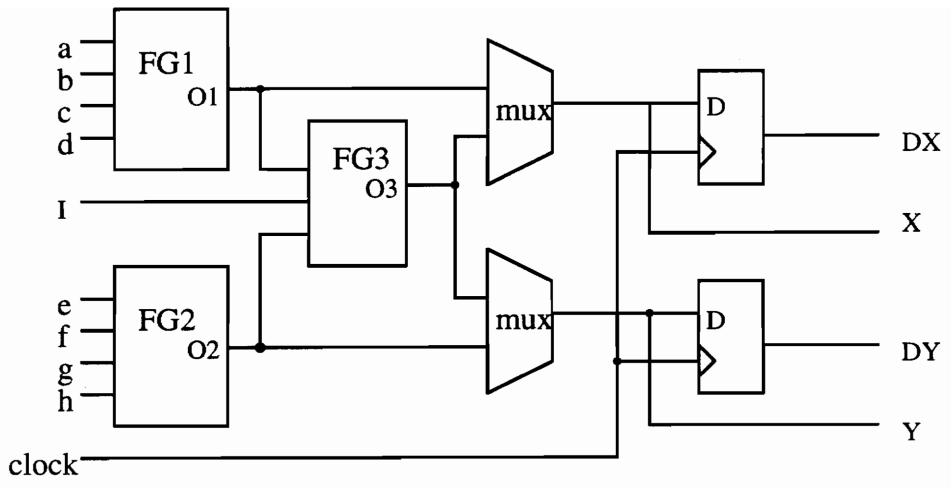


Figure 2.3. Simplified block diagram of XC4000 family CLB.

switching points and switching matrices. Figure 2.4 illustrates the basic single–length interconnect lines surrounding one CLB in the array.

The XC4000 family FPGAs also include on–chip static memory resources, and a library of macros. The fast on–chip RAMS create possibilities in the system design such as index registers, DMA counters and status registers, etc. The macros contain information of partitioning, placement and routing. These predefined and tested functions permit the user to build timing–critical designs with optimized performance.

The XC4010 FPGA is a 84–pin device used as the major processing element on SPLASH II board. Each chip includes 400 CLBs and 144 IOBs, with a total number of 1120 flip–flops. The chip also has maximum of 12800 bits of on chip RAM.

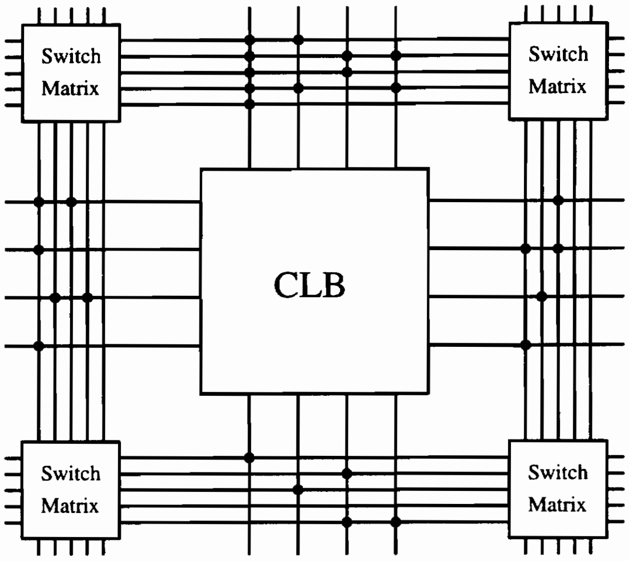


Figure 2.4. Typical CLB connections to adjacent single–length lines. Each dot indicates the interconnection between CLB pins and its I/O lines. Each switch matrix consists of programmable logic to establish connections between the lines.

2.5 Data/Control path

The 36–bit linear data path connects all FPGA processing elements in a one dimensional array. A 4–bit tag is added to the 32–bit data stream from the host in the input DMA channels. The 36–bit data pass through the input FIFOs and XL onto the SIMD Bus, and into PE X1 of the first SPLASH array board. On the interface board, the 4–bit tag is removed by XR, and the remaining 32–bit data stream return to the host through output FIFOs and their DMA channels.

The control paths between the SUN host and the SPLASH cabinet include a set of handshake registers and a global AND/OR mechanism. These features add flexibility to the SPLASH II system. However, these features are not used in this particular application.

2.6 SPLASH II software environment

The programming environment for SPLASH II is based upon the VHSIC (Very High Speed Integrated Circuit) Hardware Description Language (VHDL). The implementation of an application is first manually partitioned into a set of SPLASH computing elements. The VHDL behavioral model for each element is then developed, and refined and debugged within the SPLASH II simulator. During simulation, the VHDL model is able to interact with the system exactly as it would with the actual hardware, including the timing and I/O constraints.

Once the implementation is determined to be functionally correct, the Synopsys logic synthesis tool converts the VHDL file into an EDIF format netlist file. The EDIF file is then translated into the XILINX Netlist Format (XNF). The Xilinx Automated CAE Tool (XACT) will conduct the process of Partitioning, Placement and Routing (PPR) automatically. The completed design will be stored in a Logic Cell Array (LCA) file, from which a Xilinx bitstream format (.bit) file can be generated. More details about the SPLASH programming environment are covered in Chapter 6.

3. Image Pyramid Generation

Multiresolution and multirate image processing techniques have become increasingly popular over the past decade because of the advantages of processing image data at different scales. A basic data structure used in multiresolution and multirate processing is the image pyramid, which is a complete image representation at different levels of resolution. A image pyramid is constructed by recursively applying two basic operations, filtering and subsampling, to an input image, creating a set of images of decreasing size and spatial resolution. Filtering is performed to convolve the input image with a family of local, symmetric weighting functions. Subsampling then produces samples for the two most common image pyramids, the Gaussian (low–pass) and the Laplacian (band–pass) pyramids.

3.1 Gaussian pyramid

The sequence of images g_0, g_1, ..., g_{k-1} as shown in Figure 3.1(a) is called a k–level Gaussian pyramid [10]. A weighting function which resembles the Gaussian probability distribution is applied to each pixel of the original video image g_0 to generate the lower–resolution image g_1, which is used in turn to generate g_2, etc. The level–to–level filtering and resampling can be expressed as a function *REDUCE* as shown below:

$$g_k = REDUCE\ (\ g_{k-1}\ )$$

where each pixel value in g_k is obtained by a weighted sum of pixels from g_{k-1}, computed over a 5×5 neighborhood as follows [2]:

$$g_k(i,j) = \sum_{m=-2}^{2} \sum_{n=-2}^{2} \omega(m,n) g_{k-1}(2i - m, 2j - n) \qquad (1)$$

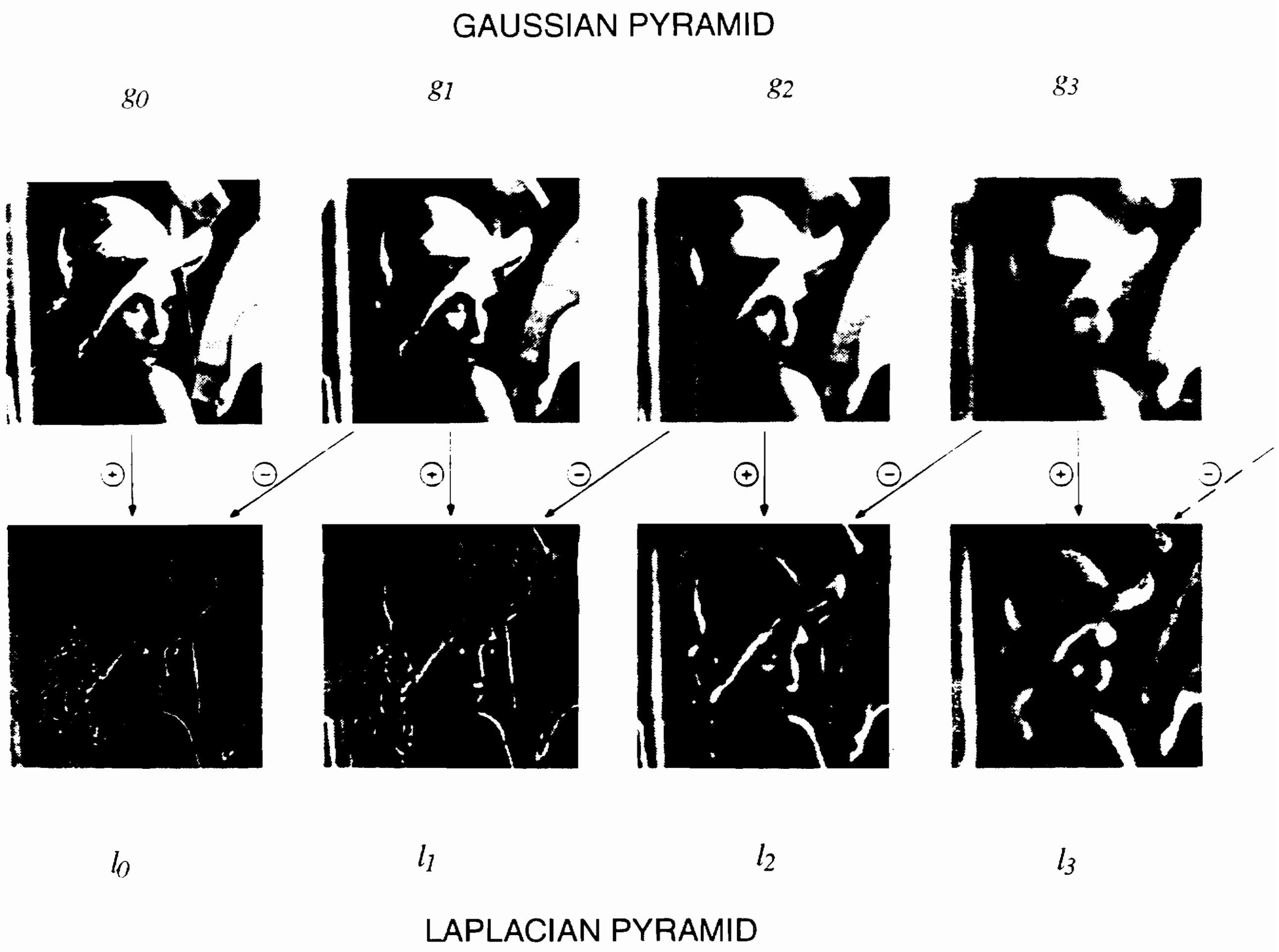


Figure 3.1. (a) Gaussian (low–pass) and (b) Laplacian (band–pass) pyramids [10].

To simplify the computational requirements, the 5×5 weighting function w is often chosen to be separable: $w(m, n) = w_x(m)\, w_y(n)$. Therefore, the image can be filtered by two one–dimensional (1D) filters w_x and w_y in sequence. As illustrated in Figure 3.2, each row of dots represents an image row within one level of the pyramid. Each dot represents one image pixel, which is computed as the weighted average of 5 pixel values from the level below it. The same weighting pattern is used to generate all pyramid levels.

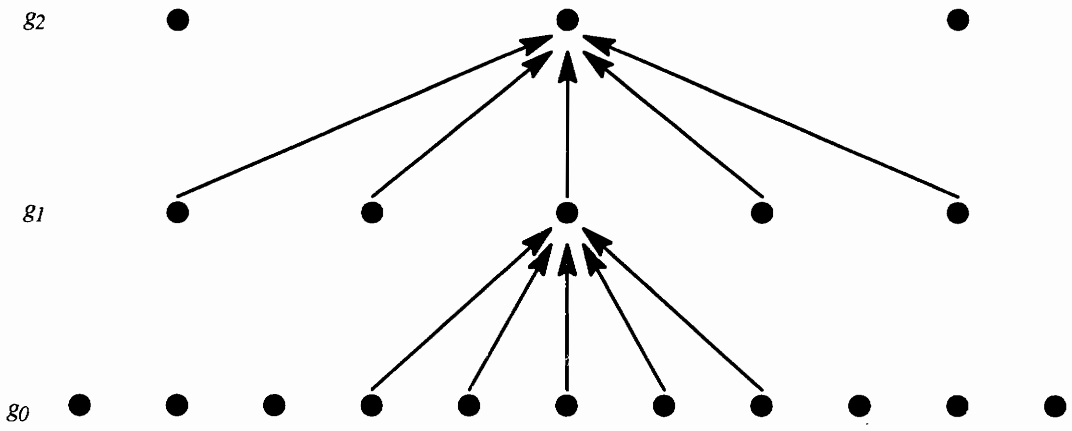


Figure 3.2. One dimensional representation of Gaussian pyramid generation.

The function *REDUCE* in Equation (1) is then split into two functions *REDUCEX* and *REDUCEY*:

$$g_{k,x}\,(i, j) = REDUCEX\,(\,g_{k-1}\,) = \sum_{m=-2}^{2} \omega_x(m) g_{k-1}(2i - m, j) \qquad (2a)$$

$$g_k\,(i, j) = REDUCEY\,(\,g_{k,x}\,) = \sum_{n=-2}^{2} \omega_y(n) g_{k,x}(i, 2j - n) \qquad (2b)$$

The 1D weighting function in the vertical direction, w_y is usually the transpose of the function in the horizontal direction, w_x. Functions w_x and w_y commonly have the following constraints [10]:

1. The function is normalized:

$$\sum_{i=-2}^{2} w(i) = 1 \qquad (3)$$

2. It is symmetric:

$$w(i) = w(-i), \qquad \text{for } i = 0, 1, 2. \qquad (4)$$

3. The function follows the equal contribution rule. In this case the equal contribution requires that

$$a + 2c = 2b, \qquad (5)$$

   where $a = w(0)$, $b = w(-1) = w(1)$, $c = w(-2) = w(2)$.

Although other solutions are possible, these three constraints are satisfied when

$$w(0) = a$$

$$w(-1) = w(1) = 1/4$$

$$w(-2) = w(2) = 1/4 - a/2.$$

The equivalent weighting function is particularly Gaussian–like when a is around 0.4. For implementation in digital logic, it is convenient to choose $a = 3/8$, $b = 1/16$, and $c = 1/4$.

3.2 Laplacian pyramid

The Laplacian pyramid as illustrated in Figure 3.1(b) is a sequence of difference images, in which each image is the difference between two successive Gaussian levels. Two types of Laplacian pyramid are in common use: the filter–subtract–decimate (FSD) structure and the reduce–expand (RE) structure [2].

The FSD Laplacian is formed by subtracting a filtered image of the next higher Gaussian pyramid level from the same level of the pyramid image. The *k*th level of the FSD Laplacian pyramid can be expressed as below:

$$L_k^{FSD}(i,j) = g_k(i,j) - g_{k+1}^F(i,j) \qquad (6)$$

Where g_{k+1}^F is the $(k+1)$th level of the filtered Gaussian image before subsampling. The FSD Laplacian is very intuitive, and relatively easy to obtain. It is appropriate for computer vision applications.

The structure of the RE Laplacian is more complicated than for the FSD Laplacian. An interpolated image is computed by convolving the same weighting function w over the $(k+1)$th level of Gaussian image. The mathematical representation can be written as follows:

$$g_{k+1}^e(2i, 2j) = g_{k+1}(i,j); \qquad (7a)$$

$$g_{k+1}^e(2i \pm 1, 2j \pm 1) = 0; \qquad (7b)$$

$$g_{k+1}^{intp}(i,j) = 4 \sum_{n=-2}^{2} \omega_y(n) \sum_{m=-2}^{2} \omega_x(m) g_{k+1}^e(i-m, j-n). \qquad (7c)$$

Where g^e is an expanded version of Gaussian image g by simply inserting a zero value between each row and column of the Gaussian image, and g^{intp} is the interpolated image from g^e. The operation above can be expressed as a function *EXPAND* referring as the reverse of the function *REDUCE*.

$$g_{k+1}^{intp}(i,j) = EXPAND(g_{k+1}(i,j)) \qquad (8)$$

The RE Laplacian L^{RE} is then computed as follows:

$$L_k^{RE}(i,j) = g_k(i,j) - g_{k+1}^{intp}(i,j) \qquad (9)$$

Notice the *k*th level of the Gaussian pyramid can be recovered from the sum of L_k^{RE} and g_{k+1}^{intp}. Furthermore, the original image I_0 can be recovered exactly by expanding each i^{th} level Laplacian image (L_i^{RE}) *i* times, and then summing all the levels: $I_0 = \sum_{i=1}^{n} EXPAND^i(L_i)$. Therefore, the RE Laplacian structure is appropriate for image processing applications involving image reconstruction and coding.

This thesis describes an implementation of a 5–level RE Laplacian pyramid. The FSD pyramid has not been implemented.

3.3 Previous implementations for Gaussian and Laplacian pyramids

It is desirable to process the image data in real–time and in a cost–efficient manner. A direct implementation for pyramid generation is to use a separate digital filter for each filter stage, where all the filters are linearly pipelined. This *linear pipeline* configuration is illustrated in Figure 3.3(a). The filters for the lower resolution levels will have to operate at reduced clock rates to keep all calculations synchronized. This ensures the minimum processing time, at the expense of extra hardware.

An alternative configuration is to use only one digital filter and construct a pyramid one level at a time as illustrated in Figure 3.3(b). This *recirculating pipeline* structure generates each level of the pyramid in sequence at the same clock rate. It requires extra memory to store intermediate results, and a dual port memory is needed to support simultaneous read and write operations. The processing time is directly proportional to the size of the image level. Since each successive level has one quarter as many pixels, each takes one quarter the

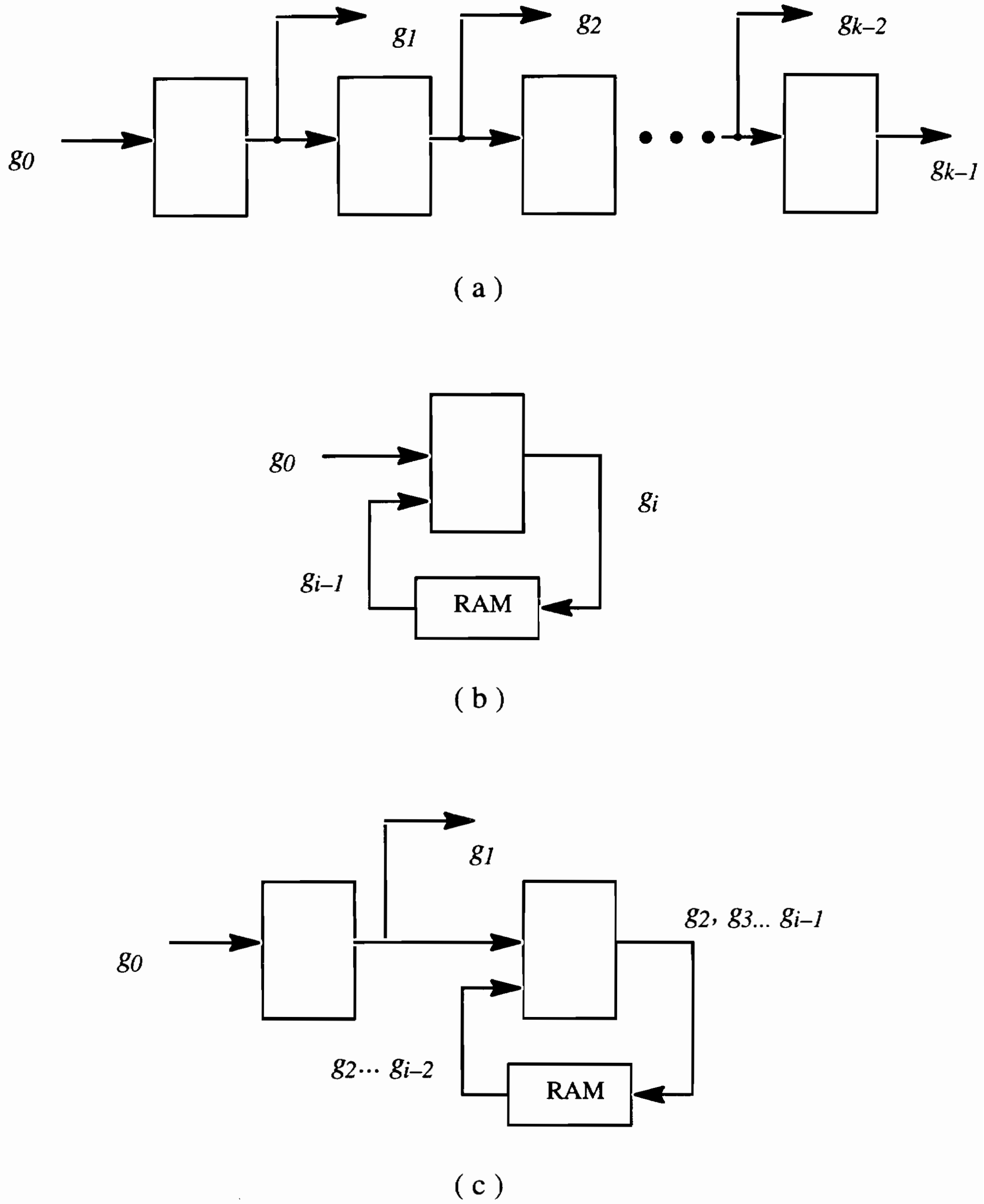


Figure 3.3. Three pyramid generation architectures: (a) linear pipeline structure, (b) recirculating pipeline structure and (c) hybrid structure implemented in this research. Each block represents a digital filter with resampling.

time to generate. The upper bound on processing time can be mathematically calculated as below:

$$1 + \frac{1}{4} + \frac{1}{4^2} + \ldots = \sum_{i=0}^{\infty} \left(\frac{1}{4} \right)^i = \frac{1}{(1 - \frac{1}{4})} = \frac{4}{3}$$

If g_1 requires T time units to be produced, then the total processing time for this procedure is (4/3) T. Therefore, the linear pipeline architecture is 25% faster than the recirculating pipeline design, but with less efficient use of hardware.

Figure 3.3 (c) shows a *hybrid pipeline* structure which was developed and implemented in this research. It is composed of two stages. The first stage produces the first level of the pyramid, and passes the data to the second stage. The second stage is a recirculating pipeline which generates levels g_2 to g_{k-1}. This architecture provides the same data processing speed as the the linear pipeline structure, but utilizes the hardware more efficiently.

A VLSI chip called PYR has been developed at the David Sarnoff Research Center in Princeton, NJ, to perform the standard operations required in pyramid transforms [2, 11]. The chip is designed to build Gaussian, Laplacian, subband, and related pyramid structures, including gradient and moment pyramids through some off–chip processing.

The functional components of the PYR chip are shown above in Figure 3.4. The chip has 3 input ports and 2 output ports. Two images can be added/subtracted before or after the filter at P1 and P2, respectively. The Gaussian pyramid can be produced from output port OUT1 by passing the input image from IN1 through the filter module. The Laplacian pyramid can be obtained from port OUT2 by subtracting the Gaussian level data from the bypassed input data at P2. The third input port IN3 can be used in implementation of double precision structures. Two PYR chips process 8 of the 16–bit input images, and the clip module on one PYR chip combines the results at 16 bits.

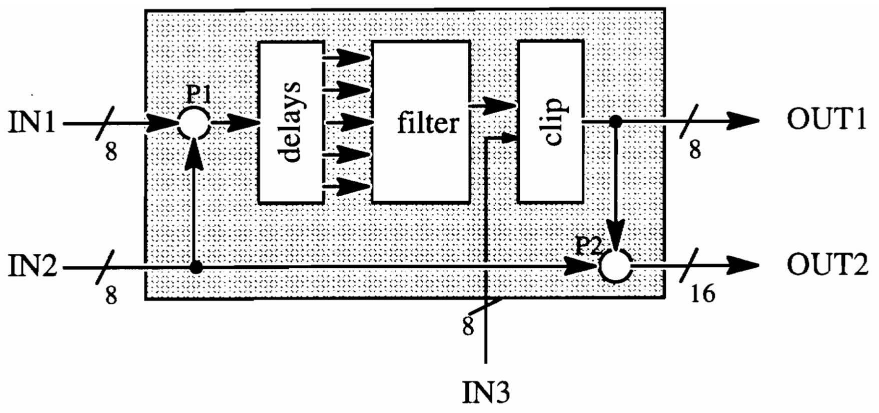


Figure 3.4. Functional diagram for the PYR chip. Gaussian pyramids can be generated from port OUT1, and Laplacian pyramids from port OUT2.

This pyramid transform system uses the PYR chip as major filter logic unit to provide relatively sophisticated implementations of image pyramid generation at a low cost. The system consists of a pyramid generation board, a frame store system, an ALU with video data multiplexing and an analog interface board. It is capable of constructing a Gaussian and a Laplacian pyramid from a 512x480 image in 22.7 ms, using a clock rate of 15 MHz.

3.4 Other useful image pyramid architectures

In addition to the Gaussian and Laplacian pyramid implementation introduced by Burt, quite a few alternative algorithms have been developed. These include the Difference of Low–Pass transform (DOLP) [12], the multiresolution wavelet representation [13], the binary tree pyramid [14] and the edge and curve pyramid [15]. These algorithms are useful for the analysis of certain images based on their construction algorithms. This section describes each of these briefly.

3.4.1 The DOLP transform

The DOLP transform uses an algorithm that is similar to Burt's algorithm of Laplacian pyramid generation. A set of bandpass images are formed by differences of two low–pass images [12]. These bandpass images comprise the DOLP space, in which the representation is constructed by detecting peaks and ridges. This representation of object shape is used for stereo matching.

3.4.2 The wavelet representation

The wavelet representation [13] is computed by decomposing the original image with a wavelet orthonormal basis. The computation is performed with a pyramidal algorithm based on convolutions with quadrature mirror filters (QMF) [2]. The wavelet image pyramid can discriminate several spatial orientation in different resolutions. Typically, for a 2D image, the wavelet pyramid consists of four component pyramids. The filters are formed as combinations of 1D horizontal and vertical high– and low–pass filters. The mathematical expression is shown below:

$$P_k^{LL}(x,y) = \sum_n \omega_y^L \sum_m \omega_x^L P_{k-1}^{LL}(x - m, y - n)] \;;$$

$$P_k^{LH}(x,y) = \sum_n \omega_y^H \sum_m \omega_x^L P_{k-1}^{LL}(x - m, y - n)] \;;$$

$$P_k^{HL}(x,y) = \sum_n \omega_y^L \sum_m \omega_x^H P_{k-1}^{LL}(x - m, y - n)] \;;$$

$$P_k^{HH}(x,y) = \sum_n \omega_y^H \sum_m \omega_x^H P_{k-1}^{LL}(x - m, y - n)] \;.$$

where P_0^{LL} is the original image, and $\omega_x^L\omega_y^L$, $\omega_x^L\omega_y^H$, $\omega_x^H\omega_y^L$ and $\omega_x^H\omega_y^H$ are the four wavelet functions.

The wavelet representation introduces spatial orientation selectivity into the decomposition process. The data at separate levels are not correlated since the wavelet functions are orthogonal to each other. The wavelet pyramid can be used in for compact image coding, texture discrimination and fractal analysis.

3.4.2 The binary–tree pyramid

The binary–tree pyramid [14] is formed according to a one–dimensional binary tree structure. Figure 3.5 shows a graphical representation of a three–level binary pyramid. Pixels at each pyramid level are weighted averages of two successive pixels from the previous level at x coordinate and y coordinate alternatively. All pixels within the same pyramid level are processed in parallel. But the operation applied to different pyramid levels may vary with the application.

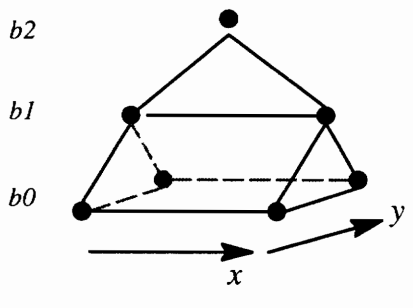


Figure 3.5. Graphical representation of a three–level binary pyramid .

The major difference between the binary tree pyramid and the pyramids introduced previously is that no neighboring pixels overlap during the construction of a higher level of a binary pyramid. The algorithm requires large numbers of simple processing elements, but is easy to implement. The key application of the binary–tree pyramid is to efficiently centralize global image features such as the mean of an image, histogram and connected region count. The binary pyramidal structure is also used for progressive refinement of binary 3D images.

3.4.3 The edge– and curve– based pyramid

The image pyramid does not necessarily represent only averages of the gray levels of the input image. It can also represent information about lines and curves [15]. The edge pyramid is constructed by applying an edge detector at each level of the gray level pyramid. Alternatively, the gray level pyramid can be first processed to produce an edge image, from which the pyramid is formed. These two pyramid generation algorithms are not equivalent, and each has its own applications

The advantage of the edge pyramid is its ability of extracting the major edges efficiently from an image at low resolution levels of the pyramid. These edges can then be projected down to the original image without the smaller edges having been detected. The edge and curve based pyramid can also be used to extract ribbon–like regions from an image. As the local curving detector is applied repeated to successively higher levels of the pyramid, the parallel pair of edges are guaranteed to become close enough together to be considered as a single curve at some level.

3.5 Applications of Gaussian and Laplacian image pyramids

3.5.1 Image compact coding

It is not efficient to represent an image directly in terms of the gray level of each pixel because it requires large amount of storage space, and neighboring pixels of an image are usually highly correlated. Ideally, an image should be represented in a form which stores the most critical information for image analysis tasks. This representation should be robust, compact and complete.

The Laplacian pyramid meets all three of these requirements for image coding [5, 10]. The Laplacian pyramid is complete for image representation because the original image can be recovered exactly by summing all pyramid levels. Each pixel in the Laplacian pyramid is the difference between the original input and the corresponding Gaussian node. It has the effect of removing pixel–to–pixel correlations, and shifting the pixel values towards zero. Some image pyramids can therefore be represented with fewer than 8 bits per pixel. Experiments show that this compact code does not degrade the image appearance or the results of image analysis. Therefore, the Laplacian coding is both compact and robust.

3.5.2 Motion estimation

One technique used in motion estimation is correlation matching [10]. The displacement vector field between two successive images is computed by using correlation to search matching image regions. This correlation approach tends to be computationally costly. A Gaussian pyramid–based procedure uses a coarse estimate strategy at low resolution level to minimize the size of the search area. The estimate is then refined progressively with higher resolution images. The coarse–fine search is terminated when the iteration reaches a certain level of resolution so that the displacement is within $\pm 1/2$ of the sample distance at level k, which corresponds to $\pm 2^{k-1}$ for the original image, g_0.

3.5.3 Object recognition and texture analysis

Edge detection and region matching are two of the most important operations used in object recognition. However, the objects that we want to recognize often have very different sizes. It is not possible to define an optimal resolution for analyzing all images. Multiresolution decomposition allows us to interpret the image in a scale–invariant way.

The Gaussian image pyramid contains image information at different levels of resolution. Small–scale objects in an image are removed by smoothing at low resolution levels, and only large–scale objects can be detected. Therefore, different objects can best be detected at the appropriate level of image resolution in parallel. The computation cost will be largely reduced if the operations are only applied to a certain resolution level and a certain image region.

The same technique is useful for characterizing certain textures. Density or local power spectrum estimations can be computed at multiple levels of resolution. These sets of information are then passed on to other processes which perform the texture classification.

4. Implementation for Gaussian Pyramids on SPLASH II

4.1. Weighting function and image convolution

The most basic operation for generating a Gaussian pyramid is image filtering. As presented in Equation (2a) and (2b), the operation is broken into two 1D convolutions in the horizontal and vertical directions. The 1D weighting function used here is $\omega_x = \omega_y^T = [\frac{1}{16}, \frac{1}{4}, \frac{3}{8}, \frac{1}{4}, \frac{1}{16}]$.

Since the denominators of all weighting factors are powers of 2, the multiplication of image pixels by the weighting factor can be simply implemented using binary shift operations. For instance, a pixel multiplied by 1/16 is equivalent to shifting the pixel value 4 binary places to the right. A pixel multiplied by 3/8 is the sum of the value shifted two places to the right, plus the original value shifted three places to the right. The one dimensional convolution by this weighting function can now be calculated as six partial sums.

To maintain numerical accuracy, the summation elements have been expanded to 12 bits each. Four bits with values of 0 are appended to the right of each image pixel value before computation. The 8 most significant bits of the final results are retained.

4.2. Two implementations for Gaussian pyramid on SPLASH II

This section presents a high–level description of the architectures that have been implemented. Section 4.3 provides implementation details for each processing element. The system was designed to generate 5 pyramid levels for images of size 512×512 pixels. Higher levels are not produced because they do not carry much useful image information. However, the design could be updated easily to produce all 9 pyramid levels without adding any processing delay.

4.2.1 The recirculating pipeline structure

Figure 4.1 shows the block diagram of a five–chip pyramid generation architecture that has been developed for SPLASH II. This implementation is based on the recirculating pipeline structure, and is designed to produce 5–level pyramids ($g_0 - g_5$). The control element X0 simply receives image pixels and passes them to FPGA X1 through the crossbar. Then the processing steps of this architecture, in sequence, are horizontal convolution by w_x (X1), vertical convolution by w_y (X2 and X3), and recirculating and output of image pyramid data (X4).

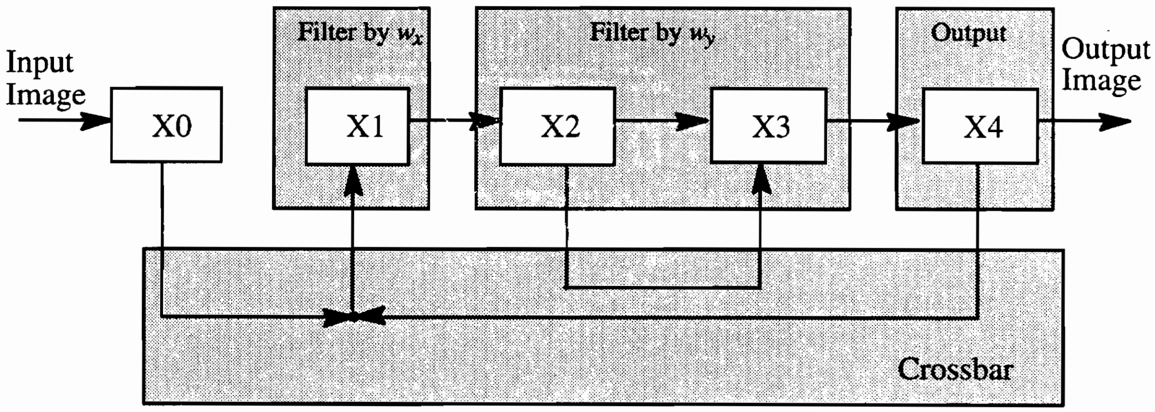


Figure 4.1. SPLASH II implementation of Gaussian pyramid generator using the recirculating pipeline architecture.

The control element X0 broadcasts image pixels, representing g_0, through the crossbar to processing elements X1 – X3 to compute the first level of Gaussian pyramid, g_1. Image data is then recirculated through the crossbar to X1, and processed through the same path to form the higher pyramid levels. Two different crossbar configurations are used to multiplex the original image data and the feedback pyramid data. X0 controls which crossbar configuration to use during processing.

Device X1 receives image data from either X0 or X4 through the crossbar, computes the convolution by w_x and passes the results to X2. Resampling in the horizontal dimension is performed during the convolution to eliminate half of the computations. The image data that is passed to X2 has half the pixels per image row.

The image data is presented into the SPLASH FPGAs one row at a time in a raster order. The 8–bit image pixels that are presented to X1 are grouped so that four pixels are passed simultaneously on the data bus (data path 0 – 31). Four control bits on data path (32 – 35) are appended as follows. The most significant bit (bit 35) is activated when image data is valid. The remaining three bits represent the pyramid level of input pixels. They control the number of pixels per image row and per pyramid level. The input data is updated every four SPLASH II clock periods. The output data from X1 contains only two 8–bit pixels, along with four control bits. The remaining 16 bits are all zeros.

X2 and X3 together implement the convolution by w_y. Unlike the convolution in the horizontal direction, the five pixels required by each computation are not presented in the same image row. Instead they are from five successive rows. The image data therefore needs to be stored in a delayed line, which has been implemented using the external RAMs of the device. One memory write and four memory reads are involved in each 5×1 convolution. Only one memory write and two memory reads are allowed during four SPLASH clock periods because of memory access constraints. X2 computes three of the five partial sums, and passes the 12–bit partial result directly to X3. X3 performs the remaining three partial sums, and passes the rounded 8–bit value to X4.

X4 resamples the image data in the vertical dimension to reduce the number of pixels per image column by half. The data is then recirculated to X1 through the crossbar to form the next level of the pyramid. Each pyramid level is also made available to the next FPGA device (X5) for further analysis.

4.2.2 The hybrid pipeline structure

The block diagram of a nine–chip hybrid structure of a Gaussian pyramid generator is given in Figure 4.2. The original image pixels (g_0) are passed to X1 directly from the input

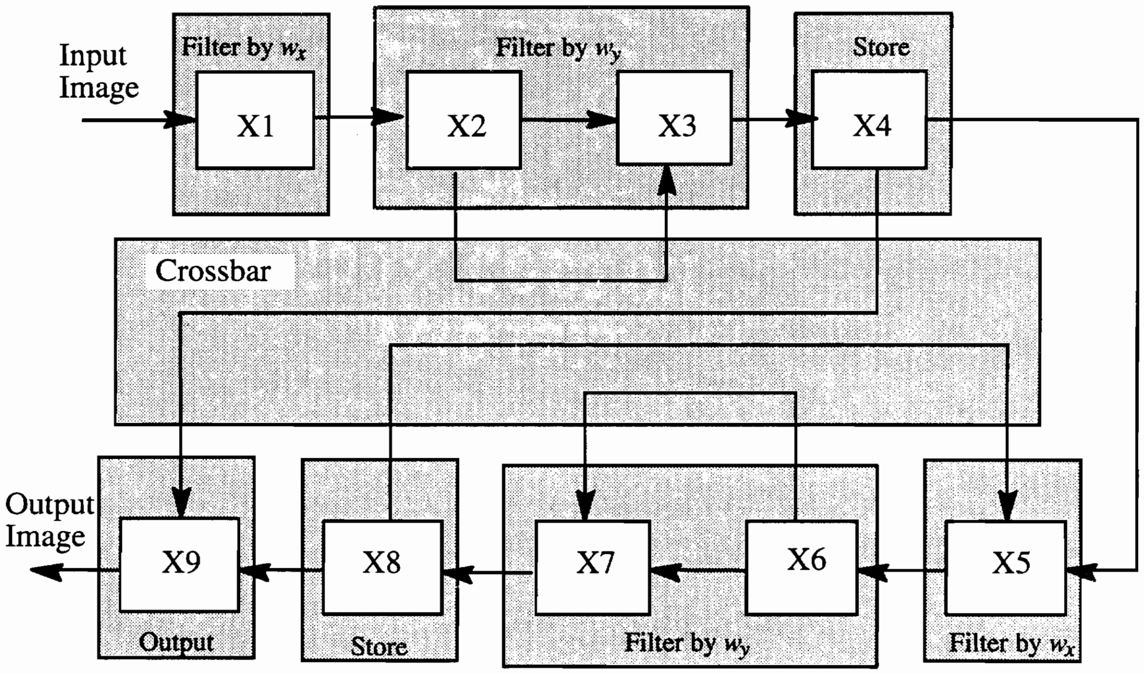


Figure 4.2. SPLASH II implementation of Gaussian pyramid generator using the hybrid pipeline architecture.

FIFO, and are processed through PEs X1 – X4 to form the first level Gaussian pyramid $g1$. Xilinx devices X5 – X8 generate the remaining four levels of the pyramid. Chip X9 takes the data from X4 and X8, and forms a complete pyramid from $g1$ through $g5$. 52 to 55.

The hybrid implementation requires five more PE devices than the recirculating implementation. It does not increase the design cycle because two groups of circuitry (X1 – X4 and X5 – X8) are almost identical to chips X1 – X4 from the previous design, described in Section 4.2.1. Only one crossbar configuration is used in this architecture, and therefore the control element X0 does not need to be programmed. The key advantage of this algorithm is that it is capable of generate Gaussian pyramid in real time. A complete pyramid is generated for every input image frame.

4.3 The design of Xilinx devices

This section describes the design in detail for Xilinx devices X1 through X4 in the recirculating pipeline structure (Section 4.2.1). The modifications required by the hybrid design structure (Section 4.2.2) are then described briefly.

4.3.1 Structure of Xilinx chips on SPLASH II

The SPLASH II array board uses Xilinx XC4010 chips, each of which contains an array of 20×20 CLBs. Appendix A shows a symbol of the Xilinx processing elements with all the signal pins listed. A brief overview of the signal ports used in this design are given in this section.

The device signals are composed of data signals and control signals. Signals XP_Left, XP_Right and XP_XBar are 36–bit data paths from or to the left neighbor chip, the right neighbor chip and the crossbar, respectively. XP_Mem_A is a 18–bit wide address path to the external memory, and XP_Mem_D is the 16–bit bi–directional data bus from or to the memory. The control signals include the SPLASH clock (XP_Clk), a memory read signal (XP_Mem_RD_L), a memory write signal (XP_Mem_WR_L) and a 5–bit crossbar output enable (XP_XBar_EN_L).

Each Xilinx chip is associated with an external RAM of size 256×16 KBit (1/2 MByte). The data bus to and from the external memory is 16 bits wide and bidirectional. This external tristate signal (*OUT*) is driven by a bidirectional buffer in the Xilinx chip. Figure 4.3 shows the graphical symbol of the I/O buffer. Two internal data paths (*IN1* and *IN2*) are connected to the buffer. An internal memory enable signal (*EN_mem*) determines when the data is driven to the external RAMs. When *EN_mem* is high (logic '1'), the latch drives the internal data from *IN1* to *OUT*. When the enable signal is low (logic '0'), the latch drives a high impedance state to *OUT*. The internal signal *IN2* always follows the value of *OUT*, regardless of the enable signal. The *EN_mem* signal stays at '0' during memory idle cycle and memory reads, and is asserted to '1' when a memory write is issued.

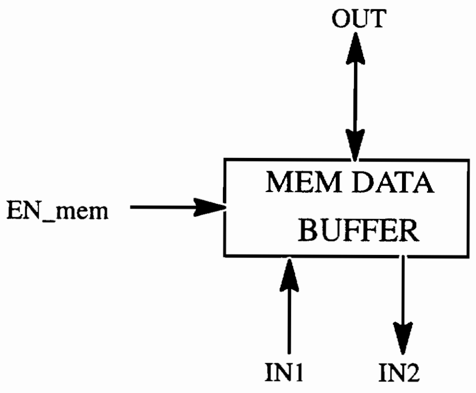


Figure 4.3. A graphical symbol of memory data I/O buffer.

In addition to the signals listed above, control chip X0 has a 3–bit crossbar configuration selector (X0_XBar_Set), which can select up to 8 pre–loaded configurations. Since X0 and X16 share the same crossbar port, signal X0_X16_Disable determines which device receives or sends data to the crossbar. When X0_X16_Disable is logic '1', chip X16 is isolated from the crossbar. The signal X0_XBar_Send controls the direction of crossbar data flow, and it is only meaningful when X0_X16_Disable is '1'.

4.3.2 The structure of X1

As described in Section 4.2.1, processing element X1 performs convolution of an input image by w_x. This involves the weighted sum of every five successive pixels in a image row. Since only four pixels are received at a time, buffering is required to save the previous input pixels. The block diagram of chip X1 is given in Figure 4.4. The circuitry is composed of two major blocks: the state machine and the shift–and–add block.

The image data is updated every four SPLASH cycles. To get the best pipeline performance, the convolution therefore must be completed within four cycles. As explained in section 4.1, each convolution contains five 12–bit additions. Therefore, a minimum of two 12–bit adders are required for the computation.

Since there are always eight pixels available during every four–cycle period, two convolutions are calculated within the period. This is graphically represented in Figure 4.5. Eight image pixels $(a - h)$ from the higher resolution level will form pixels 1–4 after the convolution. However, only two pixels 1 and 3 are stored as the next level of the pyramid after resampling. The shift–and–add block therefore requires four 12–bit adders, two for each convolution.

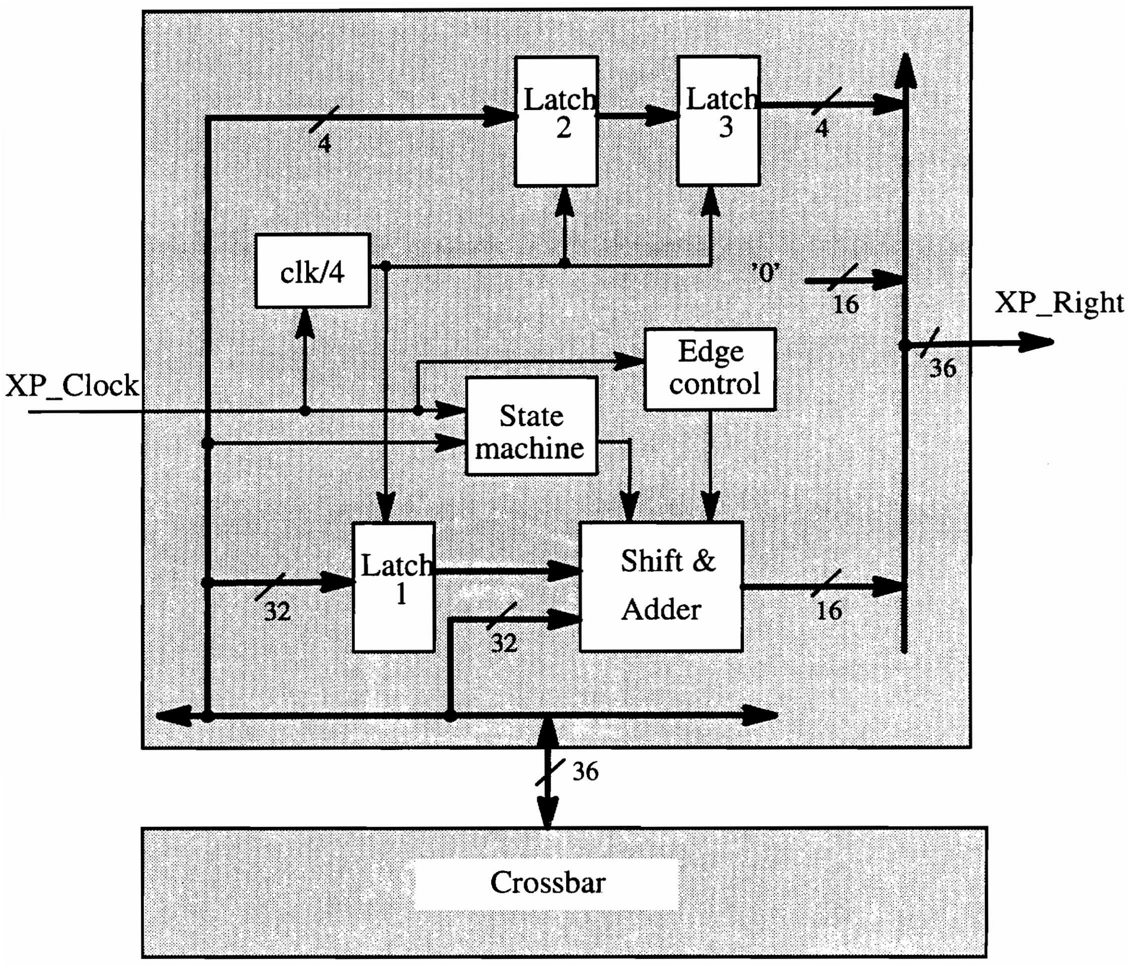


Figure 4.4. Block diagram of Xilinx device X1. This PE performs the convolution in the horizontal direction. The output image pixels are delayed by 8 Splash clocks, therefore, two latches are needed for the control codes.

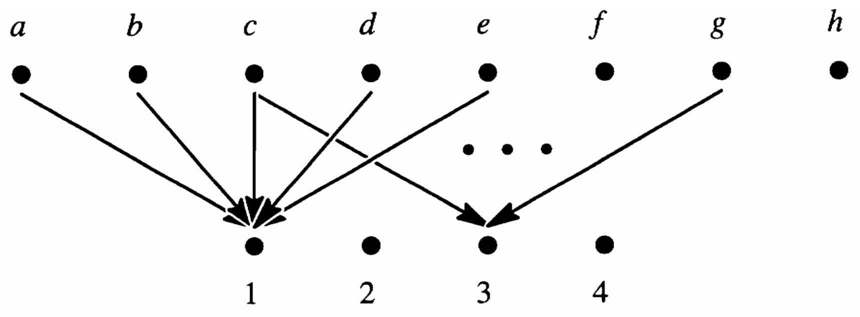


Figure 4.5. Graphical representation of the adder block in chip X1.

The state machine is simply implemented as a 2–bit counter. It is initialized to the starting state when the image valid code in image tag bits is set. Four states, three shift–add states and one data output state are executed in sequence. Logic for the input operands of the adders are controlled by the state machine.

The number of pixels per output image is now half of the original image after resampling in the horizontal direction. Only the lower 16 bits of the data bus (bits 15 to 0) are used to send the image data to element X2. Bits 35 to 32 still carry the control data. The remaining 16 bits are all set to the logic '0'.

The output values along the image border are not defined mathematically during the filtering. These pixels are normally ignored in other image processing tasks. However, border effects can not be ignored in the multiresolution image analysis. At low–resolution pyramid levels each row or column becomes a significant part of the image. Since the neighboring pixels are highly correlated, the simplest way to extend the image edge is to repeat the left–most and right–most filtered pixels. The edge control block detects and duplicates the first and the last pixels of each convolved image row.

4.3.3 The structure of X2 and X3

Xilinx processing elements X2 and X3 implement the 5×1 convolution by ω_y in the vertical direction. Unlike the convolution in the horizontal direction, the five image pixels are not available in a successive order. Pixels values must therefore be stored in the memory associated with the Xilinx chip. The computation results will have at least four image rows of delay relative to the input image.

As mentioned in Chapter 2, the SPLASH II processor board does not support dual–ported memory read and write operations, and the bi–directional memory data path is used to convey both the read and write data. Each convolution requires five memory access operations: one memory write and four memory reads. Because of the memory access constraint, that a read operation cannot be followed immediately by a write operation, at least six SPLASH clock cycles are required for each convolution in one chip. But the image data is updated every four cycles. The operations are thus distributed equally to two processing elements to achieve real time performance.

Figure 4.6 gives the block diagram of chip X2. The processing element X2 reads the image data from the left port connected to X1, and sends the data directly to X3 through the crossbar. A memory write operation is issued to store the image values in both RAMs for X2 and X3 each time the input data is updated. Chip X2 then reads image pixels from the previous third and forth rows into the chip, and calculates the first three partial additions in the convolution. And chip X2 reads data from the previous first and second row, and computes the remaining partial additions

The sequence of memory accesses and data processing is illustrated in Figure 4.7. Both memory write (WR_mem) and memory read (RD_mem) signals are active low logic. One memory write followed by two memory reads are issued every four SPLASH cycles.

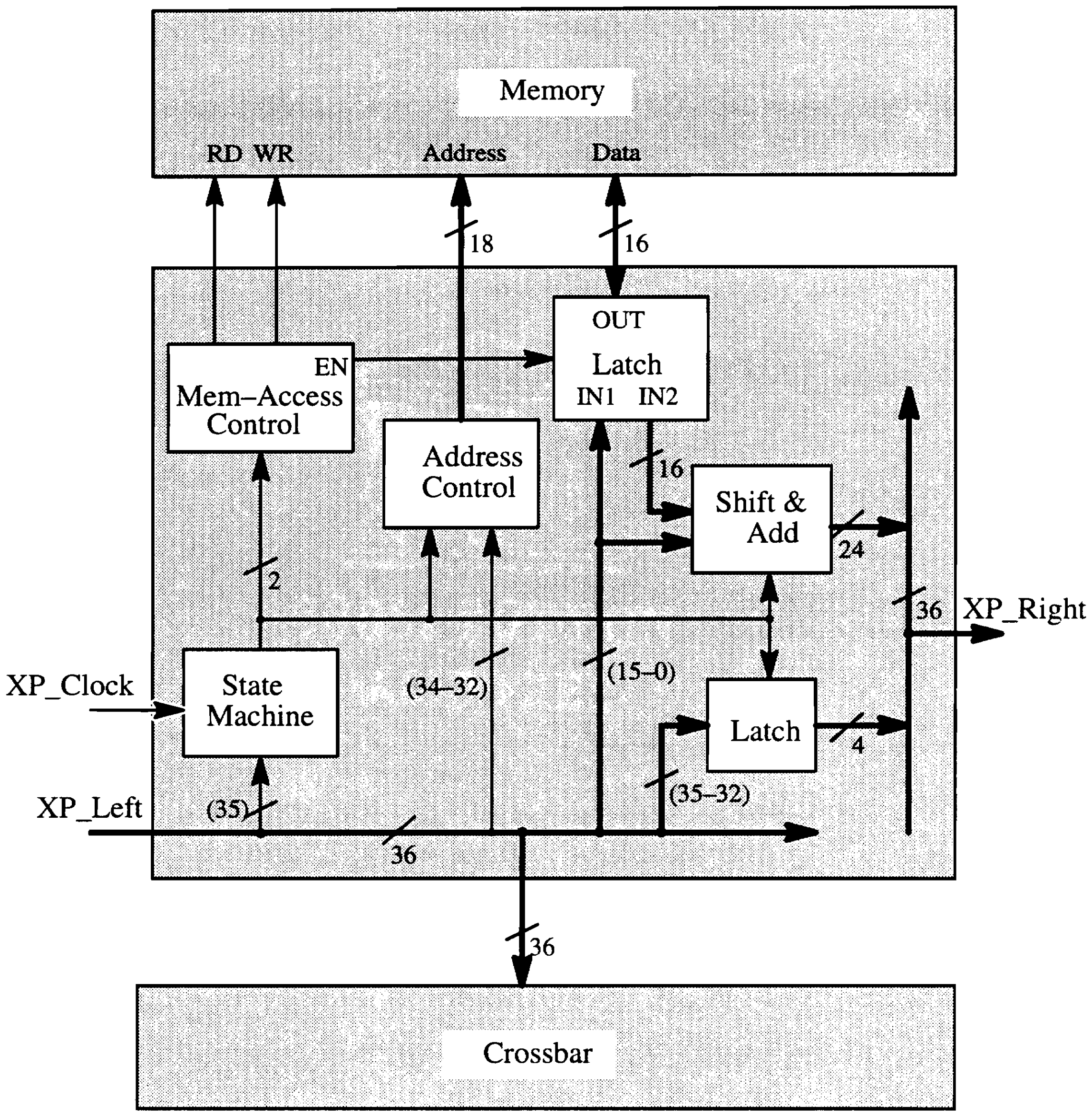


Figure 4.6. Block diagram of Xilinx device X2. This PE performs the convolution operation in the vertical direction together with PE X3. X3 has similar structure to device X2.

The memory read signal is disabled for one clock cycle before the next memory write operation is issued. The memory enable signal is logically the complement of the memory write signal, so that the image data from the left port of the chip (XP_Left) will be latched to the bidirectional memory data path when a write operation is issued, and the latch_in (IN2) signal to the chip will remain as the memory read data.

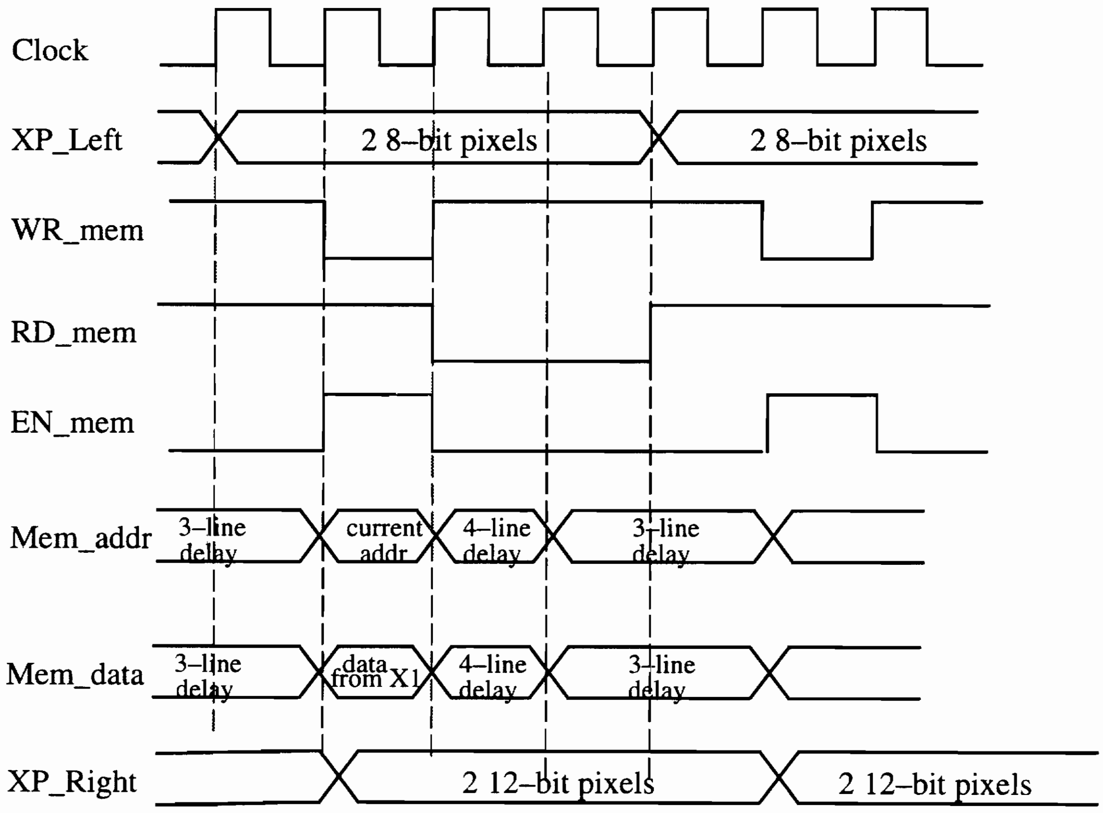


Figure 4.7. Memory access sequence of device X2.

Since X2 computes only partial results of the filtering by ω_y, the 12–bit partial sum is passed directly to X3 without being rounded to 8 bits. Chips X2 and X3 contain two and three additions for each convolution, respectively. Therefore only two 12–bit adders are required for the shift–and–add block.

The structure of processing element X3 is very similar to chip X2 except for the source of input data. The chip receives the original image data from the crossbar and the partial sum from X2 through XP_Left port. The original data is stored into the external RAMs. The image pixels from the previous first and second rows are read into the chip. The shift–and–add block adds the partial sum to the remaining weighted pixel values. Results from the adder are rounded to 8 bits, and are passed to the next Xilinx chip (X4) from the XP_Right port.

4.3.4 The structure of X4

Xilinx chip X4 functions as an I/O interface chip in this design. The image data has been filtered in both horizontal and vertical directions, but has not yet been subsampled in the vertical directional. Chip X4 implements the subsampling simply by storing only the odd numbered image rows into its memory. The X4 chip can be programmed to output the data stream in different formats. Two designs have been provided to accommodate both Gaussian and Laplacian pyramid generation.

Figure 4.7 shows the high–level structure of the chip. The 2–bit state machine configures the memory access operations and data flow directions. The output control block determines the data format of outputs to the next Xilinx chip and the crossbar. For both Gaussian and Laplacian pyramid generation, the crossbar is used to recirculate the pyramid data to the beginning of the processing path, and the output data to the crossbar must therefore have the same format as the input video data. When the entire image frame of the previous level has been processed, chip X4 reads the processed data from the memory, and pass them

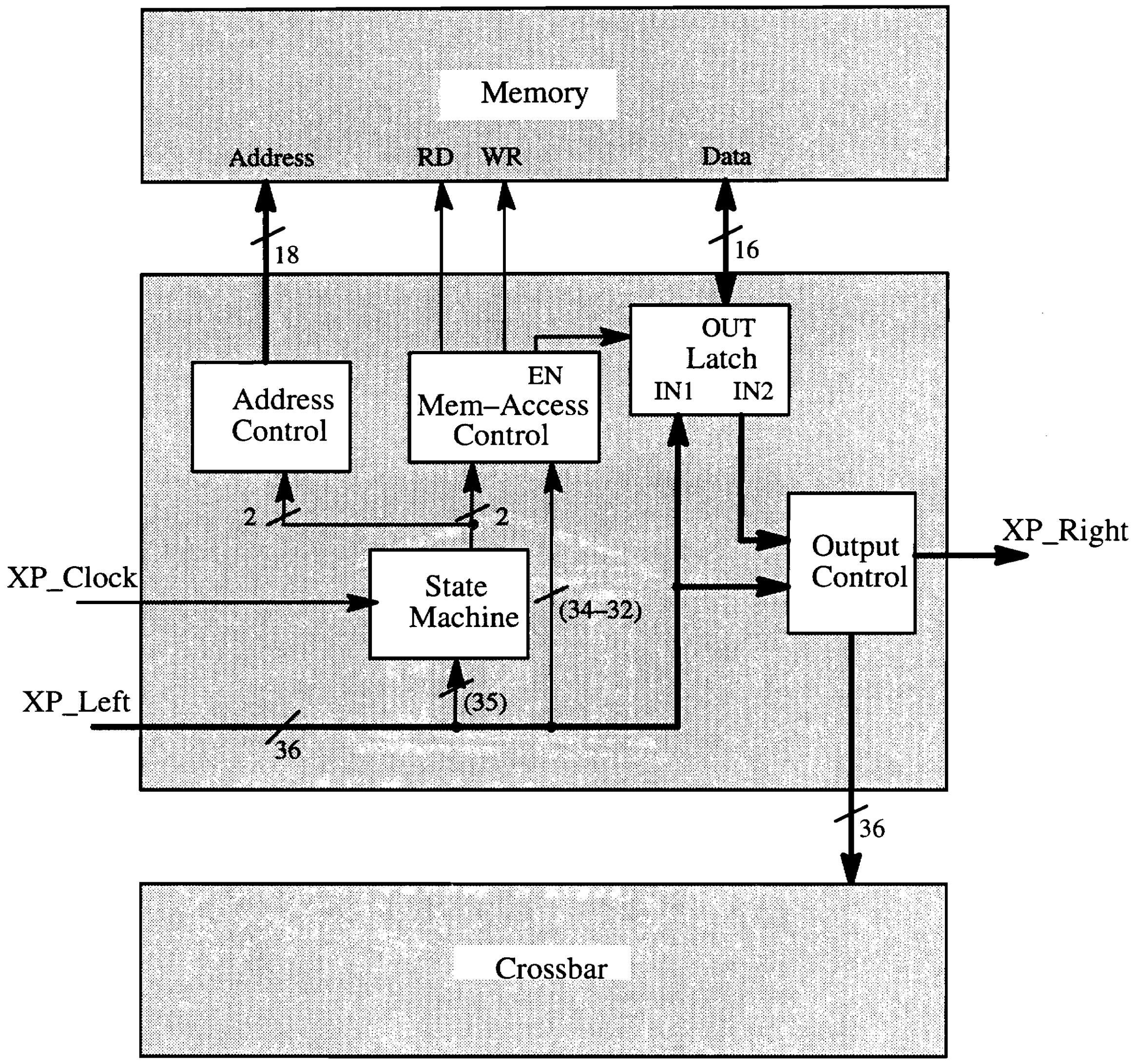


Figure 4.8. Block diagram of Xilinx device X4. This PE performs I/O interfacing between this design and the next PE (X5). It recirculates the image data back to PE X1 through the crossbar.

to X1 through the crossbar. As for the data to the next device, the data format can vary for different applications. The data from XP_Left is passed directly to chip X5 in the Laplacian pyramid generation. For Gaussian pyramid generation, the output data to the right port of X4 has the same format as to the crossbar.

5. Implementation for Laplacian Pyramids on SPLASH II

5.1 Two basic operations in RE Laplacian pyramid generation

A Laplacian pyramid is a sequence of difference images, each obtained between two successive Gaussian pyramid levels. The RE pyramid generation structure includes two basic operations: image expansion and image subtraction. The mathematical representations for these two operations are given in Equations (7) – (9).

The *EXPAND* operation can be regarded as the reverse of the *REDUCE* function in Gaussian pyramid generation. First, the image size is doubled by inserting a pixel with a grey level of '0' between two successive pixels in every row and column. The expanded image is then convolved by the same Gaussian–like weighting function. As was done for the *REDUCE* function, the *EXPAND* operation is split into two 1D identical convolutions applied to the image in both horizontal and vertical direction. The 1D operation can be expressed as below:

$$g^i(x) = 2 \sum_{m=-2}^{2} \omega(m) g^e(x-m)$$

and

$$g^e(x) = \begin{cases} g(\frac{x}{2}) & \text{if } x \text{ is even} \\ 0 & \text{if } x \text{ is odd} \end{cases}$$

where *g(x)* is the Gaussian pyramid image, and $g^i(x)$ and $g^e(x)$ are the 1D interpolated and expanded image, respectively. The above equations can also be represented in a more explicit way:

$$g^i(x) = \begin{cases} 2 \times [\omega(-2) \times g(\frac{x}{2}+1) + \omega(0) \times g(\frac{x}{2}) + \omega(2) \times g(\frac{x}{2}-1)], & \text{if } x \text{ } is \text{ } even \\ 2 \times [\omega(-1) \times g(\frac{x+1}{2}) + \omega(1) \times g(\frac{x-1}{2})], & \text{if } x \text{ } is \text{ } odd \end{cases}$$

Replacing the weighting factors *(w(–2), ... , w(2))* with their values [$\frac{1}{16}, \frac{1}{4}, \frac{3}{8}, \frac{1}{4}, \frac{1}{16}$], the equation can be simplified as follows:

$$g^i(x) = \begin{cases} \frac{1}{8} \times [g(\frac{x}{2}+1) + g(\frac{x}{2}-1)] + \frac{3}{4} \times g(\frac{x}{2}), & \text{if } x \text{ } is \text{ } even \\ \frac{1}{2} \times [g(\frac{x+1}{2} + g(\frac{x-1}{2})], & \text{if } x \text{ } is \text{ } odd \end{cases} \qquad (10)$$

Figure 5.1 illustrates the *EXPAND* operation graphically. The odd numbered pixel of the

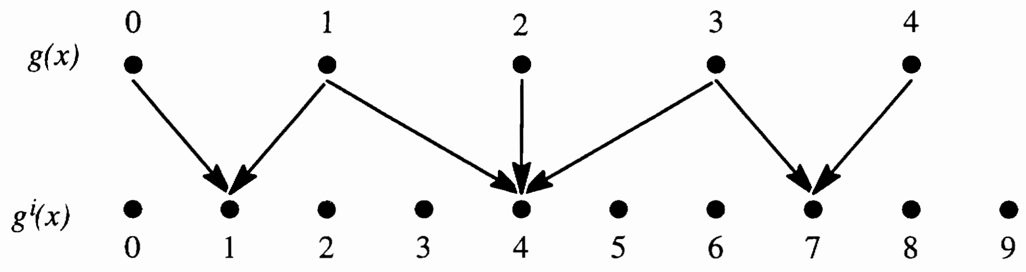


Figure 5.1. Graphical representation of 1D *EXPAND* operation.

expanded image is equal to the weighted sum of two pixels in the Guassian pyramid, and the even numbered pixel is the weighted sum of three pixels, for instance pixels 1 and 4 as shown in the figure. The 1D *EXPAND* operation can be considered as functions of 2–by–1 convolutions and 3–by–1 convolutions, with weighting functions of [$\frac{1}{2}, \frac{1}{2}$] and [$\frac{1}{8}, \frac{3}{4}, \frac{1}{8}$], respectively. Both weighting functions are normalized and symmetric as well. The edge pixels 0, 8 and 9 are not defined in Equation (8). In this design, the first and last calculated values, pixels 1 and 7, are duplicated to form the edge.

Once the pyramid is expanded to have the same size as the next higher resolution pyramid, the *subtraction* operation is applied to obtain one Laplacian pyramid level. The function is expressed in Equation (9), and is repeated here:

$$L_k^{RE}(i,j) = g_k(i,j) - g_{k+1}^{intp}(i,j)$$

5.2 Implementation for Laplacian pyramid on SPLASH II

5.2.1 Overview

The Laplacian pyramid generation system consists of two major parts: Guassian pyramid generation, and image subtraction. The system uses the recirculating pipeline structure as presented in the previous chapter to generate a Gaussian image pyramid. After the Gaussian pyramid is generated from PEs X0 – X4, the Laplacian pyramid is computed by PEs X5 – X10, as shown in Figure 5.2. The data is passed directly to PE X5, and to X7 and X8 through the crossbar. Devices X5 and X6 implement the *EXPAND* operation in the horizontal and vertical directions, respectively. The pixel–by–pixel *SUBTRACTION* operation is then implemented in chips X7 and X8 to generate a difference image. X9 and X10 reformat the images for output.

As described in the previous chapter, the data output from X4 to X5 is the image data directly from the "XP_Right" port of device X3. The 36–bit wide bus carries only 20 bits of useful information: two 8–bit image pixels and 4 control bits. Since PE X3 does not perform the subsampling function in the vertical direction, the even numbered rows of the image data are ignored in future data processing.

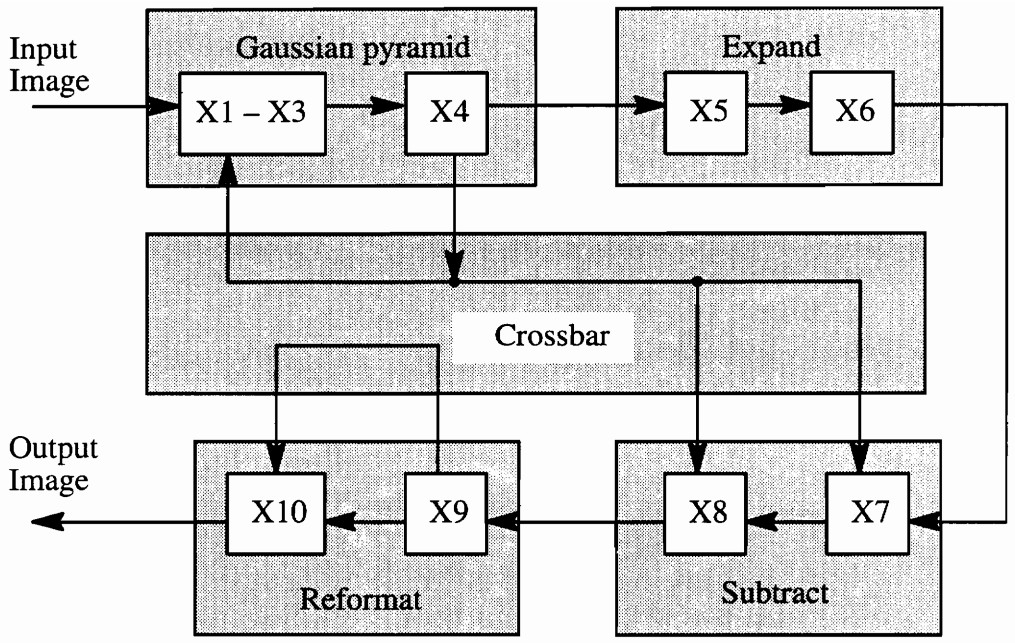


Figure 5.2. SPLASH II implementation for Laplacian pyramid.

5.2.2 The structure of X5

X5 performs the *EXPAND* function in the horizontal direction. Figure 5.3 gives a block diagram of the chip. Bit 35 of the input data, which indicates the starting and ending of an image frame, is used to initiate the state machine. Bits 34–32 carry the information which specifies the pyramid levels, and are used to configure the number of pixels per image rows. Bits 15–0 are two 8–bit Gaussian pyramid pixel values.

The state machine, which has four state, is implemented by a two–bit up–counter. It controls the data processing, memory access and output logic of the block. The shift–and–add block reads the input pixels, and calculates the corresponding weighted sums. Since the operation involves the weighted sums of two and three pixels, buffering is required to store

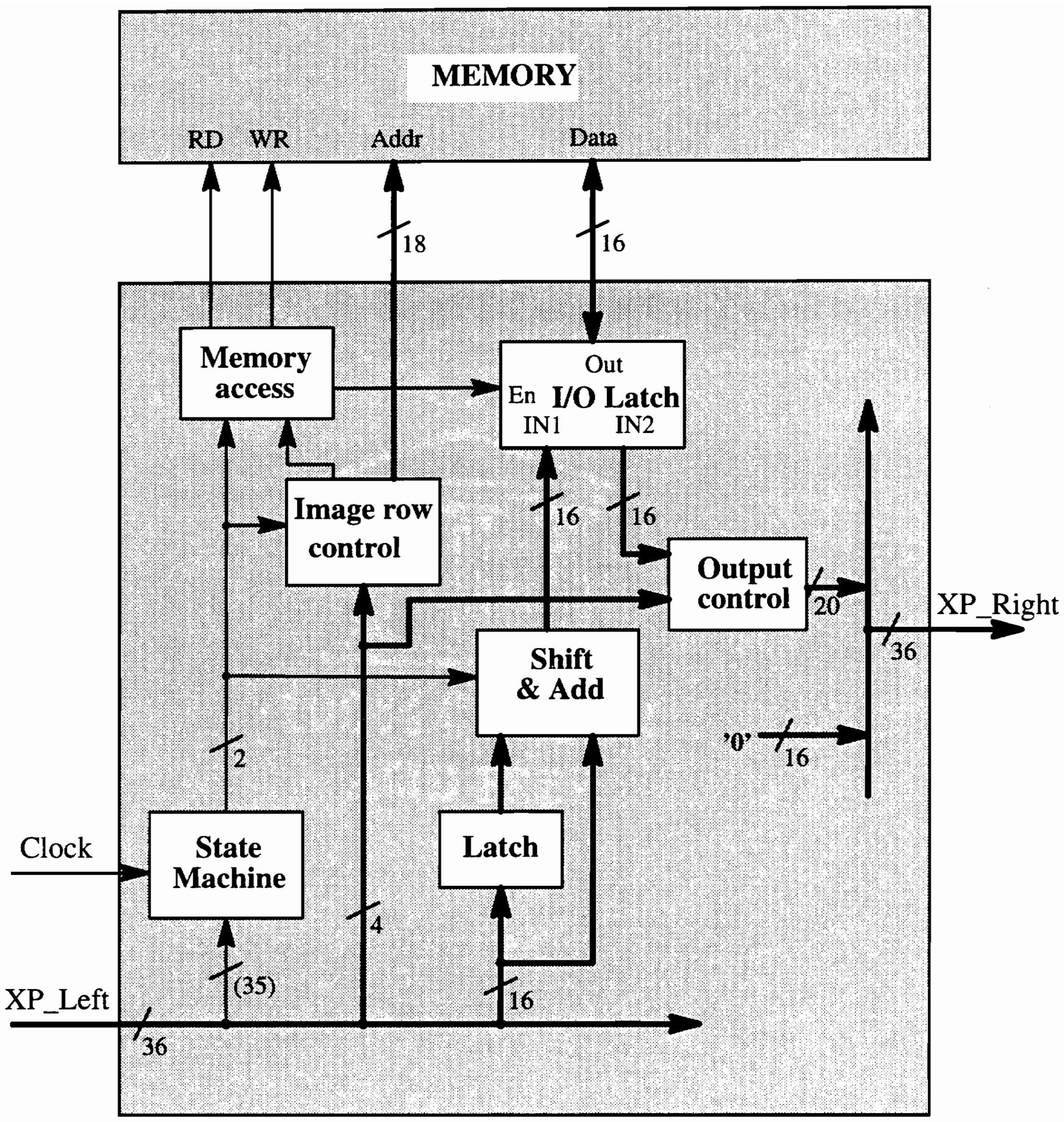


Figure 5.3. Block diagram of Xilinx device X5.

two previous input pixels. One addition is required for the 2×1 convolution, and three additions are needed for 3×1 convolution. Therefore, 8 additions are computed every four clock cycles. Also two 12–bit adders are included in the shift–and–add block.

The input data is updated every four SPLASH cycles. Therefore, every four clock cycles are called one data processing cycle. Four new pixels can be formed every data processing cycle. For instance, if current input pixels have values of *c* and *d*, and *a* and *b* are pixel values stored in the buffer from the previous processing cycle, four new pixels will be the weighted sums of pixel groups *ab*, *abc*, *bc* and *bcd*.

Since the Gaussian pyramid data is not subsampled in the vertical direction, the input image has the size of $n \times 2n$, where n is the number of pixels per image row. The chip processes all the image data but ignores data in even numbered rows. The image size is doubled to $2n \times 2n$ after the expansion in the horizontal direction. The image data is reordered by being stored into the memory first, and read back to the chip. The four newly formed pixels in odd numbered image rows are stored into the external RAM in two SPLASH cycles. A memory read following the writes produces two image pixels, which are passed to the next chip with the control data. The image data flow and memory access sequence are illustrated in Figure 5.4.

The memory address control block is implemented as a 18–bit up counter controlled by the state machine. The state machine also configures the memory access control signals. The output control block performs the edge extension by simply duplicating the edge column of each image frame.

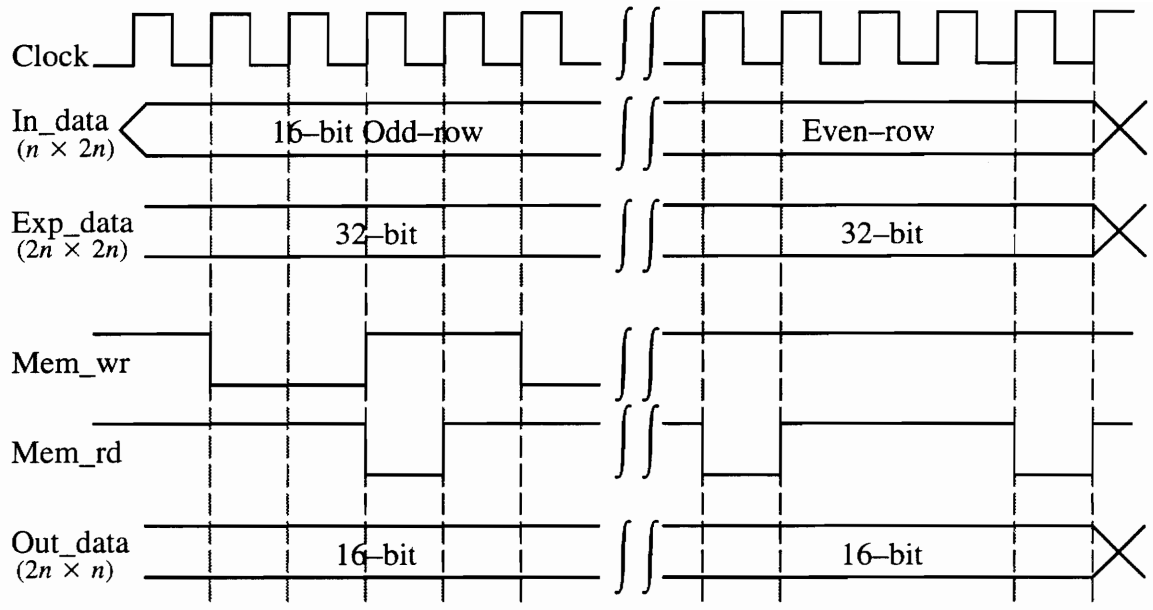


Figure 5.4. Memory access sequence of PE X5.

5.2.2 The structure of X6

PE X6 performs the *EXPAND* function in the vertical direction. The chip structure is similar to devices X2 and X3, which perform the *REDUCE* function in the vertical direction. The input data needs to be stored in memory because the convolution elements are from different image rows. However, since only two memory reads and one memory write are needed in every processing cycle, one Xilinx device is able to implement the operation.

Figure 5.5 illustrates the structure and data flow of the chip. X6 gets image data two pixels at a time from the chip X5 and issue a memory write to store them in the external memory. The chip then reads the image pixels of two previous rows from the memory, and calculates the interpolated values.

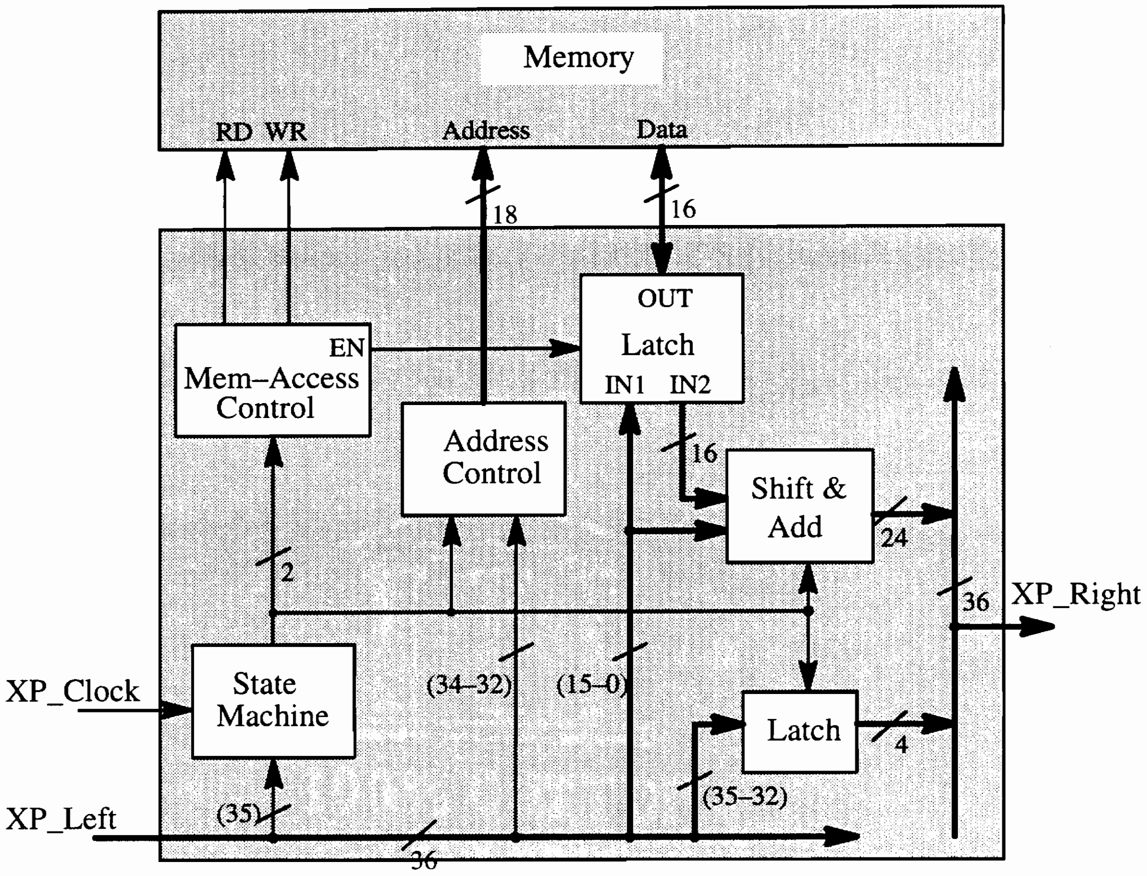


Figure 5.5. Block diagram of Xilinx device X6.

Again, the block is composed of major blocks such as the state machine, the address control block, the memory data latch and the shift–and–adder block, etc. The edge–effect correction has not been extended in the vertical direction because it will complicate the circuit much more than the horizontal edge extension does.

The output data are computed after two rows of delay on the input image. Four pixels are formed every processing cycle. But they are not successively from one but two image rows. The 36–bit wide output data path includes four controls bits and four 8–bit image pixels. The output image will have three rows less than the standard image due to the edge effect.

5.2.3 The structure of X7 and X8

Chips X7 and X8 perform a very simple subtraction operation. Together the two Xilinx chips calculate the pixel–by–pixel difference between a Gaussian pyramid level and the expanded image of the next lower resolution level of the pyramid. The requirement of two Xilinx devices is again due to memory access limitations.

Both chips receive the Gaussian pyramid data from X0 or X4 through the crossbar. The data need to be stored into the memory associated with each chip for two reasons. First, the data from the next level of pyramid is not yet ready. Second, the two input sources do not pass image data in same format. One source passes data in ordinary row–by–row order, and the other source passes four pixels from two image rows every processing cycle.

The expanded Gaussian data is passed to the chip directly from X6. Once the expanded Gaussian data is available, X7 reads in the corresponding pixel values from its memory, and calculates the pixel differences. The pixel differences are passed to chip X8 along with the other two pixels which have not been processed. X8 calculates the other two pixel differences, and sends the data to the next processing chip.

The block diagram of the chips is shown in Figure 5.6. The chip structure is relatively simple. It is composed of a state machine, a memory controller, a memory data latch and a subtracter.

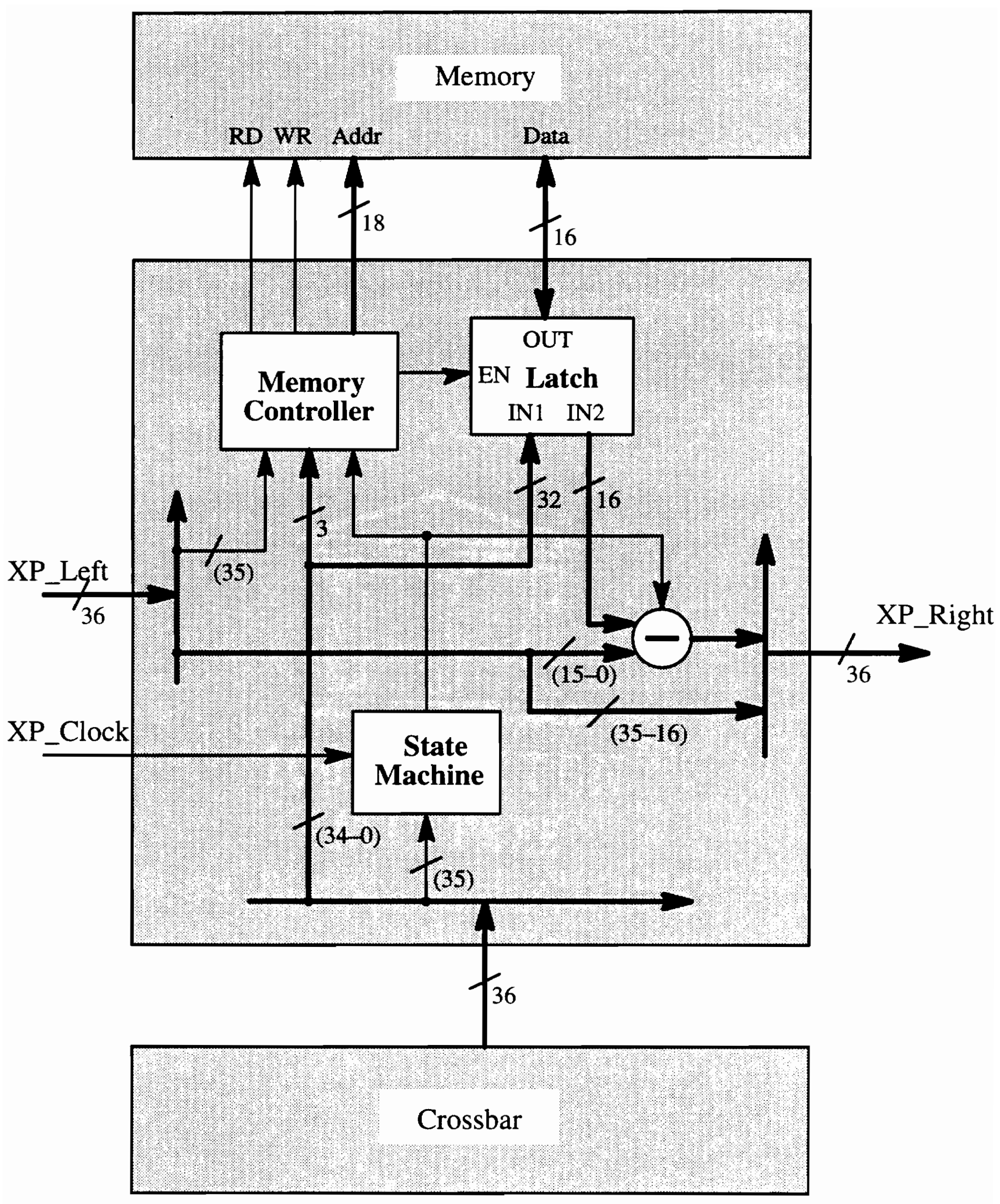


Figure 5.6. Block diagram of Xilinx devices X7 and X8.

5.2.4 The structure of X9 and X10

The image data leaving Xilinx chip X8 needs to be reordered in a format that can be displayed on a monitor. Chips X9 and X10 perform this reordering. The structure of X9 is shown in Figure 5.7. In this chip, the data is received from the XP_Left port, and is stored

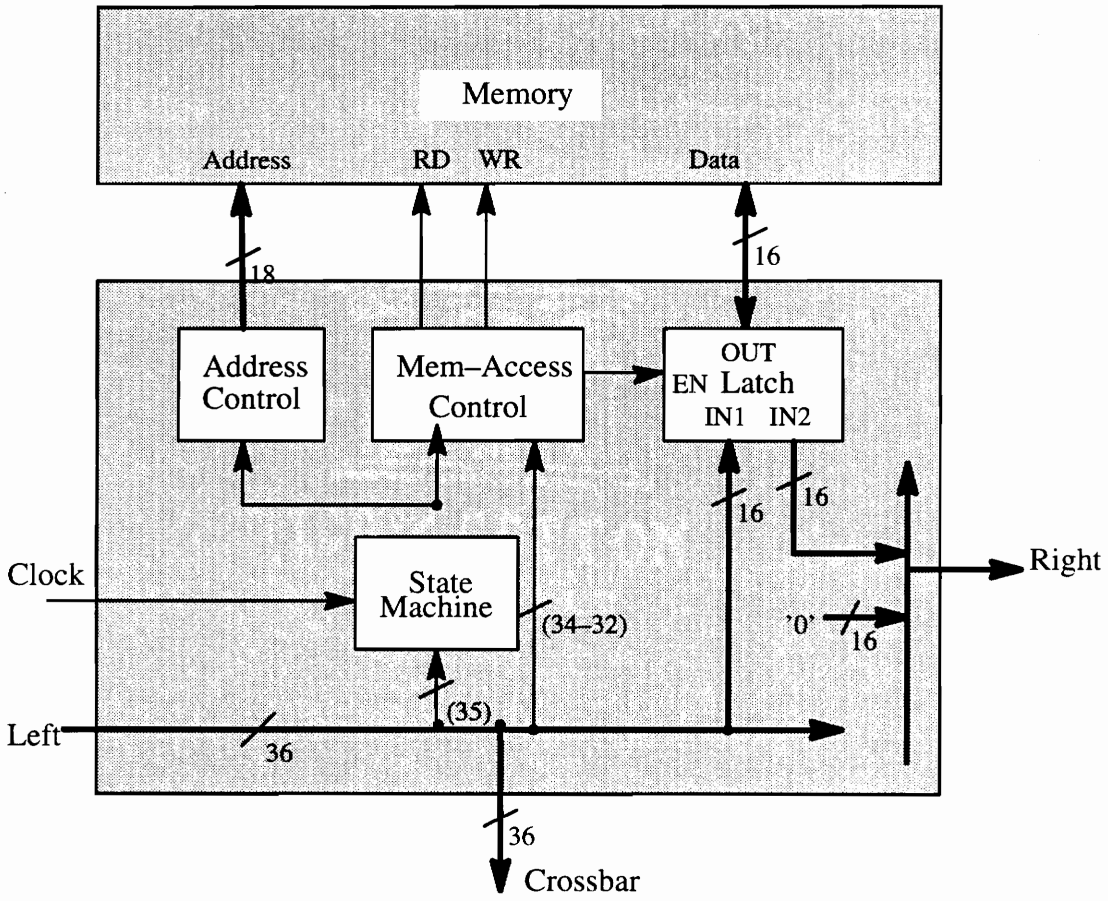


Figure 5.7. Block diagram of Xilinx chip X9.

into memory by performing two memory writes per processing cycle. The first memory write stores the lower 16–bit image data into memory locations indicating odd numbered rows. The second write stores the higher 16–bit data into even numbered rows. The chip then reads the data back from sequential address locations.

X10 receives the same input data as X9 does from the crossbar. Each of the two chips performs one memory read operation, and each read two pixel values back into the chip. Four 8–bit image pixels along with 4 tag bits are send to the display system or to the next processing chip through XP_Right port every processing cycle. The output data is therefore in the same format as the original input image.

Devices X9 and X10 can be viewed as a single module which provides the output interface for this design. It could be tailored to output the image data in many formats, as required by the next processing step, without changing the circuitry of the rest of the design. This shows one significant advantage of a design using FPGA based devices over traditional digital design.

6. Design Process for SPLASH II

6.1. Design procedure

The SPLASH II programming process includes four steps: design description, simulation, logic synthesis, and placement and routing. The Xilinx FPGAs of the SPLASH II system are programmed using VHDL. First, the application is developed by writing behavioral descriptions of algorithms in VHDL. The program is then debugged and simulated within the SPLASH II simulator. The SPLASH II simulator is also written in VHDL, and is based on the Model Technology simulator (MTI) and the Synopsys simulator. Either simulator provides the user with a full source level debugging interface.

Figure 6.1 shows the SPLASH II programming flow. Once the description model is determined to be functionally correct through simulation, the next step is the synthesis of FPGA logic from the VHDL specifications. The Synopsys logic synthesis package is supported as the SPLASH compiler. The Synopsys tool reads in the source code, synthesizes the logic, and generates an output file in EDIF netlist format. Three ways to invoke the compiler can be used: 1) the shell script *vhdl2edif*, 2) the command line based program *dc_shell*, and 3) the graphical *design_analyzer*.

The EDIF netlist needs to be translated into the Xilinx Netlist Format (XNF) before the Xilinx placement and routing process can be performed. The shell script *edif2xnf* translates the EDIF file to a XNF netlist, maps the VHDL port names to physical pins, synthesizes IOB logic and adds in the global clock buffers.

The last step of a SPLASH II application design is to place and route the XNF netlist. Xilinx software is provided for the task. The Xilinx tools include 1) the placement and rout-

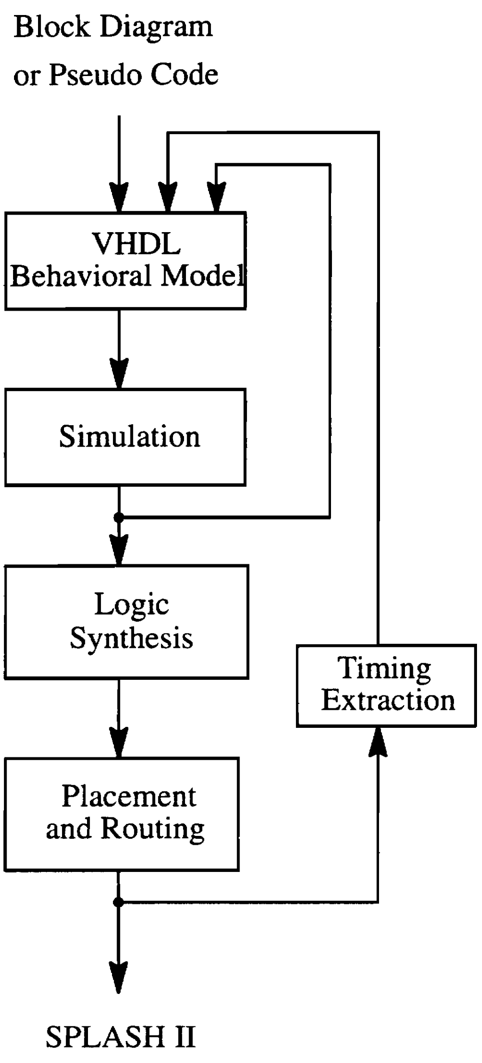


Figure 6.1. SPLASH II program development process.

ing software *ppr*, the bit stream file translator *makebits*, and the graphical FPGA editor *xact*. The *ppr* tool reads the XNF design file, and generates a placed and routed Logic Cell Array (LCA) file along with a output transcript file and a report file. The software *makebits* converts the LCA netlist into the binary FPGA configuration format. The Xilinx program *xact* is a graphical tool that allows the user manually editing the FPGA configuration. A shell script *xnf2bit* is provided for the conversion from the XNF netlist to Xilinx BIT file. The output bit stream file can be loaded into the Xilinx chip at power up.

6.2 Programming the Xilinx FPGAs

6.2.1 SPLASH II programming environment

The SPLASH II simulator provides the user with a set of VHDL models for each component of the system. These models, together with the user application program, form a behavioral model of the SPLASH II system. The SPLASH II simulator can be categorized into five major parts:

1. **The top level system definition.** A file "system.vhd" contains the entity and architecture for the top level SPLASH_System component. All of the top level signals, such as the host SBus and the SPLASH SIMD bus are defined in this file. The source code of the "system.vhd" file is listed in Appendix B.

2. **The SPLASH II configuration file.** The file "config.vhd" describes the main components of the simulator configuration. The components that must be declared in this file include the interface board, the SPLASH board array, the name of the file to be read/written by FIFOs, and the frequency of the system clock. The structural components of the interface board include the XL and XR FPGA parts. The SPLASH board array component include the number of boards and the structural model for each board. Appendix C gives a list of the "config.vhd" file of the Gaussian generation system.

3. **The SPLASH interface board.** The programmable component of the interface board contains two VHDL models "XL.vhd" and "XR.vhd". In these files, data paths and control signals for the input/output FIFOs are declared.

4. **The SPLASH II library.** The SPLASH II simulator provides a set of very useful VHDL packages for the development of user applications on SPLASH II board. These packages are TYPES, SPLASH2, COMPONENTS, ARITHMETIC, and HMACROS. The TYPES packages defines the data types used for interchip communication in the SPLASH II system. The SPLASH2 package contains a variety of constraints, data types, and functions which are specific to the SPLASH II simulator. The COMPONENTS package provides a set of useful procedures including some "Pad" procedures and data type translation procedures. The ARITHMETIC package defines the signed and unsigned arithmetic on bit vectors. The HMACROS package are a set of component declarations and simulation models for the Xilinx provided Hard Macros. The hard macro models include byte (or word) accumulators, comparators, parallel load up counters, and on–chip RAM, etc.

5. **The SPLASH II board.** The FPGA–based array board is the key component of the SPLASH II system, and is the main area where the user application code is programmed and hardware mapped. The structural model of the component consists of the SPLASH_Crossbar, Xilinx_Control_Part, Xilinx_Processing_Part and Memory_part. The structure of the board is defined in the "config.vhd" file. The complete port declaration of the Xilinx_Processing_Part and Xilinx_Control_Part is shown in Appendix D.

6.2.2 User application programming

User applications (here, Gaussian and Laplacian pyramid generation) must be manually partitioned into a set of FPGA processing elements. The VHDL behavioral model for each processing element has been written to perform the functions described in Chapters 4 and 5. A complete list of the VHDL code is shown in Appendix E. It includes a model of the Xilinx control chip, behavioral models of ten processing elements, and a user–defined package "MYPACK". The package defines a set of up counters and a D flip–flop used in the design. The entire design is synchronous, which simplifies the debugging and timing analysis.

The control element X0 reads the image data from the SPLASH SIMD bus, and sends the data to the first processing element X1 through the crossbar. The chip sets the signal X0_X16_Disable to '1' to disable chip X16 from using the crossbar. Two sets of crossbar configurations are used in Gaussian pyramid generation. X0 controls which setting to use by assigning the appropriate setting code to the vector X0_XBar_Set.

```
configuration 0
1     16
4     3

configuration 1
1     5
4     3
```

Figure 6.2. An example crossbar configuration file.

The crossbar model is simply a text file which contains up to eight chip connection settings. Figure 6.2 shows an example crossbar file, which is used in the four–chip Gaussian pyramid generation. Two configurations are defined in this file. The first line of each setting consists of the keyword "configuration" followed by the integer indicating the setting ID. The first column of the following lines indicates the output port number, and the second column is the input port specifier. The output port number ranges from 1 to 16, and corresponds to the Xilinx chip number. The input port specifier can be a single integer (0 to 16) or 5 integers (0 to 16 each). In this example, all input port specifiers are single integers, which means that the output port has single source port. Since the X0_X16_Disable signal on chip X0 is set to '1', the PE X1 receives data from X0 when crossbar configuration 0 is set, and gets data from chip X5 when configuration 1 is set. Chip X4 always receives the data from X3 through the crossbar.

The SPLASH II simulator allows the user to specify a set of initial contents for the on board memory from an ASCII input file. An initialization file is used in this design to initialize the entire memory to zero. The file contains a single line command "clear" as shown in Figure 6.3. The file name is passed as a generic parameter to the Memory_Part load file in the "config.vhd" file.

```
clear
— End of file
```

Figure 6.3. An example memory clear initialization file.

6.2.3 Hardware realization

The VHDL code has been debugged and refined under the SPLASH II environment using the Synopsys simulation tool [8]. The behavioral model was tested until functionally correct. A simple SPLASH II script was used to convert the VHDL model into a SPLASH executable netlist. The script is listed below:

```
vhdl2edif file.vhd
edif2xnf file.edif
xnf2bit file.xnf
```

Each of the above commands is described in Section 6.1. A placed and routed LCA file and a bit stream (.bit) file will be created through the process. The Xilinx program *xdelay* is used to perform a static timing analysis of the LCA file. A list of all critical path delays is produced through the timing analysis, and the maximum processing frequency can be calculated from the timing information.

The bit stream file is a SPLASH "executable" file. It can be directly mapped into the FPGAs on the SPLASH board, and may be resimulated by the SPLASH II simulator for functionality verification and timing analysis. Two host software interfaces to the SPLASH II system are provided: a C library which can be linked into an application–specific driving program, and a symbolic debug tool T2 [8]. The C library interface is used to simulate this design.

7. Results and Future Enhancements

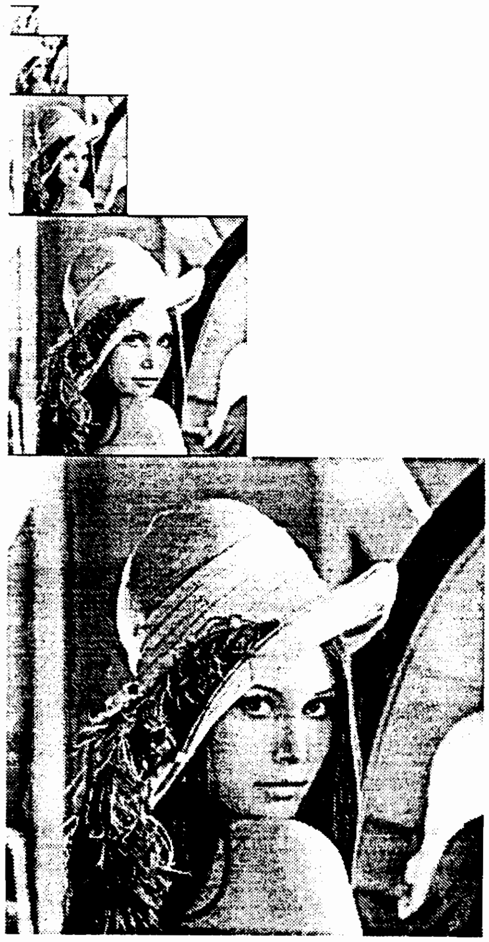
7.1. Results

The objective of implementing high–speed image pyramid generation on the SPLASH II system has been accomplished. Two types of image pyramids, Gaussian and Laplacian, are generated based on the definition in [2]. This section evaluates the system from three perspectives: image results, system performance, and design flexibility.

7.1.1 Image Results

SPLASH II has been programmed to process 512×512 input image frames. It generates 5 levels of the Gaussian pyramid and 4 levels of the Laplacian RE pyramid. A C language–based software model was built to generate image pyramids based on the same algorithms. The image pyramids obtained from SPLASH system and the software model have been compared, and are the same. Figure 6.4 shows an image of Gaussian and Laplacian pyramids generated from the SPLASH II system.

The two outermost rows and columns of the Guassian image pyramid are corrupted due to the edge effect caused by the 5×5 filter convolution, as described in Section 4.3.2. This effect is minimized in the horizontal direction by simply duplicating the edge pixel values obtained by convolution. However, the effect has been ignored in the vertical direction since the design will be significantly complicated by performing the edge extension. Therefore, each Gaussian pyramid is two rows less than expected after filtering and subsampling. The effect doubles when the Gaussian pyramid is used to generate a Laplacian pyramid. The convolution in the *EXPAND* operation of Laplacian pyramid generation will corrupt another


(a)

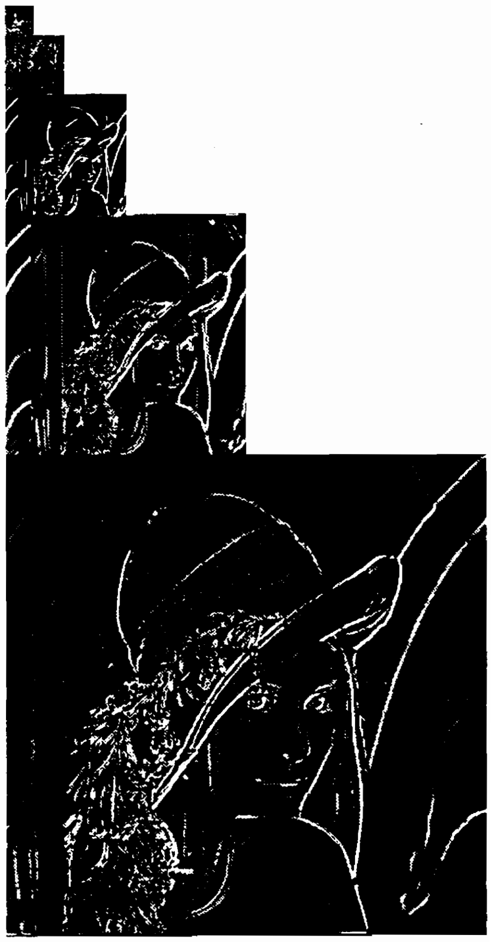

(b)

Figure 7.1. Pyramid images generated using the SPLASH II system.
(a) Gaussian pyramid and (b) Laplacian pyramid.

three pixels each image column; therefore, each Laplacian pyramid is seven rows less than expected. Fortunately, the loss of useful data will only impact the uppermost rows of the image because input images contain only 480 rows of valid video information, and the lower 32 rows of data are filled by pixels with '0' values.

7.1.2 System performance

The system has been designed to generate Gaussian and Laplacian pyramids at high speed. The linearly connected Xilinx FPGAs on the SPLASH board are programmed in a fully pipelined structure. The nine–chip Gaussian pyramid generation system processes the image data and produces image pyramids in real time. The standard video camera collects 30 frames of 512×512 images per second. Therefore, the system needs to be able to operate at minimum pixel rate of 7.87 MHz.

For the five–chip Gaussian pyramid generation system and the eleven–PE Laplacian system, the Gaussian pyramid data is recirculated within the pipeline during processing. The system processes every other video image frame, which means 15 image frames will be processed per second. It takes only ($\frac{4}{3} \cdot \frac{1}{30}$) seconds to process one image frame. Therefore, the system can operate at a reduced pixel rate. A minimum clock frequency of 5.25 MHz is required.

The Xilinx *xdelay* software reports detailed timing information about the design. The report contains the worst case path delays, so that the user can put more effort into improving the speed of those critical paths. Figure 7.2 lists the maximum processing frequencies of X1 to X10, as obtained from running the *lca* file through the *xdelay* software. The minimum pixel rate of all chips is 6.6MHz, which is above the calculated minimum of 5.25 MHz. The system therefore operates as expected.

X1	X2	X3	X4	X5	X6	X7	X8	X9	X10
10.3M	7.5M	7.2M	7.4M	11.7M	6.9M	7.5M	7.2M	6.8M	6.6M

Figure 7.2. A list of maximum operating clock frequencies of PEs in (MHz).

The design implements a large number of arithmetic operations. In the Gaussian pyramid generation, an $n \times n$ image will require $(n - 4) \times n$ convolutions in both horizontal and vertical directions to generate the next level of the Gaussian pyramid. Each convolution contains five additions. Therefore, the total number of additions implemented in this 5–level Gaussian pyramid design can be calculated as below:

$$N_G = \sum_{n=5}^{9} [(2^n - 4) \times 2^n \times 2 \times 5]$$

where 2^n is the number of pixels per row or column of each pyramid level, and N_G is number of additions per Gaussian pyramid. From this equation, the Gaussian pyramid generation system will compute 3.55M arithmetic operations for every 5–level image pyramid.

Both two operations in the Laplacian pyramid generation require arithmetic operations. The *EXPAND* operation is composed of a 2×1 convolution and a 3×1 convolution, where each contains 1 and 3 additions, respectively. It contains $[(n-1) \times n + 3 \times (n-2) \times n]$ computations to expand an $n \times n$ image. The total number of additions required for four levels of Gaussian image expansion can be calculated as follows:

$$N_L^e = \sum_{n=5}^{8}[(2^n - 1) \times 2^n + 3 \times (2^n - 2) \times 2^n]$$

$$= \sum_{n=5}^{8}(4 \times 2^n - 7) \times 2^n$$

$$= \sum_{n=5}^{8}(2^{2n+2} - 7 \times 2^n)$$

where N_L^e is the number of additions per Laplacian pyramid. Approximately 0.345 M arithmetic computations are calculated in the *EXPAND* operation. In addition to computations required here, the Laplacian pyramid generation systems contains about 0.353 M pixel–by–pixel subtractions. The number of subtractions per Laplacian pyramid, N_L^s is simply computed using the following equation:

$$N_L^s = \sum_{n=6}^{9}(2^{2n})$$

In total, the 11–PE Laplacian pyramid generation system requires approximately 4.25 million arithmetic computations to process one 512×512 video image. Figure 7.3 lists the number of fixed–point additions and subtractions per second implemented in the three designs and the supporting arithmetic operations for the memory access.

SPLASH II DESIGN	Arith. Oper. (M/pyramid)	Oper. Speed (frames/s)	Arith. Oper. (M/sec.)	Supporting Arith. (M/pyramid)
9–chip Gaussian	3.55	30	106.5	0.69
5–chip Gaussian	3.55	15	53.25	0.69
11–chip Laplacian	4.25	15	63.75	1.38

Figure 7.3. Number of fixed–point additions/subtractions implemented in each design.

7.1.3 Design Flexibility

The pipeline feature of SPLASH II Xilinx devices and the reconfiguration feature of the SPLASH crossbar structure provide wide flexibility for user applications in image processing. The pyramid generation system has been designed by programming XC4010 FPGA devices on the SPLASH II array board. Each Xilinx chip can be tested and debugged separately. The VHDL software models can be functionally simulated and then mapped directly into the Xilinx chip by synthesizing the models and converting the netlist into a Xilinx formatted bit stream file. Compared with the regular ASIC (application specified integrated circuit) design cycle, programming the Xilinx elements on the SPLASH II system results in a shorter design cycle and helps avoid manufacturing errors. The re–programmability of the Xilinx devices is therefore well suited for research tasks and fast application development.

The I/O interface and data flow format can be changed by simply rewriting the VHDL model of the part, and then mapping this into the appropriate Xilinx chip. For example, the system can be programmed to process image pixels one at a time by adding extra buffering logic in the first or the last chip in the processing element chain. This makes it easy to couple the design with other image processing components.

The design is comparable to the PYR chip described in Chapter 3. As a commercial VLSI chip, the PYR chip is more sophisticated than this design. Not only can it generate Gaussian and Laplacian pyramids, but also some related structures, such as the sub–band pyramid and the gradient pyramid are supported through appropriate choices of filter kernels. One advantage of the implementation on SPLASH system is that the SPLASH array board provides internal memories to store the intermediate results while the PYR chip requires external dual–ported memories.

7.2 Future Enhancements

Improvements can always be made to a design. These include system refinement, new technology, or alternative designing algorithm. The possible enhancements in this project are evaluated from two perspectives: enhancements to the SPLASH II system, and improvements on the pyramid generation design.

7.2.1 Suggested improvements on SPLASH II system

The SPLASH II system provides 1/2 MB (256K $\times$ 16) external static memory associated with each processing element. This arrangement is well suited for image processing purposes because of the large amount of data. However, one weakness of the SPLASH II system is that the memory is not dual–ported. Memory write after read may cause a data hazard (known as a WAR hazard). Since many operations are of the form read–operate–write, the single–port constraint seriously limits the utilization of the Xilinx devices, and significantly affects system performance.

Dual–ported memory access would require a considerable increase in complexity of data and address buses. One compromise is to have separate memory–read and memory–write data buses so that the WAR hazard can be eliminated. In this way, two memory reads and two memory writes would be supported in every processing cycle.

A big part of the design effort and circuit logic deals with data communication between the local memory blocks of Xilinx processing elements. The lack of global system memory can be considered another drawback of the SPLASH II system. The memory system of SPLASH II could be redesigned to include a block of global memory, to which all SPLASH FPGAs have access by the crossbar. In this case, the local RAMs could function as cache memory for each chip, storing some intermediate results, and the global memory could be used to store image data to be used by several chips. The usage of global memory would also solve some of the problems caused by the single–ported memory access as stated previously.

7.2.2 A pyramid generation system for images with arbitrary size

The system is designed to work with input images of fixed size with 512×512 pixels. Without great difficulty, the chips could be programmed to process images of arbitrary size. However, the number of pixels per image row and column cannot easily exceed 512 because of the data width of internal up–counters used in this design. Although it is not necessary to have image dimensions which are powers of two, even numbers of pixels per row and column are always required for each level of the pyramid so that each level has half the number of pixels in each direction than its previous level.

7.2.3 Timing analysis and extraction

This design has emphasized the development of behavioral models in VHDL, and model simulations and debugging with the SPLASH II simulator. The performance obtained from the current system is reasonably good. However, the standard video rate is 30 frames of images per second, which implies a pixel processing frequency of 7.8 MHz. The slowest chip of this design can process image data at a rate of 6.6 MHz, which implies a difference of 19 ns in the clock period. This means that the system is not guaranteed to operate correctly at full video rate.

More timing analysis and extraction could be done to improve the design performance. There are usually two ways to achieve this: changing the VHDL model, and manually editing the schematic.

When the timing analysis shows that a design is not close to meeting requirements, remodeling the design in VHDL is usually required. Buffering logic or wait states may be needed to break a long timing path into two or more clock cycles. When slight timing problems occur only in a few critical paths, a more effective way is to go into the schematic, and edit the circuit directly.

The Xilinx *xdelay* program reports detailed timing information about the design, which can be used for overall performance analysis. The user should be aware that the critical paths reported are not always true. For instance, an output signal may be driven by a signal which is available long before the current clock edge. Some timing analysis tools such as *quickpath* of Mentor Graphics allow the user to set multi–clock cycle inputs.

In summary, one suggested improvement for the design described in this thesis is to improve the calculated operational speed of the system.

7.4.4 FSD Laplacian pyramid

This design implements the RE Laplacian pyramid generation on the SPLASH II system, as described in Section 5.1. Another useful type of Laplacian pyramid is the FSD (filter–subtract–decimate) pyramid. Each level of the FSD pyramid is defined as the difference between the corresponding level of the Gaussian pyramid and the next lower resolution Gaussian pyramid prior to subsampling.

No new models need to be developed to generate the FSD pyramid, but more Xilinx devices are required because the system cannot take the advantage of subsampling and filtering the image simultaneously. The FSD pyramid generation system requires no *EXPAND* operations, but doubles the number of additions implemented in the Gaussian pyramid generation and the image *SUBTRACTION* operation.

The FSD pyramid is appropriate for computer vision application because it is simple and requires no reinterpolation. The RE pyramid is usually used for image compression or enhancement because of its reconstructable feature.

8. Conclusions

This thesis has described the implementation of a system that can generate two types of image pyramids: the Gaussian pyramid and the Laplacian pyramid. The project utilizes the SPLASH II attached processor, which is a reconfigurable platform based on FPGAs. Seven unique Xilinx designs are developed. The design was first modeled in VHDL, and then simulated and synthesized to a gate list using SPLASH II simulator and the Synopsys syntheses tool. The gate list was then mapped onto Xilinx XC4010 FPGA architectures, and tested functionally.

The resulting system is capable of generating five levels of Gaussian pyramids at either half or full standard video rate (30 images per second) by using 5 or 9 FPGA chips, respectively. The 5–chip design uses a recirculating pipeline structure, and the 9–chip design implements a hybrid pipeline architecture. Four–level Laplacian pyramids are produced using an 11–FPGA system. This operates at half of the standard video rate. If two SPLASH II processor boards are available, the system can be programmed to use both of them to process alternate image frames.

The overall FPGA chip utilization of the design is approximately 50% – 60%. The system is not able to make full use of each FPGA chip due to the extensive usage of the single–port local RAMs. A large amount of computations are involved in the system. Approximately 4.25 million fixed–point additions and subtractions are computed to generate one Gaussian and Laplacian pyramid.

This research has demonstrated the ability of SPLASH II to perform a very useful high–speed image processing task. Reconfigurable custom–computing platforms provide their users with an attractive alternative to current processors, and therefore will become increasingly popular in the future design world.

References

1. J. M. Arnold, D. A. Buell and E. G. Davis, "SPLASH 2", *Proceedings: ACM Symposium on Parallel Algorithms and Architectures*, pp. 316–322, 1992.

2. G. S. VanDerWal and P. J. Burt, "A VLSI Pyramid Chip for Multiresolution Image Analysis", *International Journal of Computer Vision, 8:3*, pp.177–189, 1992.

3. A. Rosenfeld, "Some Useful Properties of Pyramids", *Multiresolution Image Processing and Analysis*, ed. by A. Rosenfeld, (Springer–Verlag Berlin), pp. 1–3, 1984.

4. T. J. Olson and R. J. Lockwood, "Fixation–based Filtering", *Proceedings: SPIE Intelligent Robots and Computer Vision XI*, vol. 1825, pp. 685–695, 1992.

5. P. J. Burt, "The Pyramid as a Structure for Efficient Computation", *Multiresolution Image Processing and Analysis*, ed. by A. Rosenfeld, (Springer–Verlag Berlin), pp. 4–35, 1984.

6. A. Tarmaster, "Median and Morphological Filtering of Images in Real Time Using an FPGA–based Custom Computing Platform", M. S. Thesis, Virginia Polytechnic Institute and State University, 1994.

7. A. L. Abbott, P. M. Athanas, L. Chen and R. L. Elliott, "Finding Lines and Building Pyramids with Splash 2", to appear in *Proceedings*: *IEEE Workshop on FPGAs for Custom Computing Machines*, April, 1994.

8. J. M. Arnold and M. A. McGarry, *Splash 2 Programmer's Manual*, Supercomputing Research Center, Bowie, Maryland, 1993.

9. Xilinx, Inc., *The Programmable Logic Data Book*, San Jose, California, 1994.

10. 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, 1983.

11. G. S. VanDerWal and J. O. Sinniger, “Real Time Pyramid Transform Architecture”, *Processing: SPIE Intelligent Robots and Computer Vision*, Vol. 579, pp. 300–305, 1985.

12. J. L. Crowley, “A Multiresolution Representation for Shape”, *Multiresolution Image Processing and Analysis*, ed. by A. Rosenfeld, (Springer–Verlag Berlin), pp. 169–189, 1984.

13. S. G. Mallat, “A Theory for Multiresolution Signal Decomposition: The Wavelet Representation”, *IEEE Transactions on Pattern Analysis and Machine Intelligence,* vol.11, no. 7, pp. 674–693.

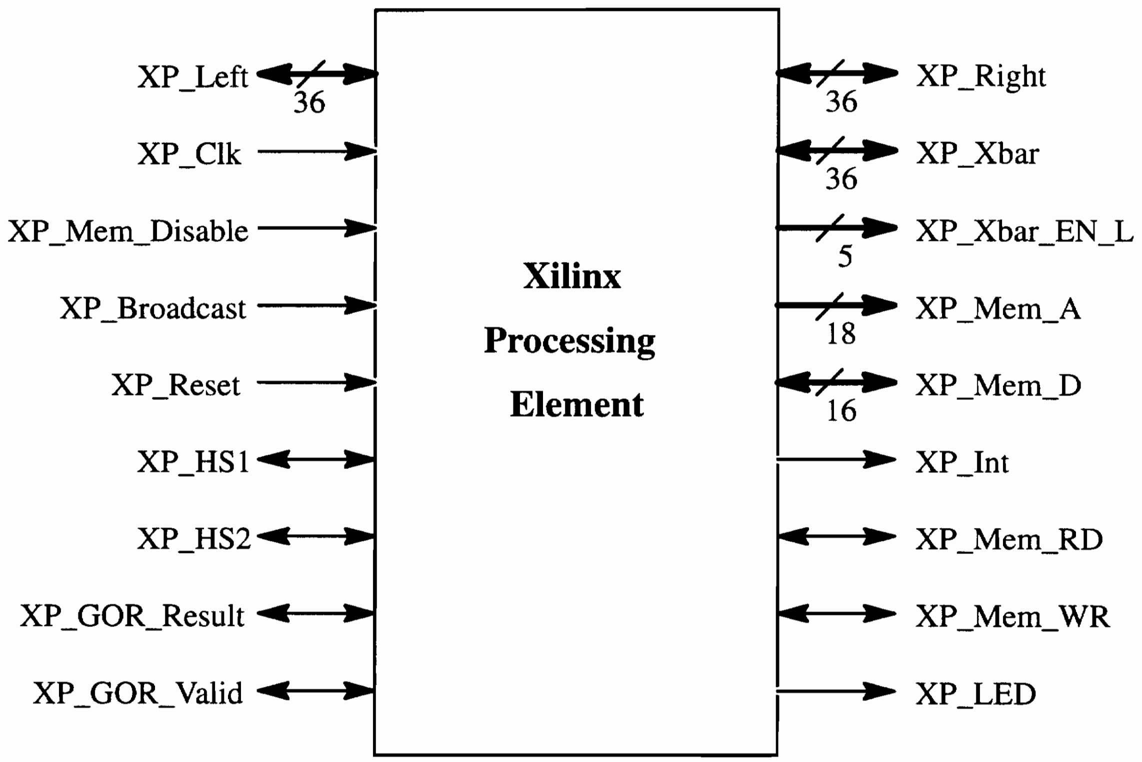
14. A. Merigot, P. Clermont and J. Mehat, “A Pyramidal System for Image Processing”, *Pyramidal Systems for Computer Vision*, ed. by A. Rosenfeld, (Springer–Verlag Berlin), pp. 109–124, 1986.

15. M. Shneier, “Multiresolution Feature Encoding”, *Multiresolution Image Processing and Analysis*, ed. by A. Rosenfeld, (Springer–Verlag Berlin), pp. 190–199, 1984.

9. Appendix A

External Connection of a SPLASH II Processing Element

The following is a schematic symbol for the processing element of the SPLASH II board. The detailed definition of each signal is described in [8].



10. Appendix B

VHDL List of "System.vhd"

```
__*********************************************************
—*                    SYSTEM.VHD                          *
__*********************************************************

— Top level of the SPLASH II system

library SPLASH2, INTERFACE, S2BOARD;
use SPLASH2.TYPES.all;
use SPLASH2.SPLASH2.all;
use SPLASH2.COMPONENTS.all;
use INTERFACE.INTERFACE_COMPONENTS.all;
use SPLASH2.ARITHMETIC.all;

ENTEITY Splash_System IS
        Generic (First_Splash_Board: Integer :=0;

                 Last_Splash_Board: Integer :=0);
END Splash_System;

ARCHITECTURE Structure of Splash_System is

        type IBoard_Type is array (0 to BOARDS_PER_SYSTEM_MAX) of DataPath;
        subtype HandShake is RBit3_Vector (0 to XLINX_PER_BOARD);
        type AllHandShake is array (0 to BOARDS_PER_SYSTEM_MAX) of HandShake;
        type Slot_Type is array (0 to BOARDS_PER_SYSTEM_MAX) of Bit_Vector(4
                        downto 0);

        signal SYS_CLK        : Bit;
        signal GOR_Result     : OCBit;
        signal GOR_Valid      : OCBit;
        signal SIMD_Bus       : DataPath;
        signal R_Bus          : DataPath;
        signal SIZE           : Bit_Vector(4 downto 0);
        signal Set_Empty1     : Boolean := FALSE;
        signal Set_Empty2     : Boolean := FALSE;
        signal Set_Empty3     : Boolean := FALSE;
        signal Set_Full1      : Boolean := FALSE;
        signal Set_Full2      : Boolean := FALSE;

        signal InterBoard     : IBoard_Type;
        signal Mem_Disable    : Bit := '0';
        signal S2Ints         : Bit_Vector(0 to BOARDS_PER_SYSTEM_MAX-1);
        signal Slot           : Slot_Type;
        signal RDir           : Bit;
        signal Broadcast      : Bit;
        signal Reset          : Bit;
        signal HS0            : HS0Vector;
        signal HS0Reg         : Bit_Vector(0 to Xilinx_Per_Board);
        signal HS1            : Bit_Vector(0 to BOARDS_PER_SYSTEM_MAX-1);
        signal SRC_Bus        : SRC_Bus;
        signal Last_Board     : Integer;

        signal Int_En         : Bit_Vector (0 to BOARDS_PER_SYSTEM_MAX -1);
        signal Int_Mask       : IntMaskVector;
        signal Int_Status     : IntStatusVector;

BEGIN
        IFACE : Interface_Board

                Port_Map (

                        SYS_CLK         => SYS_CLK,
                        GOR_Result      => GOR_Result,
                        GOR_Valid       => GOR_Valud,
                        SIMD_Bus        => SIMD_Bus,
                        INT_REG         => S2Ints,
                        R_Bus           => R_Bus,
                        RDir            => RDir,
                        SIZE            => SIZE,
                        Last_Board      => Last_Board,
                        Set_Empty1      => Set_Empty1,
                        Set_Empty2      => Set_Empty2,
                        Set_Empty3      => Set_Empty3,
                        Set_Full1       => Set_Full1,
                        Set_Full2       => Set_Full2
                        );

        SPLASH : Splash2_Boards
                Port Map (
                        SB_SIMD         => SIMD_Bus,
                        SB_R            => R_Bus,
                        SB_Clk          => Sys_Clk,
                        SB_Size         => Size,
                        SB_RDir         => RDir,
                        SB_Broadcast    => Broadcast,
                        SB_Interrupt    => S2Ints,
                        SB_GORValid     => GOR_Valid,
                        SB_GORResult    => GOR_Result,
                        SB_Reset        => Reset,
                        SB_HS0          => HS0,
                        SB_HS1          => HS1,
                        SB_Int_En       => Int_En,
                        SB_Int_Mask     => Int_Mast,
                        SB_Int_Status   => Int_Status,
                        SB_Last_Board   => Last_Board
                        );
        Pad_Output ( HS0(0), HS0Reg);
        TEST: PROCESS
        BEGIN
                WAIT UNTIL Sys_Clk'EVENT AND Sys_Clk = '1';
                Broadcase       <= NOT Broadcast;
                HS1             <= NOT HS1;
                HS0Reg(1)       <= NOT HS0Reg(1);
                IF ( S2Ints(0) = '1' ) THEN
                        Int_En(0)       <= '1';
                        Int_Status(0)   <= "0000000000000000";
                        Int_Mask(0)     <= "0000000000000000";
                ELSE
                        Int_En(0)       <= '0';
                        Int_Mask(0)     <= "1111111111111111";
                END IF;
        END PROCESS TEST;
END structure;
```

11. Appendix C.

VHDL List of "Config.vhd"

```
__*******************************************
—*                    CONFIG.VHD                    *
__*******************************************

— Sample configuration file for SPLASH II system

library S2Board, Interface;

configuration TOP of Splash_System is
        for Structure

          for IFACE : Interface_Board
                use entity interface.Interface_Board (structure)

                Generic Map  (input_file1     => "xinput1",
                              input_file2     => " ",
                              J3_file1        => " ",
                              output_file1    => "xoutput1",
                              output_file2    => " ",
                              File_Type       => Hex,
                              Clock_Freq      => 20);

                for Structure
                  for all: XL
                        use entity interface.XL(Valid);
                  end for;
                  for all: XR
                        use entity interface.XR(Valid);
                  end for;
                end for;
          end for;

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

                Generic Map (Number_Of_Boards    => 1);

                for Structure
                  for SBOARDS (0)
                        for BD : Splash2_Board
                          use entity work.Splash2_Board(Structure);
                          for Structure
                                for XBAR : Splash_Crossbar
                                  use entity S2Board.Splash_Crossbar (Behavior)
                                  Generic Map (
                                        Config_File     => "xcrossbar"
```

```
            );
end for;

for all : Xilinx_Control_Part
  use entity work.Xlinx_Control_Part (behavior);
end for;

for XPARTS (1)
  for all : Xilinx_Processing_Part
        use entity work.Xlinx_Processing_Part(X_one);
  end for;
  for all : Memory_Part
        use entity S2Board.Memory_Part(Dynamic)
        Generic Map ( Load_file => "memory.ini");
  end for;
end for;

for XPARTS (2)
  for all : Xilinx_Processing_Part
        use entity work.Xlinx_Processing_Part(X_two);
  end for;
  for all : Memory_Part
        use entity S2Board.Memory_Part(Dynamic)
        Generic Map ( Load_file => "memory.ini");
  end for;
end for;

for XPARTS (3)
  for all : Xilinx_Processing_Part
        use entity work.Xlinx_Processing_Part(X_three);
  end for;
  for all : Memory_Part
        use entity S2Board.Memory_Part(Dynamic)
        Generic Map ( Load_file => "memory.ini");
  end for;
end for;

for XPARTS (4)
  for all : Xilinx_Processing_Part
        use entity work.Xlinx_Processing_Part(X_four);
  end for;
  for all : Memory_Part
        use entity S2Board.Memory_Part(Dynamic)
        Generic Map ( Load_file => "memory.ini");
  end for;
end for;

for XPARTS (5 to 16)
  for all : Xilinx_Processing_Part
  use entity S2Board.Xilinx_Processing_Part(Left_to_right);
  end for;
  for all : Memory_Part
        use entity S2Board.Memory_Part(Blank);
```

```
                              end for;
                            end for;

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

12. Appendix D

FPGA Entity Declaration

```
—              Xilinx_Control_Part Entity Declaration              —

ENTITY Xilinx_Control_Part IS
        Generic (BD_ID              : Integer := 0;
                 PE_ID              : Integer:= 0);

        Port  (X0_SIMD              : inout DataPath;
               X0_XB_Data           : inout DataPath;
               X0_Mem_A             : inout MemAddr;
               X0_Mem_D             : inout MemData;
               X0_Mem_RD            : inout Bit;
               X0_Mem_WR            : inout Bit;
               X0_Mem_Disable       : in Bit;
               X0_GOR_Result_In     : inout RBit3_Vector(1 to 16);
               X0_GOR_Valid_In      : inout RBit3_Vector(1 to 16);
               X0_GOR_Result        : out Bit;
               X0_GOR_Valid         : out Bit;
               X0_Clk               : in Bit;
               X0_XBar_Set          : out Bit_Vector(0 to 2);
               X0_X16_Disable       : out Bit;
               X0_XBar_Send         : out Bit;
               X0_Int               : out Bit;
               X0_Broadcast_In      : in Bit;
               X0_Broadcast_Out     : out Bit;
               X0_Reset             : in Bit;
               X0_HS1, X0_HS2       : inout RBit3;
               X0_LED               : out Bit);
END Xilinx_Control_Part;
```

—   Xilinx_Processing_Part Entity Declaration   —

```
ENTITY Xilinx_Processing_Part IS
        Generic (BD_ID                  : Integer := 0;
                 PE_ID                  : Integer:= 0);

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

END Xilinx_Processong_Part;
```

13. Appendix E

Application Program Listing

The following are the VHDL code listings of FPGA devices X0, X1 – X10, and a user–defined package, in which up–counters and D Flip–Flops are defined.

```
—              Xilinx_Control_Part X0               —

ARCHITECTURE behavior of Xilinx_Control_Part IS
        SIGNAL x0_in        : Bit_Vector(35 downto 0);
        SIGNA L x0_out      : Bit_Vector(35 downto 0);
BEGIN
        PROCESS
        BEGIN
                wait until X0_Clk'event and X0_Clk = '1';
                Pad_Input (X0_SIMD, X0_in);
                Pad_Output(X0_XB_Data, X0_out);

                X0_out <= X0_in;
                X0_X16_Disable <= '1';

        — Crossbar configuration 0 is used when the input image data is valid;
        — Configuration 1 is used during the gap between two frames
                IF X0_in(35) = '1' THEN
                        X0_XBar_Set <= "000";
                        X0_XBar_EN_L <= '0' ;
                        X0_XBar_Send <= '1';
                ELSE
                        X0_XBar_Set <= "100"
                        X0_XBar_EN_L <= '1';
                        X0_XBar_Send <= '0';
                END IF;
        END PROCESS;
END behavior;
```

```
—            Xilinx_Processing_Part X1                  —

libary SPLASH2;
use SPLASH2.TYPES.all;
use SPLASH2.ARITHMETIC.all;
use SPLASH2.SPLASH2.all;
use SPLASH2.COMPONENTS.all;
use work.mypack.all;
ARCHITECTURE X_one OF Xilinx_Processing_Part IS
        SIGNAL right        : Bit_Vector(35 downto 0);
        SIGNAL xbar         : Bit_Vector(35 downto 0);
        SIGNAL temp1        : Bit_Vector(35 downto 0);
        SIGNAL temp2        : Bit_Vector(7 downto 0);

        SIGNAL Rega1        : Unsigned (11 downto 0);
        SIGNAL Rega2        : Unsigned (11 downto 0);
        SIGNAL Regb1        : Unsigned (11 downto 0);
        SIGNAL Regb2        : Unsigned (11 downto 0);
        SIGNAL Regc1        : Unsigned (11 downto 0);
        SIGNAL Regc2        : Unsigned (11 downto 0);
        SIGNAL Regd1        : Unsigned (11 downto 0);
        SIGNAL Regd2        : Unsigned (11 downto 0);
        SIGNAL Sum1         : Unsigned (11 downto 0);
        SIGNAL Sum2         : Unsigned (11 downto 0);
        SIGNAL Sum3         : Unsigned (11 downto 0);
        SIGNAL Sum4         : Unsigned (11 downto 0);

        SIGNAL Cnt_load     : Bit_Vector(8 downto 0);
        SIGNAL PE           : Bit;
        SIGNAL CE           : Bit;
        SIGNAL RD           : Bit;
        SIGNAL TC           : Bit;
        SIGNAL Col          : Bit_Vector(8 downto 0);

        SIGNAL Cnt_load2    : Bit_Vector(1 downto 0);
        SIGNAL PE2          : Bit;
        SIGNAL TC2          : Bit;
        SIGNAL CNT          : Bit_Vector(1 downto 0);
        SIGNAL Q, Q_not     : Bit;
BEGIN
        — Disable memory write/read;

        XP_Mem_WR_L <= '1';
        XP_Mem_RD_L <= '1';

        — Enable the up counters;
        — Reset the counters when image data is valid;

        Cnt_load <= (others => '0');
        Cnt_load2 <= (others => '0');
```

```
CE <= '1';
RD <= xbar(35) AND Q_not
PROCESS
        VARIABLE End_of_Row     : Unsigned (8 downto 0);
        VARIABLE Colv           : Unsigned (8 downto 0);
        VARIABLE Data_tag       : Bit_Vector (3 downto 0);
        VARIABLE four           : Unsigned (8 downto 0);
        VARIABLE eight          : Unsigned (8 downto 0);
BEGIN
        WAIT UNTIL XP_Clk'event AND XP_Clk = '1';

        Pad_Input(XP_XBar, xbar);
        XP_XBar_EN_L <= "00000";
        Pad_Output(XP_Right, right);
        __*************************
        — Number of pixels per image row:
        — pyr. level 0:        512 pix
        — pyr. level 1:        256 pix
        — pyr. level 2:        128 pix
        — pyr. level 3:         64 pix
        — pyr. level 4:         32 pix
        __*************************
        CASE xbar(34 downto 32) IS
                WHEN "000" => End_of_Row := "111111111";
                WHEN "001" => End_of_Row := "011111111";
                WHEN "010" => End_of_Row := "001111111";
                WHEN "011" => End_of_Row := "000111111";
                WHEN "100" => End_of_Row := "000011111";
                WHEN others => End_of_Row := "111111111";
        END CASE;

        — Reset counter 1 when end_of_row is reached;
        COLV := COL;
        IF (COLV = End_of_Row – 1) THEN
                PE <= '1';
        ELSE
                PE <= '0';
        END IF;

        four := "000000100";
        eight:= "000001000";

        — State machine
        CASE CNT IS
                WHEN "00" =>
                        — [P1 × 1/16 + P5 × 1/16]
                        — image convolves by [1/16, 0, 0, 0, 1/16]
                        Rega1 <= "0000" & temp1(7 downto 0);
```

```
                    Rega2 <= "0000" & xbar(7 downto 0);
                    Regb1 <= "0000" & temp1(23 downto 16);
                    Regb2 <= "0000" & xbar(23 downto 16);
                    — [P2 × 1/4 + P4 × 1/4]
                    — image convolves by [0, 1/4, 0, 1/4, 0]
                    Regc1 <= "00" & temp1(15 downto 8);
                    Regc2 <= "00" & temp1(31 downto 24);
                    Regd1 <= "00" & temp1(31 downto 24);
                    Regd2 <= "00" & xbar(15 downto 8);
          WHEN "01" =>
                    — [1/16, 1/4, 0, 1/4, 1/16]
                    Rega1 <= Sum1;
                    Rega2 <= Sum3;
                    Regb1 <= Sum2;
                    Regb2 <= Sum4;

                    — [0, 0, 3/8, 0, 0]
                    Regc1 <= "00" & temp1(23 downto 16) & "00";
                    Regc2 <= "000" & temp1(23 downto 16) & '0';
                    Regd1 <= "00" & xbar(7downto 0) & "00";
                    Regd2 <= "000" & xbar(7 downto 0) & '0';

                    data_tag := temp1(35 downto 32);
                    temp1 <= xbar(35 downto 0);
          WHEN "10" =>
                    — [1/16, 1/4, 3/8, 1/4, 1/16]
                    Rega1 <= Sum1;
                    Rega2 <= Sum3;
                    Regb1 <= Sum2;
                    Regb2 <= Sum4
          WHEN "11" =>
                    — Image output with edge extension;
                    — End of an image row
                    IF Colv <= four THEN
                              right <= Data_tag & "0000000000000000" &
                                        temp2(7 downto 0) & temp2(7 downto 0);
                    — Beginning of an image row
                    ELSIF Colv <= eight THEN
                              right <= Data_tag & "0000000000000000" &
                                        Sum1(11 downto 4) & Sum1(11 downto 4);
                    ELSE
                              right <= Data_tag & "0000000000000000" &
                                        Sum1(11 downto 4) & temp2(7 downto 0);
                    END IF;

                    temp2 <= Sum2 (11 downto 4);
     END CASE;
END PROCESS;
```

```
        Sum1 <= Rega1 + Rega2;
        Sum2 <= Regb1 + Regb2;
        Sum3 <= Regc1 + Regc2;
        Sum3 <= Regd1 + Regd2;

        COL_counter: cup9h
                port map (Cnt_load, PE, XP_Clk, CE, RD, Col(8 downto 0), TC);

        Counter2: cup2h
                port map (CNT_load2, PE2, XP_Clk, CE, RD, CNT(1 dowto 0) TC2);

        Rset: D_FF
                port map ( xbar(35), XP_Clk, Q, Q_not);

END X_one;


________________________________________________________
—               Xilinx_Processing_Part X2               —
________________________________________________________

library SPLASH2;
use SPLASH2.TYPES.all;
use SPLASH2.SPLASH2.all;
use SPLASH2.COMPONENTS.all;
use SPLASH2.ARITHMETIC.all;
use work.mypack.all;

ARCHITECTURE X_two of Xilinx_Processing_Part IS

        SIGNAL left             : Bit_Vector (35 downto 0);
        SIGNAL right            : Bit_Vector (35 downto 0);
        SIGNAL xbar             : Bit_Vector (35 downto 0);
        SIGNAL mem_addr         : Bit_Vector (17 downto 0);
        SIGNAL mem_write        : Bit;
        SIGNAL mem_read         : Bit;
        SIGNAL mem_rd_data      : Bit_Vector (15 downto 0);
        SIGNAL mem_wr_data      : Bit_Vecotr (15 downto 0);
        SIGNAL en_mem           : Bit;

        SIGNAL rega1            : Unsigned (11 downto 0);
        SIGNAL rega2            : Unsigned (11 downto 0);
        SIGNAL regb1            : Unsigned (11 downto 0);
        SIGNAL regb2            : Unsigned (11 downto 0);
        SIGNAL sum1             : Unsigned (11 downto 0);
        SIGNAL sum2             : Unsigned (11 downto 0);

        SIGNAL Cnt_load         : Bit_Vector(19 downto 0);
        SIGNAL CE               : Bit;
        SIGNAL PE               : BIT;
        SIGNAL RD               : BIT;
        SIGNAL TC               :BIT;
        SIGNAL CNT              : Bit_Vector(19 downto 0);
        SIGNAL Q, Q_not              : Bit;
        SIGNAL valid                 : Bit;
        SIGNAL temp                  : Bit;
        SIGNAL temp1                 : Bit_vector (15 downto 0);
BEGIN
        CE <= '1';
        RD <= left (35) AND Q_not;
        PROCESS
                VARIABLE row4_del         : Unsigned (17 downto 0);
                VARIABLE row3_del         : Unsigned (17 downto 0);
                VARIABLE valid_out        : Bit;
        BEGIN
                WAIT UNTIL XP_Clk'event AND XP_Clk = '1';
                Pad_Input (XP_Left, left);
                Pad_Output (XP_Xbar, left);
                XP_XBar_EN_L <= "11111";
                Pad_Output (XP_Right, right);
                Pad_Output (XP_Mem_WR_L, memwrite);
                Pad_Output (XP_Mem_RD_L, memread);
                Pad_Output (XP_Mem_A, mem_addr);
                Pad_InOut (XP_Mem_D, mem_wr_data, mem_rd_data, en_mem);
                mem_wr_data <= left (15 downto 0);
                __*************************************************
                — memory address offsets for four–row delay and three–row delay
                — pyr. level 0:           r4 = X"200",   r3 = X"180"
                — pyr. level 1:           r4 = X"100",   r3 = X"0C0"
                — pyr. level 2:           r4 = X"080",   r3 = X"060"
                — pyr. level 3:           r4 = X"040",   r3 = X"030"
                — pyr. level 4:           r4 = X"020",   r3 = X"018"
                __*************************************************
                CASE left(34 downto 32) IS
                        WHEN "000" => row4_del := "000000001000000000";
                                      row3_del := "000000000110000000";
                        WHEN "001" => row4_del := "000000000100000000";
                                      row3_del := "000000000011000000";
                        WHEN "010" => row4_del := "000000000010000000";
                                      row3_del := "000000000001100000";
                        WHEN "011" => row4_del := "000000000001000000";
                                      row3_del := "000000000000110000";
                        WHEN "100" => row4_del := "000000000000100000";
                                      row3_del := "000000000000011000";
                        WHEN others => row4_del := "000000000000000000";
                                      row3_del := "000000000000000000";
                END CASE;
                CASE CNT(1 downto 0) IS
                        WHEN "00" =>
                                memwrite <= '1';
                memread <= '0';
            WHEN "01" =>
                — issure a memory write;
                memwrite <= '0';
                memread <= '1';
                en_mem <= '1';
                mem_addr <= CNT (19 downto 2);

                — $[\frac{1}{16}, 0, 0, 0, \frac{1}{16}]$
                Rega1 <= "0000" & mem_rd_data (7 downto 0);
                Regb1 <= "0000" & temp1 (7 downto 0);
                Rega2 <= "0000" & mem_rd_data (15 downto 7);
                Regb2 <= "0000" & temp1(15 downto 7);
                valid <= left (35);
            WHEN "10" =>
                — issue the first memory read
                memwrite <= '1';
                memread <= '0';
                en_mem <= '0';
                mem_addr <= CNT(19 downto 2) – row4_del;
                tmp1 <= mem_wr_data;

                — $[\frac{1}{16}, \frac{1}{4}, 0, 0, \frac{1}{16}]$
                Rega1 <= sum1;
                Regb1 <= "00" & mem_rd_data(7 downto 0) & "00";
                Rega2 <= sum2;
                Regb2 <= "00" & mem_rd_data(15 downto 8) & "00";
            WHEN "11" =>
                — issue the second memory write;
                memwrite <= '1';
                memread <= '0';
                en_mem <= '0';
                mem_addr <= CNT(19 downto 2) – row3_del;

                IF (CNT(19 downto 2) > row4_del AND temp = '1') THEN
                        valid_out := '1';
                ELSE
                        valid_out := '0';
                END IF;
                right <= valid_out & "0000000000000000" & sum2 & sum1;
                temp <= valid;
        END CASE;
END PROCESS;

Sum1 <= Rega1 + Regb1;
Sum2 <= Rega2 + Regb2;

Counter : cup20h
        port map (Cnt_load, PE, XP_Clk, CE, RD, CNT, TC);

Rest: D_FF
        port map (left(35), XP_Clk, Q, Q_not);
```

```
END X_two;

________________________________________________________
—                 Xilinx_Processing_Part X3                  —
________________________________________________________

library SPLASH2;
use SPLASH2.TYPES.all;
use SPLASH2.SPLASH2.all;
use SPLASH2.COMPONENTS.all;
use SPLASH2.ARITHMETIC.all;
use work.mypack.all;
ARCHITECTURE X_three of Xilinx_Processing_Part IS
        SIGNAL left                 : Bit_Vector (35 downto 0);
        SIGNAL right                : Bit_Vector (35 downto 0);
        SIGNAL xbar                 : Bit_Vector (35 downto 0);
        SIGNAL mem_addr             : Bit_Vector (17 downto 0);
        SIGNAL memwrite             : Bit;
        SIGNAL memread              : Bit;
        SIGNAL mem_rd_data          : Bit_Vector (15 downto 0);
        SIGNAL mem_wr_data          : Bit_Vector (15 downto 0);
        SIGNAL en_mem               : Bit;

        SIGNAL rega1                : Unsigned (11 downto 0);
        SIGNAL rega2                : Unsigned (11 downto 0);
        SIGNAL regb1                : Unsigned (11 downto 0);
        SIGNAL regb2                : Unsigned (11 downto 0);
        SIGNAL sum1                 : Unsigned (11 downto 0);
        SIGNAL sum2                 : Unsigned (11 downto 0);

        SIGNAL Cnt_load             : Bit_Vector (19 downto 0);
        SIGNAL CE                   : Bit;
        SIGNAL PE                   : Bit;
        SIGNAL RD                   : Bit;
        SIGNAL TC                   : Bit;
        SIGNAL CNT                  : Bit_Vector (19 downto 0);
        SIGNAL Q, Q_not             : Bit;
BEGIN
        CE <= '1';
        RD <= xbar (35) AND Q_not;

        PROCESS
                VARIABLE row4_del           : Unsigned (17 downto 0);
                VARIABLE row2_del           : Unsigned (17 downto 0);
                VARIABLE row1_del           : Unsigned (17 downto 0);
                VARIABLE level              : Unsigned (2 downto 0);
                VARIABLE valid_out          : Bit;
        BEGIN
                WAIT UNTIL XP_Clk'event AND XP_Clk = '1';
                Pad_Input (XP_Xbar, xbar);
```

```
XP_Xbar_EN_L <= "00000";
Pad_Input (XP_Left, left);
Pad_Output (XP_Right, right);
Pad_Output (XP_Mem_WR_L, memwrite);
Pad_Output (XP_Mem_RD_L, memread);
Pad_Output (XP_Mem_A, mem_addr);
Pad_InOut (XP_Mem_D, mem_wr_data, mem_rd_data, en_mem);
mem_wr_data <= xbar(15 downto 0);

level := xbar (34 downto 32);
CASE level IS
        WHEN "000" => row4_del := "000000001000000000";
                      row2_del := "000000000100000000";
                      row1_del := "000000000010000000";
        WHEN "001" => row4_del := "000000000100000000";
                      row2_del := "000000000010000000";
                      row1_del := "000000000001000000";
        WHEN "010" => row4_del := "000000000010000000";
                      row2_del := "000000000001000000";
                      row1_del := "000000000000100000";
        WHEN "011" => row4_del := "000000000001000000";
                      row2_del := "000000000000100000";
                      row1_del := "000000000000010000";
        WHEN "100" => row4_del := "000000000000100000";
                      row2_del := "000000000000010000";
                      row1_del := "000000000000001000";
        WHEN others => row4_del := "000000000000000000";
                      row2_del := "000000000000000000";
                      row1_del := "000000000000000000";
END CASE;

CASE CNT (1 downto 0) IS
        WHEN "00" =>
                memwrite <= '1';
                memread <= '1';

                — Step 3: $[\frac{1}{16},\frac{1}{4},\frac{3}{8},\frac{1}{4},\frac{1}{16}]$
                Rega1 <= sum1;
                Regb1 <= left (11 downto 0);
                Rega2 <= sum2;
                Regb2 <= left (23 downto 12);
        WHEN "01" =>
                memwrite <= '0';
                memread <= '1';
                en_mem <= '1';
                mem_addr <= CNT(19 downto 2);

                — Step 1: $[0,0,\frac{3}{8},0,0]$
                Rega1 <= "000" & mem_rd_data (7 downto 0) & '0';
                Regb1 <= "00" & mem_rd_data (7 downto 0) & "00";
```

```
                    Rega2 <= "000" & mem_rd_data (15 downto 8) & '0';
                    Regb2 <= "00" & mem_rd_data (15 downto 8) & "00";
                    IF (CNT (19 downto 2) > (row4_del + 1)
                       AND (left(35) = '1') THEN
                            valid_out <= '1';
                    ELSE
                            valid_out <= '0';
                    END IF;
                    right <= valid_out &level & "0000000000000000" &
                              sum2(11 downto 4) & sum1(11 downto 4);
                WHEN "10" =>
                    memwrite <= '1';
                    memread <= '0';
                    en_mem <= '0';
                    mem_addr <= cnt (19 downto 2) – row2_del;
                    — Step 2: [0,0,3/8,1/4,0]
                    Rega1 <= sum1;
                    Regb1 <= "00" & mem_rd_data (7downto 0) & "00";
                    Rega2 <= sum2;
                    Regb2 <= "00" &mem_rd_data(15 downto 8) & "00";
                WHEN "11" =>
                    memwrite <= '1';
                    memread <= '0';
                    en_mem <= '0';
                    mem_addr <= CNT (19 downto 2) – row1_del;
            END CASE;
        END PROCESS;
        sum1 <= rega1 + regb1;
        sum1 <= rega2 + regb2;
        Counter1 : cup20h
            port map (Cnt_load, PE, XP_Clk, CE, RD, CNT, TC);
        D_FF
            port map (xbar(35), XP_Clk, Q, Q_not);
END X_three;


———————————————————————————————————————————
—              Xilinx_Processing_Part X4                 —
———————————————————————————————————————————

library SPLASH2;
use SPLASH2.TYPES.all;
use SPLASH2.SPLASH2.all;
use SPLASH2.COMPONENTS.all;
use SPLASH2.ARITHMETIC.all;
use work.mypack.all;
ARCHITECTURE X_four of Xilinx_Processing_Part IS
        SIGNAL left                  : Bit_Vector (35 downto 0);
        SIGNAL right             : Bit_Vector (35 downto 0);
        SIGNAL xbar              : Bit_Vector (35 downto 0);
        SIGNAL mem_addr          : Bit_Vector (17 downto 0);
        SIGNAL memwrite          : Bit;
        SIGNAL memread           : Bit;
        SIGNAL mem_wr_data       : Bit_Vector (15 downto 0);
        SIGNAL mem_rd_data       : Bit_Vector (15 downto 0);
        SIGNAL en_mem            : Bit;

        SIGNAL Cnt_load1         : Bit_Vector (1 downto 0);
        SIGNAL PE1               : Bit;
        SIGNAL CE1               : Bit;
        SIGNAL RD1               : Bit;
        SIGNAL TC1               : Bit;
        SIGNAL Clk_CNT           : Bit_Vector(1 downto 0);

        SIGNAL Cnt_load2         : Bit_Vector (19 downto 0);
        SIGNAL PE2               : Bit;
        SIGNAL CE2               : Bit;
        SIGNAL RD2               : Bit;
        SIGNAL TC2               : Bit;
        SIGNAL mem_cnt           : Bit_Vector (19 downto 0);

        SIGNAL Cnt_load3         : Bit_Vector (19 downto 0);
        SIGNAL PE3               : Bit;
        SIGNAL CE3               : Bit;
        SIGNAL RD3               : Bit;
        SIGNAL TC3               : Bit;
        SIGNAL mem_rd_addr       : Bit_Vector (19 downto 0);

        SIGNAL pre_level         ; Unsigned (2 downto 0);
        SIGNAL level             : Unsigned (2 downto 0);
        SIGNAL bas_wr_addr       : Unsigned (17 downto 0);
        SIGNAL bas_rd_addr       : Unsigned (17 downto 0);
        SIGNAL num_of_mem        : Unsigned (17 downto 0);
        SIGNAL Bit_per_row       : Natural;
        SIGNAL Q1, Q1_not        : Bit;
        SIGNAL valid_in          : Bit;
        SIGNAL valid_out         : Bit;
        SIGNAL temp              : Bit_Vector (15 downto 0);
BEGIN

        RD1 <= '0';
        RD2 <= left (35) AND Q1_not;
        RD3 <= NOT (left (35) OR Q1_not);
        CE1 <= left (35) OR level (1) OR (NOT (level(2)) AND level(0))
                OR (level(2) AND NOT(level(0)));
        CE2 <= '1';
        PE1 <= '0';
        PE2 <= '0';
        PE3 <= '0;
```

```
valid_in <= left(35);
right <= left;

PROCESS
BEGIN
        WAIT UNTIL valid_in'event AND valid_in = '1';
        CASE left(34 downto 32) IS
                WHEN "000" => bas_wr_addr <= "000000000010000000";
                                Bit_per_row <= 9;
                WHEN "001" => bas_wr_addr <= "001000000001000000";
                               Bit_per_row <= 8;
                WHEN "010" => bas_wr_addr <= "001010000000100000";
                                Bit_per_row <= 7;
                WHEN "011" => bas_wr_addr <= "001010100000010000";
                                Bit_per_row <= 6;
                WHEN "100" => bas_wr_addr <= "001010101000001000";
                                Bit_per_row <= 5;
                WHEN others => bas_wr_addr <= "000000000000000000";
                                Bit_per_row <= 1;
        END CASE;
        pre_level <= left(34 downto 32);
END PROCESS;

PROCESS
BEGIN
        WAIT UNTIL valid_in'event AND valid_in = '0';
        CASE pre_level IS
                WHEN "000" => num_of_mem <= "000100000000000000";
                                bas_rd_addr <= "000000000000000000";
                                CE3 <= '1';
                WHEN "001" => num_of_mem <= "000001000000000000";
                                bas_rd_addr <= "001000000000000000";
                                CE3 <= '1';
                WHEN "010" => num_of_mem <= "000000010000000000";
                                bas_rd_addr <= "001010000000000000";
                                CE3 <= '1';
                WHEN "011" => num_of_mem <= "000000000100000000";
                                bas_rd_addr <= "001010100000000000";
                                CE3 <= '1';
                WHEN "100" => num_of_mem <= "000000000000000000";
                                bas_rd_addr <= "000000000000000000";
                                CE3 <= '0';
                WHEN others => num_of_mem <= "000000000000000000";
                                bas_rd_addr <= "000000000000000000";
                                CE3 <= '0';
        END CASE;
        level <= pre_level + 1;
END PROCESS;

PROCESS
        VARIABLE temp_addr : Unsigned (17 downto 0);
```

```
BEGIN
        WAIT UNTIL XP_Clk'event and XP_Clk = '1';
        Pad_Input (XP_Left, left);
        Pad_Output (XP_Right, right);
        Pad_Output (XP_Mem_WR_L, memwrite);
        Pad_Output (XP_Mem_RD_L, memread);
        Pad_Output (XP_Mem_A, mem_addr);
        Pad_Output (XP_Xbar, xbar);
        XP_Xbar_EN_L <= "11111";
        Pad_InOut (XP_Mem_D, mem_wr_data, mem_rd_data, en_mem);
        mem_wr_data <= left (15 downto 0);
        CASE Clk_CNT IS
                WHEN "00" =>
                        memwrite <= '1';
                        memread <= '1';
                WHEN "01" =>
                        xbar <= valid_out & level & mem_rd_data & temp;
                        IF mem_cnt (Bit_per_row) = '0' AND left(35) = '1' THEN
                                memwrite <= '0';
                                memread <= '1';
                                en_mem <= '1';
                                IF Bit_per_row = 9 THEN
                                        temp_addr := '0' & mem_cnt(19 downto 10)
                                                        & mem_cnt(8 downto 2);
                                ELSIF Bit_per_row = 8 THEN
                                        temp_addr := '0' & mem_cnt(19 downto 9)
                                                        & mem_cnt(7 downto 2);
                                ELSIF Bit_per_row = 7 THEN
                                        temp_addr := '0' & mem_cnt(19 downto 8)
                                                        & mem_cnt(6 downto 2);
                                ELSIF Bit_per_row = 6 THEN
                                        temp_addr := '0' & mem_cnt(19 downto 7)
                                                        & mem_cnt(5 downto 2);
                                ELSIF Bit_per_row = 5 THEN
                                        temp_addr := '0' & mem_cnt(19 downto 6)
                                                        & mem_cnt(4 downto 2);
                                ELSE
                                        temp_addr := mem_cnt (19 downto 2);
                                END IF;
                        ELSE
                                memwrite <= '1';
                                memread <= '1';
                        END IF;
                        mem_addr <= temp_addr + bas_wr_addr;
                        temp <= mem_rd_data;
                WHEN "10" =>
                        IF CE3 = '1' AND
                           mem_rd_addr (19 downto 2) < num_of_mem THEN
                                memwrite <= '1';
```

```
                                        memread <= '0';
                                        en_mem <= '0';
                                        temp_addr <= mem_rd_addr(18 downto 2) & '0';
                                        valid_out <= '1';
                                ELSE
                                        memwrite <= '1';
                                        memread <= '1';
                                        en_mem <= '0';
                                        valid_out <= '0';
                                END IF;
                                mem_addr <= temp_addr + bas_rd_addr;
                        WHEN "11" =>
                                memwrite <= '1';
                                memread <= memread;
                                en_mem <= '0';
                                mem_addr <= mem_addr + 1;
                END CASE;
        END PROCESS;

        Counter1: cup2h
                port map (Cnt_load1, PE1, XP_Clk, CE1, RD1, Clk_CNT, TC1);

        Counter2: cup20h
                port map (Cnt_load2, PE2, XP_Clk, CE2, RD2, mem_cnt, TC2);

        Counter3: cup20h
                port map (Cnt_load3, PE3, XP_Clk, CE3, RD3, mem_rd_addr, TC3);

        Rest: D_FF
                port map (left(35), XP_Clk, Q1, Q1_not);

END X_four;



------------------------------------------------------------
—               Xilinx_Processing_Part X5               —
------------------------------------------------------------

library SPLASH2;
use SPLASH2.TYPES.all;
use SPLASH2.SPLASH2.all;
use SPLASH2.COMPONENTS.all;
use SPLASH2.ARITHMETIC.all;
use work.mypack.all;

ARCHITECTURE X_five of Xlinx_Processing_Part IS
        SIGNAL left             : Bit_Vector(35 downto 0);
        SIGNAL right            : Bit_Vector(35 downto 0);
        SIGNAL Cnt_load1        : Bit_Vector(1 downto 0);
        SIGNAL PE1              : Bit;
        SIGNAL CE1              : Bit;
        SIGNAL RD1                  : Bit;
        SIGNAL TC1                  : Bit;
        SIGNAL Clk_Cnt              : Bit_Vector(1 downto 0);

        SIGNAL Cnt_load2            : Bit_Vector(9 downto 0);
        SIGNAL PE2                  : Bit;
        SIGNAL CE2                  : Bit;
        SIGNAL RD2                  : Bit;
        SIGNAL TC2                  : Bit;
        SIGNAL CNT2                 : Bit_Vector(9 downto 0);

        SIGNAL Rega1                : Unsigned (11 downto 0);
        SIGNAL Rega2                : Unsigned (11 downto 0);
        SIGNAL Regb1                : Unsigned (11 downto 0);
        SIGNAL Regb2                : Unsigned (11 downto 0);
        SIGNAL sum1                 : Unsigned (11 downto 0);
        SIGNAL sum2                 : Unsigned (11 downto 0);
        SIGNAL temp_sum1            : Bit_Vector(7 downto 0);
        SIGNAL temp_sum2            : Bit_Vector(7 downto 0);
        SIGNAL temp_sum3            : Bit_Vector(7 downto 0);
        SIGNAL temp_sum4            : Bit_Vector(7 downto 0);

        SIGNAL mem_addr             : Bit_Vector(17 downto 0);
        SIGNAL memwrite             : Bit;
        SIGNAL memread              : Bit;
        SIGNAL mem_rd_data          : Bit_Vector(15 downto 0);
        SIGNAL mem_wr_data          : Bit_Vector(15 downto 0);
        SIGNAL en_mem               : Bit;

        SIGNAL Q1, Q1_not           : Bit;
        SIGNAL Q2, Q2_not           : Bit;
        SIGNAL temp1                : Bit_Vector(19 downto 0);
        SIGNAL temp2                : Bit_Vector(19 downto 0);
        SIGNAL valid_out            : Bit;
        SIGNAL temp_outh            : Bit_Vector(15 downto 0);
        SIGNAL temp_outl            : Bit_Vector(15 downto 0);
BEGIN
        Cnt_load1 <= "00";
        Cnt_load2 <= "0000000000";
        CE1 <= '1';
        CE2 <= '1';
        RD1 <= left(35) and Q1_not;
        RD2 <= valid and Q2_not;

        PROCESS
                VARIABLE End_of_Row      : Unsigned (7 downto 0);
                VARIABLE pix_cnt         : Unsigned (7 downto 0);
        BEGIN
                WAIT UNTIL XP_CLK'event AND XP_CLK = '1';
                Pad_Input(XP_Left, left);
                Pad_Output(XP_Right, right);
                Pad_Output(XP_Mem_WR_L, memwrite);
Pad_Output(XP_Mem_RD_L, memread);
Pad_Output(XP_Mem_A, mem_addr);
Pad_InOut(XP_Mem_D, mem_wr_data, mem_rd_data, en_mem);
IF left(35) = '1' THEN
        CASE left(34 downto 32) IS
                WHEN "000" => End_of_Row := "01111111";
                WHEN "001" => End_of_Row := "00111111";
                WHEN "010" => End_of_Row := "00011111" ;
                WHEN "011" => End_of_Row := "00001111";
                WHEN "100" => End_of_Row := "00000111";
                WHEN others => End_of_Row := "11111111";
        END CASE;
END IF;
IF (CNT2 = End_of_Row(6 downto 0) & "110") THEN
        PE2 <= '1';
ELSE
        PE2 <= '0';
END IF;
CASE Clk_CNT IS
        WHEN "00" =>
                memwrite <= '1';
                memread <= '1';
                temp_sum3 <= sum1(10 downto 3);
                temp_sum4 <= sum2(10 downto 3);
                Rega1 <= "000" & temp1(15 downto 8) & '0';
                Rega2 <= "00" & temp1(15 downto 8) & "00";
                Regb1 <= "00" & temp1(15 downto 8) & "00"
                Regb2 <= "00" & temp1(7 downto 0) & "00";
        WHEN "01" =>
                pix_cnt := CNT2 (9 downto 2);
                IF pix_cnt <= End_of_Row THEN
                        memwrite <= NOT(valid);
                ELSE
                        memwrite <= '1';
                END IF;
                memread <= '1';
                en_mem <= '1';
                IF pix_cnt = 0 THEN
                        mem_wr_data <= temp_sum1 & temp_sum1;
                ELSE
                        mem_wr_data <= temp_outh;
                END IF;
                mem_addr <= "000000000" & pix_cnt & '0';
                temp_outh <= temp_sum4 & temp_sum3;
                temp_outl <= temp_sum2 & temp_sum1;
                temp_sum1 <= sum2(10 downto 3);
                Rega1 <= "0000" & left(7 downto 0);
                Rega2 <= sum1;
                Regb1 <= "000" & left(7 downto 0) & '0';
```

```
                    Regb2 <= “00” & left(7 downto 0) & “00”;
            WHEN “10” =>
                    valid_out <= valid;
                    right <= valid_out & temp2(18 downto 16) &
                            “0000000000000000” & mem_rd_data;
                    IF pix_cnt <= End_of_Row THEN
                            memwrite <= NOT(valid);
                    ELSE
                            memwrite <= ’1’;
                    END IF;
                    memread <= ’1’;
                    en_mem <= ’1’;
                    IF pix_cnt = End_of_Row THEN
                            mem_wr_data <= mem_wr_data(15 downto 8) &
                                            mem_wr_data(15 downto 8);
                    ELSE
                            mem_wr_data <= temp_outl
                    END IF;
                    mem_addr <= “000000000” & pix_cnt & ’1’;
                    Rega1 <= “0000” & temp1(7 downto 0);
                    Rega2 <= sum1;
                    Regb1 <= “0000” & temp1(15 downto 0);
                    Regb2 <= “sum2;
                    temp2 <= temp1;
                    temp1 <= left(35 downto 32) & left(15 downto 0);
            WHEN “11” =>
                    memwrite <= ’1’;
                    memread <= NOT(valid);
                    en_mem <= ’0’;
                    mem_addr <= “0000000000” pix_cnt;
                    temp_sum2 <= sum1(10 downto 3);
                    Rega1 <= “00” & temp2(15 downto 8) & “00”;
                    Rega2 <= “00” & temp1(7 downto 0) & “00”;
                    Regb1 <= “0000” & temp2(15 downto 8);
                    Regb2 <= sum2;
                    valid <= temp2(19);
    END CASE;
END PROCESS;
Sum1 <= Rega1 + Rega2;
Sum2 <= Regb1 + Regb2;
Counter1 : cup2h
    port map (CNT_load1, PE1, XP_Clk, CE1, RD1, Clk_CNT, TC1);

Counter2: cup10h
    port map (CNT_load2, PE2, XP_Clk, CE2, RD2, CNT2, TC2);

Reset1 : D_FF
    port map (left(35), XP_Clk, Q1, Q1_not);

Reset2: D_FF
    port map (valid, XP_Clk, Q2, Q2_not);
```

```
END X_five;
```

— Xilinx_Processing_Part X6 —

```
library SPLASH2;
use SPLASH2.TYPES.all;
use SPLASH2.SPLASH2.all;
use SPLASH2.COMPONENTS.all;
use SPLASH2.ARITHMETIC.all;
use work.mypack.all;

ARCHITECTURE X_six of Xilinx_Processing_Part IS
        SIGNAL left             : Bit_Vector (35 downto 0);
        SIGNAL right            : Bit_Vector(35 downto 0);
        SIGNAL mem_addr         : Bit_Vector(17 downto 0);
        SIGNAL memwrite         : Bit;
        SIGNAL memread          : Bit;
        SIGNAL mem_rd_data      : Bit_Vector(15 downto 0);
        SIGNAL mem_wr_data      : Bit_Vector(15 downto 0);
        SIGNAL en_mem           : Bit;

        SIGNAL rega1            : Unsigned (11 downto 0);
        SIGNAL rega2            : Unsigned (11 downto 0);
        SIGNAL regb1            : Unsigned (11 downto 0);
        SIGNAL regb2            : Unsigned (11 downto 0);
        SIGNAL regc1            : Unsigned (11 downto 0);
        SIGNAL regc2            : Unsigned (11 downto 0);
        SIGNAL regd1            : Unsigned (11 downto 0);
        SIGNAL regd2            : Unsigned (11 downto 0);
        SIGNAL sum1             : Unsigned (11 downto 0);
        SIGNAL sum2             : Unsigned (11 downto 0);
        SIGNAL sum3             : Unsigned (11 downto 0);
        SIGNAL sum4             : Unsigned (11 downto 0);
        SIGNAL temp_sum1        : Unsigned (11 downto 0);
        SIGNAL temp_sum2        : Unsigned (11 downto 0);
        SIGNAL temp_sum3        : Unsigned (11 downto 0);
        SIGNAL temp1            : Bit_Vector (15 downto 0);
        SIGNAL temp2            : Bit_Vector (15 downto 0);
        SIGNAL temp_out1        : Bit_Vector (15 downto 0);
        SIGNAL valid1           : Bit;
        SIGNAL valid2           : Bit;

        SIGNAL CNT_load1        : Bit_Vector(19 downto 0);
        SIGNAL PE1              : Bit;
        SIGNAL CE1              : Bit;
        SIGNAL RD1              : Bit;
        SIGNAL TC1              : Bit;
        SIGNAL CNT1                 : Bit_Vector(19 downto 0);
        SIGNAL Q, Q_not             : Bit;
BEGIN
        CE1 <= '1';
        RD1 <= left(35) and Q_not;
        PROCESS
                VARIABLE valid_out: Bit;
                VARIABLE row1_del: Unsigned (17 downto 0);
                VARIABLE row2_del: Unsigned (17 downto 0);
        BEGIN
                WAIT UNTIL XP_CLK'EVNT AND XP_CLK = '1';
                Pad_Input (XP_Left, left);
                Pad_Output(XP_Right, right);
                Pad_Output(XP_Mem_WR_L, memwrite);
                Pad_Output(XP_Mem_RD_L, memread);
                Pad_Output(XP_Mem_A, mem_addr);
                Pad_InOut(XP_Mem_D, mem_wr_data, mem_rd_data, en_mem);
                mem_wr_data <= left(15 downto 0);
                IF left(35) = '1' THEN
                        CASE left(34 downto 32) IS
                                WHEN "000" => row2_del := "000000001000000000";
                                              row1_del := "000000000100000000";
                                WHEN "001" => row2_del := "000000000100000000";
                                              row1_del := "000000000010000000";
                                WHEN "010" => row2_del := "000000000010000000";
                                              row1_del := "000000000001000000";
                                WHEN "011" => row2_del := "000000000001000000";
                                              row1_del := "000000000000100000";
                                WHEN "100" => row2_del := "000000000000100000";
                                              row1_del := "000000000000010000";
                                WHEN others => row2_del := "000000000000000000";
                                              row1_del := "000000000000000000";
                        END CASE;
                END IF;
                CASE CNT1(1 downto 0) IS
                        WHEN "00" =>
                                memwrite <= '1';
                                memread <= '1';
                                IF ((CNT1(19 downto 2) > (row2_del(17 downto 1) & '1'))
                                        AND (valid2 = '1')) THEN
                                                valid_out := '1';
                                ELSE
                                                valid_out := '0';
                                END IF;
                                right <= valid_out & left(34 downto 32) &
                                        sum2(10 downto 3) & sum1(10 downto 3) &
                                        temp_out1;
                        WHEN "01" =>
                                memwrite <= NOT ( left(35 );
                    memread <= '1';
                    en_mem <= '1';
                    mem_addr <= CNT1(19 downto 2);
                    temp2 <= mem_rd_data;
                    rega1 <= "0000" & temp1(7 downto 0);
                    rega2 <= "0000" & mem_rd_data (7 downto 0);

                    regb1 <= "0000" & temp1(15 downto 8);
                    regb2 <= "0000" & mem_rd_data (15 downto 8);
                WHEN "10" =>
                    memwrite <= '1';
                    memread <= NOT ( left(35) );
                    en_mem <= '0';
                    mem_addr <= CNT1(19 downto 2) -row2_del;

                    valid1 <= left(35);
                    valid2 <= valid1;
                    temp1 <= left(15 downto 0);
                    temp_sum1 <= sum1;
                    temp_sum2 <= sum2;
                    rega1 <= "00" & temp2 (7 downto 0) & "00";
                    rega2 <= "00" & mem_rd_data ( 7 downto 0) & "00";
                    regb1 <= "00" & temp2(15 downto 8) & "00";
                    regb2 <= "00" & mem_rd_data (15 downto 8) & "00";
                    regc1 <= "00" & mem_rd_data(7 downto 0) & "00";
                    regc2 <= "000" & mem_rd_data(7 downto 0) & '0';
                    regd1 <= "00" & mem_rd_data(15 downto 8) & "00";
                    regd2 <= "000" & mem_rd_data(15 downto 0) & '0';
                WHEN "11" =>
                    memwrite <= '1';
                    memread <= memread;
                    en_mem <= '0';
                    mem_addr <= CNT1 (19 downto 2) - row1_del;
                    temp_out1 <= sum2(10 downto 3) & sum1(10 downto 3);
                    rega1 <= sum3;
                    rega2 <= temp_sum1;
                    regb1 <= sum4;
                    regb2 <= temp_sum2;
        END CASE;
    END process;

    sum1 <= rega1 + rega2;
    sum2 <= regb1 + regb2;
    sum3 <= regc1 + regc2;
    sum4 <= regd1 + regd2;

    Counter1: cup20h
        port map(CNT_load1, PE1, XP_Clk, CE1, RD1, CNT1, TC1);

    Rest : D_FF
        port map ( left (35), XP_Clk, Q, Q_not);
END X_six;
```

```
—              Xilinx_Processing_Part X7             —

library SPLASH2;
use SPLASH2.TYPES.all;
use SPLASH2.SPLASH2.all;
use SPLASH2.COMPONENTS.all;
use SPLASH2.ARITHMETIC.all;
use work.mypack.all;
ARCHITECTURE X_seven of Xilinx_Processing_Part IS
        SIGNAL left              : Bit_Vector (35 downto 0);
        SIGNAL right             : Bit_Vector (35 downto 0);
        SIGNAL xbar              ; Bit_Vector (35 downto 0);
        SIGNAL mem_addr          ; Bit_Vector (17 downto 0);
        SIGNAL memwrite          : Bit;
        SIGNAL memread           : Bit;
        SIGNAL mem_rd_data       : Bit_Vector (15 downto 0);
        SIGNAL mem_wr_data       : Bit_Vector (15 downto 0);
        SIGNAL en_mem            : Bit;

        SIGNAL Cnt_load1         : Bit_Vector (19 downto 0);
        SIGNAL PE1               : Bit;
        SIGNAL CE1               : Bit;
        SIGNAL RD1               : Bit;
        SIGNAL TC1               : Bit;
        SIGNAL CNT1              : Bit_Vector (19 downto 0);
        SIGNAL Cnt_load2         : Bit_Vector (19 downto 0);
        SIGNAL PE2               : Bit;
        SIGNAL CE2               : Bit;
        SIGNAL RD2               : Bit;
        SIGNAL TC2               : Bit;
        SIGNAL CNT2              : Bit;

        SIGNAL Q1, Q1_not        : Bit;
        SIGNAL Q2, Q2_not        : Bit;
        SIGNAL temp_left         : Unsigned (35 downto 0);
        SIGNAL sub_h             : Bit_Vector (7 downto 0);
        SIGNAL sub_l             : Bit_Vecotr(7 downto 0);
BEGIN
        CE1 <= '1';
        RD1 <= xbar(35) and Q1_not;
        PE1 <= '0';
        CNT_load1 <= "00000000000000000000";

        CE2 <= '1';
        RD2 <= left(35) AND Q2_not;
        PE2 <= '0';
        CNT_load2 <= "00000000000000000000";
```

```
PROCESS
        VARIABLE temp_addr : Bit_Vector (17 downto 0);
        VARIABLE tmp_data : Bit_Vector (15 downto 0);
        VARIABLE Bit_per_row: Natural;
        VARIABLE row4_del : Unsigned (17 downto 0);
BEGIN
        WAIT UNTIL XP_Clk'EVENT AND XP_Clk = '1';
        Pad_Input(XP_Left, left);
        Pad_Input(XP_XBar, xbar);
        XP_XBar_EN_L <= "00000";
        Pad_Output(XP_Right, right);
        Pad_Output(XP_Mem_WR_L, memwrite);
        Pad_Output(XP_Mem_RD_L, memread);
        Pad_Output(XP_Mem_A, mem_addr);
        Pad_InOut(XP_Mem_D, mem_wr_data, mem_rd_data, en_mem);
        IF xbar(35) = '1' THEN
                CASE xbar(34 downto 32) IS
                        WHEN "000" => row4_del := "000000010000000000";
                                      Bit_per_row := 10;
                        WHEN "001" => row4_del := "000000001000000000";
                                      Bit_per_row := 9;
                        WHEN "010" => row4_del := "000000000100000000";
                                      Bit_per_row := 8;
                        WHEN "011" => row4_del := "000000000010000000";
                                      Bit_per_row := 7;
                        WHEN "100" => row4_del := "000000000001000000";
                                      Bit_per_row := 6;
                        WHEN others => row4_del := "000000000000000000";
                                      Bit_per_row := 0;
                END CASE;
        END IF;

        CASE CNT1 (1 downto 0) IS
                WHEN "00" =>
                        memwrite <= '1';
                        memread <= '1';
                WHEN "01" =>
                        IF ((CNT1 (19 downto 2) >= ('0' & row4_del (17 downto 1)))
                           AND (xbar(35) = '1')) THEN
                                memwrite <= '0';
                        ELSE
                                memwrite <= '1';
                        END IF;
                        memread <= '1';
                        en_mem <= '1';
                        mem_wr_data <= xbar(15 downto 0);
                        mem_addr <= CNT1(18 downto 2) & '0' – row4_del;
                WHEN "10" =>
                        memwrite <= memwrite;
                        memread <= '1';
```

```
                    en_mem <= '1';
                    mem_wr_data <= xbar(31 downto 16);
                    mem_addr <= CNT1 (18 downto 2) & '1' – row4_del;
                    tmp_data := mem_rd_data;
                    IF tmp_data (15 downto 8) > temp_left(15 downto 8) THEN
                            sub_h <= tmp_data(15 downto 8)
                                    – temp_left(15 downto 8);
                    ELSE
                            sub_h <= temp_left(15 downto 8)
                                    – tmp_data(15 downto 8);
                    END IF;
                    IF tmp_data (7 downto 0) > temp_left(7 downto 0) THEN
                            sub_h <= tmp_data(7 downto 0)
                                    – temp_left(7 downto 0);
                    ELSE
                            sub_h <= temp_left(7 downto 0)
                                    – tmp_data(7 downto 0);
                    END IF;
                WHEN "11" =>
                    memwrite <= '1';
                    memread <= not (left (35));
                    en_mem <= '0';

                    right <= temp_left(35 downto 16) & sub_h & sub_l;
                    temp_left <= left(35 downto 0);
                    IF Bit_per_row = 10 THEN
                            temp_addr := CNT2(18 downto 10) & '0'
                                        & CNT2(9 downto 2);
                    ELSIF Bit_per_row = 9 THEN
                            temp_addr := CNT2(18 downto 9) & '0'
                                        & CNT2(8 downto 2);
                    ELSIF Bit_per_row = 8 THEN
                            temp_addr := CNT2(18 downto 8) & '0'
                                        & CNT2(7 downto 2);
                    ELSIF Bit_per_row = 7 THEN
                            temp_addr := CNT2(18 downto 7) & '0'
                                        & CNT2(6 downto 2);
                    ELSIF Bit_per_row = 6 THEN
                            temp_addr := CNT2(18 downto 6) & '0'
                                        & CNT2(5 downto 2);
                    ELSE
                            temp_addr := "000000000000000000";
                    END IF;
                    mem_addr <= temp_addr;
        END CASE;
END PROCESS;

Counter1 : cup20h
        port map ( Cnt_load1, PE1, XP_Clk, CE1, RD1, CNT1, TC1);
        Counter2 : cup20h
                port map (Cnt_load2, PE2, XP_Clk, CE2, RD2, CNT2, TC2);

        Rest1 : D_FF
                port map ( xbar(35), XP_Clk, Q1, Q1_not);

        Rest2 : D_FF
                port map ( left(35), XP_Clk, Q2, Q2_not);

END X_seven;
```

```
—               Xilinx_Processing_Part X8                    —
```

```
library SPLASH2;
use SPLASH2.TYPES.all;
use SPLASH2.SPLASH2.all;
use SPLASH2.COMPONENTS.all;
use SPLASH2.ARITHMETIC.all;
use work.mypack.all;

ARCHITECTURE X_eight of Xilinx_Processing_Part IS
        SIGNAL left                 : Bit_Vector (35 downto 0);
        SIGNAL right                : Bit_Vector (35 downto 0);
        SIGNAL xbar                 ; Bit_Vector (35 downto 0);
        SIGNAL mem_addr             ; Bit_Vector (17 downto 0);
        SIGNAL memwrite             : Bit;
        SIGNAL memread              : Bit;
        SIGNAL mem_rd_data          : Bit_Vector (15 downto 0);
        SIGNAL mem_wr_data          : Bit_Vector (15 downto 0);
        SIGNAL en_mem               : Bit;

        SIGNAL Cnt_load1            : Bit_Vector (19 downto 0);
        SIGNAL PE1                  : Bit;
        SIGNAL CE1                  : Bit;
        SIGNAL RD1                  : Bit;
        SIGNAL TC1                  : Bit;
        SIGNAL CNT1                 : Bit_Vector (19 downto 0);
        SIGNAL Cnt_load2            : Bit_Vector (19 downto 0);
        SIGNAL PE2                  : Bit;
        SIGNAL CE2                  : Bit;
        SIGNAL RD2                  : Bit;
        SIGNAL TC2                  : Bit;
        SIGNAL CNT2                 : Bit;

        SIGNAL Q1, Q1_not           : Bit;
        SIGNAL Q2, Q2_not           : Bit;
        SIGNAL temp_left            : Unsigned (35 downto 0);
        SIGNAL sub_h                : Bit_Vector (7 downto 0);
        SIGNAL sub_l                    : Bit_Vecotr(7 downto 0);
BEGIN
        CE1 <= '1';
        RD1 <= xbar(35) and Q1_not;
        PE1 <= '0';
        CNT_load1 <= "00000000000000000000";

        CE2 <= '1';
        RD2 <= left(35) AND Q2_not;
        PE2 <= '0';
        CNT_load2 <= "00000000000000000000";

        PROCESS
                VARIABLE temp_addr : Bit_Vector (17 downto 0);
                VARIABLE tmp_data : Bit_Vector (15 downto 0);
                VARIABLE Bit_per_row: Natural;
                VARIABLE row4_del : Unsigned (17 downto 0);
        BEGIN
                WAIT UNTIL XP_Clk'EVENT AND XP_Clk = '1';
                Pad_Input(XP_Left, left);
                Pad_Input(XP_XBar, xbar);
                XP_XBar_EN_L <= "00000";
                Pad_Output(XP_Right, right);
                Pad_Output(XP_Mem_WR_L, memwrite);
                Pad_Output(XP_Mem_RD_L, memread);
                Pad_Output(XP_Mem_A, mem_addr);
                Pad_InOut(XP_Mem_D, mem_wr_data, mem_rd_data, en_mem);
                IF xbar(35) = '1' THEN
                        CASE xbar(34 downto 32) IS
                                WHEN "000" => row4_del := "000000010000000000";
                                              Bit_per_row := 10;
                                WHEN "001" => row4_del := "000000001000000000";
                                              Bit_per_row := 9;
                                WHEN "010" => row4_del := "000000000100000000";
                                              Bit_per_row := 8;
                                WHEN "011" => row4_del := "000000000010000000";
                                              Bit_per_row := 7;
                                WHEN "100" => row4_del := "000000000001000000";
                                              Bit_per_row := 6;
                                WHEN others => row4_del := "000000000000000000";
                                              Bit_per_row := 0;
                        END CASE;
                END IF;

                CASE CNT1 (1 downto 0) IS
                        WHEN "00" =>
                                memwrite <= '1';
                                memread <= '1';
                        WHEN "01" =>
                                IF ((CNT1 (19 downto 2) >= ('0' & row4_del (17 downto 1)))
                                   AND (xbar(35) = '1')) THEN
                memwrite <= '0';
            ELSE
                memwrite <= '1';
            END IF;
            memread <= '1';
            en_mem <= '1';
            mem_wr_data <= xbar(15 downto 0);
            mem_addr <= CNT1(18 downto 2) & '0' – row4_del;
    WHEN "10" =>
            memwrite <= memwrite;
            memread <= '1';
            en_mem <= '1';
            mem_wr_data <= xbar(31 downto 16);
            mem_addr <= CNT1 (18 downto 2) & '1' – row4_del;
            tmp_data := mem_rd_data;
            IF tmp_data (15 downto 8) > temp_left(31 downto 24) THEN
                sub_h <= tmp_data(15 downto 8)
                         – temp_left(31 downto 24);
            ELSE
                sub_h <= temp_left(31 downto 24)
                          – tmp_data(15 downto 8);
            END IF;
            IF tmp_data (7 downto 0) > temp_left(23 downto 16) THEN
                sub_h <= tmp_data(7 downto 0)
                         – temp_left(23 downto 16);
            ELSE
                sub_h <= temp_left(23 downto 16)
                          – tmp_data(7 downto 0);
            END IF;
    WHEN "11" =>
            memwrite <= '1';
            memread <= not (left (35));
            en_mem <= '0';

            right <= temp_left(35 downto 32) & sub_h & sub_l
                     & temp_left (15 downto 0);
            temp_left <= left(35 downto 0);
            IF Bit_per_row = 10 THEN
                temp_addr := CNT2(18 downto 10) & '1'
                             & CNT2(9 downto 2);
            ELSIF Bit_per_row = 9 THEN
                temp_addr := CNT2(18 downto 9) & '1'
                             & CNT2(8 downto 2);
            ELSIF Bit_per_row = 8 THEN
                temp_addr := CNT2(18 downto 8) & '1'
                             & CNT2(7 downto 2);
            ELSIF Bit_per_row = 7 THEN
                temp_addr := CNT2(18 downto 7) & '1'
                             & CNT2(6 downto 2);
            ELSIF Bit_per_row = 6 THEN
```

```
                                        temp_addr := CNT2(18 downto 6) & '1'
                                                        & CNT2(5 downto 2);
                                ELSE
                                        temp_addr := "000000000000000000";
                                END IF;
                                mem_addr <= temp_addr;
                END CASE;
        END PROCESS;

        Counter1 : cup20h
                port map ( Cnt_load1, PE1, XP_Clk, CE1, RD1, CNT1, TC1);

        Counter2 : cup20h
                port map (Cnt_load2, PE2, XP_Clk, CE2, RD2, CNT2, TC2);

        Rest1 : D_FF
                port map ( xbar(35), XP_Clk, Q1, Q1_not);

        Rest2 : D_FF
                port map ( left(35), XP_Clk, Q2, Q2_not);

END X_eight;



------------------------------------------------------------------
--              Xilinx_Processing_Part X9                       --
------------------------------------------------------------------

library SPLASH2;
use SPLASH2.TYPES.all;
use SPLASH2.SPLASH2.all;
use SPLASH2.COMPONENTS.all;
use SPLASH2.ARITHMETIC.all;
use work.mypack.all;

ARCHITECTURE X_nine of Xlinx_Processing_Part IS
        SIGNAL left             : Bit_Vector(35 downto 0);
        SIGNAL right            : Bit_Vector (35 downto 0);
        SIGNAL xbar             : Bit_Vector (35 downto 0);
        SIGNAL mem_addr         : Bit_Vector (17 downto 0);
        SIGNAL memwrite         : Bit;
        SIGNAL memread          : Bit;
        SIGNAL mem_rd_data      : Bit;
        SIGNAL mem_wr_data      : Bit;
        SIGNAL en_mem           : Bit;
        SIGNAL Cnt_load1        : Bit_Vector (19 downto 0);
        SIGNAL PE1              : Bit;
        SIGNAL CE1              : Bit;
        SIGNAL RD1              : Bit;
        SIGNAL TC1              : Bit;
        SIGNAL CNT1             : Bit_Vector (19 downto 0);
        SIGNAL valid                : Bit;
        SIGNAL Q, Q_not             : Bit;
BEGIN
        CE1 <= '1';
        RD1 <= left (35) AND Q_not;
        PE1 <= '0';
        Cnt_load1 <= "000000000000000000";
        PROCESS
                VARIABLE temp_addr : Bit_Vector (17 downto 0);
                VARIABLE Bit_per_row : Natural;
                VARIABLE row1_del : Unsigned (17 downto 0);
                VARIABLE valid_out : Bit;
        BEGIN
                WAIT UNTIL XP_Clk'EVENT AND XP_Clk = '1';
                Pad_Input (XP_Left, left);
                Pad_Output(XP_XBar, left);
                XP_XBar_EN_L <= "11111";
                Pad_Output(XP_Right, right);
                Pad_Output(XP_Mem_WR_L, memwrite);
                Pad_Output(XP_Mem_RD_L, memread);
                Pad_Output(XP_Mem_A, mem_addr);
                Pad_InOut(XP_Mem_D, mem_wr_data, mem_rd_data, en_mem);
                IF left (35) = '1' THEN
                        CASE left (34 downto 32) IS
                                WHEN "000" => row1_del := "000000000100000000";
                                              Bit_per_row := 10;
                                WHEN "001" => row1_del := "000000000010000000";
                                              Bit_per_row := 9;
                                WHEN "010" => row1_del := "000000000001000000";
                                              Bit_per_row := 8;
                                WHEN "011" => row1_del := "000000000000100000";
                                              Bit_per_row := 7;
                                WHEN "100" => row1_del := "000000000000010000";
                                              Bit_per_row := 6;
                                WHEN others => row1_del := "000000000000000000";
                                              Bit_per_row := 0;
                        END CASE;
                END IF;
                CASE CNT1(1 downto 0) IS
                        WHEN "00" =>
                                memwrite <= '1';
                                memread <= '1';
                        WHEN "01" =>
                                memwrite <= NOT (left(35));
                                memread <= '1';
                                en_mem <= '1';
                                mem_wr_data <= left(15 downto 0);
                                IF Bit_per_row = 10 THEN
```

```
                                        temp_addr := CNT1(18 downto 10) & '1'
                                                    & CNT1(9 downto 2);
                                ELSIF Bit_per_row = 9 THEN
                                        temp_addr := CNT1(18 downto 9) & '1'
                                                    & CNT1(8 downto 2);
                                ELSIF Bit_per_row = 8 THEN
                                        temp_addr := CNT1(18 downto 8) & '1'
                                                    & CNT1(7 downto 2);
                                ELSIF Bit_per_row = 7 THEN
                                        temp_addr := CNT1(18 downto 7) & '1'
                                                    & CNT1(6 downto 2);
                                ELSIF Bit_per_row = 6 THEN
                                        temp_addr := CNT1(18 downto 6) & '1'
                                                    & CNT1(5 downto 2);
                                ELSE
                                        temp_addr := "000000000000000000";
                                END IF;
                                mem_addr <= temp_addr;
                        WHEN "10" =>
                                memwrite <= NOT(left(35));
                                memread <= '1';
                                en_mem <= '1';
                                mem_wr_data <= left(31 downto 16);
                                mem_addr <= row1_del + temp_addr;
                                valid <= left(35);
                                IF CNT1(19 downto 2) > row1_del AND valid = '1' THEN
                                        valid_out := '1';
                                END IF;
                                right <= valid_out & left(34 downto 32) &
                                        "0000000000000000" & mem_rd_data;
                        WHEN "11" =>
                                memwrite <= '1';
                                memread <= '0';
                                en_mem <= '0';
                                mem_addr <= CNT1(18 downto 2) & '0' - row1_del;
                END CASE;
        END PROCESS;
        Counter1 : cup20h
                port map (Cnt_load1, PE1, XP_Clk, CE1, RD1, CNT1, TC1);
        Rest : D_FF
                port map (left(35), XP_Clk, Q, Q_not);
END X_nine;
```

— Xilinx_Processing_Part X10 —

```
library SPLASH2;
use SPLASH2.TYPES.all;
use SPLASH2.SPLASH2.all;
use SPLASH2.COMPONENTS.all;
use SPLASH2.ARITHMETIC.all;
use work.mypack.all;
ARCHITECTURE X_ten of Xlinx_Processing_Part IS
        SIGNAL left                 : Bit_Vector(35 downto 0);
        SIGNAL right                : Bit_Vector (35 downto 0);
        SIGNAL xbar                 : Bit_Vector (35 downto 0);
        SIGNAL mem_addr             : Bit_Vector (17 downto 0);
        SIGNAL memwrite             : Bit;
        SIGNAL memread              : Bit;
        SIGNAL mem_rd_data          : Bit;
        SIGNAL mem_wr_data          : Bit;
        SIGNAL en_mem               : Bit;
        SIGNAL Cnt_load1            : Bit_Vector (19 downto 0);
        SIGNAL PE1                  : Bit;
        SIGNAL CE1                  : Bit;
        SIGNAL RD1                  : Bit;
        SIGNAL TC1                  : Bit;
        SIGNAL CNT1                 : Bit_Vector (19 downto 0);
        SIGNAL valid                : Bit;
        SIGNAL Q, Q_not             : Bit;
        SIGNAL temp_right           : Bit_Vector(15 downto 0);

BEGIN
        CE1 <= '1';
        RD1 <= xbar (35) AND Q_not;
        PE1 <= '0';
        Cnt_load1 <= "000000000000000000";
        PROCESS
                VARIABLE temp_addr : Bit_Vector (17 downto 0);
                VARIABLE Bit_per_row : Natural;
                VARIABLE row1_del : Unsigned (17 downto 0);
                VARIABLE valid_out : Bit;
        BEGIN
                WAIT UNTIL XP_Clk'EVENT AND XP_Clk = '1';
                Pad_Input (XP_Left, left);
                Pad_Input(XP_XBar, left);
                XP_XBar_EN_L <= "00000";
                Pad_Output(XP_Right, right);
                Pad_Output(XP_Mem_WR_L, memwrite);
                Pad_Output(XP_Mem_RD_L, memread);
                Pad_Output(XP_Mem_A, mem_addr);
                Pad_InOut(XP_Mem_D, mem_wr_data, mem_rd_data, en_mem);
                IF left (35) = '1' THEN
                        CASE left (34 downto 32) IS
                WHEN “000” => row1_del := “000000000100000000”;
                                      Bit_per_row := 10;
                WHEN “001” => row1_del := “000000000010000000”;
                                      Bit_per_row := 9;
                WHEN “010” => row1_del := “000000000001000000”;
                                      Bit_per_row := 8;
                WHEN “011” => row1_del := “000000000000100000”;
                                      Bit_per_row := 7;
                WHEN “100” => row1_del := “000000000000010000”;
                                      Bit_per_row := 6;
                WHEN others => row1_del := “000000000000000000”;
                                      Bit_per_row := 0;
        END CASE;
END IF;


CASE CNT1(1 downto 0) IS
        WHEN “00” =>
                memwrite <= ’1’;
                memread <= ’1’;
        WHEN “01” =>
                memwrite <= NOT ( xbar(35) );
                memread <= ’1’;
                en_mem <= ’1’;
                mem_wr_data <= xbar(15 downto 0);
                IF Bit_per_row = 10 THEN
                        temp_addr := CNT1(18 downto 10) & ’1’
                                        & CNT1(9 downto 2);
                ELSIF Bit_per_row = 9 THEN
                        temp_addr := CNT1(18 downto 9) & ’1’
                                        & CNT1(8 downto 2);
                ELSIF Bit_per_row = 8 THEN
                        temp_addr := CNT1(18 downto 8) & ’1’
                                        & CNT1(7 downto 2);
                ELSIF Bit_per_row = 7 THEN
                        temp_addr := CNT1(18 downto 7) & ’1’
                                        & CNT1(6 downto 2);
                ELSIF Bit_per_row = 6 THEN
                        temp_addr := CNT1(18 downto 6) & ’1’
                                        & CNT1(5 downto 2);
                ELSE
                        temp_addr := “000000000000000000”;
                END IF;
                mem_addr <= temp_addr;
        WHEN “10” =>
                memwrite <= NOT( xbar(35));
                memread <= ’1’;
                en_mem <= ’1’;
                mem_wr_data <= xbar(31 downto 16);
                mem_addr <= row1_del + temp_addr;
```

```
                                temp_right <= mem_rd_data;
                        WHEN "11" =>
                                memwrite <= '1';
                                memread <= '0';
                                en_mem <= '0';
                                mem_addr <= CNT1(18 downto 2) & '1' - row1_del;
                                right <= left(35 downto 32) & temp_right
                                        & left(15 downto 0);
                END CASE;
        END PROCESS;
        Counter1 : cup20h
                port map (Cnt_load1, PE1, XP_Clk, CE1, RD1, CNT1, TC1);
        Rest : D_FF
                port map ( xbar(35), XP_Clk, Q, Q_not);
END X_ten;


————————————————————————————————————————————
—               User defined package "MYPACK"            —
————————————————————————————————————————————

library SPLASH2;
use SPLASH2.TYPES.all;
use SPLASH2.SPLASH2.all;
use SPLASH2.COMPONENTS.all;
use SPLASH2.ARITHMETIC.all;
PACKAGE mypack IS
        COMPONENT cup2h
          port (Cnt_load                 : in Bit_Vector (1 downto 0);
                PE, Clk, CE, RD      : in Bit;
                CNT                  : out Bit_Vector (1 downto 0);
                TC                   : out Bit);
        END COMPONENT;
        — 8–bit, 9–bit, 18–bit and 20–bit up counter are defined in this package.
        — They have the same structure of the cup2h does.
        COMPONENT D_FF
          port (D               : in Bit;
                Clk     : in Bit;
                Q       : out Bit;
                Q_not   : out Bit);
        END COMPONENT;
END mypack;
__*********************************
—*              2–bit up counter            *
__*********************************
```

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

ENTITY cup2h IS
        port (  CNT_load            : in Bit_Vector(1 downto 0);
                PE, Clk, CE, RD     : in Bit;
                CNT                 : out Bit_Vector(1 downto 0);
                TC                  : out Bit);
END cup2h;

ARCHITECTURE behavior of cup2h IS
        SIGNAL temp : Unsigned (1 downto 0);
BEGIN
        CNT <= temp;
        TC <= AND_REDUCE(temp);
        PROCESS
        BEGIN
                WAIT UNTIL Clk'EVENT AND Clk='1';
                IF (RD = '1') THEN
                        temp <= "00";
                ELSE
                        IF (PE = '1') THEN
                                temp <= Cnt_load;
                        ELSE
                                IF (CE = '1') THEN
                                        temp <= temp + 1;
                                END IF;
                        END IF;
                END IF;
        END PROCESS;
END behavior;


--****************************************
--*                 D Flip–flop          *
--****************************************

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

ENTITY D_FF IS
        port ( D : in Bit; Clk : in Bit;
                Q: out Bit; Q_not : out Bit);
END ENTITY;

ARCHITECTURE behavior of D_FF IS
BEGIN
```

```
          PROCESS
          BEGIN
                    WAIT UNTIL Clk'event AND Clk = '1';
                    Q <= D;
                    Q_not <= NOT (D);
          END PROCESS;
END behavior;
```

Vita

Luna Chen was born in Suzhou, Jiangsu Province, the People's Republic of China, on January 27, 1969. She attended several schools in Sichuan and Tianjin, and graduated in July, 1985, from No. 1 Tianjin High School. In July, 1989, Ms. Chen received a Bachelor of Science degree in Electrical Engineering from Fudan University in Shanghai. The following August, she continued her studies in Applied Physics at Appalachian State University, and received a Master of Science degree in August, 1991. She then enrolled in the Bradley Department of Electrical Engineering at Virginia Polytechnic Institute and State University in August, 1992, and hopefully will receive her Master of Science in Electrical Engineering in September of 1994. Ms. Chen is currently working as a hardware engineer at GE Fanuc Automation Inc. in Charlottesville, Virginia.

Ms. Chen's parents are Mr. Renkang Chen of Changzhou, Jiangsu, and Ms. Weizhen Yu of Suzhou, Jiangsu. They are currently living in Tianjin, China.