FPGA Implementation of Video Noise Estimation for Real-Time Processing

François-Xavier Lapalme

A Thesis

in

The Department

of

Electrical and Computer Engineering

Presented in Partial Fulfillment of the Requirements
for the Degree of Master of Applied Science (Electrical Engineering) at
Concordia University
Montréal, Québec, Canada

August 16, 2005

Your file Votre référence
ISBN: 0-494-10239-X
Our file Notre référence
ISBN: 0-494-10239-X

Abstract

FPGA Implementation of Video Noise Estimation for Real-Time Processing

François-Xavier Lapalme

Digital video processing algorithms are computationally intensive and their performance worsens dramatically as image resolution and pixel data size grow larger. Effective techniques are required to contend with this shortcoming in performance. One solution is to make use of a fast-prototyping, flexible and reprogrammable Field Programmable Gate Array (FPGA) technology. High density FPGAs offer compelling platforms for real-time performance of complex video processing algorithms. An important application of this type of hardware approach involves the treatment of noise, often introduced during the transmission and processing of video signals, which can significantly impact the effectiveness of video processing algorithms.

This thesis proposes an FPGA implementation of a video noise estimation algorithm capable of real-time processing. The objectives of this thesis consist of adapting a computationally demanding noise estimation algorithm to a synthesizable VHDL design and achieving real-time processing performance. Hardware feasibility is determined through a study of the mathematical operations used in the estimation process. The proposed architecture provides a satisfactory compromise between area and processing speed. Furthermore, parameterization of the architecture allows additional flexibility with the scaling of features, such as filter size, to operate on 3 x 3 or 5 x 5 blocks of pixels. The proposed design is targeted to an FPGA Xilinx XC2V4000 chip and is evaluated through computer simulations on the basis of its throughput, minimum resource allocation, and quality of noise estimation results compared to the original software implementation.

Acknowledgments

I wish to express my sincere gratitude to both of my supervisors. Dr. Amer, for proposing this research topic to me and for always providing me with prompt feedback, Dr. Wang, for overseeing and helping me prepare this research dissertation and thesis defence.

I would like to thank Mr. Tadeusz Obuchowicz for introducing me to VHDL and hardware design, which has lead to my career path, and for always having been available to help me solve my technical problems relating to FPGA CAD tools. I am specially grateful to Kumara Ratnayake for his generous help and technical insight and to Eric Lewis for his teachings of fundamental hardware principles, guidance and patience that has been invaluable to my academic and working experience.

None of this would have been possible without the constant love and support given to me by my family, grand-parents, parents and sister. They have encouraged me throughout my university studies and I am grateful for their moral and financial assistance.

Je dédie cet ouvrage à mes parents.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Background

The rapid development of products and services offering digital video processing suggests that, in the near future, digital video will greatly impact the communications and imaging industries. Video processing such as video compression, filtering, and analysis (e.g., edge detection and image segmentation) are at the core of numerous technologies applied in digital television, DVD players, video conferencing, cellular phones displays, and digital cameras.

Some parts of the video processing will need to be carried out immediately by digital signal processing (DSP) systems instead of being executed subsequently by a computer whereby providing embedded video processing and analysis capabilities. Smart cameras adopt embedded features in order to increase the performance of simple video processing

manipulations, to become more cost effective and to reduce the system overall hardware requirements and bandwidth. Thus, having the majority of the video processing work be performed inside a single hardware unit. High performance machine vision systems using smart camera technology [1], suggest to use embedded hardware in complementary effort with PC-based vision systems for tasks such as edge detection or binarization. The Smart cameras in [2] consisting of embedded hardware and high quality video cameras perform two-dimensional convolution in the medical field for retinal vascular tracing. Smart cameras in [3] capture high-level descriptions of a video sequence by performing real-time video processing analysis such as region extraction, contour following, ellipse fitting and graph matching.

Video signal processing has become increasingly complex and entails high computational demands on large quantities of data. Indeed, realizing video analysis on huge streams of data for broadcasting purposes in real-time has become challenging. Real-time digital signal processing can be attained by means of dedicated hardware. Traditionally, embedded systems equipped with instruction set processors (ISP) and a set of hard-wired integrated circuits (IC) were used to process signals. These large embedded systems used DSP specific cores to perform individual arithmetic task. They were controlled by a microcontroller chip which acted as the system central processing unit (CPU). This arrangement of chips were connected on a printed circuit board (PCB) and composed an un-flexible processing scheme. Application-specific integrated circuits (ASIC), a more powerful single chip processing system, were also designed for specific digital signal processing purposes. However, custom-made fixed hardware have a long design cycle and are usually used in high-volume commercial applications, ASICs are often considered too costly for many designs. A suitable

alternative to these DSP-based or ASIC chips signal processing designs are low-cost and flexible Field-Programmable Gate Array (FPGA). The current high density FPGAs offer a compelling platform for hardware acceleration of complex software algorithms which are otherwise too slow to meet real-time requirements. FPGA-based designs become popular because of their reconfigurable capability and short design-time (fast prototyping) for implementing computation intensive applications. Although FPGAs have limited on-chip memory capacity, proper architecture organization, external memory usage, pipelined circuit design, and custom implementation permit fast performance for real-time video processing.

Noise can significantly impact the performance of video processing algorithms. Noise estimation techniques are some of many video pre-processing algorithms that are used to help reduce the noise in a video sequence. Noise estimation calculates the level of white noise contained in a corrupted video signal (e.g., due to video acquisition or transmission). Noise estimation algorithms help enhance the image quality by adapting the main video processing procedures (e.g., video noise reduction or motion estimation) to the amount of noise for improved performance. Therefore, this pre-processing noise estimation procedure is imperative to furnish important information about the content of the captured video and its high-performance hardware implementation is equally important to accelerate the overall execution of an embedded video signal processing system.

This work proposes a FPGA implementation of video white-noise estimation technique [4] for real-time processing. This noise estimation algorithm has not been considered for FPGA implementation before. The noise estimation method used [4] is based on finding homogeneous blocks in noisy video frames where the estimated noise variances is calculated on these set of blocks. The final estimated noise variance of a video frame is generated from

the most intensity-homogeneous regions in the video frames. This estimation task is done using a 2 dimensional digital FIR filter based on high pass convolution masks (or kernels) to allow implicit detection of structure and to stabilize the homogeneity estimation.

1.2 Motivations and Objectives

Implementing computation intensive video processing algorithms in real-time has become challenging. Although software algorithms may possess multi-threading capabilities and granting today's CPUs offer powerful performances in terms of processing speeds, some software algorithms are unable to process data fast enough due to the lack of parallelism in the sequentially driven execution of the software commands. This constraint leads to slow processing speeds when successive video processing manipulations (e.g., noise estimation, noise reduction, edge detection) require to be executed on subsequent video frames. In today's fast advancements in technology, the increasing demand in bandwidth and the increasingly complex video algorithms, software programs may encounter timing complications when attempting to perform the many video processing algorithms "in real-time". "In real-time" signifies completing the processing of a current video frame before a next video frame is grabbed and digitized for video processing purposes. Thus, hardware solutions are suitable to respond to this high demand in real-time computational capabilities. In this work, a real-time processing hardware implementation of a video noise estimation technique is proposed.

The first motivation is to find a hardware solution suitable to accelerate the computation speed of video processing algorithms such as noise estimation, to reach real-time

performance. Future video processing embedded systems will require more and more video processing algorithms to be fully implemented in hardware. Furthermore, noise estimation algorithms help better the performance of other video processing algorithms; it is therefore essential to develop its hardware implementation for future embedded hardware devices.

The second motivation is driven by the necessity for a flexible hardware design. The strength of software programs resides in the facility with which they can be parameterized to help fine-tune the accuracy of their results. Video processing algorithms such as noise estimation contain features like various window scanning sizes, data sizes and threshold values. These parameters help improve the noise estimation algorithm depending on the input video sequence. Developing a scalable and flexible architecture capable of supporting different video processing parameters is important to make the hardware design as easy to utilize and as versatile as software applications.

The main objective is to develop a hardware implementation of the software noise estimation algorithm capable of performing real-time processing. In order to reach the primary objective, the optimum hardware architecture must be determined. The architecture is to render the most accurate noise estimation results as compared to the original software results. The architecture must also be designed to operate at the highest possible frequency and must be best structured to produce real-time performances. Furthermore, the architecture is to be built to consume a minimum amount of logic resources to keep the overall area of the design as condensed as possible. The general concept behind the desire to minimize a design's circuit size (area) is to decrease the total cost of the FPGA hardware system. Moreover, the architecture is to be scalable and capable of supporting different noise estimation parameters such as various filter sizes.

Notwithstanding the fact that many algorithms can be directly mapped onto a hardware platform, some software algorithms need to be adapted and often modified to generate a high performance hardware implementation. The FPGA design is to be programmed and developed using Very High Speed Integrated Circuit Hardware Description Language (VHDL). This noise estimation algorithm can not be directly mapped to the FPGA hardware with the VHDL language. Some sections of the algorithm contain some arithmetic operations such as logarithmic calculations for which the VHDL language does not have an equivalent operator. Other software specific structures such as floating-point representation can not be instantiated in VHDL. Sorting of large arrays is also exploited in this noise estimation algorithm. Sorting algorithms can easily be implemented with data structures such as pointers and link lists that are incorporated in high level software languages like C and C++. However, the VHDL hardware description language does not support data structures. Hence, the software noise estimation algorithm needs to be properly adapted and modified into a synthesizable hardware design. These changes must minimize the affects on the quality of the hardware noise estimation results. Hardware complications and bottlenecks that arise from algorithm translation need to be solved to comply with the resource-constrained FPGA platform.

Scalability and parameter passing for design of complex structures (e.g., data structures) in software is also much simpler because memory management is not a design issue. Software programs take the responsibility of allocating and releasing any memory spaces that it needs via the operating system. In hardware design, memory management must be developed manually and the designers must select the internal (registers) and external (DRAM, cache) memory sizes, bus capacity and bus topology to build complex structures. Designers

must also build the interfaces and develop transmission protocols to communicate with the external memories. Hence, building a scalable hardware architecture to incorporate flexible parameter passing features is a complex procedure that requires a thorough analysis of the design's circuitry and control logic.

Throughout the FPGA design process, a fundamental principal in hardware design is to construct a design using the least possible amount of logic. In depth circuit analysis, optimizations at the architecture level, at the VHDL code level, and at the synthesis level, are key procedures to minimize the overall area in the final hardware implementation. Furthermore, proper design optimizations should eliminate extra logic, increase the processing speed and help reach real-time performances.

1.3 Thesis Outline

The thesis is organized as follows. In Chapter 2, background material on the principles of noise estimation is discussed, various noise estimation techniques are indicated, and the selected homogeneity-based noise estimation algorithm is described. FPGA-based DSP related work is also discussed in this chapter, along with certain benefits of this thesis's design as contrasted to similar implementations. In Chapter 3, the design flow is laid out, the noise estimation algorithm's necessary adaptations are explained to attain hardware compliance. This chapter also describes and justifies the design strategies that are adopted to construct the design's architecture. In Chapter 4, the FPGA implemented design organization is depicted at the top level view. The details of the sub-modules that compose this variance estimation system are individually described. In Chapter 5, the experimental results show

the implementation's accuracy of the variance generation as compared to the original software results. This chapter also illustrates the synthesis and routing results and validates the real-time performance of this implementation through the analysis of the design's timing results. In Chapter 6, the results of this noise estimation implementation are summarized, the contributions are enumerated and additional design suggestions are proposed.

Chapter 2

State of Art

In this chapter, some noise estimation techniques in video processing are discussed. The theory relating to the modeling of noise in video signals is described. The homogeneity-oriented noise estimation algorithm in [4], which is chosen for the implementation of this thesis, is detailed. Video processing implementations on FPGA platforms of image enhancement and restoration related research work are also examined. The benefits of the proposed noise estimation design is contrasted to similar FPGA implementations.

2.1 Noise Estimation and Video Processing

As stated in [5], future TV-receiver applications using image processing has to take into account fundamental image enhancement processing such as noise reduction. In a video system, an important pre-processing step is noise reduction and, as stated in [4], it is important that its computational cost stays low while keeping a reliable performance. The

human visual system (HVS) is sensitive to image content such as edges, textures, or moving areas and must be preserved because they are important performance features. The main objective of noise reduction is to eliminate most of the noise perceptible to the HVS. Images always contain noise which can be produced through image acquisition, recording, processing and transmission. Noise reduction techniques attempt to restore the original image from a deteriorated noisy copy.

If the noise level is determined, video processing can be made to adapt to the amount of noise to improve the stability of its results. For instance, the Canny edge detector in [6, 7] uses a noise estimator to help smooth the image to reduce the effects of noise. Another example, an image segmentation processing in [8], uses a Weibull noise index to estimate the noise in a volume pixel (voxel) and to obtain more precise standard deviation values for each voxel. A multi-resolution motion estimation scheme using noise estimation is devised for video encoding in [9]. An image restoration scheme is presented in [10] that combines a nonlinear smoothing algorithm and a noise estimation technique to recover images corrupted by Gaussian noise. Another smoothing algorithm found in [11] is significantly improved with the estimated noise variance.

In this sub-section, an overview on noise modeling theory in video processing is presented. A numerical measure to evaluate the quality of the noise estimation algorithm is described. Various noise estimation methods are enumerated and the selected noise estimation technique for the proposed implementation utilizing the homogeneity-oriented noise estimation algorithm [4] is detailed.

2.1.1 Modeling of Image Noise

Real world signals usually contain deviations from the ideal signal. Such deviations are denoted as noise. A signal is corrupted with noise because of transmission, storage or some other process as stated in [12].

The noise signal can be modeled as a stochastic signal which is additive or multiplicative to an image signal. Noise can generally be grouped into two classes, signal-dependent or signal-independent. For example, the noise produced in a quantization process is additive and signal-independent. Image noise can render different spectral characteristics; it can be white or colored. It is known that white light includes all parts of the visual spectrum. Similarly, white noise incorporates all parts of the frequency spectrum. On the other hand, colored noise has light low-frequency content and large high-frequency content. Thus, it can be said that any kind of filtered noise signal can be called "colored noise". The noise signal in images represented in Eq. 2.1 is assumed to be *independent identically distributed* (i.i.d.) additive and stationary zero-mean noise (e.g., white Gaussian noise). In Eq. 2.1, the $S(n)$ is the original image signal where n represents the time or frame index, $I(n)$ is the deteriorated noisy image signal and $\eta(n)$ is the added noise signal. The i and j stand for the lattice's X and Y coordinates of each pixel in the image.

$$I(i,j,n) = S(i,j,n) + \eta(n). \tag{2.1}$$

For Eq. 2.1 it is assumed that all data samples are independent from each other, which means that no information about any one sample can be inferred from knowledge of any

other. It is also said to have identical distribution, which means that the statistical distribution of each of the samples is the same (i.e., where the mean of all the samples is the same).

The quality of the noise estimation technique in this thesis is evaluated using the peak signal to noise ratio (PSNR) numerical measure as seen in Eq. 2.2.

$$PSNR = 10 \cdot \log \frac{(255)^2}{\frac{1}{XY}\sum_{i=1}^{X}\sum_{j=1}^{Y}(I_p(i,j,n) - I_r(i,j,n))^2}. \tag{2.2}$$

The PSNR technique will compare the hardware noise estimation results with the software original results. In Eq. 2.2 the image size is $X \cdot Y$, where $I_p(i,j,n)$ and $I_r(i,j,n)$ denote the pixel amplitudes (i.e., luminance) of the processed and reference image, respectively.

2.1.2 Noise Estimation Methods

Noise can be estimated within an image (intra-image estimation) or between two or more successive images (inter-image estimation) in a video sequence. Inter-image estimation methods store one or many images of a video sequence for processing purposes which necessitate a large amount of memory. Intra-image noise estimation can be achieved through two methods such as smoothing-based processing and block-based processing. In the smoothing-based method the process of smoothing is first applied to the image using an averaging filter, then the noise is calculated by the difference between the noisy and enhanced image. Noise is estimated at each pixel where the gradient is smaller than a given threshold. This

method tends to usually overestimate the noise variance in images characterized by refined textures. In the block-variance based method, the variance is calculated over a set of blocks that segment the image. The average of the smallest variances is then taken as an estimate. This method generally tends to overestimate the noise variance in good quality images and underestimate it in highly noisy images.

The block-variance based method is less complex and its computation speed is several times faster than the smoothing-based method as demonstrated in [13]. This research ([13]) evaluates six methods for estimating the standard deviation of white additive noise in images and states that the sequential hardware implementation of the block-variance based method is at least 6 times faster than the other evaluated noise estimation methods. The main drawback with the block-variance based methods is that the noise estimates may fluctuate considerably depending on the image structure and noise level.

In this thesis, the homogeneity-based noise estimation technique is studied for hardware implementation. The details of this technique are described in the following sub-section.

2.1.3 Homogeneity-Oriented Noise Estimation Algorithm

The noise estimation algorithm selected for this thesis estimates the variance of additive white noise in video images. This homogeneity-based technique, developed in [4], achieves reliable noise estimates in images containing smooth and textured areas by taking image structure into account and by using a novel homogeneity measure rather than the variance to determine if a block is homogeneous. A homogeneous analyzer uses specific masks to identify homogeneous blocks by detecting lines in the image and reject highly structured blocks.

The noise estimation algorithm concentrates on averaging noise variances of homogeneous image blocks, where only blocks showing similar homogeneities are included in the averaging process. The algorithm is described as follows:

First, the structure analyzer, based on high pass operators, utilizes eight masks (seen in Fig. 2.1) to calculate a homogeneity measure (ξ_{Bh}) for each block of the segmented image frame.

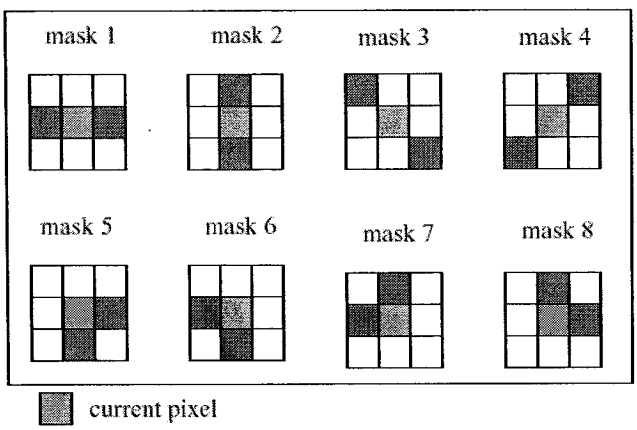


Figure 2.1: Diagram of 8 masks of the homogeneity analyzer.

The mathematical equation of the horizontal direction mask (e.g., mask 1 shown in Fig. 2.1) of 3x3 scanning window is given in Eq. 2.3. The term $I(i)$ represents the center pixel and the terms $I(i-1)$ and $I(i+1)$ represent the left and right neighboring horizontal pixels, respectively.

$$I_{o_1}(i) = -I(i-1) + 2 \cdot I(i) - I(i+1). \tag{2.3}$$

The homogeneity measure (ξ_{Bh}) is then produced from the sum of the 8 mask filter

results as shown in Eq. 2.4. It is important to note that the variance-based homogeneity measures fail to detect fine structure and texture. On the other hand, the proposed homogeneity measure, that uses high-pass operators, distinguishes well lines, steps and shoulders of ramps edges along 8 different directions. Furthermore, special corners masks allow implicit detection of center of ramp edge structures and help stabilize the homogeneity evaluation.

$$\xi_{Bh} = I_{o_1} + I_{o_2} + \cdots + I_{o_8}. \tag{2.4}$$

Second, the sample mean (μ_{Bh}) is calculated in Eq. 2.5 for each block in the image frame which is then used to generate the variance (σ^2_{Bh}) as seen in Eq. 2.6 for each block in the image frame. The term $I(i,j)$ represents the luminance of the pixels included within a block of height W.

$$\mu_{Bh} = \frac{\sum\limits_{(i,j)\in W_{ij}} I(i,j)}{W \times W}. \tag{2.5}$$

$$\sigma^2_{Bh} = \frac{\sum\limits_{(i,j)\in W_{ij}} (I(i,j) - \mu_{Bh})^2}{W \times W}. \tag{2.6}$$

Third, the homogeneity measures (ξ_{Bh}) previously calculated in step 1 for each block in the image frame are sorted in ascending order.

Fourth, the block variances (σ^2_{Bh}) of the corresponding top 10% of the sorted block's ho-

mogeneous measures (ξ_{Bh}) are selected. This intricate sorting and block selection procedure is the main complexity residing in this noise estimation algorithm . This complexity is illustrated in Fig. 2.2, where the ordering of the homogeneous measures ($\xi_{Bh_1}, \xi_{Bh_2}, \xi_{Bh_3}, \cdots, \xi_{Bh_n}$) are represented in the central row, but it is in fact their corresponding variances, seen in the last row, that are used for further processing to generate the final variance. Hence, this procedure could be illustrated in software terms as ordering a sub-list of a linked list and represents the primary challenge of this hardware design.

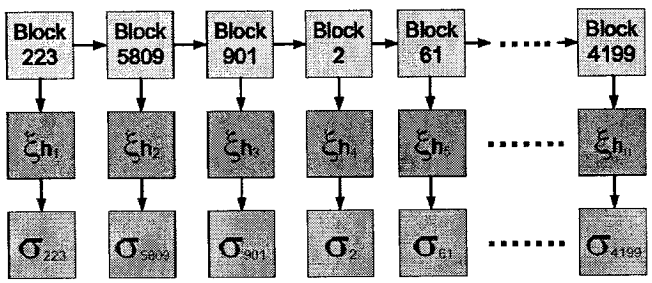


Figure 2.2: Homogeneous measure sorting.

Fifth, a reference variance (σ^2_{REF}) is calculated from the median of the variances of the 3 most-homogeneous blocks as shown in the Eq. 2.7. The top three values are represented by a, b, and c. The MAX and MIN functions find the maximum and minimum values amongst the a, b, and c terms.

$$MEDIAN(a,b,c) = a+b+c-MIN(a,b,c)-MAX(a,b,c) \tag{2.7}$$

Sixth, valid variances are selected by the difference between the logarithmic value of the reference variance and the logarithmic value of the individual block variances. This difference is then compared to a threshold (t_σ). This condition illustrated in Eq. 2.8 will

determine the similar homogeneities between blocks.

$$|log(\sigma^2_{Bh}) - log(\sigma^2_{REF})| < t_\sigma. \tag{2.8}$$

Seventh, the final variance (σ^2_n) resulting from this noise estimation process is produced from the average sum of the valid variances that pass the condition evaluated in Eq. 2.8.

2.2 Related Work on FPGA Implementations

The main focus in FPGA implementations of video processing algorithms is to process data rapidly to achieve real-time performances. Using different methods such as adapting algorithms to comply with hardware structures, parameterizing design architectures, pipelining and simplifying algorithms by reducing their order of complexity allows real-time performances to be attained.

Recent technological development in the field of FPGAs has increased their logic density and clock speeds to comparable ASIC performances at a much lower cost. It has now become feasible to implement DSP algorithms onto FPGAs to reach real-time performances. Nevertheless, complex algorithms implemented on resource constrained hardware systems can cause serious speed and quality downgrades.

In this section, related work on FPGA implementations of image enhancement algorithms such as de-interlacing, median filter and variance-based filtering techniques for image restoration, and convolution are discussed. The noise estimation algorithm used in this

research that was developed in [4] and described in section 2.1.3 has not been implemented in an FPGA before. Therefore, it has been impossible to find papers that have published exactly similar or equivalent hardware designs. The proposed FPGA implementation incorporates some interesting benefits which are drawn from comparisons to similar research material.

The paper in [14] focuses on a de-interlacing mechanism for image interpolation problems using non-linear filters. Since FPGAs are limited in resources, the Volterra model used in the de-interlacing mechanism was truncated at the third order to decrease its algorithmic computational complexity. The paper in [14] suggests to take a finite number of sums of higher order impulse responses to decrease the resource consumption. Thus, the decision to truncate the Volterra model seen in Eq. 2.9 of an M^{th} order to a third order allows the identification of non-linear systems which symmetrically distorts the input signal.

$$\begin{aligned} y(n) &= \sum_{k_1=0}^{L} h_1(k_1)x(n-k_1) \\ &+ \sum_{k_1=0}^{L}\sum_{k_2=0}^{L} h_2(k_1,k_2)x(n-k_1)x(n-k_2) \\ &+ \sum_{k_1=0}^{L}\sum_{k_2=0}^{L}\sum_{k_3=0}^{L} h_3(k_1,k_2,k_3)x(n-k_1)x(n-k_2)x(n-k_3) \\ &\vdots \\ &+ \sum_{k_1=0}^{L}\cdots\sum_{k_M=0}^{L} h_M(k_1,\cdots,k_M)x(n-k_1)x(n-k_M) \end{aligned} \tag{2.9}$$

Truncating to a lower order filter, which would contain linear and quadratic filters, could only detect non-linear elements which would skew the input. Hence, higher order Volterra models allow for even greater generalized analysis of a video data. Although the

algorithm was modified and a loss of quality was recorded, the use of a 2D aperture and pipelining technique gave significantly improved results for the Virtex II FPGA implementation. The timing analysis showed sufficient clock speeds for real-time broadcast television. The simulation results in [14] showed acceptable correlation between the original C code implementation and the VHDL hardware implementation.

Using a parallel and pipelined structure in [15], an adaptive temporal Kalman filter for white Gaussian noise filtering is implemented onto a Xilinx FPGA. The adaptation of the filter is executed by adjusting its parameters based on the variation of the noise statistics and the motion detected in the image sequence to render an optimum noise filter. The paper in [15] uses the Xilinx *Coregen*^{TM} tool extensively to yield optimal performance and reduce the design time. The papers in [15] makes clear that Kalman filter algorithms should be used in conjunction with pre-spatially non-linear filtered frames when impulsive noise is present in image sequences.

A 2D convolution core is implemented onto an FPGA in [16] for real-time image processing applications. Convolution is an essential operation in video and image processing. The paper in [16] presents a scalable high-level core generator capable of operating with various filter window size, window coefficients and input pixel data size for multiple 2D convolution configurations. The core uses a recoding scheme that exploits the window coefficients in order to reduce the number of add and subtracts exercised by the multiplications in the algorithm. The canonical signed digit recoding helps minimize the amount of partial products in the multiplication with the integer coefficients in the convolution. The noise estimation method implemented in this thesis is also constructed in such a way as to use power of twos coefficients to reduce the hardware complexity of the multiplication being executed by

the masks that detect the regions of homogeneity. The scalability feature presented in [16] allows the convolution design to be reused in many real-time video processing applications.

The paper in [2] demonstrates the use of embedded hardware directly into a camera to accelerate image processing applications. Application such as retinal vascular tracing used in the medical field is implemented in hardware since the software execution is too slow for real-time operations. The FPGA implementation in [2] of a smart camera technology allows surgeons to visualize the highlighted retinal image in real-time during surgery. By feeding the image data directly from the camera to the FPGA processing unit, the number of data transfers is reduced and the data are transfered in a fast and efficient manner due to the proximity of the hardware platforms. The software equivalent implementation would otherwise be too slow of a process if executed through the memory subsystem of a PC or workstation. Different masks are used to compose the 16 directional matched filters shown in Fig. 2.3, the same technique is used to compose the 8 masks shown in Fig. 2.1 which are implemented in this thesis to determine the homogeneity measure. The hardware architecture takes advantage of parallel design to compute the matched filter results and offer a great deal of speedup compared to the serial execution style of software programs. A similar design strategy is used in this thesis to calculate the filter responses of the masks in parallel to reduce the processing time and render real-time performances. An efficient pipelining strategy shown by the vertical bars in Fig. 2.4 of every add or subtract stages of the design in [2] is also used in this thesis to accelerate the frequency of operations and increase the performance of the system. The work in [2] maximizes its pipelining efforts by using a "ping-pong" tactic to store data in one memory while reading data from another memory to keep the system from having to stop its continuous processing. The

pipelining technique is also applied in this thesis to enable the processing of multiple frames continuously without having to stall the input video feed and to help achieve overall real-time performances.

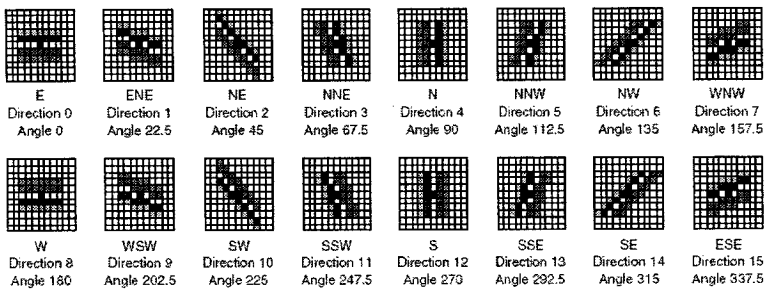


Figure 2.3: Diagram of 16 masks of the match filters [2].

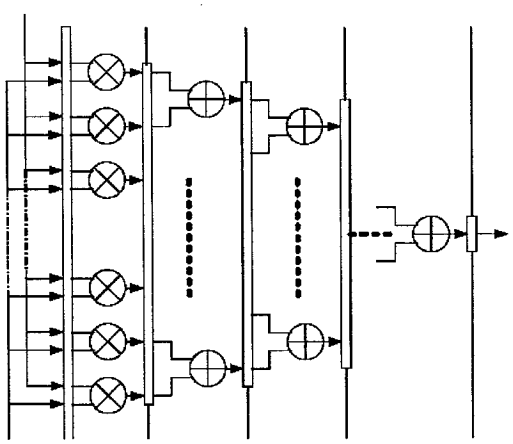


Figure 2.4: Diagram of pipeline stages between adder stages [2].

Nevertheless, the development of a complex 2D convolution system in [2] using 16 different mask sizes set in a large fixed scanning window to trace vasculature in retinal images is deemed un-flexible. The masks help the tracing algorithm find the direction to continue looking to find the next point on the vessel. The smart camera project in [2] uses wide-window matching filters but is limited to a set filter width. The masks used in this thesis to find the direction of homogeneity are scalable and the design's architecture is equipped to support many more filter sizes.

Some work has been done modifying the design of image enhancement filters in [17] to improve the quality of the results produced by the FPGA implementation of a median filter. Minor modifications to the 2D multi-shell median filter used for noise suppression is introduced to simplify the hardware implementation and speed up the processing time.

In the master thesis [18], the author implements popular DSP algorithms onto a Xilinx Virtex FPGA and a Altera FLEX FPGA chip. The video processing algorithms, rank order filter, morphological operators, and convolution processes, are designed in $Matlab^{TM}$ and in VHDL. Similar design methodology is used in this thesis to verify the quality of the results with respect to the original algorithm and to compare the functionality to the hardware VHDL results. The rank order filter, such as the median filter, is useful for smoothing and noise removal. The morphological image processing implementation is used in edge detection, restoration, and texture analysis. The implementation of the convolution spatial filter is used for low-pass filtering processes. Convolution involves a division by the number of pixels in the scanning window. The convolution algorithm in [18] was modified to reduce the complexity of the devision. Instead of dividing by the 3x3 window size (9), the author decided to divide by the closest power of 2 number such as 8. The algorithm change in [18] concluded an acceptable modification that still produced quality results.

The papers in [17] and [18] both implement 2D window-based filters using a fixed width. Both FPGA implementations are neither parameterized and do not offer a flexible solution. As mentioned in [4], different filter sizes are needed to adjust the noise estimation algorithm to better its performance according to the type of image that is being processed. Hence, it has been proven in [4] that using a filter window size of 3x3 results in a better estimation in less noisy images (PSNR $>$ 40 dB), whereas using a window size of 5x5 gives better results

in noisy images.

Case studies [19] performed on Xilinx FPGAs on how to improve or adapt image processing algorithms have been investigated to determine the most suitable architecture for image restoration designs. An important trade-off exists between the processing quality and the amount of logic available on an FPGA chip. In [19], the iterative image restoration algorithm is modified to process segmented parts of the image instead of the whole image at one time. Nevertheless, the FPGA is limited in the amount of parallel segments it can process. Therefore a finite number of segments are processed and stored once the restoration iterations have converged to an acceptable residual result. The rest of the image segments are processed subsequently. Another FPGA constraint, which also affected this thesis's work, was the limited amount of on-chip memory. In the paper [19], the degraded image is fetched from an external memory since the currently available FPGAs can not provide sufficient on-chip memory to contain images of practical sizes. The performance is decreased due the large amount of time used to transfer and load the data onto the FPGA. The papers in [19] finds its solution in increasing the number of bytes transfered onto the FPGA per clock cycles to maximize the bandwidth. Hence, [19] shows that resource planning and proper algorithm adaptation help find the optimal FPGA design with respect to run time and result quality.

```
1  initialize residual
2  while residual < ε
3      for i ← 0 to image_height - 1
4          for j ← 0 to image_width - 1
5              conv ← r0 x_{k-1} + r1 ∑ eight 1-neighbor pixels
6              x_k ← βy + x_{k-1} − βconv
7      update residual
```

Figure 2.5: Iterative Restoration Algorithm [19].

Nonetheless the effort set forth in [19] to adapt the algorithm to reach high quality results lack the speed to yield real-time performance. The image restoration paper in [19] uses an iterative algorithm that does not deliver real-time performance. The restoration process shown in Fig. 2.5 iterates until the criteria of convergence is met. The algorithm is modified to process segmented parts of the image independently instead of the whole frame at once to comply with the resource and bandwidth limitations. The resulting time taken to restore the 256x256 gray-scale images shown in [19] is below real-time performance standards. In [19], it is concluded that the resource constraints of the FPGA affects the design outcome and the decision is made to trade quality of results over processing speed. Furthermore, it is also concluded that speedup could be attained through higher end FPGAs and a pipelined architecture which is one of the main design strategies endorsed in this thesis.

Other researchers concentrate on adaptive and reusable image enhancement algorithms using a histogram modification technique based on statistical processing [20]. The system calculates the mean and the variance of each image that are used to influence and change the value of each pixel of the following image. The implementation and simulation is performed on an Altera FPGA platform. The reusability strategy of the design in [20] has been obtained through the technique of using VHDL generics and global packages to parameterize the design. Thus, the paper in [20] illustrates that the number of bits per pixels or image resolution can be easily modified without changing the Resistor Transistor Level (RTL) behavior by reusing the same architecture. It is important to note that the paper in [20] concentrates on single or specific arithmetic procedure where a simple scalable variance generation algorithm is implemented.

In contrast to the FPGA implementation developed in [20], the work accomplished in

this thesis combines many arithmetic procedures and works to implement a more complete system for noise estimation. In [20] the mean is calculated with each pixel in the frame (image is not segmented into blocks) and the variance is evaluated using this mean. This thesis proceeds in a similar fashion to [20] in that it calculates the mean and the variance. Eq. 2.10 shows the calculations of the mean and the squared mean. Eq. 2.11 generates the variance where I_i is the luminance of the pixels and n is the total number of pixels per image.

$$\begin{aligned} m = E(I) &= \frac{\sum_{i=1}^{n} I_i}{n}. \\ E(I^2) &= \frac{\sum_{i=1}^{n} I_i^2}{n}. \end{aligned} \tag{2.10}$$

$$\sigma^2 = E(I^2) - E(I)^2. \tag{2.11}$$

This thesis includes a higher level of complexity in that it takes into account the direction of the homogeneity, segments the images into blocks for local processing, sorts the homogeneity measure to find a reference variance and proceeds to further analysis with respect to a threshold (to determine the maximal affordable difference between the true variances and the estimated variances) to generate the variance. The hardware complexity of these extra algorithmic steps represents a more intricate implementation in this thesis. The same reusability strategies used in [20], such as generic design methodology and global packages, are also incorporated in this thesis's architecture. Although the paper in [20] presents a scalable design in terms of pixel data size and image resolution, the architecture

in this thesis is built to support multiple filter sizes and could be modified to support the same parameters.

2.3 Summary

In this chapter, the importance of noise reduction and the significance of noise estimation in image enhancement and general DSP algorithms have been discussed. Noise modeling theory has been defined to better understand the composition of video signals and distinguish the different types of noisy components than can deteriorate a video signal. The PSNR numerical method that serves to evaluate the quality of the noise estimation algorithm has been recapitulated. Noise estimation has then been studied in more details, various types of estimation methods have been enumerated and described. The thesis's homogeneity-oriented noise estimation algorithm has been reviewed. The intricacies and the complexity of this estimation process have been detailed and the sequence of operation has been explained with the help of equations and figures. Related DSP implementations on FPGAs have been mentioned and the the interesting advantages of this thesis's work have been put in contrast to other similar research material. The following chapter will discuss the design flow, the algorithm complications and necessary adaptations for hardware compliance, and the architecture strategies will be analyzed and justified.

Chapter 3

Design Flow and Architecture Analysis

In this chapter, the FPGA design flow of the proposed hardware architecture is described through a series of design steps. Some design assumptions are stated. The necessary algorithm adaptation sequence that this noise estimation method requires to comply with FPGA constraints is presented. The main noise estimation algorithm modifications are described and justified. The design strategies utilized to attain the thesis's objectives are explained. The reasoning for choosing such design strategies is demonstrated. Further discussion on additional scalability features regarding the proposed architecture is addressed.

3.1 FPGA Design Flow

The design flow of the proposed noise estimation algorithm mentioned in section 2.1.3 is illustrated in Fig. 3.1. The algorithm must first be examined, then the hardware creation and implementation is performed. The design sequence consists of 7 general steps which include the algorithm analysis and adaptation, architecture development, software model construction and validation, VHDL program creation and compilation, VHDL code functionality validation through hardware simulations, synthesis, and netlist place and route procedure.

Step 1 is to examine the noise estimation algorithm selected to establish whether its hardware implementation is feasible. The algorithm is broken down into many parts and each of these parts is studied to determine if the arithmetic procedures of the algorithm could be implemented in a hardware description language (e.g., VHDL) to program the FPGA. For example, some arithmetic procedures such as logarithmic functions can not be programmed in VHDL and alternative solution must be studied. Some modifications and adaptations to the software algorithm had to be performed to make the selected noise estimation algorithm compliant to the FPGA hardware.

Step 2 begins by establishing the design strategies which must be formulated before the design of the architecture is to commence. These design strategies are important to justify the way to design the hardware architecture of the noise estimation system. These design strategies encompass the design goals that this thesis's hardware implementation strives to attain. Consequently, the final design's organization and hardware optimizations might take on very different architectures depending on the adopted strategies. Substantial

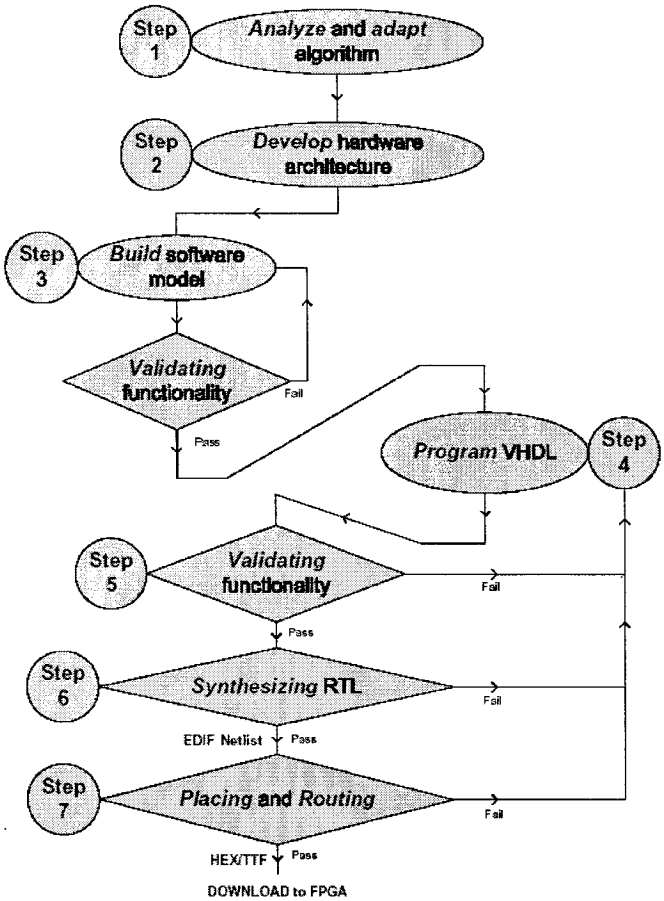


Figure 3.1: FPGA Design Flow.

efforts must be put in step 2 (i.e., the construction of the hardware architecture) to deliver an optimum hardware implementation that will have been modeled by following the initial design strategies. In hardware design, changing the architecture during the latter stages of the design flow can be very time consuming because it may require a complete reorganization of the design structure. Although it is not fatal in FPGA design, the turn around time is longer than modifying a software program.

In step 3, once the architecture is considered stable, a software model of the modified algorithm is programmed using high-level languages such as C or C++ to imitate the

hardware architecture. The software model is then simulated and used to validate the correct functionality of the new hardware architecture. This time-saving design step is an efficient way to test the newly developed architecture without having to go through the entire hardware design flow.

In step 4, the newly developed and validated architecture is then programmed in VHDL and compiled using the *Synopsys VHDL Analyzer*^{TM} to detect syntax errors.

In step 5, the compiled version of the VHDL code is simulated and its functionality is validated with the *Synopsys VHDL Simulator*^{TM} hardware simulation tool. The hardware simulation results are compared to the software simulation results to validate the hardware design's correctness and level of quality. If the hardware results are not satisfiable, then a change or optimization in the VHDL code or even the hardware architecture might be needed. It must be noted that the FPGA design flow is iterative and that many corrections must be added concurrently while creating the hardware design.

In step 6, once the VHDL design is deemed stable, the synthesis of the VHDL code is performed using the *Synoposys Design Compiler*^{TM} tools. This step will translate the design from a resistor transistor level (RTL) description into an optimized *netlist*, e.g., a file containing technology specific digital cells required for the actual design formally known as an EDIF (Electronic Design Interchange Format) file. The synthesis procedures will allow the designer to get preliminary timing results and will also give a general overview of the amount of resources being used. In other words, successful synthesis results will confirm that the design is hardware compatible.

In step 7, the *Xilinx Project Navigator*^{TM} tools are used to mapping the synthesized

netlist design, placing the logic and routing the signals. This step is specific to the Xilinx FPGA Virtex technology. In the first part of step 7, the mapping consists in grouping the logical elements of a netlist into configurable logic blocks (CLBs) and input/output blocks (IOBs). These fundamental building blocks, depicted in Fig. 3.2, are organized in an array and they are used to build combinational and synchronous logic designs. A CLB element, shown in Fig. 3.3, is comprised of 4 similar slices and a general switch matrix that permits communication to other neighboring CLBs. The slices, seen in Fig. 3.4, are composed of a grouping of 4-input look-up tables (LUTs), 16 bits of distributed SelectRAM memory (RAM), 16-bit variable-tap shift register, general purpose flip-flops and arithmetic logic (adders, logic gates, etc.). The IOBs include a group of buffers, latches, flip flops, and input/output pads used for sending signals off of the FPGA and receiving signals onto the FPGA. These IOB interfacing elements border the FPGA as seen in Fig. 3.2.

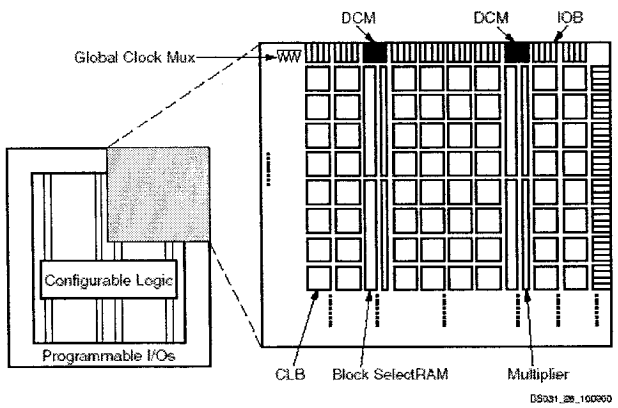


Figure 3.2: Internal FPGA Composition [21].

The second part on step 7 is the place and route procedure. This is the process of examining logic that has been mapped to CLBs and IOBs to determine which locations of the target FPGA are occupied and what optimum routing (wiring) should be used to connect them. The resulting HEX (Hexadecimal Intel-format file), TTF (Tabular Text File)

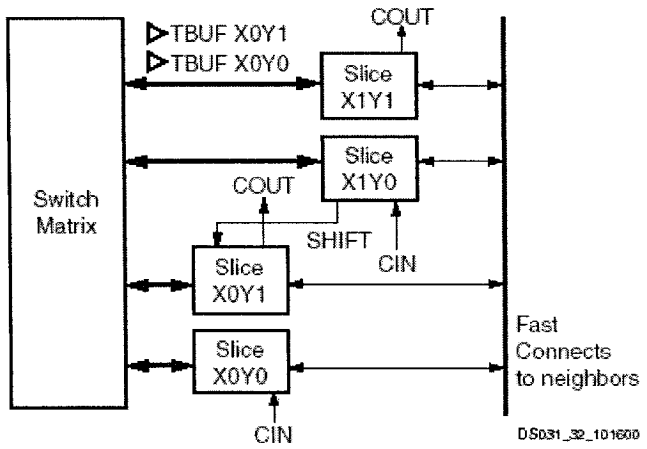


Figure 3.3: Configurable Logic Block (CLB) [21].

or BIT (binary Xilinx-format configuration file) files can be loaded into an FPGA. These bit patterns will end up controlling logic gates and filling memory and registers to be operated on a Xilinx FPGA.

It is important to note that in the case where one of these previous steps fail, the designer may have to re-evaluate the VHDL program, or even the hardware architecture, and run through the design flow numerous times to make sure that the designs complies with the hardware specifications and design rules. It is also important to note that this design flow applies mainly to Xilinx FPGA-oriented designs. Thus, this design flow would have to be modified, from step 2 and on, to render an ASIC implementation since some of the design techniques used to build the architecture (step 2) rely heavily on Xilinx internal components which would not be available when fabricating an ASIC chip.

Some design assumptions are described in the following sub-section.

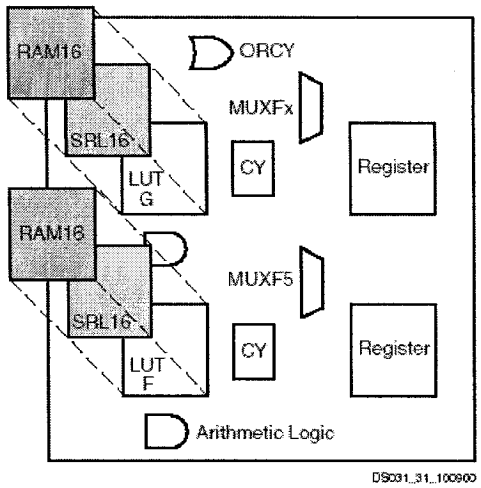


Figure 3.4: Slice Composition [21].

3.1.1 Design Assumptions

The proposed noise estimation design was based a some assumptions. The pixel data size being used is an 8 bit grayscale signal. The video frame rate is set to the PAL format which follows the ITU-R-BT-601 recommended standard (formerly known as CCIR-601). The PAL video resolution is 864x625 pixel per frame. The streaming video is interlaced which renders resolution of 864x312 pixel per field. The active pixels of the video frame is 720x576 pixels per frame and that of the interlaced field is 720x288 pixel per field. The assumption resides in the approximation of the vertical blanking time also known as the downtime. Thus, the input interlaced video sequence is assumed to have an active and a passive pixel type enclosed within the entire global resolution of 864x312 pixel per field. It is presumed that the active pixels are grouped together and are acquired in a row without ever having any horizontal blanking. The vertical blanking is subsequently appended to the list of active pixels. Therefore, 269568 valid pixels are streamed as input. Incorporated within this input stream is a set of 207360 active pixels followed by a set of 62208 downtime

pixels as described in Eq. 3.1.

$$\begin{aligned} \text{Valid pixels per field} &= 864\frac{pixel}{row} * 312\frac{row}{field} = 269\ 568 \text{ pixels/field} \\ \text{Active pixels per field} &= 720\frac{pixel}{row} * 288\frac{row}{field} = 207\ 360 \text{ pixels/field} \\ \text{Downtime pixels per field} &= \text{Pixels per field - Active pixels per field} \\ &= 269\ 568 - 207\ 360 \Rightarrow 62208 \text{ pixels/field} \end{aligned} \tag{3.1}$$

These downtime pixels are necessary too pad the incoming stream of data. This scheme is mainly used to incorporate the vertical blanking to the input signal to mimic the composition of a real life video stream. Table 3.1 summarizes the basic elements of the input video stream.

Element Specification	Value
Valid resolution	864 * 312 pixels
Active resolution	720 * 288 pixels
Active pixel count	269568 pixels
Passive pixel count	207360 pixels
Passive pixel count	62208 pixels
Pixel rate	13.5 MHz
Valid field time	0.02 seconds
Active field time	0.01536 seconds
Downtime time	0.004608 seconds
Pixel data size	8 bit

Table 3.1: Input video signal specifications

The next section describes the necessary adaptations that the algorithm experienced to become hardware compliant as explained in first part of the design flow.

3.2 Algorithm Adaptation for Hardware Compliance

The original algorithm underwent several modifications to conform to the VHDL hardware description language limitations and to comply with the hardware constraints of the FPGA platform. This necessary process helped reduce the hardware complexity and worked to make the software algorithm portable to various FPGA systems. It is known that some mathematical expressions (e.g., logarithmic functions), ordering functions and number precision (e.g., floating-point precision) are difficult to realize in hardware. Upon studying the original algorithm, the important modifications adopted consists in reducing the precision of the software algorithm to a fixed-point representation, using a look up table to perform logarithmic functions and implement an alternate sorting procedure. The goal is to find the optimum hardware design for an FPGA platform that would compare closely in terms of quality and accuracy with the original algorithm results without having to significantly change the original algorithm. The evolution process that the original algorithm undergoes and the main modifications that are applied to this noise estimation algorithm are described. The strategies utilized to reach the main design objectives (to build a real-time performance noise estimation hardware design, to minimize logic consumption, and to support a scalable architecture) are explained and justified in the following sub-sections. Furthermore, this section also discusses various architecture issues such as the expansion and shrinkage of filter sizes, the support of other parameters like variable image resolution, the implementation of variations of the noise estimation algorithm and the possible implementation of external look-up table for the logarithmic function.

3.2.1 Floating-Point to Fixed-Point Precision Alternative

In the original noise estimation software C program, the algorithm's calculations are performed with floating-point variables. Implementing floating-point numbers in hardware is complicated and must be developed manually. Unlike the C language, VHDL does not support floating-point packages that an FPGA chip is capable of synthesizing. In other words, the designer must build the floating-point structure by implementing a single-precision 32-bit or 64-bit floating-point variable type as recommended by the IEEE Society standards (Standard 754-1985). Numerous research work developing floating-point structures in hardware, as in [22, 23] to cite a few, have been implemented on FPGA platforms. Nevertheless, in this thesis a different strategy was chosen to use integer signal types that would consequently result in a simple and fast prototyping hardware design. A software model of the original algorithm was implemented using integer variables, it revealed that the integer-based design rendered imprecise results. The calculations using multiplying, dividing and squaring functions produced truncated fractional parts. Knowing that the algorithm required further precision, the following more complex fixed-point variable scheme was developed to improve the accuracy of the results.

The fixed-point method implies using fractional numbers without resorting to using floating-point values to perform arithmetic calculations. Fixed-point representation requires that the designer create a virtual decimal place in between two bits of a given signal that will be instantiated in the VHDL module. The main concern for the designer of this technique is to keep track of the decimal point throughout the arithmetic calculations. The fixed-point implementation is achieved by multiplying the decimal terms by a power of 2 number (e.g.,

2, 4, 8, 16, etc.) and performing the arithmetic calculations with these up-scaled values. In hardware, this implementation is achieved by simply shifting the binary data to the left and by padding the least significant bits with zeros. A fixed-point transformation example of 4 bit data with an added 2 bits of precision can be as seen in the Fig. 3.5. The proposed hardware design uses these shifted values to process the noise estimation algorithm and generate the final variance.

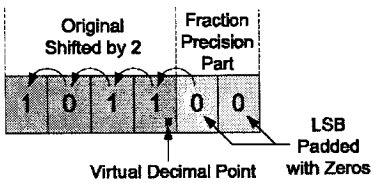


Figure 3.5: Fixed-Point Representation using 2 bits of precision.

In the fixed-point representation, the accuracy added to binary terms is measured by the amount of shifted bits which represent the fractional part. As seen in Fig. 3.5, two bits of extra precision allows for $2^{-1} = 0.5$ and $2^{-2} = 0.25$ added fractional accuracy. During the experimental phase of this work, using an integer-based implementation of the algorithm, some problems arose. For example, a particular frame in a video sequence generated a variance which was approximated to an integer value of 3. The precision problem was apparent in this case since its original floating-point-based implementation had produce a variance value of 3.95. The approximation using integer arithmetic was deemed unacceptable because of the huge loss in accuracy. A fixed-point-based implementation with two bits of extra precision was then used in this same context and generated a resulting variance of 3.75. The new fixed-point variance (3.75) added a fractional part (.75) to the arithmetic calculations which compared much better to the original floating-point value (3.95) than the previous integer value (3). Consequently, when more bits of accuracy are

added to the terms in the equations then the approximation factor of integer only arithmetic will be reduced and the overall accuracy of the results will approach floating-point precision.

The downside to this technique is that it initially widens the size of the data signals which, in turn, increases the size of the overall hardware circuit. Hence, this floating-point scheme increases the amount of hardware logic needed to implement larger arithmetic units that employ larger input signals. Since an FPGA has a restricted amount of logic, there exists a trade-off in choosing a number of extra bits of accuracy between, on one hand, attaining a maximum amount of arithmetical precision and, on the other hand, implementing a minimum sized hardware circuit.

Software experimentations with different levels of accuracy and the study of the hardware feasibility of these different tests have been performed to find the optimum relationship between the quality of the results and cost in circuit size of the implementation. The software test trials were carried out to compare the original C code with a modified version of the C code replicating the hardware implementation. The PSNR plots are generated using *Matlab*^{TM} tool where the software results (e.g., *PSNR-AVG-30-ORIG*) are represented by the solid line patterns and the hardware results (e.g., *PSNR-AVG-30-FPGA*) are represented by the dashed line pattern as depicted in Fig. 3.6.

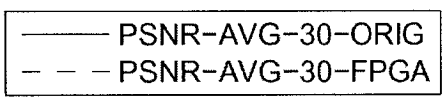
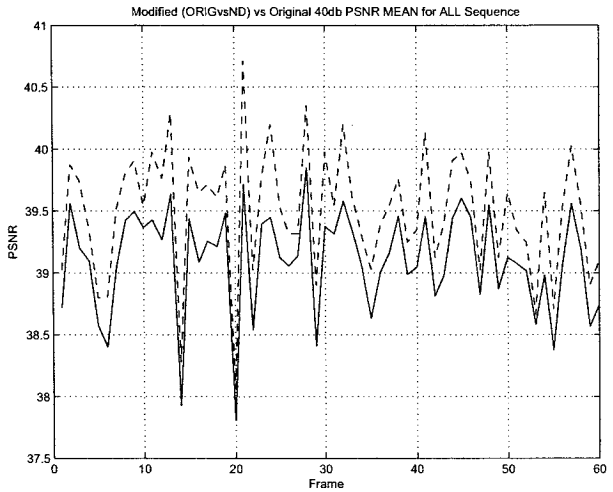


Figure 3.6: Matlab line legend.

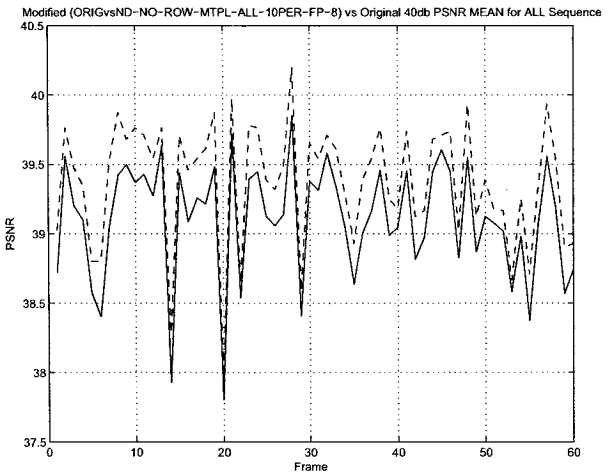
The results seen in part a) of Fig. 3.7 show the difference between integer-based (modified version) and floating-point-based (original version) implementations. The difference between the fixed-point-based (modified version using 6 bits of accuracy) and the floating-

point-based (original version) implementation of the noise estimation algorithm is shown in part b) of Fig. 3.7. These PSNR results have been obtained using an average of 3 different video sequences (*Train*, *Prlcar* and *Flower_Garden*) containing a noise level of 40db. These results clearly show the advantage of using fixed-point implementation over integer-based variables. Other tests validating the accuracy of the fixed-point implementation have also been performed on video sequences containing noise levels of 20db, 25db, 30db, and 50db.

After having tested in software the fixed-point-based implementation using additional bits of precision (e.g., exceeding 6 bits), the tests showed no significant increase in the resulting PSNR precision. Thus, the number of bits of accuracy for the final variance results was chosen to be 6 bits for this thesis. This decision was based on the in depth study of the corresponding hardware architecture and the acceptable quality of the PSNR test trial results as shown in part b) of Fig. 3.7. The design's architecture had to be examined thoroughly before a final decision could be taken because the increase in input data size has important repercussion the entire circuit. Furthermore, the size of all of the arithmetic units must be re-evaluated to determine the feasibility of the circuit. Using 6 bits of precision rendered a realizable FPGA circuit except for a problem with one division core. This division core, to be generated by the Xilinx *Coregen*^{TM} tool, occurs at the end of the noise estimation algorithm when the sum of the valid variances is divided by the number of valid variances. The problem this architecture presented was to create the need for a 37 bit divider which can not be generated by the Coregen tool. This tool can only generate divider cores having dividend sizes of up to a maximum of 32 bits. This complication was solved by the creation of a scalable divider unit capable of supporting up to 128 bit dividend sizes. A scalable non-restoring pipelined array divider module was implemented and tested "in-



(a) 40db PSNR of Orig vs Mod integer implementation



(b) 40db PSNR of Orig vs Mod fixed-point implementation

Figure 3.7: 40db PSNR of Original versus Various Modified C code implementations

house" to replace the divider cores. The details and the theory of this array-based divider are described in section 4.2.

The algorithm modification from a floating-point representation to a 6 bit extra precision fixed-point hardware structure was the first of the adaptations needed to comply with the VHDL hardware description language. The next section presents a different adaptation which deals with the complications brought on by the sorting of the homogeneous measures in the noise estimation algorithm. This intricate task of ordering the homogeneous measures is fairly difficult to implement in hardware to provide real-time performance.

3.2.2 Hardware Sorting Complications

The main bottleneck in the proposed algorithm was the sorting of the block's homogeneous measures of the segmented image in real-time. Sorting data in hardware is a complicated and time consuming task due to the fact that logic constrained FPGA hardware devices are not capable of processing data in the same way as software applications do. Some hardware sorting techniques are analyzed, and different attempts to solve the sorting complications are evaluated.

Noise estimation without sorting

At first, attempts were made to eliminate the sorting step of the algorithm which would greatly reduce the complexity of the circuit. Modified software models were implemented to test the noise estimation algorithm without the sorting of the homogeneous measures but the quality of the results were significantly affected and were not considered satisfactory

due to the difference of more than 3 dB in some cases between the original and modified simulation results as seen in Fig. 3.8. Thus, the sorting step was deemed an essential part of the algorithm and needed to be implemented.

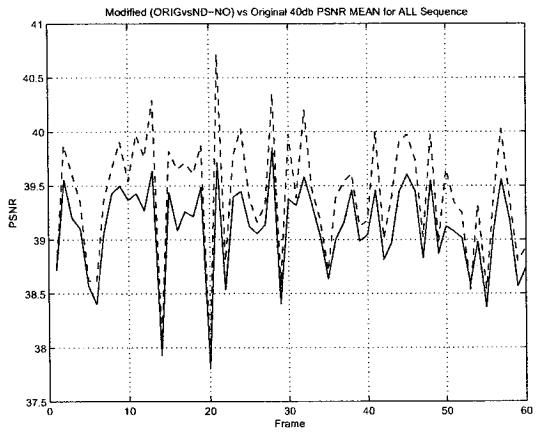


Figure 3.8: 40db PSNR of Orig vs Mod without sorting.

Noise estimation with localized sorting by row

The second attempt consisted in trying a localized ordering approach to sort the block's homogeneity measure by row instead of trying to sort all of the block's homogeneity measures contained in a field. The reason for choosing to sort the block's data in a row fashion is to make use of the extra time at the end of a line during the occurrence of horizontal blanking. Hence, the time to store N lines of pixel data (where N is the vertical length of the scanning window) can be used between the last active window of the previous row and the first active window of the next row. This time would allow the hardware to sort all of the block's homogeneous values contained in a row and take the top 10% of this row's ordered list. The number of data blocks to be sorted would be greatly reduced instead of having to sort all of the blocks included in a frame. The results were still not satisfactory

in some cases between the original and modified simulations as seen in Fig. 3.9 due to the fact that single row processing was too localized to grasp the total homogeneity of the field. Other attempts were devised to evaluate multiple row processing to verify if the analysis of a broader portion of the field would render better results. The outcome did not show much improvement from the previous row processing attempt and this idea was also rejected. Therefore the research proved that it had become imperative to sort all of the blocks in a frame to generate the best qualitative noise estimation results.

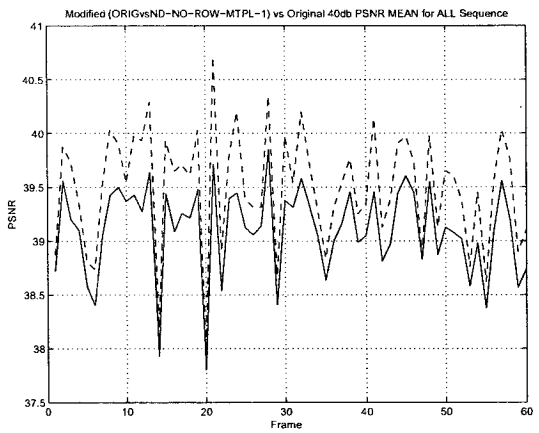


Figure 3.9: 40db PSNR of Orig vs Mod sorting by row.

Analysis of popular sorting algorithms

Popular sorting algorithms were investigated to determine their feasibility in hardware. Sorting algorithms usually process data in a sequential fashion by comparing and swapping data contained in an un-ordered list until the final list of data is ordered. For example, one of the oldest and simplest sorting method, the *bubble sort* algorithm works by comparing each item in the list with the item next to it, and swapping them if required. The algorithm repeats this process until it passes through the entire list without swapping any items. This

method is known to be the slowest sorting method and would be very time consuming in hardware due to the large amount of memory accesses it needs to perform to fetch and store data every time a swapping action is required. Furthermore, this technique's sequential processing style, needing many passes to complete it's sorting, is also a contributing factor to increasing the processing time making this method impractical for real-time performance. Other sorting algorithms have been considered such as the extremely fast *quick sort* algorithm. This massively recursive technique is based on a complex divide-and-conquer approach which is simple in theory but very difficult to put into software code. This method was discarded because of its recursive nature. Other standard algorithms for sorting and searching computations (e.g., merge sort, tree sort, binary search, etc.) also work with recursive formulations, yet the problems is that some PC-based synthesis tools do not support recursion while other tools (e.g., *Synoposys Design Compiler*^{TM}) can render recursive functions under strict conditions (i.e., when the compiler can determine a priori the number of recursive calls). Hence, other alternatives were examined to resolve the sorting problem.

Related material on hardware sorting revealed that some work has been implemented onto an FPGA such as median filters used for noise removal. The median of this non-linear filter is defined by the middle element of a ordered list. In [24], a 3x3 sliding window is used to scan the video frame. This paper's implementation is capable of sorting the 9 elements in the sliding window on every clock cycles rendering a real-time performance. In [16] the sorting technique also uses a 3x3 sliding window and is based on bit voting system to help reduce the area consumption. These methods provide real-time performance but are only required to sort a few items. In the case of this noise estimation algorithm, a large number of data must be sorted. The algorithm's sorting complexities are presented in the next

section to help the reader understand the problematic task of ordering large lists of data in hardware.

Analysis of the noise estimation's sorting complexity

The complication resides in the fact that the input data rate (fixed PAL or NTSC standard rate) frequency is extremely fast and makes it difficult to sort large list of data between subsequent frames. For a better understanding, it is necessary to take a look at the numbers. The video processing algorithm follows the Phase Alternation by Line (PAL) standard format, using a resolution of 864x625 pixel per frame (864x312 per field) according to the ITU-R-BT-601 (International Telecommunication Union) standard. The international ITU organization was formerly known as CCIR-601 (International Radio Consultative Committee). The valid PAL resolution is 720x576 pixels per frame generating an interlaced video sequence with a field resolution of 720x288 pixels per field. For this example, the homogeneous measure (ξ_{B_h}) and the variances ($\sigma^2_{B_h}$) are generated from a 5x5 sliding window (using a total of 25 pixels). This data is calculated for each block contained in a field. The total number of blocks per field using a 5x5 sliding window amounts to 8294 blocks and is calculated in Eq. 3.2.

$$\begin{aligned}
\text{Number of pixels per field} &= 720\tfrac{pixel}{row} * 288\tfrac{row}{field} \\
&= 207\ 360 \text{ pixels/field} \\
\text{Number of pixels per block} &= \text{block height * block length} \\
&= 5 * 5
\end{aligned}$$

$$
\begin{aligned}
&= 25\frac{pixels}{block} \\
\text{Number of blocks per field} &= \frac{207360\frac{pixel}{field}}{25\frac{pixel}{block}} \\
&= 8294 \text{ blocks/field} \qquad (3.2)
\end{aligned}
$$

Only the top 10% of the $\sigma^2_{B_h}$ (approximately 800 values) of the corresponding 8294 sorted values of ξ_{B_h} are needed for future processing. The rate at which the pixel data enter the processing unit is 13.5MHz. The rate at which the value ξ_{B_h} and $\sigma^2_{B_h}$ are generated is every 5 pixels or every 5 clock cycles. Thus providing approximately 370 nanoseconds (NS) to sort the ξ_{B_h} values between valid consecutive blocks as seen in Eq. 3.3.

$$
\begin{aligned}
\text{Consecutive block generation rate} &= \text{input pixel frequency / number of clock cycles} \\
&= \text{13.5 MHz / 5 clock cycles} \\
&= \text{67.5 MHz} \\
\text{Time between consecutive blocks} &= \frac{1}{67.5MHz} \\
&= \text{370 ns} \qquad (3.3)
\end{aligned}
$$

In review, 144 data block values are generated in each row of an image field, as seen in Eq. 3.4, and 48 rows exist in one field, as seen in Eq. 3.5. Using these values, Eq. 3.6 illustrates that only 8208 entire blocks are considered instead of 8294 blocks because of the way the field is segmented.

$$
\text{Blocks per row} = \frac{720\frac{pixel}{row}}{5 \ block \ length}
$$

$$= 144 \text{ blocks/row} \tag{3.4}$$

$$\begin{aligned} \text{Rows per field} &= \frac{288 \frac{rows}{field}}{5 \ block \ hight} \\ &= 57.6 \text{ rows/field} \\ &\approx 57 \text{ rows/field} \end{aligned} \tag{3.5}$$

$$\begin{aligned} \text{Number of blocks per field} &= 144 \frac{blocks}{row} * 57 \frac{rows}{field} \\ &= 8208 \text{ blocks/field} \end{aligned} \tag{3.6}$$

From this study, we conclude that approximately 8000 ξ_{B_h} block values are to be sorted when processing with a 5x5 sliding window. However, the sorting section of the noise estimation algorithm must generate the variances that correspond to this ordered list of ξ_{B_h} values. The question remains, how many blocks can be sorted using hardware logic in a relatively small period of time that would allow real-time performances?

Feasibility analysis for combinational hardware sorting

Various combinational hardware designs were implemented and simulated in an attempt to sort the large list. Several architectures were built to operate similarly to a software ordered link list without using the data structure pointers. The list is to be ordered in terms of the homogeneity measure (ξ_{B_h}) block values. Each ξ_{B_h} element of this list is individually linked to their corresponding variance ($\sigma^2_{B_h}$) block values. The study aimed at finding

the maximum possible number of elements that could be ordered using a combinational hardware sorting scheme within the minimum allotted time of 370 ns determined previously in Eq. 3.3. Thus, sorting the elements "on the fly" between subsequent generations of block data.

The combinational design was simple and worked to place the new generated block data values at their proper position in an ordered list. The new data was compared to each element of a ordered list. Once the position of the new data was found within the ordered list, the rest of the list was consequently shifted to make place for the new data. This hardware design placed new elements into the ordered list in a single clock cycle. The design implementation was strictly combinational which, in turn, did not include any pipelined structures (no latency). Fig. 3.10 illustrates the hardware sorting circuit diagram of this sorting scheme.

Initially, a combinational VHDL design was implemented using this sorting scheme to order 10 ξ_{B_h} values. Different VHDL coding techniques were used to optimize the design to reduce the critical path. The resulting synthesized design revealed fast processing times and minimal area consumption. Another design was then implemented to sort 50 ξ_{B_h} values. The synthesis and mapping procedures performed on a Xilinx XC2V2000-FF896 (Virtex II Family) chip showed that a substantial amount of logic was used just to sort 50 elements. According to Fig. 3.11, approximately 16% of the FPGA logic was needed to implement this sorting scheme. Sorting 50 elements seemed feasible but the realization of this sorting scheme demanded quite enough hardware. The timing of the sorting scheme was deemed unacceptable because this method needed to traverse the entire ordered list every time a new value was to be added. The larger the list, in the case of a 5x5 processing window is

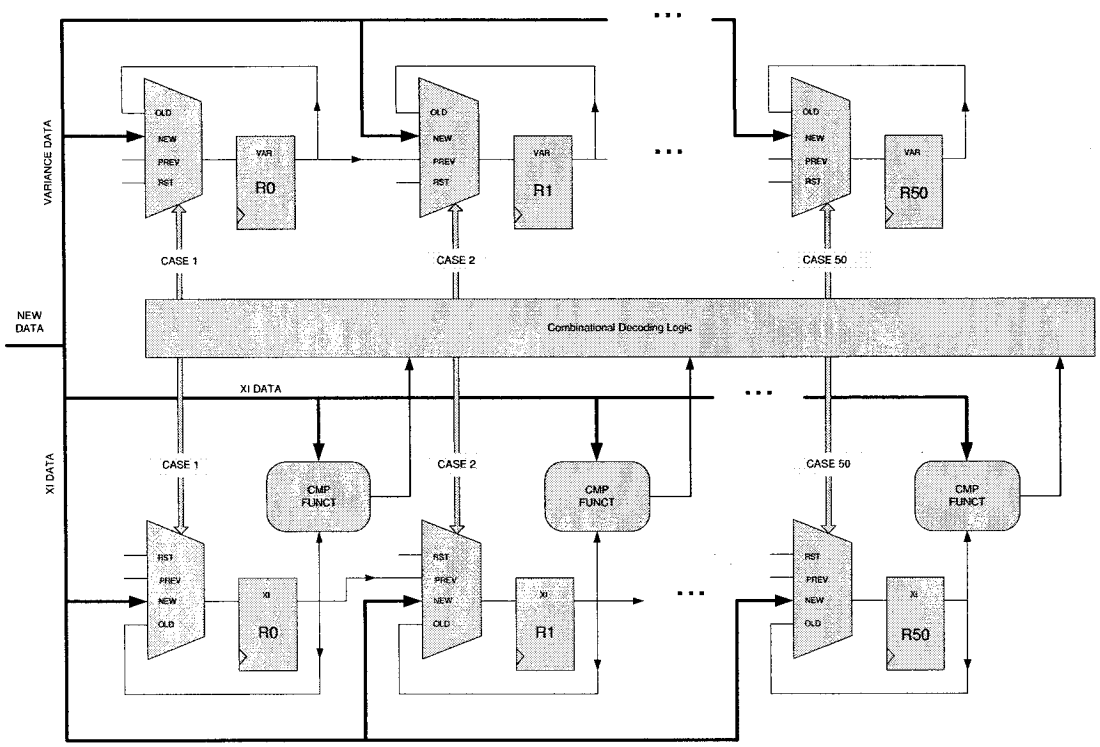


Figure 3.10: Combinational hardware sorting diagram.

approximately 8000 values, the longer the sorting time. These results have forced us to find a better solution to sorting large lists.

```
Mapping results:
-----------------
Number of Slices:                1,730 out of 10,752   16%
Number of Slices containing
Number of Slice Flip Flops:      1,369 out of 21,504    6%
Number of 4 input LUTs:          3,330 out of 21,504   15%
Number of bonded IOBs:              43 out of    624    6%
-----------------
Total equivalent gate count for design:  33,332
```

Figure 3.11: Mapping results from the 50 element sorting implementation.

Approved hardware sorting method

The study of combinational circuits to effectuate the sorting was considered too slow and an alternative hardware sequential method was evaluated to solve this sorting issue. The

sequential sorting algorithm adopted is mathematically known as *counting sort* and was devised in 1954 by Harold H. Seward in [25]. Harold H. Seward was a computer scientist at the Massachusetts Institute of Technology where he published works on LSD (Least Significant Digit) radix sort, using a stable counting sort. In 1956, Edward H. Friend developed an efficient utilization of an electronic computer system for the sorting of large amounts of data using an improved version of the counting sort algorithm. In [26], Edward H. Friend suggested counting the digits in the next column while rearranging numbers in the current column allowing this technique to reduce the number of memory accesses. The counting sort technique is adapted and optimized for FPGA designs in [27] as it works to reduce the consumption of FPGA logic resources. Indeed, the proposed design applies the counting sorting algorithm from [27] using a single memory element, and reusing it to execute the multiple sequential steps of the algorithm.

The counting sort algorithm is a linear time sorting algorithm used to sort items belonging to a fixed and finite set of keys. In the case of the noise estimation algorithm, the finite set is determined by the fixed size of the homogeneity measure (ξ_{B_h}). The range of the finite set is determined by the size in bits of the homogeneity measure values. The algorithm proceeds by defining an ordering relation between the items from which the set to be sorted is derived. In the case of the noise estimation algorithm, the relation exists between the the homogeneity measures (ξ_{B_h}) and their corresponding variance values ($\sigma^2_{B_h}$).

Using a simpler notation, given an original unsorted array $A\{1 \cdots n\}$ of length n (where n = maximum number of items to sort), the algorithm requires two more arrays to perform the counting sort. An array $C\{1 \cdots k\}$ (where k is the maximum possible value of an item) provides temporary working storage to execute the histogram sort and to calculate the

address indexes and an array $B\{1 \cdots n\}$ to hold the sorted output. In the first pass, the array A is initialized with zero values. The second pass counts the occurrences of each key to build the histogram in the auxiliary C array. The third pass calculates the index addresses by accumulating a running total of the keys. This is essential in order for each auxiliary entry to get the running total of the preceding keys. The fourth pass puts each item, whether its occurrences has been repeated or not, in its final place in array B according to the auxiliary entry for that key. The algorithm sequence is illustrated in the Table 3.2.

Steps	Descriptions
First Pass $\{1 \Rightarrow n\}$	Initialize Array A to zero
Second Pass $\{1 \Rightarrow n\}$	Build Histogram by counting the occurrences of each key
Third Pass $\{1 \Rightarrow k\}$	Calculate the index addresses by accumulating a running total
Fourth Pass $\{1 \Rightarrow k\}$	Extract sorted list into array B

Table 3.2: Counting Sort Algorithm Sequence.

The Eq. 3.7 takes into account each pass of algorithm and renders the total complexity of the counting sort.

$$\theta(n+k) = 2 \cdot \theta(n) + 2 \cdot \theta(k). \tag{3.7}$$

The Counting sort is a stable sort (i.e., where multiple keys with the same value are placed in the sorted array in the same order that they appear in the input array). Its main drawback, that can be understood from the complexity Eq. 3.7, is that the 4-pass sorting technique is efficient when the range of the finite set is small and when there are many duplicate keys in the sets as illustrated in [28, 29]. Nevertheless, this sorting algorithm is perfectly acceptable for the real-time processing demands of this noise estimation implementation as demonstrated in the timing results that are reported in section 5.3.

3.2.3 Logarithmic Arithmetic Complexity

A logarithm is an exponent and it is this exponent (represented by n) to which a base number (represented by b) must be raised to produce a given number (represented by x) as seen in Eq. 3.8. The logarithm function, which calculates the n term, is depicted in Eq. 3.9. The logarithmic arithmetics applied in this noise estimation algorithm are better known as common logarithms with base 10.

$$x = b^n \tag{3.8}$$

$$f(x) = \log_b(x) = n \tag{3.9}$$

Common logarithms can be converted into a natural logarithms as proven in Eq. 3.10. The natural logarithmic functions can be expressed as the Mercator series as illustrated in Eq. 3.11, which is an expansion of the Taylor series.

$$\begin{aligned} \log_{10}(x) &= \frac{\log_e(x)}{\log_e(10)} \\ &= \frac{\ln(x)}{\ln(10)} \\ &= \frac{\ln(x)}{2.30258} \\ &= 0.4343 \cdot \ln(x) \end{aligned} \tag{3.10}$$

$$\begin{aligned} \ln(1+x) &= \sum_{k=1}^{\infty} \frac{(-1)^{k+1}}{k} x^k \\ &= x - \frac{1}{2}x^2 + \frac{1}{3}x^3 - \cdots \end{aligned} \tag{3.11}$$

The previous Eq. 3.11 demonstrates the exponential order of complexity comprised in the logarithmic function. This exponential property requires excessive amount of logic to implement onto a resource constraint hardware device. Therefore, the logarithmic function was implemented using a look-up table to realize this part of the algorithm. This technique consequently reduces the accuracy because the results of the logarithmic arithmetics are abbreviated. In this case, the accuracy level is directly proportional to the size of the look-up table or, in other words, the size of the memory being used. Since the logarithmic function generates fractional numbers and that this FPGA design does not support floating-point representation, the look-up table contains fixed-point logarithmic values. As mentioned, at the beginning of this chapter, fixed-point representation precision is proportional to the number extra bits of precision added to the original value. For the proposed design, a concession was approved to use a 16 bit internal memory (inside the FPGA) look-up table to calculate the logarithmic values. The simulation spawned acceptable results with respect to the software results as seen in section 5.1.

Taking into account the necessary algorithm modifications, the next sub-section discusses the architecture design strategies and the justifications supporting these design goals.

3.2.4 Design Strategies and Justifications

In this sub-section, the conception of the proposed hardware architecture is reviewed, the architecture design strategies are explained and the reason why they have been chosen are justified. The main design objectives followed in the proposed architecture are to build a real-time performance noise estimation hardware design, to minimize logic consumption

and to support a scalable architecture. Many design strategies are being employed to reach these ultimate goals and they are described in the following sub-sections.

Reaching Real-time Results

The first objective of this thesis is to render a real-time performance of the software-based noise estimation algorithm. The idea is to design an effective architecture capable of speeding-up the noise estimation processing time. In order to reach this design goal, some well known hardware design techniques such as data path pipelining and parallel processing design must be adopted and the acceleration of local clock must be integrated into the architecture.

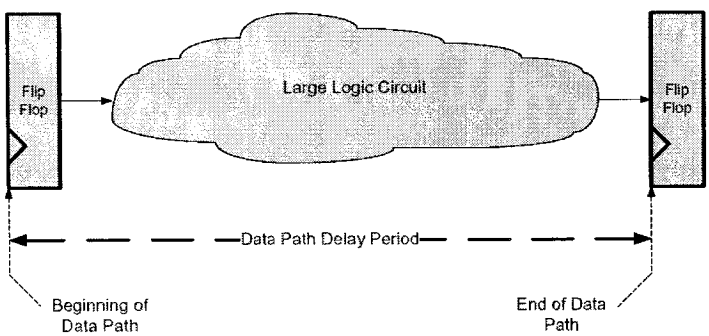
The first hardware design technique adopted is the data path pipelining method. As cited in [30, p. 592], a pipeline is a series of stages, where some part of the processing is carried out at each stage. The processing work is not completed until it has passed through all stages. The number of stages that need to be traversed can be interpreted as the latency of the system being pipelined. A popular pipelining depiction of a 3 stage pipeline example is explained in the following analogy:

> *"In everyday life, people do many tasks in stages. For instance, when we do the laundry, we place a load in the washing machine. When it is done, it is transferred to the dryer and another load is placed in the washing machine. When the first load is dry, we pull it out for folding or ironing, moving the second load to the dryer and start a third load in the washing machine. We proceed with folding or ironing of the first load while the second and third loads*

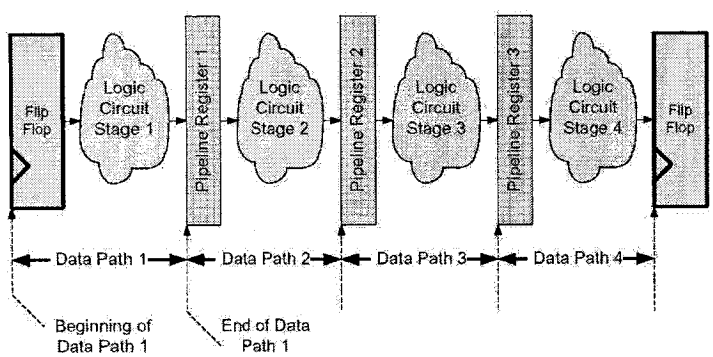
are being dried and washed, respectively. We may have never thought of it this way but we do laundry by pipeline processing." [31]

Pipelining performed on a computationally complex equation, which is interpreted in hardware as a large logic circuit, consists in segmenting the chunk of logic, seen in part a) of Fig. 3.12, into parts (e.g., stages) as seen in part b) of Fig. 3.12. The advantage in segmenting a data path is to reduce its critical path into ensuing smaller circuits containing shortened path delays. Contracted path delays reduces the completion time (e.g., period) of each circuit segments. Consequently, this results in an increase in the operating frequency since its period is inversely proportional ($p = \frac{1}{f}$). An increase in frequency furnishes an increase in processing speed which in turn creates an increase in latency. The latency, as mentioned before, is homologous to the number of stages in the segmented circuit. Furthermore, the execution of the circuit functions (i.e., the tasks) in each stages can be overlapped. Whereas each stage of the pipeline can execute subsequent tasks at the same time as explained in the washing machine analogy example. The overlapping of the laundering tasks is explained by the concurrent operation of the washing machine and drying machine on two separate loads.

The pipelining technique is integrated throughout the entire design of the noise estimation implementation. The addition, subtraction and division calculations including the scalable array divider are thoroughly pipelined. Moreover, the data path and control structure is also fully pipelined between internal modules by input and output registers. This important practice will avoid numerous timing problems due to enormous data paths that can arise at the system level when the modules are put together to construct the top level



(a) Before Pipelining



(b) After Pipelining

Figure 3.12: Pipelining Logic Circuit

entity. The module's interconnected data paths which are not separated by pipeline stages can create exceedingly long data paths and increase the overall period.

The second hardware design technique consists in maximizing the parallelism of the architecture. This method attempts to eliminate the sequential processing software structures (e.g., loops) and works to increase throughput by parallelizing arithmetic operations; thus completing operations concurrently. Parallelism in hardware is performed through replication of hardware blocks. The notion of design parallelism is used at the system level (high level) and at the arithmetic level (low level) in this noise estimation algorithm implementation.

At the system level, the data is acquired and processed in parallel using a "ping-pong" concept. The term "ping-pong" refers to the back and forth processing scheme that is used to enable rapid parallel processing. In this thesis, the "ping-pong" scheme is used to allow simultaneous but separate access to two external memory components to store and read data concurrently. This concept is illustrated in Fig. 3.13 as the variance ($\sigma^2_{B_h}$) and homogeneity (ξ_{B_h}) measures generated by the acquisition module during the current frame are being stored in memory bank "A", the same data from the previous frame is being retrieved by the sorting module from memory bank "B" concurrently. Chronologically, once the entire current frame has been stored and that the data of the previous frame has been read and processed then the external memory component exchange roles; this role reversal is seen in Fig. 3.14. The reason for using this "ping-pong" architecture is due to the time consuming sequential sorting procedure. As mentioned in the sorting complexity section, the sorting of the homogeneity measures can not be execute during the acquisition of the frame. Rather, the counting sorting algorithm is performed after the entire frame has been acquired.

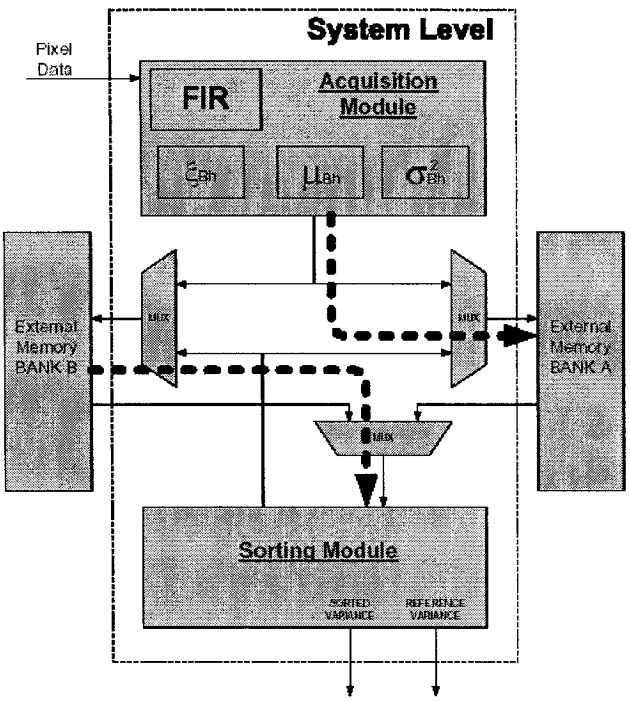


Figure 3.13: "Ping-pong" scheme : data stored in bank "A".

At the lower level, the architectural parallelism is integrated in the arithmetic processing units. In the sample mean equation seen in Eq. 2.5, the software program loops through the scanning window's pixel data to accumulate the luminance sum of the segmented image block. The corresponding hardware implementation incorporates a parallel configuration by adding the luminance values concurrently by duplicating the adders and creating parallel data paths to achieve the summation procedure. This parallel structure eliminates the sequential accumulation process employed by the software looping procedure and generates mean values faster. The same method is utilized to optimize the squaring operation in the generation of the variance values as seen in Eq. 2.6. The squaring operation is performed for each pixel in the scanning window on the resulting value of the subtraction calculation between the sample mean and each pixel's luminance data. The subtraction and squaring

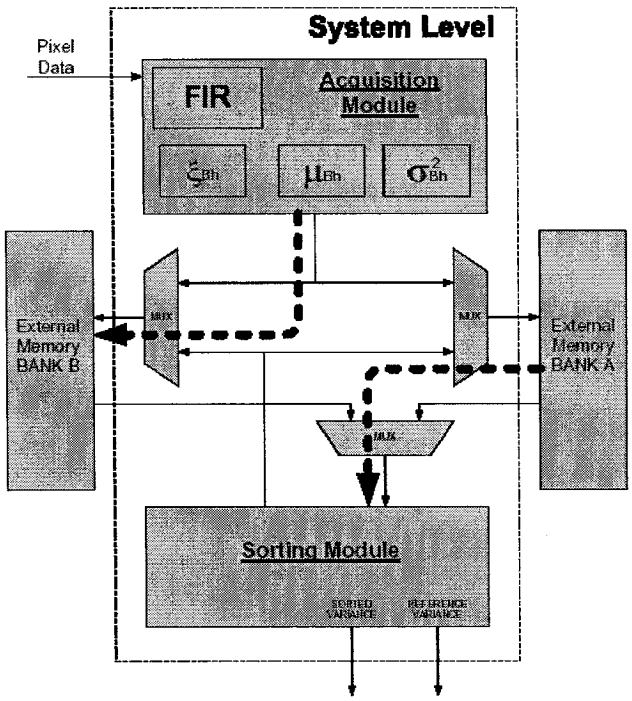


Figure 3.14: "Ping-pong" scheme reversed : data stored in bank "B".

operations are implemented using a parallel structure of 25 subtracter and multiplier blocks concurrently as seen in Fig. 3.15 for a 5x5 window size.

To further accelerate the processing time to reach real-time performances, local clock frequencies can be increased within the FPGA platform. The Xilinx Virtex II FPGA chips are equipped with specific Digital Clock Manager (DCM) components capable of de-skewing internal clock signals (using a clock delay locked loop (DLL) sub-components to eliminate clock distribution delays), performing frequency synthesis (clock multiplication and division manipulations) and achieving clock phase shifting. The primary goal in increasing the internal clock speed is to accelerate the sequential sorting mechanism. The sorting section of the noise estimation algorithm is directly affected by the amount of blocks contained in a frame. Smaller window filter sizes generate more segments per frame; thus more block data

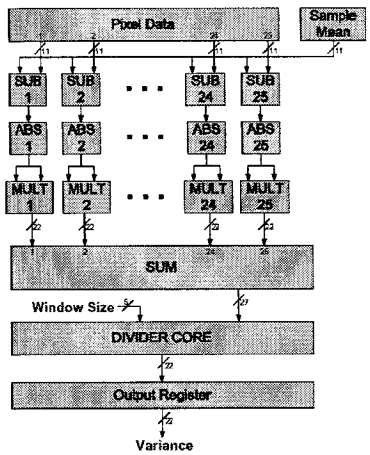


Figure 3.15: Variance generation using parallel multiplier blocks.

(e.g., homogeneous measure values) must be sorted within a fixed time frame (e.g., PAL 50 Hz standard or 0.02 seconds). In this thesis, the DCM's frequency synthesis feature is used to multiply the internal clock frequency by three. Studies have shown that even for small filter sizes (e.g., 3x3), an accelerated internal clock frequency of 40.5 MHz (initially 13.5 MHz multiplied by 3) is sufficient to meet real-time requirements. Further justification through experimental results are explained in section 5.3.

Minimizing Area

Increasing the pipeline stage by including additional register pipe stages and maximizing the parallelism of the operations by duplicating hardware blocks comes at the expense of increasing the FPGA resource consumption. The second objective of this thesis is to minimize the FPGA logic consumption. This is achieved by an efficient use of the internal Xilinx FPGA optimized components (primitives) and external memory devices.

In the variance generation, large multiplier blocks (e.g., 22 bit multiplication as seen

in Fig. 3.15 are required to perform the squaring operation. Studies have shown that the incorporation of specific Xilinx Virtex II *MULT18x18* primitives (18x18-bit 2's complement signed multipliers) to perform the multiplication have rendered much more condensed implementations than the "*" VHDL operator or the Coregen multiplication cores. The Table 3.3 illustrates the resource consumption of these various multiplication instances implemented on a Xilinx XC2V2000-FF896 (Virtex II Family) FPGA chip. These tabulated FPGA resource usage statistics show the advantage in using multiplication primitives in that their implementation use half of the amount of the CLB's look-up tables (LUT) and provide the least timing delays (i.e., fastest processing time).

Multiplication Methods	Slice	Flip Flop	LUT	Max Delay (ns)	Latency (clk cycles)
VHDL operator	32%	9%	22%	24	37
Coregen Cores	34%	30%	22%	19	41
Xilinx *MULT18x18* Primitives	14%	8%	10%	13	37

Table 3.3: Multiplication Resource Consumption

Similarly, specific embedded 16-bit shift registers (Xilinx Virtex II *SRL16*) primitives are included in the proposed design to implement data delay modules. The FPGA's CLBs can be configured as shift registers to store and shift data synchronously with the clock pulse. The advantage in using these primitives is to enable data storage and data shifting without using CLB internal flip-flops or global routing matrix to construct shift registers elements. Furthermore, 16K byte dual-port RAM components can be cascaded to implement large embedded storage blocks or large delay elements. These Xilinx Virtex II (*BRAM*) blocks can be programmed to various sizes for internal FPGA storage purposes. The RAM blocks are utilized throughout the design; for the counting sorting implementation (e.g., sorting

array), the finite impulse response (FIR) filter (e.g., line delays) and the logarithmic look-up table. Hence, a significant amount of flip-flops and CLBs are spared by using embedded Xilinx FPGA features such as primitives.

The most important strategy used to minimize the logic consumption is to store the maximum amount of data off-chip. For example, external memory is much larger than internal RAM block memory. Nevertheless, the read or write access time to external memory is usually much slower. Designers may opt for external memory devices when excessive quantities of data need to be stored but do not need to be accessed immediately. In this thesis, external memory was used to store the long lists of block variances ($\sigma^2_{B_h}$) and block homogeneous measures (ξ_{B_h}). For a 5x5 and 3x3 window filter size, the Eq. 3.12 calculates the number of internal RAM blocks ($BRAMs$) required to implement the adequate amount of memory space within a Xilinx Virtex II FPGA to store the long lists of data. In this Equation, the $MemoryDevices$ stands for the number of parallel memory devices (e.g., Bank A and B) necessary to realize the ping-pong structure, the $Blocks$ stands for the number of blocks (segments within a frame) and the $DataWidth$ stands for the data width (consisting of the block data $\sigma^2_{B_h}$ and ξ_{B_h} bit width calculated and analyzed in section 4.2). These terms are multiplied together and then divided by $BRAMSize$ which stands for the size of RAM blocks (standard 16 KBytes for Xilinx Virtex II FPGA).

$$\begin{aligned} BRAMs^{NxN} &= \frac{MemoryDevices*Blocks*DataWidth}{BRAMSize} \\ BRAMs^{5x5} &= \frac{2*8208*(2*22)}{16*1024} \approx 45BRAM \\ BRAMs^{3x3} &= \frac{2*23040*(2*22)}{16*1024} \approx 124BRAM \end{aligned} \tag{3.12}$$

Considering the Xilinx Virtex II FPGA family resource distribution Table 3.4, the number of RAM blocks (*BRAM*) available in the various FPGA family members (devices) are in limited quantities. Using approximately 124 RAM blocks (calculated in Eq. 3.12 for a 3x3 filter window size) to store the variances and homogeneous measures is considered excessive and unreasonable since only the high-end Virtex II FPGA chips could support that range of storage capacity. External memory is therefore employed to store the long lists of block data as opposed to internal RAM block memory. Thus, the design can be implemented on a smaller FPGA chip, in turn, reducing the cost of the IC.

Devices	System Gates	Slices	Multiplier Blocks	RAM Blocks (16KByes)	DCM	I/O pins
XC2V40	40K	256	4	4	4	88
XC2V80	80K	512	8	8	4	120
XC2V500	500K	3,072	32	32	8	264
XC2V1000	1M	5,120	40	40	8	432
XC2V2000	2M	10,752	56	56	8	624
XC2V3000	3M	14,336	96	96	12	720
XC2V4000	4M	23,040	120	120	12	912
XC2V6000	6M	33,792	144	144	12	1,104
XC2V8000	8M	46,592	168	168	12	1,108

Table 3.4: Xilinx Virtex II Member Resource Distribution

Design Scalability

Once the primary objective of realizing real-time performance has been confirmed feasible and the secondary objective of minimizing the area of the design has been analyzed, the scalability of the architecture can be examined. Hence, the third objective is to build a flexible architecture capable of supporting various parameters such as window filter sizes. VHDL generics, global packages and special purpose scalable array divider permit the pa-

rameterization of the design.

VHDL generics are used to construct parameterized hardware components by passing information into a module. The generics may specify a module's time delay value for signal assignment statements, the depth of a RAM component or the width of a data bus. The generic structure is incorporated into the VHDL code before the entity instantiation since the generic variables can affect the input and output signals as well as the internal signals bus widths. Generics are employed in this thesis to declare the size the data and address bus in order for the entire design to be aware of the signal sizes being used to interface with the external memory. This technique was used so that the design could be independent of external memory type or size. Knowing that memory control structures (e.g., chip enable, read and write control signals, etc) are usually standard, only the interface signal sizes would have to be modified. This would allow the connections to be scalable and adaptable to any other external memory devices.

A VHDL global packages contain subprograms, constant definitions and type definitions to be used throughout the design. These package definition are included in a separate file. The constants declared in packages are visible in all levels of the design's hierarchy. The modules infer a particular package with the "use" VHDL statement which is incorporated before the module's entity declaration. This instantiation method is similar to utilizing the "include" statement in "C" software that makes use of pre-compiled libraries. A global package was created for this thesis to hold the global constants that model the different window filter size configurations. The architecture has been built to support two window filter sizes (e.g., 3x3 and 5x5) and has been designed to be easily modifiable to add other filter sizes or other future parameters (e.g., data size, image resolution, etc). The configurations

can be changed by a simple code modification in the VHDL global package file and a recompilation of the design's packages and modules. The modules do not need to be updated or modified for the behavior conversion to take place.

For example, the global package in this thesis contain constants that are employed in the pixel counting mechanism (in the input data acquisition module) to define the pixel count limits. The control structure embedded in the module's architecture uses predefined constants to know when the total count of input pixels have reached the end of a valid window filter boundary, end of line or end of frame. Hence, the window filter can take on different sizes which will affect the number of blocks (segments) that appear in a row, the number of rows that appear in a field and the total number of blocks in a field as is described in Eq. 3.13 for a 5x5 filter size. The same calculations performed for a 3x3 filter size are seen in Eq. 3.14.

$$\begin{aligned}
\text{Blocks per row} &= \frac{720\frac{pixel}{row}}{5\ block\ length} \\
&= 144 \text{ blocks/row} \\
\text{Rows per field} &= \frac{288\frac{rows}{field}}{5\ block\ hight} \\
&= 57 \text{ rows/field} \\
\text{Number of blocks per field} &= 144\frac{blocks}{row} * 57\frac{rows}{field} \\
&= 8208 \text{ blocks/field}
\end{aligned} \tag{3.13}$$

$$\begin{aligned}
\text{Blocks per row} &= \frac{720\frac{pixel}{row}}{3\ block\ length} \\
&= 240 \text{ blocks/row}
\end{aligned}$$

$$\begin{aligned} \text{Rows per field} &= \frac{288\frac{rows}{field}}{3\ block\ hight} \\ &= 96 \text{ rows/field} \\ \text{Number of blocks per field} &= 240\frac{blocks}{row} * 96\frac{rows}{field} \\ &= 23040 \text{ blocks/field} \end{aligned} \tag{3.14}$$

The global package stores the delimiting variables (e.g., number of blocks per row, number of rows per field, length/width of the filter, number of block per field, etc) that the pixel counting control mechanism uses to determine the valid blocks within the incoming sea of pixels that compose an image field. The pixels in the valid blocks are necessary to generate the variance and homogeneous measure values. The architecture is deemed scalable in the fact that its structure supports the delimiting variables for both 3x3 and 5x5 filter configurations. The distribution of the blocks calculated in the previous equation for the two different window size is illustrated in Fig. 3.16. The field pixels are depicted as the small squares, the 3x3 filter window is designated by the medium-sized dark square and the 5x5 filter window is indicated by the largest hashed square. The block count is denoted in the parenthesis at the end of the first and last row in the field.

The package is also used to define the sorting algorithms delimiters. Thus, for different filter sizes, a constant number of elements must be read from memory, ordered and written back into memory depending on the total number of blocks that need to be processed. The architecture for the sorting module must also be adjusted to support various lengths of data lists.

A general purpose scalable pipelined array divider was developed to help construct a

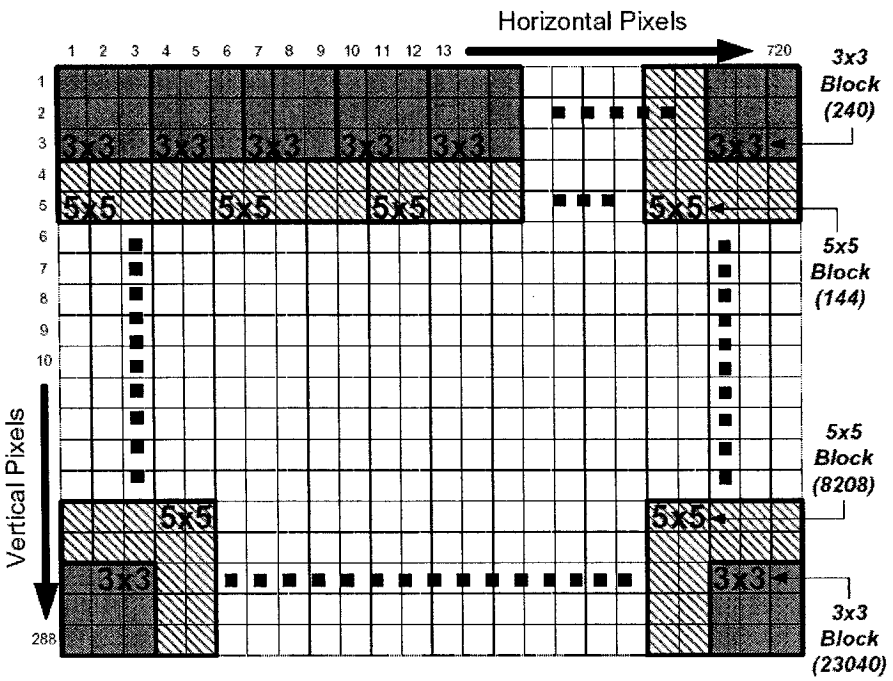


Figure 3.16: Filter window distribution for 3x3 and 5x5 blocks.

highly scalable architecture. The main reason for delving into such a complex divider design was that the Xilinx *Coregen*^{TM} tool was not capable of generating divider cores exceeding 32 bit bus widths. This situation had caused some major architectural design problems since an increase in the number of blocks (segments per field), when using smaller filter sizes (e.g., 3x3), has had an increasing affect on the size of the arithmetical units. In worst case scenarios with 3x3 filtering windows, if the variance for each block was to pass the threshold condition (mentioned in Eq. 2.8), the averaging accumulator unit would have to be capable of dividing a 37 bit value. This accumulation bus width is described in Eq. 3.15 where the $MaxVarBW$ stands for the maximum variance bus width and the NB stands for the total possible number of blocks that could be accumulated. From this equation, it is clear that a new divider unit had to be developed to replace the divider cores that

were not capable of performing 37 bit divisions. The creation of a scalable divider helps the architecture support large number arithmetic calculations for a particular 3x3 filter size configuration and for future configurations. Future configuration include supporting multiple filter size which would mean processing more blocks (as a consequence of smaller filter sizes) and/or processing less blocks which would mean processing more pixels per blocks (as a consequence of larger filter sizes). Other configuration would include increasing the fixed-point precision which would increase the data bus widths. The scalable divider becomes essential in constructing a flexible architecture and in materializing these various configuration (details of the scalable array divider design is shown in section 4.2).

$$\begin{aligned} \text{Accumulator Bus Width} &= \frac{\log\left(2^{MaxVarBW} * NB\right)}{\log(2)} \\ &= \frac{\log\left(2^{22} * 23040\right)}{\log(2)} \\ &\approx 37 \text{ bits} \end{aligned} \tag{3.15}$$

The objectives of designing a hardware implementation of a noise estimation algorithm for real-time processing with a scalable architecture capable of being easily modified for future configurations as led to further discussions. The extent of this architecture's flexibility can be limited by certain architectural complications which are discussed in the following sub-section.

3.2.5 Architecture Discussion

The architecture was conceived with the idea that it could be easily modified to incorporate new features. The extent of the architecture's flexibility with respect to expanding or shrinking filter sizes, using disparately shaped filters (as opposed to square shape filters), processing various image resolutions sizes, processing partial sections of an images and calculating the logarithmic arithmetics with the use of off-chip memory is discussed. The consequence of the possible addition of these future parameters is also explained.

Studies have shown that the current architecture could easily be modified to support filter size expansion. The analysis of an architecture using a 7x7 filter size has been performed to verify these allegations. The architecture was remodeled from a 5x5 filter structure where every module's data bus width and internal hardware organization was re-arranged to determine the hardware feasibility. The main consequence of filter size expansion is an increase in arithmetic units proportions since additional pixels are processed per scanning window (e.g., from 25 for 5x5 filter to 49 for a 7x7 filter). Thus, the effect of this architectural adaptation would require larger accumulators, larger dividers, extra pipeline stages and additional multiplier blocks. The overall ramifications of filter expansion is to increase the resource consumption. However, since the number of blocks per image is decreased, the memory utilization is also decreased since there are less blocks of data to be stored.

The opposite effect can be seen when shrinking the filter size. This type of implementation modification has been studied and integrated into the current architecture. Originally, the design was built to perform noise estimation with a 5x5 filter size. An alternate smaller filter size (e.g., 3x3) was incorporated and tested. The direct effect is an increase in the

number of blocks to process per images. The sorting mechanism must compensate by ordering a larger list of block data faster by increasing the local clock frequency. The size of the memory devices must also be increased to store the large amounts of block data. Disparately shaped filter sizes (e.g., 3x1, 5x1, 7x1, etc) will incur the same effects as square filter shrinking. Since the number of pixels per blocks are reduced, the overall number of blocks per image are increased. In this case the architecture can easily be adapted to accept filter shrinkage or un-even filter sizes.

The complexity of incorporating scalable image resolution features or being able to support other video standard resolutions (e.g., NTSC) are somewhat intricate. The current architecture acquires the pixels-based image which is made of a static field resolution (i.e., in our case the PAL standard resolution is used). Although this feature implementation is possible, the architecture would require some important modifications. The acquisition module would necessitate variable line delays components since the number of pixels per lines would change for different resolution settings. A very high resolution (e.g., HDTV) would increase the number of pixels per field which would effect the architecture in the same manner as filter shrinkage feature. A low resolution would decrease the number of pixels per field and would simply reduce the size of the overall design. A change in video standard would reduce or increase the number of lines per frame. High resolutions can limit the FPGA feasibility in that they can consume excessive amount of resources (e.g., extremely large memory and arithmetical units required) to implement the noise estimation algorithm.

Processing partial sections of the field is not feasible with this architecture. Studies have shown a lack of accuracy in employing single row processing or multiple row processing to

estimate the noise variance. The algorithm would have to be slightly modified to sort the block data in a row fashion. Nevertheless, if this technique were to be adopted, the current architecture would need to be changed considerably to calculate and sort homogeneous measures at the end each row or multiple rows instead of at the end of a field. New memory devices would have to store the resulting row ordered variances that would be averaged out at the end of the field's acquisition procedure. Alternatively, if the noise estimation was only performed on the first few rows of an image using the same algorithm, this would result in using a limited version of the current architecture. This technique would require to "shut off" the hardware processing during the acquisition of the remainder of the image's unwanted rows. The architecture would still contain the same amount of logic since the processing sequence would not change, it would only stop earlier. Consequently, the size of the memory could be reduced for this type of feature. Furthermore, applying the noise estimation hardware to process only the n^{th} field (e.g., every 5 fields) could be implemented easily. This type of feature would have the same effect as processing only the first few rows of an image, the hardware design would simply disregard the estimation of the noise variance during non-valid fields. Hence, the architecture could be adapted to endorse this feature.

The logarithmic arithmetic of the algorithm is implemented internally in this current architecture. The logarithmic arithmetics, as mentioned at the beginning of this chapter, are evaluated using look-up tables (LUT) stored in an internal memory component. The present architecture retains the LUT in internal RAM blocks (*BRAM*). Alternatively external memory could have been chosen to store the LUT since RAM memory is limited in an FPGA as can be seen in Table 3.4. Moreover, an important downgrade in precision

(logarithmic value precision) is a consequence in using a constrained memory component. Although half of the RAM elements of the Xilinx XC2V4000 FPGA chip are used to implement the current logarithmic LUT (as described in detail in section 4.2), a substantially larger memory space could have been used to store additional bits of data; making the logarithmic calculations more precise. Upon studying the simulation results from this internal LUT implementation, the accuracy was deemed acceptable but not perfect. The following positive trend was observed in the PSNR results for the noise estimation of video sequences with low noise levels (e.g., 20 db) and relatively large filter windows (e.g., 5x5) in that they produced very accurate results as can be seen in section 5.1. A negative trend was noticed when using a smaller filter window sizes (necessitating to process more blocks per field) which resulted in less accurate results. The conclusions drawn from these simulation results is that when many blocks need to be processed, there is more of a likelihood that many block variances might not be considered when being evaluated by the threshold condition in Eq. 2.8. Knowing that the logarithmic LUT fixed-point values are not as precise as the software floating-point values, some logarithmic values of variances which are very close to the threshold limit are omitted, thus not contributing to the calculations of the final noise variance thereby reducing its accuracy. Another effect is noticed from these simulations by the fact that high noise levels (PSNR $>$ 40 db) translate into small variance values and that a considerable asymmetry (large difference) can be distinguished between logarithmic results of subsequent small numbers. Consequently, evaluating the threshold condition of images that contain high noise levels using a less precise logarithmic look-up table also leads to un-accurate results. In this case, the threshold condition is being evaluated with logarithmic values which are far apart and which also have more of a likelihood of being

outside of the acceptable threshold range and being omitted. Considerable attention had to be invested into the selection of a logarithmic table size and logarithmic data precision capable of generating acceptable results without having its implementation consume excessive quantities of FPGA internal memory space.

The decision to keep the logarithmic LUT internally stored was made for several practical reasons. The goal of this research is to develop an academic study and implementation of a noise estimation algorithm into a hardware design. The scope of this research includes the implementation and simulation of the algorithm's functionality as opposed to developing an industry "ready-to-use" board application. A fair amount of work would need to be invested in building a prototype PCB board, interconnecting memory ICs with the FPGA and developing an entire real-life system. This system would contain a physical interface between external memory and the FPGA's direct memory access module (DMA) that would communicate using a certified communication protocol (e.g., *PCI*, I^2C, *RS232* or serial protocols). This thesis's interface design integrates a simple communication protocol with a straightforward external memory simulation model. In other words, the complex PCB board system is mimicked in a virtual simulation environment. Seeing that external memory is similar to a PC's RAM memory which is volatile, meaning that they lose their contents when the power is turned off or when a simulation session has ended, this virtual external memory is used for simple read and write procedures (e.g., storing the block's variance and homogeneous measure lists) that do not require an initialization or refreshing sequence. Once a simulation is finished, the virtual external memory is not required to keep information stored statically. Utilizing external memory to implement the logarithmic LUT would require setting up an initialization sequence and an alternate hardware system capa-

ble of loading (refreshing) the external memory with the logarithmic values at the beginning of each simulation runs. This method would have been impractical since a mandatory initialization sequence would have lengthened the simulation time. Furthermore, extra logic would have been incorporated into the design to control the sequence of the initialization phase. It was considered more reasonable and practical to use the FPGA's internal RAM blocks because they can be initialized, only once, with a text file during its creation with the Xilinx *Coregen*^{TM} tool. Designers who wish to further develop this architecture would have to extract the current oversimplified interface and integrate a more common interface that would meet internationally accepted communication protocols standards. However, the internal processing core of the noise estimation algorithm could still be employed even if a common communication protocol is installed. Furthermore, this current architecture would still produce real-time results given that the external memory interface read and write access time are kept short.

3.3 Summary

In this chapter, the design flow was detailed through a series of design steps. The important design step was the analysis of the proposed noise estimation algorithm which helped determine the required modifications that need to be incorporated to make the design hardware compliant. These modifications to the original software algorithm consisted in replacing the software-based floating-point precision to a hardware feasible fixed-point representation, using a look up table to perform logarithmic arithmetics instead of building an exponentially enormous logarithmic series equivalent circuit, and implementing an

alternate ordering procedure to replace a software-driven looping-based sorting mechanism with a hardware-efficient counting sort technique.

An optimum architecture was attained with specific design strategies (e.g., design pipelining, parallel processing, acceleration of local clocks, use of external memory, VHDL generics, use of FPGA special features, etc.) which was needed to reach the main design objectives of this thesis. Thus, these objectives consisting of building a real-time performance noise estimation hardware design, to minimizing logic consumption, and supporting a scalable architecture were therefore proven to be feasible. The subsequent experimental result chapter will validate the performance of this proposed architecture. Furthermore, this chapter also discussed various architecture feasibility issues such as the expansion and shrinkage of filter sizes, the support of other parameters like variable image resolution, the implementation of a variant of the noise estimation algorithm, and the possible implementation of an external memory look-up table for the logarithmic function.

By using an internal BRAM memory to store the logarithmic look-up table, the memory access time to fetch the table's data is greatly reduced. However, this advantage is diminished by using many instances of internal BRAM memories and by the limited accuracy that each memory location can provide due to its restraint size. It is shown that the loss in quality seen in logarithmic calculations happens mostly with images containing high levels of noise and variance estimation using small filter sizes. Although, some results are deteriorated as compared to the software results, the experimental results chapter shows acceptable variance generations. In the following chapter, the noise estimation implementation is discussed and the details of its internal organization and circuit functionality are described.

Chapter 4

FPGA Implementation

In this chapter a general overview of the noise estimation implementation is presented. The organization of the top level module data flow is described. The sub-modules that compose this top level system and their inter-relations are enumerated. The sub-module's individual purposes are explained and their internal circuitry are depicted through a series of figures and diagrams. The size of the structure in term of wire width are demonstrated and justified for both 3x3 and 5x5 filter configurations.

4.1 Top Level Organization

This section discusses in detail the design's organization from a high level of abstraction. The general organization of the FPGA top level view of the noise estimation implementation can be seen in Fig. 4.1. The long stripe line surrounding the module A, B and C represent the FPGA design's boundary; external components to the FPGA chip reside on the outside

of this delimitation. The vertical narrow stripe line divides the design into two separate clock domains. The top clock domain composed of module A operates at 13.5 MHz. The bottom clock domain, consists of module B, module C, external memory banks and logarithmic look-up table, is 3 times the speed of module A and operates at 40.5 MHz. The accelerated clock signal (i.e., 40.5 MHz) is generated by a Xilinx Virtex II special-purpose DCM primitive component.

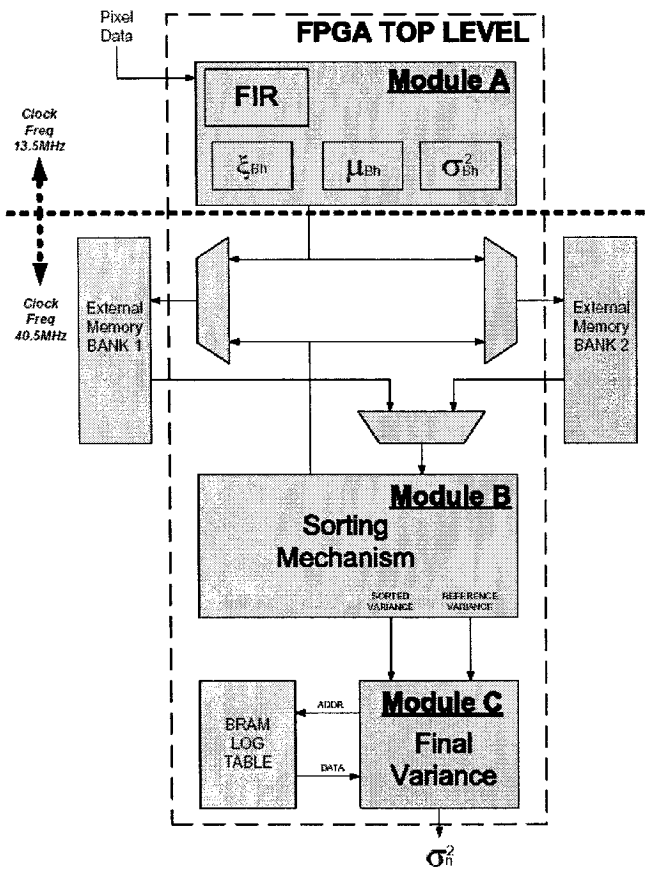


Figure 4.1: Top level view of FPGA implementation.

The input pixel data is acquired by the finite response filter (FIR) sub-module in module A. The FIR's main purpose is to sample the proper amount of input pixels, depending

on the filter sizes, to form valid data blocks that segment the input image. The data blocks contain the valid pixels that are forwarded to other sub modules to generate the homogeneous measure (ξ_{B_h}), the sample mean (μ_{Bh}) and the variance ($\sigma^2_{B_h}$). These values are produced for each block that segment the image field and are stored into the external memory banks in an alternative fashion (known as in the "ping-pong" scheme mentioned in section 3.2.4) by the memory controllers. Once the entire field has been processed into a memory bank, the other bank can be used to process and store the subsequent field data.

While the pixel data are being acquired from a current field by module A and the previous field data (e.g., the homogeneous measure (ξ_{B_h}) and the variance ($\sigma^2_{B_h}$)) are being fetched concurrently by module B. The purpose of module B is to sort the list of homogeneous measure values and to return the top 10% of the corresponding variance values of the ordered block list. The reference variance is also generated from the top 3 variances obtained from the ordered block list.

The main duty in module C is to generate the final noise variance value. Module C will evaluate the logarithmic value of each block variances. The difference between the logarithmic value of the reference variance and the logarithmic value of the individual block variances are computed. This difference is then compared to a threshold which will determine the field's most homogeneous blocks. The variance values that pass this threshold condition are labeled valid block variances. The non-valid block variances are rejected from the noise estimation process. In the end, the final variance (σ^2_n) is produced from the average sum of the valid variances.

In next section, the top level module is broken down into sub-modules where an intrinsic

description of their internal functionality is explored.

4.2 Sub-module's Internal Circuit Description

This section is aimed at helping the reader apprehend the design's internal structure at a lower level of abstraction. The internal depiction of the important sub-modules that compose the top level system are presented in this section. An in debt characterization of the digital clock manager (DCM), the scalable non-restoring pipelined array divider, the finite impulse response filter, the mask and homogeneity analyzer, the sample mean generator, the variance generator, the external memory and the direct memory access (DMA) control unit, the sequential sorting sub-module, the logarithmic look-up table, and the final variance generator are described.

4.2.1 Digital Clock Manager

The digital clock manager (DCM) is an internal Xilinx Virtex II FPGA primitive component and is instantiated in the top level of the design. The DCM is capable of executing various clock driven functions. In the proposed implementation, the DCM is used to perform frequency synthesis (i.e., clock multiplication and division manipulations). Its main purpose is to multiply the general input clock frequency by a factor of 3. The need to accelerate the clock frequency (justified in section 3.2.4) is to help speedup the sorting mechanism. The DCM component connection to the main system is illustrated in Fig. 4.2. The general clock tree feeds the first part of the system circuit in module A at a clock frequency of 13.5MHz. This general clock frequency has been chosen based on the standard ITU-R-BT-601 input

pixel rate. The second part of the system circuit is fed by the DCM component $ClkFX$ output at a clock frequency of 40.5MHz. The input buffer ($IBUFG$) and the internal general buffer ($BUFG$) are obligatory configuration components that drive the clock signal to and from the DCM primitive.

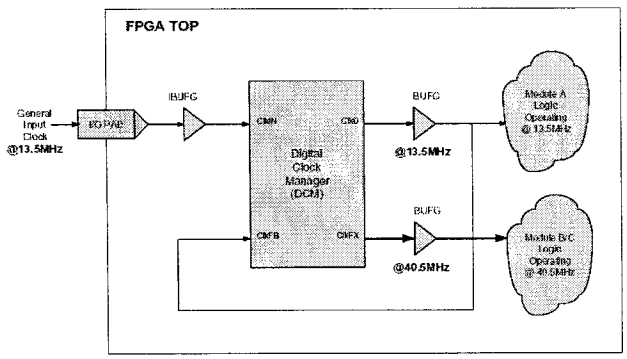


Figure 4.2: Digital clock manager circuit.

Research done in this thesis have shown that for a small filter size, a sorting clock frequency of 40.5MHz has resulted in sufficient processing speed. Thus, real-time performance was attained without the need to further accelerate the sorting clock frequency. Although this system's place and route results (seen in section 5.3) show a maximum possible operating frequency of approximately 50MHz (20ns), the decision was taken to proceed with a reasonable 40.5MHz clock frequency to avoid timing complications and possible timing violations.

4.2.2 Scalable Non-Restoring Pipelined Array Divider sub-module

The scalable non-restoring pipelined array divider has been developed for general purpose use throughout the noise estimation implementation. The need to make the architecture

flexible has led to scalability problems due to restricted divider core sizes as mentioned in section 3.2.4. This divider has been developed to expand beyond the Xilinx *Coregen*^{TM} tool limit of 32 bits and up to 128 bits.

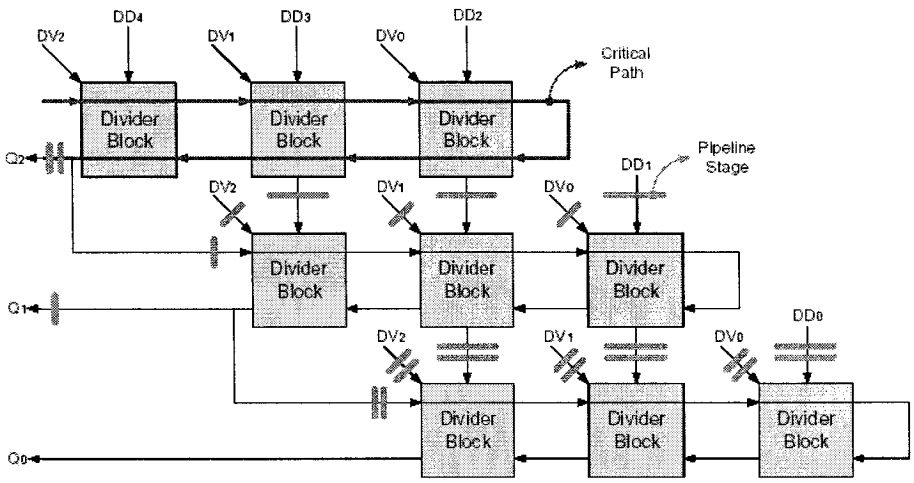
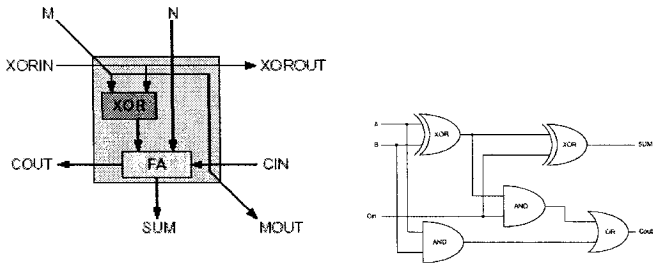


Figure 4.3: Non-restoring pipelined array divider.

The non-restoring array divider algorithm [32, 33] was chosen to implement the division unit because of its regular structure. This important architecture feature is essential to permit pipelining of the divider design. In conventional binary restoring division, a sequence of subtraction and shifts are performed to select the quotient. In each iteration the divisor is subtracted to form the partial remainder, until the difference becomes negative. Then the restoration technique is executed when the divisor is added back to the negative difference. In the case of the binary non-restoring division, the restoring procedure is eliminated to improve the division time and to reduce the amount of logic to construct the division circuit.

The array divider is composed of a regular arrangement of divider blocks. For instance, Fig. 4.3 illustrates the array structure of a 5 bit dividend ($DD_{[4\rightarrow 0]}$) being divided by a 3 bit divisor ($DV_{[2\rightarrow 0]}$); resulting in a 3 bit quotient ($Q_{[2\rightarrow 0]}$) output. The divider block contains



(a) Divider block circuit (b) Full adder circuit

Figure 4.4: Divider internal components

an "exclusive-OR" (XOR) gate and a full adder component (FA) as can be seen in part a) of Fig. 4.4. The full adder constitutes a 3 bit addition block that generates a sum bit and an output carry bit as its internal structure is seen in part b) of Fig. 4.4. An XOR operation is performed between the M and $XORIN$ input signal and controls the divisor input to the full adder. The result of this logical operation is then added to the CIN and N input signals. The full adder's sum (S), which holds the partial remainder, is transmitted to the next row in the array divider. The carry ($COUT$) produced by the full adder is transfered to the preceding divider block. The divisor value (M) is forwarded diagonally to the next row in the array and the $XORIN$ value is forwarded to the succeeding divider block in the same row of the array. The pipeline stages are inserted between rows to synchronize the input data with the generation of the quotient output.

The latency of the pipelined divider is calculated in the Eq. 4.1 and produces comparable timing results to the Xilinx $Coregen^{TM}$ tool. The critical path determines the complexity of this array-based division algorithm. Without pipeline stages, the critical path would traverses every divider block in the entire array. Thus, for a KxK array size, the complexity would be $\theta(K^2)$. Including the pipeline stages brakes up the critical path and renders an

improved complexity of $\theta(K)$. This pipeline structure reduces the complexity to the critical path of a row of K divider block delays instead of the entire array of divider block delays.

$$\text{Pipelined Divider Latency} = (\text{Dividend Width - Divisor Width}) + 7 \quad (4.1)$$

The architecture is made scalable with the help of a VHDL "generate" statement which combines concurrent statements with a looping capability. This programming feature is convenient for the construction of repetitive instantiation of regular structures (e.g., divider block). The width of the dividend and divisor signals control the looping mechanism which determines the size of the structure; making the design scalable.

4.2.3 Module A : Finite Impulse Response Filter

The finite impulse response filter (FIR) is located in module A. The objective of the FIR is to acquire the input 8 bit data pixels and to sample the valid pixels to form a block of data. For a 5x5 filter size, the block will contain 25 pixel data elements. The pixel data contains the intensity level and is stored in the array of registers seen in Fig. 4.5. The intensity is represented in an 8 bit signal capable of rendering 256 different grayscale levels.

The line delays are built with internal RAM blocks and are used to store the contents of each pixel found on a line of an image field. In the case of a 5x5 filter window, four delay elements are used to synchronize the influx of pixels with the center pixel that will eventually form a valid block of data of 25 pixels as seen in Fig. 4.6. Studies have determined that a single internal RAM block is needed to build the line delays which are capable of

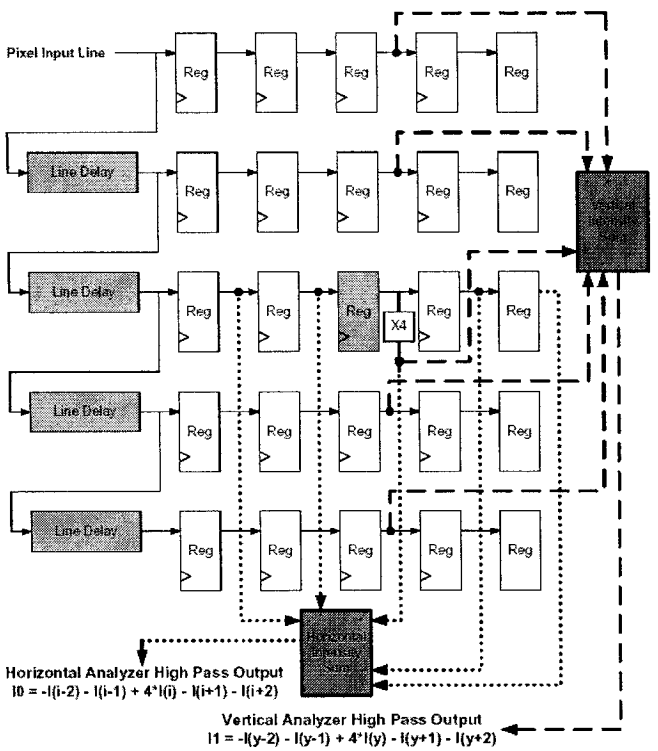


Figure 4.5: Finite impulse response filter for 5x5 window.

holding the 720 pixels contained in a line as described in Eq. 4.2.

$$\begin{aligned} \text{Number of BRAM blocks} &= (\text{ Pixel Width } * \text{ Pixel per Line }) / \text{ BRAM Size} \\ &= \frac{8 * 720}{16 * 1024} \approx 1 \text{ BRAMs} \end{aligned} \tag{4.2}$$

Alternatively, a 3x3 filter window only uses two line delays to form its valid block of data of 9 pixels. In this algorithm, the noise variance is calculated only on independent blocks separated by a filter length. Consequently, the variance is generated on subsequent blocks that do not overlap one another. The 25 pixel data included in the valid block are then forwarded to the mask sub-module to generate the homogeneous measure values. The FIR

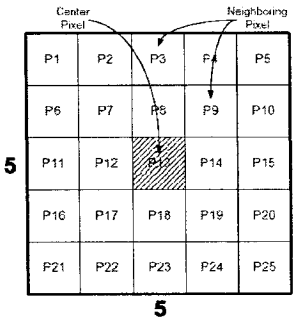


Figure 4.6: Valid block of data for a 5x5 filter.

structure with the horizontal and vertical intensity sum components are shown in Fig. 4.5. These sets of pixel data are also transfered to the sample mean sub-module and to the variance sub-module as seen in the internal connections of module A in Fig. 4.7.

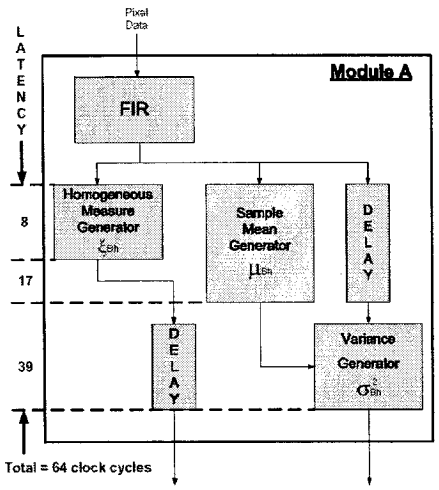


Figure 4.7: Module A sub-module circuit connection.

4.2.4 Module A : Masks and Homogeneity Analyzer

The homogeneity analyzer is made up of eight mask sub-module to calculate the homogeneous measure values (ξ_{Bh}). The eight distinctive masks, seen in Fig. 2.1, measure the high-frequency image components previously shown in Eq. 2.3. Using a 5x5 filter size, each

masks produce a worst case bus width (biggest unsigned values calculated) of 10 bits of unsigned data as can be seen in Eq. 4.3. The values inserted into the horizontal mask equation ($I_{o_{hor}}$) have been chosen to generate the largest output value (largest bus width). This large output value yields the mask data signal width (the value zero is used for the neighboring pixels and the value 255 for the center pixel of the mask).

$$\begin{aligned} I_{o_{hor}} &= \text{-}I(x-2,y) - I(x-1,y) + 4 * I(x,y) - I(x+1,y) - I(x+2,y) \\ &= -(0) - (0) + 4 * (255) - (0) - (0) \\ &= 1020 \approx 2^{10} \end{aligned} \tag{4.3}$$

The various mask (I_o) results are summed up as seen in Fig. 4.8 to generate the homogeneous value (ξ_{B_h}). This addition will generate a worst case bus width of 13 bits of unsigned data as can be seen in Eq. 4.4. The overall latency of this sub-module is 8 clock cycles.

$$\begin{aligned} \xi_{B_h} &= I_{o_1} + I_{o_2} + \cdots + I_{o_8} \\ &= 8 * 1020 \\ &= 8160 \approx 2^{13} \end{aligned} \tag{4.4}$$

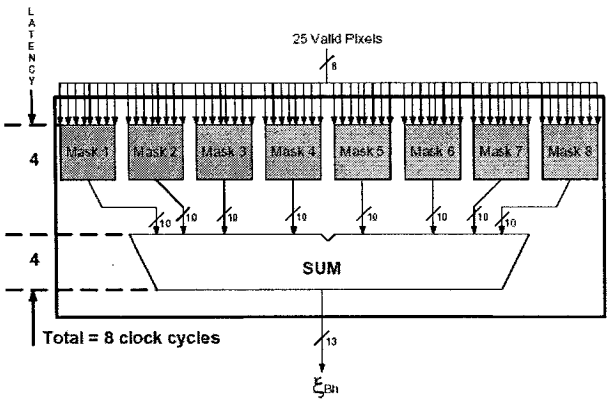


Figure 4.8: Homogeneous analyzer structure.

4.2.5 Module A : Sample Mean Generator

The sample mean (μ_{Bh}) is calculated with Eq. 4.5. The first step consists of incorporating the fixed-point representation into the pixel signal. This step will increase the width of the signal to include added precision and establish a new fixed-point by shifting the data by three bits to the left. The second step is to calculate the sum of the intensity of all the pixels contained in a block. The results of this addition generates a worst case bus width of 16 bits of unsigned data as can be seen in Eq. 4.6.

$$\mu_{Bh} = \frac{\sum_{(i,j)\in W_{ij}} I(i,j)}{W \times W}. \tag{4.5}$$

$$\begin{aligned} \text{Sum} &= I_{pixel_1} + I_{pixel_2} + \cdots + I_{pixel_{25}} \\ &= 25 * 2^{11} \\ &= 51200 \approx 2^{16} \end{aligned} \tag{4.6}$$

The third step is to calculate the average by dividing the 16 bit summation result by the filter area (W^2) of 25 to generate the mean square value. The result of this division generates a worst case bus width of 11 bits of unsigned data as can be seen in Eq. 4.7.

$$\begin{aligned} \text{Division} &= \frac{Sum}{W^2} \\ &= \frac{2^{16}}{25} \approx 2^{11} \end{aligned} \tag{4.7}$$

The overall latency of this sub-module, which includes all three steps, combine into a total of 25 clock cycles. The internal structure of the sample mean generator and its latency distribution through the pipeline levels are illustrated in Fig. 4.9.

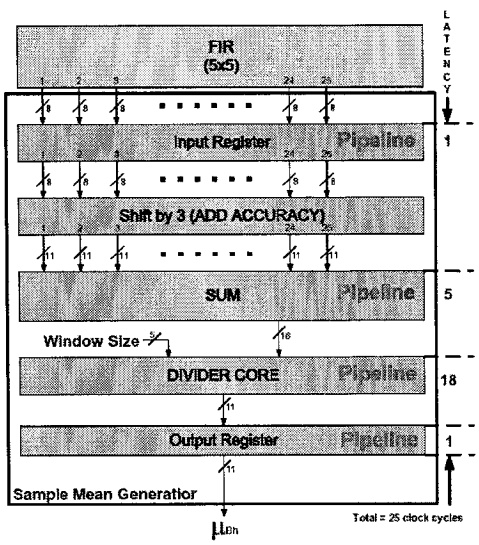


Figure 4.9: Sample mean generator structure.

4.2.6 Module A : Variance Generator

The sample mean is utilized to calculate the variance (σ^2_{Bh}) of each block as seen in Eq. 4.8. Following an in depth study of the variance equation, the decision was taken to partition it into five independent steps to facilitate its implementation.

$$\sigma^2_{Bh} = \frac{\sum_{(i,j)\in W_{ij}} (I(i,j) - \mu_{Bh})^2}{W \times W}. \tag{4.8}$$

The first step consists in increasing the inputs data's precision by shifting the data by three bits to the left. The second step involves a subtraction of each pixel value contained in the block to the "previously calculated" sample mean. That is why a delay element is needed (see Fig. 4.7) to stall the incoming pixel data during the mean latency time (e.g., the time required for the sample mean to be generated) before they can be read by the variance sub-module. The consequence of pipelining the sample mean sub-module requires the hardware designer to adjust the arrival time of the pixel data signals of each blocks to correspond with the block's sample mean value generation time, thus synchronizing the variance sub-module inputs. In the third step, the absolute value of the resulting subtractions are performed concurrently and squared in parallel with multiplier primitives (*MULT18x18*). This parallel structure is illustrated in Fig. 4.10. The squaring operation generate a worst case bus width of 22 bits of unsigned data as can be seen in Eq. 4.9.

$$\begin{aligned} Nbits * Nbits &= 2 * N \text{ bits} \\ 11bits * 11bits &= 22 \text{ bits} \end{aligned} \tag{4.9}$$

The fourth step consists in adding up the resulting squaring operations together which will render a worst case bus width of 27 bits of unsigned data as can be seen in Eq. 4.10.

$$\begin{aligned} \text{Sum} &= 25 * 2^{22} \\ &= 104857600 \approx 2^{27} \end{aligned} \tag{4.10}$$

The fifth step will average out the sum by dividing the result with the filter area (W^2) of 25. This procedure will bring back the data width of the variance ($\sigma^2_{B_h}$) to 22 bits of precision. The overall latency of the variance generator is 39 clock cycles and is mostly due to the large division core logic.

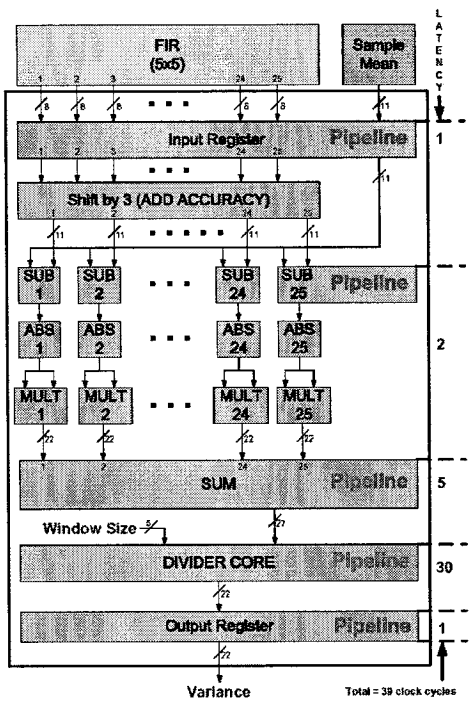


Figure 4.10: Variance generator structure.

The module A, presented in Fig. 4.7), also determines the amount of black and white

pixels that exist within a block of data. The control logic of the module detects whether the pixel's luminance value reside below the lower bound range (black pixels) or above the upper bound range (white pixels). The pixels that lie outside of the valid range are accumulated. The black or white pixel count must be smaller than half the window size to be considered for noise estimation. Otherwise the module disregards the data and renders the block in question as an invalid block since it does not respect the aforementioned condition. The ITU-R-BT-601 standard stipulates that the noise is best estimated in regions within these ranges to eliminate the clipping effects that could influence the estimation process. In this work, the valid luminance values ranges between 20 (lower bound) and 235 (upper bound) on an 8 bit gray-scale source. The module A concludes its processing duties by sending the generated homogeneous measure and variance data for each valid block that pass the black and white condition to the external memory DMA control sub-module.

4.2.7 ZBT External Memory and DMA Controller sub-module

For simulation purposes, the direct memory (DMA) controller module has been overly simplified to use a elementary communication protocol to transfer information to an external memory simulation model. This external memory component is programmed in VHDL to model the timing behavior of a zero bus turnaround (ZBT) static random access memory (SRAM). This type of memory has been chosen because of its fast read and write access time features. Generally, external memory access times are slow and complete a read or write procedure in the order of a few microseconds. The ZBT memory is capable of completing a read or write procedure in a few nanoseconds. This ZBT model can be download directly from a vendor's internet web page. In this work, a 16 MByte ZBT

SRAM model ($MT55L512Y32P^{TM}$) from *Micron Semiconductor Products Inc.* is utilized in the implementation. As mentioned in section 3.2.5, a more complex DMA control module would need to be developed to support a "real life" interface.

4.2.8 Module B : Sequential Sorting sub-module

Module B is composed of the sequential sorting circuit, the ordered variance storage devices and the reference variance generator as seen in Fig. 4.11. Module B fetches the homogeneous measure and variance values stored in the external memory (either bank A or bank B) as illustrated in Fig. 3.13 and Fig. 3.14. The sorting procedure is then executed on these recovered values.

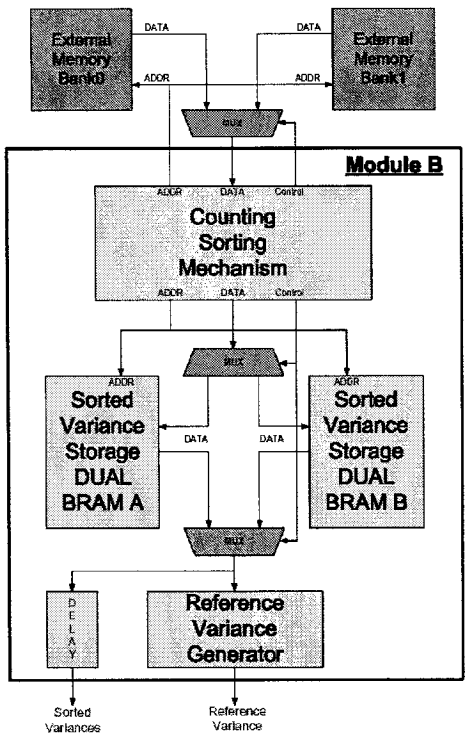


Figure 4.11: Module B sub-module circuit connection.

This sorting mechanism, described in section 3.2.2, implements the counting sort algorithm. This algorithm waits until the entire frame's block data, the homogeneous measure (ξ_{B_h}) and the variance ($\sigma^2_{B_h}$) have been generated and stored in an external memory. This technique is sequential and has been evaluated to require almost a full frame to process the sorting of all the values. The result of the processing delivers the $\sigma^2_{B_h}$ values corresponding to the top 10% of to the sorted ξ_{B_h} values of the ordered list of an entire field.

The sorting algorithm sequence is accomplished through a series of steps as mentioned in Table 3.2. A finite state machine (FSM) has been developed to control the steps of this sequence which can be seen in Fig. 4.12. It is important to note that this state machine was validated using a testbench where the sequence of state changes was verified manually.

The first step is the initialization of the internal memory RAM blocks ($BRAM$) to zero. The second step illustrated in Fig. 4.13, is the creation of the histogram which generates an ordered list of ξ_{B_h} into the internal memory RAM blocks. The ξ_{B_h} value is used as an address value to the memory component. The histogram records the number of instances the ξ_{B_h} values have been seen. It is not unlikely that the same address value may be retrieved many times because many occurrences of the same ξ_{B_h} data within one field is possible and very likely.

In the third step, shown in Fig. 4.14, the generated histogram is then manipulated to calculate the address indexing list. This list is used to locate the proper variance value corresponding to the proper ξ_{B_h} value. This list is stored back into the same internal memory RAM blocks.

In the fourth step, shown in Fig. 4.15, of the algorithm, this address index list is used

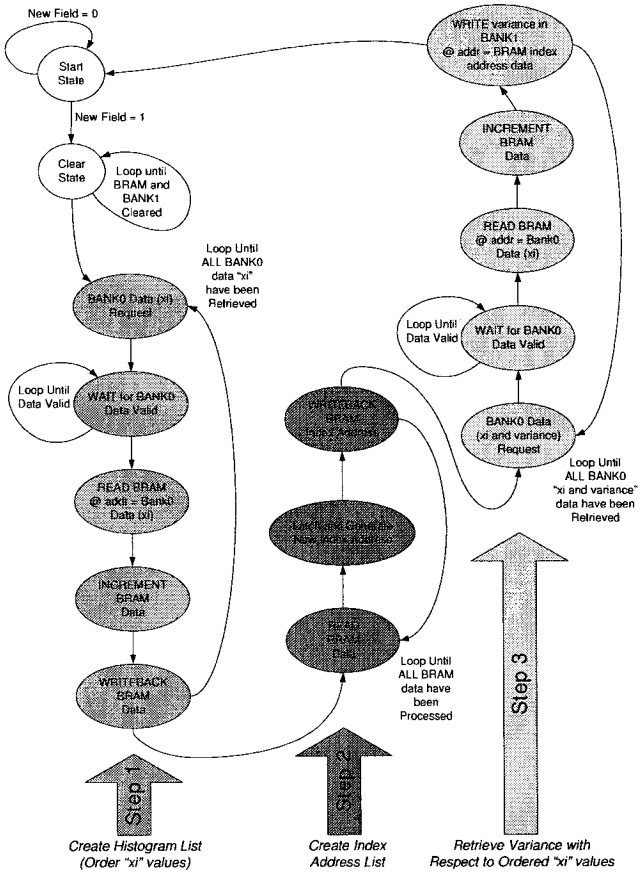


Figure 4.12: Finite State Machine (FSM) control system of the sorting algorithm.

to retrieve the variances ($\sigma^2_{B_h}$). The order in which the variances are fetched is relative to the order of their corresponding ξ_{B_h} value. Hence, the variance is fetched from external memory and is stored into another internal memory RAM block.

The hardware advantage of this sorting structure is that the digital components used to implement this internal BRAM memory is reused throughout the algorithm. For instance, the FPGA internal BRAM memory is used as a "working array" to create the histogram and to store the address index list. The counters used to cycle through the external memory bank and internal BRAM memory addresses are also being recuperated. This reduces the

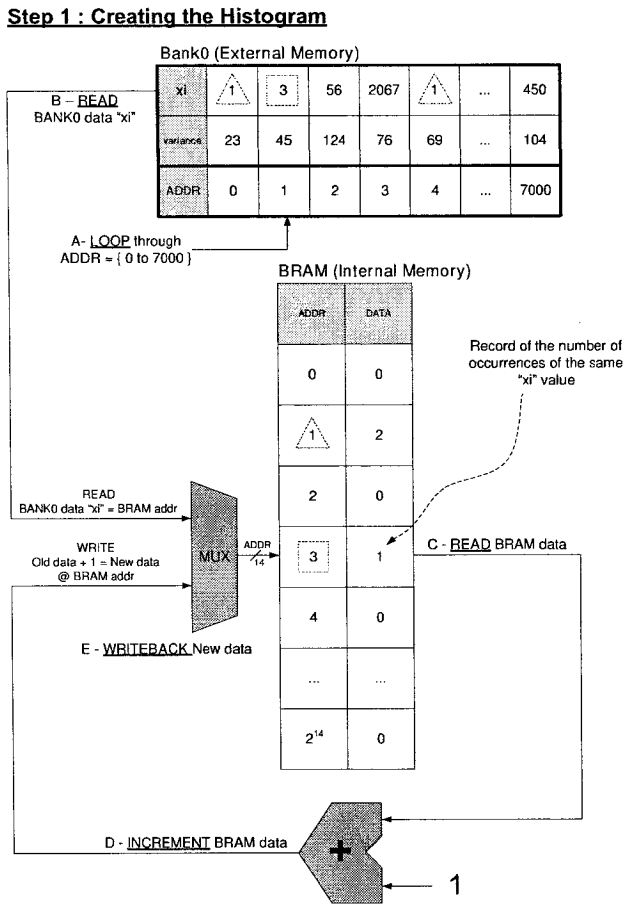


Figure 4.13: Step 2 : Histogram generation.

overall area being consumed by this sorting circuit. The most important advantage of this sorting design is that it sorts all the values of the entire field and does not modify the original algorithm.

Once the ordered list of variances has been extracted from the external memory, only the top 10% of the list is stored in another set of internal BRAM memories. Furthermore, a "ping-pong" mechanism is used to store the variance values of subsequent image fields. This technique, described in section 3.2.4, is necessary because of the risk that previously stored variance lists might be overwritten. As illustrated in Fig. 4.11, two separate internal BRAM

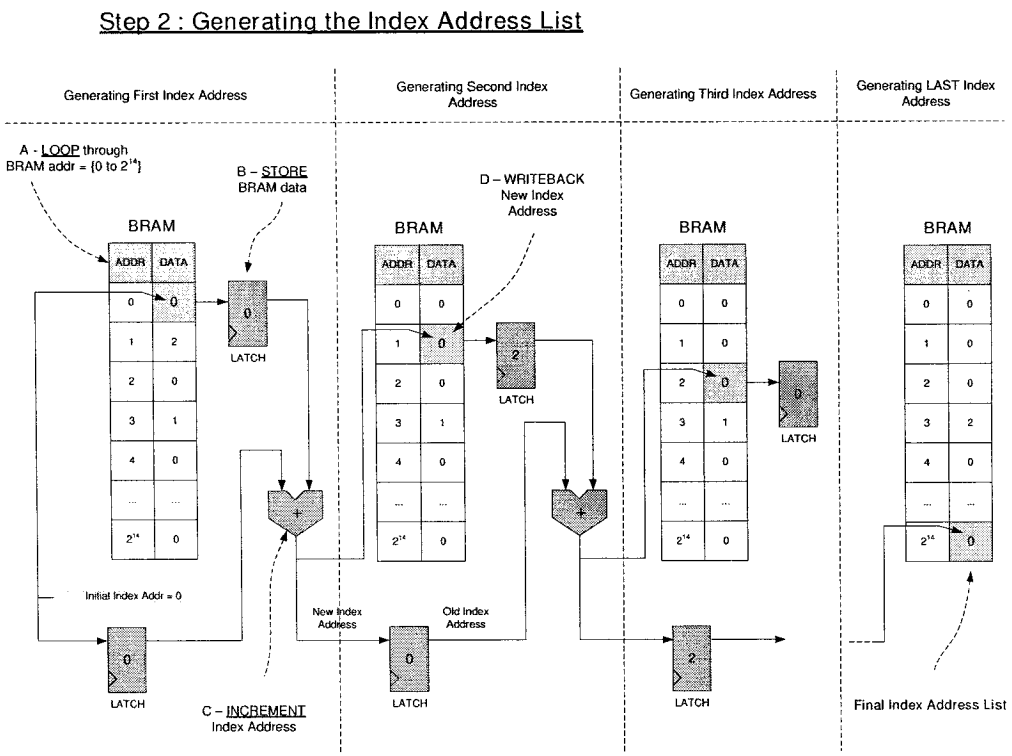


Figure 4.14: Step 3 : Index Address List generation.

memory components are used in parallel to store the ordered list of variances. Studies have determined that 3 internal BRAM blocks with dual access features are needed to build the storage components. The Eq. 4.11 describes the calculations needed to ascertain the number BRAM block for the worst case scenario using a small filter size (e.g., 3x3) and for a alternative filter size (e.g., 5x5).

$$\begin{aligned} \text{Number of BRAM blocks} &= (\text{ Variance Width } * \text{ Top 10\% Ordered List }) \text{ / BRAM Size} \\ &= \frac{22 * 2000^{3x3}}{16 * 1024} \approx 3 \text{ BRAMs} \\ &= \frac{22 * 800^{5x5}}{16 * 1024} \approx 2 \text{ BRAMs} \end{aligned} \tag{4.11}$$

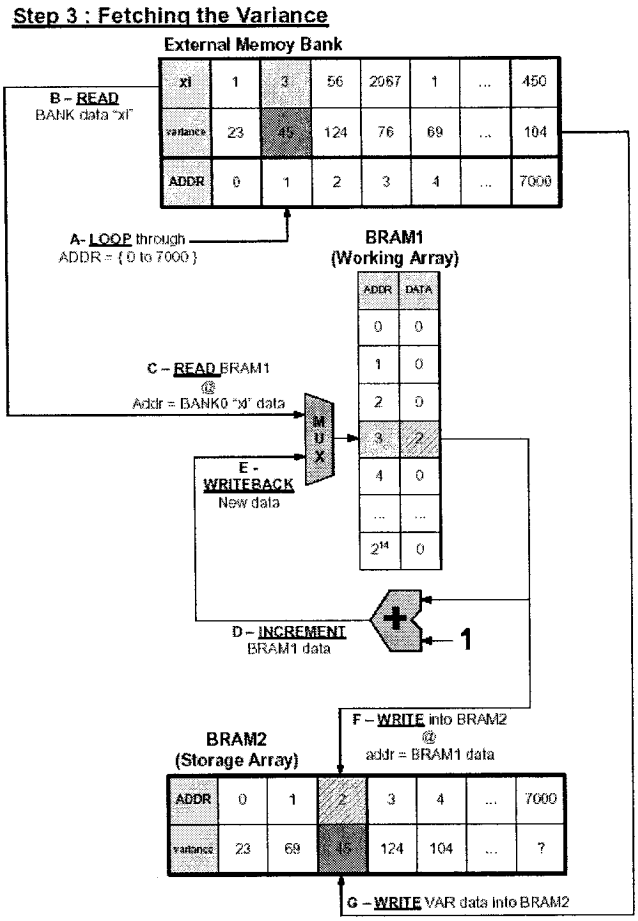


Figure 4.15: Step 4 : Variance Extraction.

The three most-homogeneous values of the ordered variance list are selected to generate the reference variance. The median of these three variances determine the reference variance which can be seen in Eq. 4.12. The top three values are represented by a, b and c and the MAX and MIN terms find the maximum and minimum values respectively. Although this equation is used in the software algorithm, it is computed in a different manner in this noise estimation implementation. To attain the same median results in a list of 3 items, it is easier to simply order the values in ascending order and to pick out the middle value. Hence, less hardware logic is consumed to build the Eq. 4.12 which would require the designer to

build adder circuits, MAX and MIN functions. This concludes the structure of Module B, which delivers the lists of the top 10% of variances values corresponding to the ordered list of homogeneous measure values and the reference variance which are transfered to the Module C.

$$MEDIAN(a,b,c) = a+b+c-MIN(a,b,c)-MAX(a,b,c) \quad (4.12)$$

4.2.9 Logarithmic Look-Up Table

The logarithmic look-up table (LUT) works hand in hand with the module C as seen in Fig. 4.16. The purpose of this LUT is to deliver the logarithmic values requested by module C during the processing of the final noise variance. As mentioned in the section 3.2.3, the logarithmic complexity is demonstrated in its inherent exponential order. Thus, the logarithmic function is realized with a look-up table using internal BRAM memory.

The LUT contains 16 bit worth of fixed-point logarithmic values that can be looked-up over 64K (2^{16}) addresses. The number of addresses are determined by the width of the address bus. The size of the LUT is 64K by 16bits and takes up 64 internal BRAM blocks. The look-up mechanism that fetches the logarithmic value of an input variance works in the following manner. In the first step, the variance data is reduced from a 22 bit signal to a 16 bit signal by shifting the data to the left by 6. The consequence is a loss of accuracy due to the truncation of the variance. Nevertheless, the decision to truncate was based on wanting to reduce the memory size needed to construct this LUT and was based on acceptable simulations results. Developing a LUT with a 22 bit (variance width) address width would

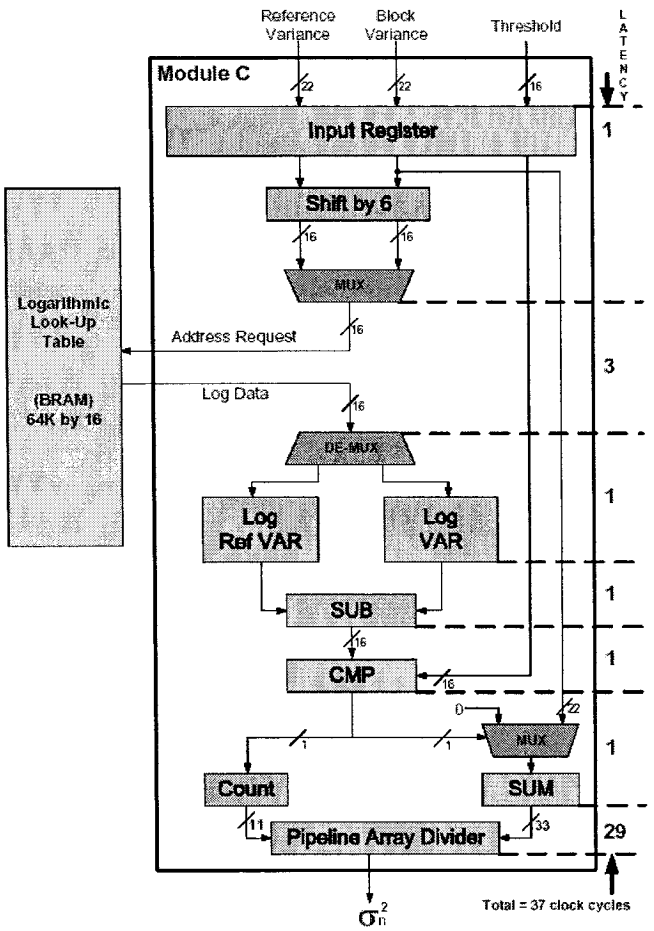


Figure 4.16: Module C internal circuit.

have required a 2^{22} location in memory, meaning that roughly 4000 internal BRAM blocks would have been consumed. Moreover, even high-end Virtex II FPGAs would not have been able to support such demand in memory space since they contain a maximum of 168 internal BRAM blocks (Xilinx XC2V8000). In the second step, this truncated variance signal is forwarded to the LUT's address bus. In the third step, the data is located at the variance's value address that corresponds to its equivalent logarithmic value. The data is then transfered out to the module C. This look-up table is illustrated in the Table 4.1. The first column represents the addresses, the second column expresses the floating-point

logarithmic values for the corresponding address values (this column is not implemented but is present to help the viewer understand the table) and the third column embodies the actual data stored in the LUT.

Address	Floating-Point Data (Decimal Format)	Actual Data (Hexadecimal Format)
0	NA	0
1	0	0
2	0.3010	0X09A2
3	0.4771	0X0F44
⋮	⋮	⋮
$2^{16}-1$	4.8164	0X9A20

Table 4.1: Logarithmic look-up table

The actual data present it this table is calculated in Eq. 4.13. The data stored in the look-up table does not correspond to a floating-point value since floating representation is not used in the proposed design. Instead, the data is up-scaled to a large number in order to keep as much accuracy as possible for the threshold condition calculations. This technique is similar to the fixed-point upgrade method discussed in section 3.2.1. Hence, the actual data stored in the LUT corresponds to the real logarithmic value of the variance multiplied by 2^{13}. Furthermore, the threshold and the reference variance used in the threshold condition calculation in Eq. 4.14 are also up-scaled to larger values to coincide with the up-scale variance terms.

$$\text{Actual LUT data} = \log(\text{LUT address}) * 2^{13} \tag{4.13}$$

Consequently, the condition is evaluated with less accuracy and reveals to be a major

source of error in the proposed implementation in particular circumstances. As mentioned in section 3.2.5, studies of the simulation results have shown that high image noise levels (PSNR $>$ 40 db) translate into small variance values and that a considerable asymmetry (large difference) can be distinguished between logarithmic results of subsequent small numbers. Therefore, evaluating the threshold condition of images that contain high noise levels using a less precise logarithmic look-up table (only 16 bits of data) also leads to unaccurate results. With high image noise levels, the threshold condition is being evaluated with logarithmic values which are far apart and which also have more of a probability of being outside of the acceptable threshold range and being omitted. When many values are missing, a discrepancy can be noted in the final variance results.

4.2.10 Module C : Final Variance Generator

Module C, seen in Fig. 4.16, generates the final noise variance. This module evaluates the condition seen in Eq. 4.14 which determines the valid variances from the list of top 10% of most-homogeneous variances values stored in Module B.

$$|log(\sigma^2_{Bh}) - log(\sigma^2_{REF})| \quad < \quad t_\sigma \tag{4.14}$$

The condition is broken down into a sequence of three steps. In the first step, the logarithmic value of the block's variances is fetched from the logarithmic look-up table. In the second step, the difference between the logarithmic value of the reference variance and the logarithmic value of the individual block variances is computed. In the third step, this

difference is compared to a threshold (t_σ) to verify that the variance is within an acceptable homogeneity range with respect to the reference variance.

The resulting valid variance that pass the threshold condition are then added together. In the worst case scenario, each block in the field is selected as being valid (homogeneous). The sum of all top 10% most-homogeneous valid variance values would generate bus widths of 33 bits unsigned data as can be seen in Eq. 4.15 for a 3x3 filter size.

$$\begin{aligned} \text{Max Variance Magnitude} &= \sigma^2_{B_{h\_MAX}} \approx 2^{22} \\ \text{Top 10\% Valid Block Count} &= 0.10 * 23040 \approx 2000 \text{ Blocks} \\ \text{Total Sum Magnitude} &= \sigma^2_{B_{h\_MAX^1}} + \sigma^2_{B_{h\_MAX^2}} + \cdots + \sigma^2_{B_{h\_MAX^{2000}}} \\ &= 2^{22} + 2^{22} + \cdots + 2^{22} \approx 2^{33} \end{aligned} \tag{4.15}$$

The resulting sum is divided by the number of variances that have been accumulated. Thus, the final noise variance (σ^2_n) is produced from the average of the valid variances that have satisfied the threshold condition.

4.3 Summary

In this chapter, the implementation of the proposed noise estimation architecture is presented. The top level organization of the internal modules and their correlations are described. The intricate functionality of each module A, B and C are detailed. The general purpose DCM module's use to accelerate local clock frequencies is presented. The scalable

pipelined divider module developed to increase the architecture's flexibility is decomposed and it's non-restoring array structure is described. Module A, acting as input module, is shown as working to capture and sample the image pixels and generating the block values such as the homogeneity measure, the sample mean and the variance. The external memory controller, serving as a DMA interface with the FPGA, is depicted. Module B is defined as managing the sorting mechanism and producing the reference variance. The internal structure of the logarithmic table is illustrated. Module C is depicted as the final variance generator. The implementation's data path and arithmetic unit sizes are analyzed and justified for both 3x3 and 5x5 filter configurations. In the following chapter, the experimental results are shown and analyzed to validate the implementation's functionality, to present the total area consumption, and to prove the real-time processing performance of the design.

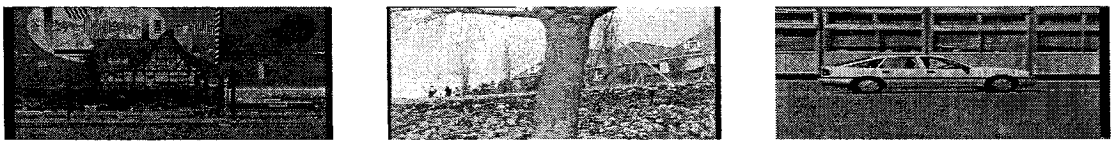
Chapter 5

Experimental Results

The experimental results are analyzed in three separate sections. In the first section, the hardware simulation results are compared to the original software results to determine the quality of the accuracy and the correct functionality of the FPGA implementation. In the second section, the design's synthesis and routing results are presented which describe the type of resources and the amount of area that has been consumed. The third section depicts the design's processing and execution timing results and validates the real-time processing performance of the noise estimation implementation.

5.1 Noise Estimation Hardware Accuracy

This section validates the quality of the design's noise variance results. The noise variance created by the hardware implementation from the processing of various video sequences are compared to noise variance results generated by the original software program. The



(a) Train (b) Flower Garden (c) Prlcar

Figure 5.1: Single Field of Video Test Sequences Used

simulations have been executed for the 3x3 and 5x5 filter size configurations. To test the accuracy of the proposed design, commonly-referenced video sequences (See Fig. 5.1) were applied to the hardware noise estimation implementation. These sequences (*Train*, *Flower_Garden*, and *Prlcar*) contain complex image structure and various motions such as pan and zoom. The design is tested on different PSNR noise levels (20, 30, and 40 dB) that are typically found in real-world images. These specific video sequences are used to make sure that the design's functionality is independent of the input image content and that the hardware behavior is reliable with respect to the original software algorithm.

The software simulations were performed using the executable file generated when compiling the C code. The hardware simulations were performed using the *Synopsys VHDL Simulator*^{TM} tool. The simulator tools acquired the same pixel files where each file contained 1 of 60 frames of a video sequence of type CCIR-601 (International Radio Consultative Committee) with a PAL resolution of 720x480 pixels per image frame. The comparison between the software results and the hardware results were then plotted by the *Matlab*^{TM} tool. The diagram of the experimental validation procedures are shown in Fig. 5.2.

Fig. 5.3 illustrates the average results of the PSNR estimation error from the hardware design compared to the original software results for two video (*Train* and *Flower_Garden*)

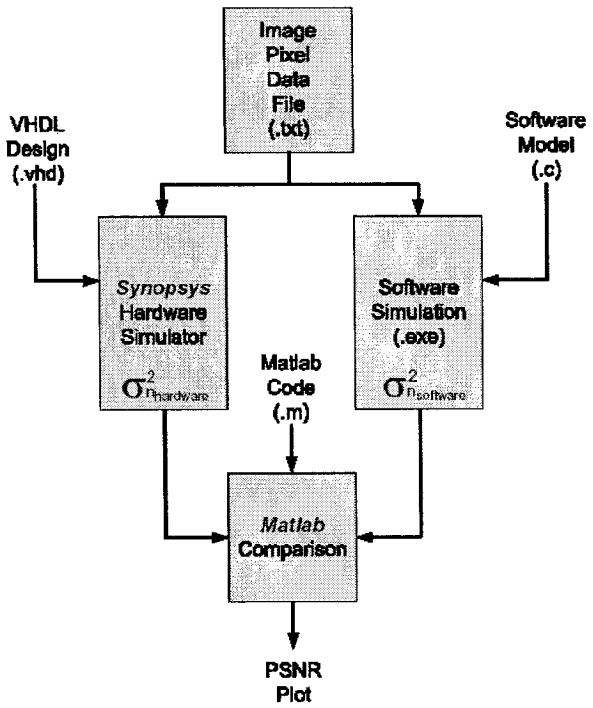
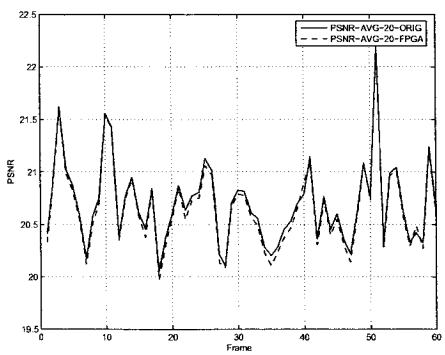


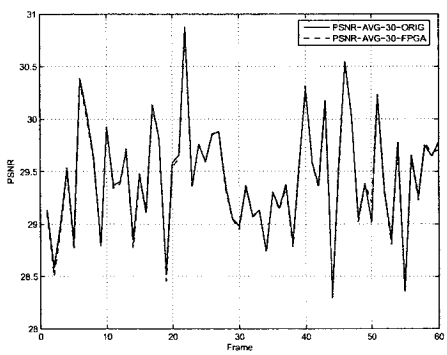
Figure 5.2: Overview of experimental validation procedures.

sequences containing 20dB, 30dB and 40dB of noise using a 5x5 filter size. The results (i.e. original software results versus the hardware results) produced by the 5x5 filter configuration are quite congruent as they almost coincide exactly.

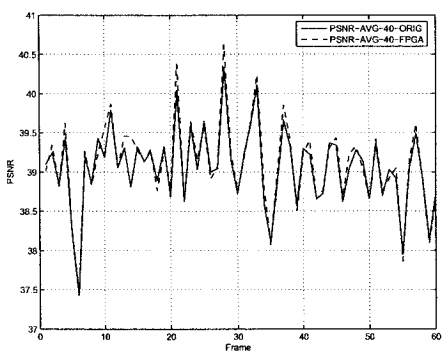
Fig. 5.4 illustrates the same average results of the PSNR estimation error for three video (*Train*, *Flower_Garden*, and *Prlcar*) sequences containing 20dB, 30dB and 40dB of noise using a 3x3 filter size. The difference between the original and hardware results generated by the 3x3 filter configuration are noticeable and they do not superimpose unequivocally. These diverging noise estimation results are largely due to the logarithmic look-up table's lack of precision for lower bound variance values creating high PSNR levels (PSNR $>$ 40db) and by the fact that smaller filter size consequently increase the number of blocks per frame and, in turn, increase the chance of discrepancies in logarithmic calculations (refer to



(a) 20dB

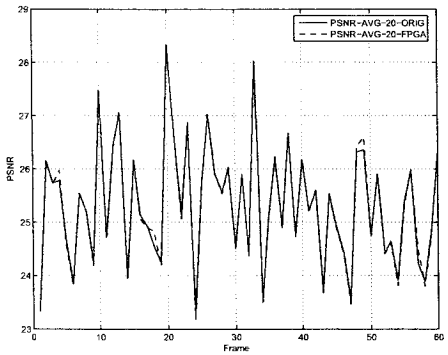


(b) 30dB

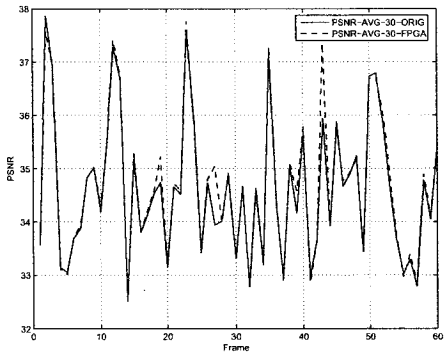


(c) 40dB

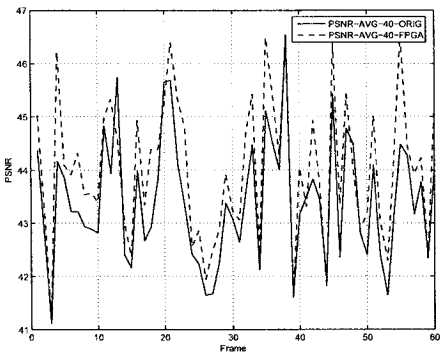
Figure 5.3: Software versus hardware implementation average estimated PSNR using 5x5 filter size



(a) 20dB



(b) 30dB



(c) 40dB

Figure 5.4: Software versus hardware implementation average estimated PSNR using 3x3 filter size

discussion in section 3.2.5 for more information).

Although the results for both configurations are not perfect, the accuracy of these simulations show acceptable quality while using a simple fixed-point logarithmic structure. In fact, the worst case error is shown to be less than 3 dB, as seen in the PSNR plot results. The accepted simulated design is then synthesized and routed; these results are presented in the next sub-section.

5.2 Synthesis and Routing Results

The register transfer level (RTL) synthesis design step is of primary importance since it will determine whether the hardware implementation is realizable. The synthesis step is performed by the *Synoposys Design Compiler*^{TM} tool. The hardware circuits are described as a collection of registers, Boolean equations and control logic (such as "if-else" statements) by the RTL hardware language (VHDL). These RTL specifications are translated into EDIF netlists through RTL synthesis. The tasks performed by the synthesis design step include :

- Flattening the architecture hierarchal levels : Changing the combinational logic from multi-level to a flat single level description.
- Logic optimization : Performing Boolean factorization and expression reduction of the combinational logic.
- Circuit Transduction : Reducing the gate count by eliminating redundant logic without changing the circuit functionality.

As mentioned in section 3.1, the synthesis procedures will allow the designer to get preliminary timing results and obtain an overview of the initial resource consumptions. The synthesis was performed on each sub-modules individually and on the final top level design. The execution of the synthesis script files produced complex log files (error and warning records) that determine the ultimate pass or fail status of the proposed design step. Although the synthesis step is necessary to optimize the design and create the EDIF file, the place and route design step output files are deemed more important to present since

they hold more accurate implementation results in terms of resource allocation (area) and operating frequency (processing speed).

The place and route design steps are performed by *Xilinx Project Navigator*^{TM} tools. These design steps work to map the synthesized netlist (EDIF), place the logic and route the signals. The tasks performed by the place and route steps include :

- Mapping : Mapping the synthesized netlist components onto the types of resources (logic cells, i/o cells, etc.) available in the specific FPGA (Virtex, Spartan, etc.) being targeted.
- Placing : The resources are assigned specific locations (Flip-flops, Slices and LUTs) on the FPGA.
- Routing : The connections between the resources are associated into the FPGAs interconnect network.

These design steps are performed on a target Xilinx XC2V4000 Virtex FPGA. The results of the place and route design steps produce a series of text files. A summary of these text files showing the resources consumed by the noise estimation design is illustrated in Table 5.1. The left column represents the resource type and the right column depicts the alloted proportion of these resource types. The resource types consist of flip-flops (single bit memory elements), LUT (4-input look-up tables), Slices (primary element of configurable logic blocks), BRAM (internal Block RAM), MULT18X18 (dedicated multiplier blocks) and DCMs (digital clock manager blocks). These resource types comprise the basic building blocks used to implement this thesis's design onto a Xilinx FPGA device.

Resource	Percent
Flip Flops	6504/46080 (14%)
LUT	4865/46080 (10%)
Slices	4821/23040 (20%)
BRAM	73/120 (60%)
MULT18X18	25/120 (20%)
DCM	1/12 (8%)

Table 5.1: Resource utilization for XC2V4000 Xilinx FPGA.

It is important to note that the most allocated resource type was the BRAM element due to the large logarithmic look-up table. This issue is discussed in section 3.2.5 which explains the practical reasons why the large logarithmic look-up table was implemented using internal memory instead of external memory. Otherwise, this noise estimation algorithm could have been implemented onto a smaller sized FPGA, which in turn, could eventually lower the overall cost of the design. The next sub-section describes the design's processing and execution timing results.

5.3 Design Timing Description

The final implementation processing time and execution timing details are presented in this sub-section. In the following paragraphs, the confirmation of real-time processing performance is proven through timing equations for the various filter configurations. The pipelining design technique is illustrated to describe the execution timing of the design.

5.3.1 Real-time Performance Validation

The thesis's initial objective of attaining real-time processing performance is authenticated. A series of equations are used to validate the processing time of the noise estimation system. It is important to note that the proposed design is entirely pipelined except for the sorting part of the algorithm. Thus, only the timing evaluation of the sorting mechanism is significant in determining if the proposed design is capable of delivering real-time performance. The standard PAL pixel input rate of an interlace video sequence is 0.02 seconds per field ($t = \frac{1}{50Hz}$). Eq. 5.1 verifies this industry standard rate by dividing the input field resolution (pixel resolution) with the field's input pixel frequency rate. This standard rate serves as a delimiting time factor that must be satisfied to respect real-time performance. In other words, the sorting mechanism must complete its ordering process within the standard field rate time (0.02 seconds).

$$\begin{aligned} \text{Field Time (seconds)} &= \text{Field Resolution (Pixels) / Pixel Rate Frequency (MHz)} \\ &= \frac{864 * 312}{13.5} = 0.01995 \approx 0.02(50Hz) \end{aligned} \tag{5.1}$$

As mentioned in Table 3.2, the counting sort algorithm completes its sorting sequence in four passes. The number of clock cycles required to execute the four passes of the counting sort sequence, CSS, is shown in Eq. 5.2. The first and third pass loop through each location of the working array (internal BRAM) which holds up to $2^{\xi_{WIDTH}}$ number of addresses. According to the VHDL code and the simulation results, the first and third pass take a total of 5 clock cycles to complete and is represented by $\text{CC}_{PASS_{1\&3}}$. Furthermore,

the second and fourth pass loop through each location of the variance and homogeneous measure storage array (external memory BANK 1 an 2) which amount to the number of blocks per field (BPF^{NxN}) depending on the filter size being used. According to the VHDL code and the simulation results, the second and fourth pass take a total of 27 clock cycles to complete and is represented by $\mathrm{CC}_{PASS_{2\&4}}$.

$$\begin{aligned}
\mathrm{CSS}^{NxN}\ (\text{clock cycles}) &= (\mathrm{CC}_{PASS_{2\&4}} * \mathrm{BPF}^{NxN}) + (\mathrm{CC}_{PASS_{1\&3}} * 2^{\xi_{WIDTH}}) \\
\mathrm{CSS}^{5x5}\ (\text{clock cycles}) &= (27{*}8208) + (5 * 2^{13}) \approx 222K \\
\mathrm{CSS}^{3x3}\ (\text{clock cycles}) &= (27{*}23040) + (5 * 2^{13}) \approx 540K
\end{aligned} \tag{5.2}$$

The processing time of the counting sort algorithm for both filter sizes is shown in Eq. 5.3. The number of clock cycles required to execute the four passes of the counting sort sequence (CSS) is divided by the sorting frequency (40.5 MHz). Originally, this sorting frequency (using the input pixel frequency of 13.5 MHz) had not been accelerated and the noise estimation algorithm was not able to complete within the standard field rate of 0.02 seconds. Nevertheless, in the current implementation, according to Eq. 5.3, both filter sizes render real-time performances. It was therefore necessary to accelerate the internal sorting clock frequency using a DCM specific component to achieve real-time processing.

$$\begin{aligned}
\mathrm{ST}^{NxN}(seconds) &= \mathrm{CSS}^{NxN} \ /\ \mathrm{SF}\ (\mathrm{MHz}) \\
\mathrm{ST}^{5x5}(seconds) &= \frac{222K}{40.5} \approx 0.00648 < 0.02 \Rightarrow \text{Real-time !} \\
\mathrm{ST}^{3x3}(seconds) &= \frac{540K}{40.5} \approx 0.01637 < 0.02 \Rightarrow \text{Real-time !}
\end{aligned} \tag{5.3}$$

In determining the maximum possible clock acceleration, the place and route tool performs a timing analysis of the entire design. The tool evaluates the maximum timing delays that exist in the system's circuit. The experimental results extracted (see Table 5.2) by the tool have produced a maximum delays of approximately 20ns, thus delivering a maximum operating frequency of 50MHz. Note that these timing results are unconstrained in that the design's implementation has no restriction on mapping procedures, block placement and/or timing specifications.

Avg Net Delay (ns)	Max Pin Delay (ns)	Max Clock Skew (ns)	Max Clock Delay (ns)
2.343	19.484	13.870	15.320

Table 5.2: Unconstrained place and route timing results for XC2V4000 Xilinx FPGA.

5.3.2 Pipelining Portrayed In Time

In order to fully comprehend the timing distribution of the various processes of this noise estimation implementation, the pipelined design is portrayed in time in Fig. 5.5. This illustration depicts the initial overhead latency found at the beginning of the noise estimation processing of the first field. The time that is consumed by the sequential sorting algorithm is clearly shown as the bottleneck of the proposed design as it takes up a large portion of the processing time. Knowing that the proposed design processes two fields at a time (current and previous fields are processed concurrently through the "ping-pong" structure), the variance is only required to be generated before the input of the third field. Hence, from this figure, the real-time performance is once again validated as it can be seen that the first field's variance is generated before the third incoming field. The general concept

of pipelining is portrayed in the fact that while pixels are being acquired, previous fields are being processed and variances are being generated. Pipelining a design is similar to allowing independent processes to work independently and yet overlap each other in time as previously mentioned in section 3.2.4.

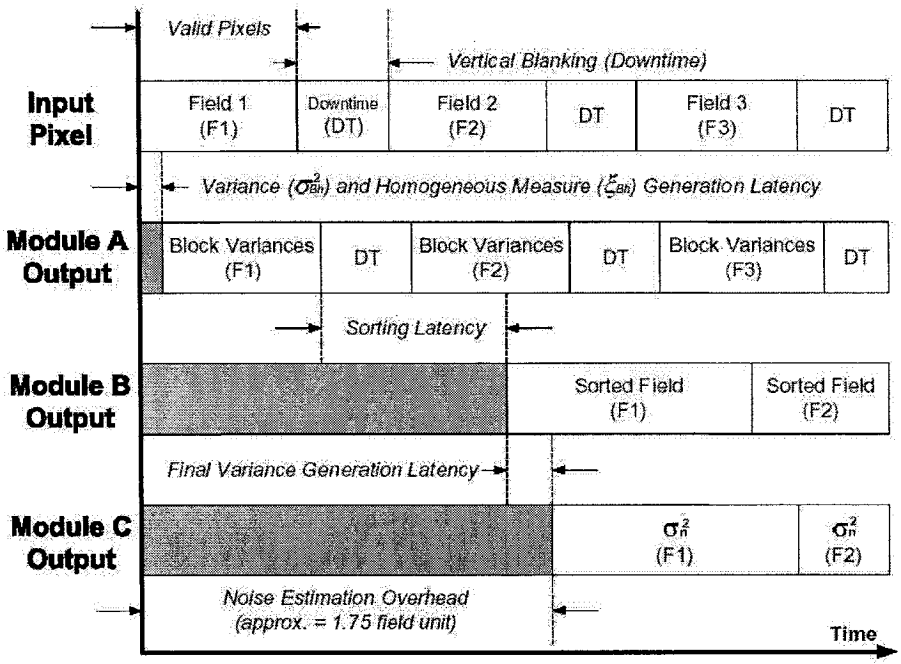


Figure 5.5: Pipeline timing diagram.

This pipelining technique can be extrapolated for uses in a larger video processing system. Fig. 5.6 demonstrates a software DSP system diagram processing data in a sequential fashion. The Fig. 5.6 depicts a general overview of an image enhancement and image analysis system which could eventually be used for automatic dynamic scene interpretation and representation. This DSP system is composed of a noise estimation block that generates a variance value that is forwarded to an edge detection block which then feeds an image segmentation block. In this type of system, the completion of the processing of a current field is necessary before a subsequent field can be analyzed.

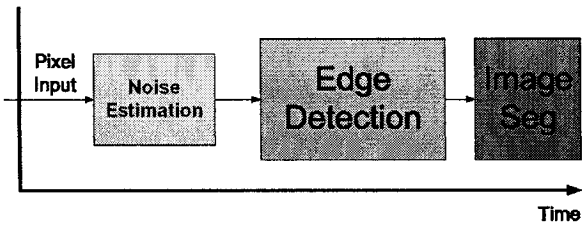


Figure 5.6: Overview of sequential video processing system.

Fig. 5.7 illustrates the same DSP system processing video data using a pipelined structure and its corresponding timing diagram depicting the components task's duration. In the timing diagram of this pipelined design, the overlapping of tasks can clearly be seen, which in turn, reduces the processing time of the overall system. For example, while the edge detection result of the first field is being produced, the noise variance of the second field is being generated and the third field is being acquired. These figures help to better portray the fundamental operation of the pipelining technique.

5.4 Summary

In this chapter, the noise variances generated by the hardware implementation processing various video sequences are compared to noise variance results generated by the original software program. The simulations were executed for the 3x3 and 5x5 filter size configurations and have produced acceptable variance estimations. Albeit the results for both configurations were not perfect, the accuracy of these simulations results have shown to give satisfactory quality while using a simple fixed-point logarithmic structure. Furthermore, the worst case error observed is shown to be less than 3 dB, as seen in the PSNR plot results. The synthesis and routing results generated the area size used to implement this proposed

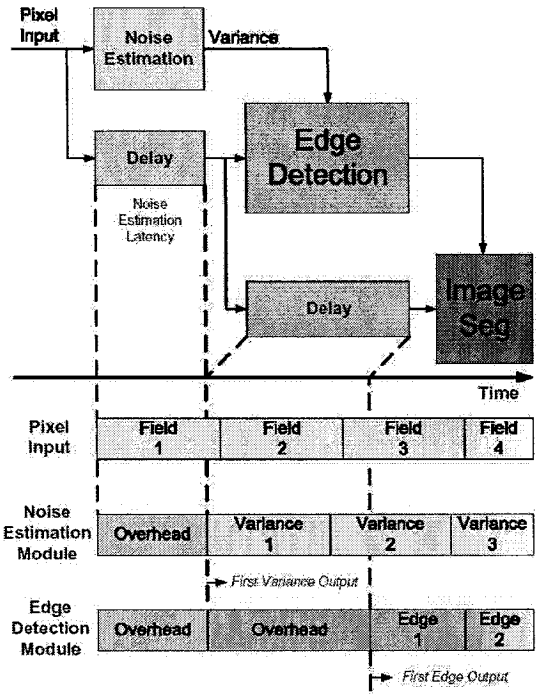


Figure 5.7: Pipeline structure and timing diagram of video processing system.

design and showed that the most allocated resource type was the BRAM element due to the large logarithmic look-up table. Otherwise, the logic consumption was minimal as less than 15% of the flip-flops and less than 20% of the slices of the FPGA ressources were used. The maximum operating frequency of the proposed architecture generated by the routing tools was approximately 50 MHz although only 40.5 MHz was required to render real-time performance. In this chapter, the pipelining and "ping-pong" design techniques have also been shown through timing diagrams to be essential in realizing real-time performance.

Chapter 6

Conclusion

6.1 Concluding Remarks and Contributions

Video signal processing necessitates considerable computation, manipulation of large quantities of data and storage capabilities. The demand for expanding image resolution and faster processing performance has reduced the effectiveness of software-based video processing methods. Notwithstanding multi-threading capabilities of software programs and powerful performances of the current CPUs, still important video processing software algorithms, are incapable of processing data fast enough. However, real-time performance can be attained by means of dedicated hardware such as FPGAs. Programmable FPGA hardware technology is well suited for fast-prototyping of system implementation of DSP algorithms as it reduces the time-to-market by allowing designers to develop abstract mathematical ideas into physical hardware implementations in a short period of time.

This thesis focused on a noise estimation algorithm in video processing that is used as a pre-processing technique to estimate the noise in a video sequence. In this thesis, an implementation of a noise estimation algorithm onto a Xilinx FPGA is presented. The primary focus of reaching real-time performance is accomplished by efficiently pipelining the VHDL design, promptly parallelizing the architecture and adequately adapting the software algorithm into a synthesizable hardware implementation. In general, the transformation of the software program is not simple and some design feasibility issues had to be overcome.

The main objective of generating real-time performance was met by adapting the algorithm to resolve many feasibility obstacles. These obstacles found in the proposed algorithm included the realization of complex mathematical expressions (i.e., logarithmic functions), the establishment of an adequate arithmetical precision structure to replace a software floating-point representation, and the creation of fast sorting functions in hardware.

The first obstacle consisted in implementing a complex logarithmic operation. Most hardware description languages (HDL's) do not support equivalent "built in" logarithmic operators. In the proposed design, this intricate arithmetic operation was solved by storing the logarithmic values into a look-up table. The consequence was noted as a minor loss of precision in the final noise estimation results for video sequences containing upper range noise levels.

The second obstacle was related to a software issue that affected the overall precision of the proposed design. The original software was programmed using a floating-point representation which is fairly complex to build in hardware. Alternatively, this thesis's design was developed with a fixed-point structure. This implied using less-accurate fractional numbers

to perform arithmetic calculations. However, a relatively precise logarithmic table and a 22-bit fixed-point representation scheme produced satisfactory noise variances which proved to be accurate compared to the original software results showing a worst-case estimation error of less than 3 dB.

The third and principal obstacle of this thesis was to overcome the time consuming sorting issue required by the noise estimation algorithm. Most popular sorting algorithms usually process data lists in a sequential fashion by comparing and swapping data positions which is time consuming. Moreover, other conventional sorting methods use recursive procedures which render large logic circuits. Consequently, an alternate counting sort algorithm was implemented which completed the ordering of variances in less than a time frame using four distinct steps and a single internal BRAM memory. Furthermore, the coupling of the counting sort technique with a pipelining method, a "ping-pong" architecture, and an acceleration of the local clock with DCM helped improve the sorting processing performances to attain real-time processing speeds.

Another objective was achieved by incorporating scalability features such as different filter sizes. VHDL generics, global packages, and a special purpose scalable array divider permitted the parameterization of the proposed design. Moreover, the advantage of the proposed architecture is that it is designed in such a way as to be easily modified for future configurations that could harbor new parameters such as various image resolution and/or a wider range of filter sizes.

Minimizing the area was conjointly a predominant objective of the proposed design. Reducing the FPGA logic consumption was achieved by efficiently utilizing the internal Xilinx

primitives and by taking advantage of external memory devices to store large quantities of data such as each block's variances and homogeneous measures.

The analysis of the experimental results between the hardware simulations and the original software simulations showed that the proposed implementation generated almost identical noise variance results as compared to the original software results. The design was also synthesized and routed for a target Xilinx XC2V4000 Virtex FPGA chip. The resource allocation results reported that only 20% of the FPGA slices had been utilized for this implementation. Small area consumption is advantageous in that the remaining slices can be employed to upgrade the data formats or to incorporate other processing algorithms (e.g., video noise reduction or motion estimation) that use noise estimated variance values.

The main contributions accomplished in this thesis are as follows:

i. Defining a computation procedure for a noise estimation algorithm that can be effectively synthesized for FPGA implementation.

ii. Proposing a convenient and flexible scalable architecture that includes such attributes as various filter sizes and that could harbor additional features such as expandable resolutions and variable pixel data sizes.

iii. Successfully developing an FPGA system of the noise estimation algorithm capable of reaching real-time processing performance.

6.2 Suggestions for Future Work

Future work to improve the proposed hardware architecture includes speeding up the processing time by increasing the number of pipeline stages. The modules that compose the proposed design could be extensively pipelined to minimize the circuit delays and increase the operating frequency. The area could be minimized by implementing the logarithmic look-up table with external memory to reduce the FPGA internal BRAM memory consumption. By establishing the look-up table on external storage devices, the logarithmic precision could be improved by augmenting the memory capacity. This would allow for additional look-up table locations where each location could incorporate more bits of data that would increase the precision of each logarithmic values. Further amelioration to the architecture could render a more flexible hardware such as the adaptation of the architecture to include added scalability features. These new features could include, i) a wider range of filter sizes, ii) the possibility of changing the image resolution, video standard type (e.g., PAL or NTSC) or pixel data sizes, and iii) the availability of processing single and multiple rows in an image field depending on the content of the input video sequence.

Bibliography

[1] Carver Mead, *Analog VLSI and Neural Systems*, Addison Wesley Publishing Company, 1989.

[2] M. Leeser, S. Miller, and Y. Haiqian, "Smart camera based on reconfigurable hardware enables diverse real-time applications," in *Proc. of 12th Annual IEEE Int. Symp. on Field-Programmable Custom Computing Machines*, Boston, USA, Apr. 2004, pp. 147–155.

[3] W. Wolf, B. Ozer, and T. Lv, "Smart cameras as embedded systems," *IEEE Computer Society, Computer*, vol. 35, no. 9, pp. 48–53, Sept. 2002.

[4] A. Amer and E. Dubois, "Fast and reliable structure-oriented video noise estimation," *Circuits and Systems for Video Technology, IEEE Transactions on*, vol. 15, no. 1, pp. 113–118, Jan. 2005.

[5] H. Schröder, "Image processing for TV-receiver applications," in *Proc. of Int. Conf. on Image Processing and its applications*, Maastricht, The Netherlands, Apr. 1992, Keynote paper.

[6] J. Canny, "A computational approach to edge detection," *IEEE Trans. Pattern Anal. Machine Intell.*, vol. 9, no. 6, pp. 679–698, Nov. 1986.

[7] L. Ping and Y.F. Wang, "Local scale controlled anisotropic diffusion with local noise estimate for image smoothing and edge detection," in *Proc. of Conf. on Computer Vision*, Riverside, USA, Jan. 1998, pp. 193–200.

[8] H. Jiuxiang, A. Razdan, G.M. Nielson, G.E. Farin, D.P. Baluch, and D.G. Capco, "Volumetric segmentation using weibull e-sd fields," *IEEE Trans. on Visualization and Computer Graphics*, vol. 9, no. 3, pp. 320–328, July 2003.

[9] C.S. Byung and W.C. Kang, "Motion-compensated noise estimation for efficient prefiltering in a video encoder," in *Proc. of Int. Conf. on Image Processing*, Barcelona, Spain, Sept. 2003, vol. 2, pp. 211–214.

[10] F. Russo, "A method for estimation and filtering of gaussian noise in images," *IEEE Trans. on Instrumentation and Measurement*, vol. 52, no. 4, pp. 1148–1154, Aug. 2003.

[11] A. Amer and H. Schröder, "A new video noise reduction algorithm using spatial sub-bands," in *Proc. of Int. IEEE Conf. on Electronics, Circuits and Systems*, Rodos, Greece, Oct. 1996, vol. 1, pp. 45–48.

[12] Tamal Bose, *Digital Signal and Image Processing*, John Wiley & Sons, Inc., New Jersey, USA, 2004.

[13] S.I. Olsen, "Estimation of noise in images: An evaluation," *CVGIP: Graphical Model and Image Processing*, vol. 55, no. 4, pp. 319–323, July 1993.

[14] P. McWilliams, S. McLaughlin, D. Laurenson, W. Collis, M. Weston, and P. White, "Non-linear filtering for broadcast television: a real-time FPGA implementation," in *Proc. of Int. Conf. on Image Processing*, Thessaloniki Greece, Oct. 2001, vol. 3, pp. 354–357.

[15] R.D. Turney, A.M. Reza, and J.G.R. Delva, "FPGA implementation of adaptive temporal kalman filter for real time video filtering," in *Proc. of Int. Conf. on Acoustics, Speech, and Signal Processing*, Phoenix, USA, Mar. 1999, vol. 4, pp. 2231–2234.

[16] K. Benkrid and S. Belkacemi, "Design and implementation of a 2d convolution core for video applications on FPGAs," in *Proc. of Int. Workshop on Digital and Computational Video*, Belfast, UK, Nov. 2003, pp. 85–92.

[17] C. J. Juan, "Modified 2d median filter for impulse noise suppression in a real-time system," *IEEE Transactions on Consumer Electronics*, vol. 41, no. 1, pp. 73–80, Feb. 1995.

[18] A. Nelson, "Implementation of image processing algorithms on FPGA hardware," M.S. thesis, Vanderbilt University, 2000.

[19] S. Memik, A. Katsaggelos, and M. Sarrafzadeh, "Analysis and FPGA implementation of image restoration under resource constraints," *IEEE transactions on Computers*, vol. 52, no. 3, pp. 390–399, Mar. 2003.

[20] U. Bidarte, J.A. Ezquerra, A. Zuloaga, and J.L. Martin, "Vhdl modeling of an adaptive architecture for real-time image enhancement," in *Proc. of VHDL International Users Forum (VIUF)*, Orlando, FL, USA, Oct. 1999, pp. 94–100.

[21] Xilinx Inc., "Virtex-ii platform FPGAs : Complete data sheet (ds031 v3.4)," Xilinx On-line, http://direct.xilinx.com/bvdocs/publications/ds031.pdf, 2005.

[22] L. Zhuo and V.K. Prasanna, "Scalable and modular algorithms for floating-point matrix multiplication on FPGAs," in *Proc. of Int. Conf. on Parallel and Distributed Processing Symposium*, Los Angeles, USA, Apr. 2004, p. 92.

[23] B. Lee and K. Lever, "Logarithmic number system and floating-point implementations of a well-conditioned rls estimation algorithm on FPGA," in *Proc. of Conf. on Signals, Systems and Computers*, Cardiff, UK, Nov. 2003, vol. 1, pp. 109–113.

[24] R. Maheshwari, S.S.S.P. Rao, and E.G. Poonach, "FPGA implementation of median filter," *Proc. IEEE Tenth Int. Conf. VLSI Design, VLSI Mult. Apps.*, p. 523, Jan. 1997.

[25] H.H. Seward, "Information sorting in the application of electronic digital computers to business operations," M.S. thesis, Massachusetts Institute of Technology (MIT), 1954.

[26] Edward H. Friend, "Sorting on electronic computer systems.," *Association for Computing Machinery Journal (ACM)*, vol. 3, no. 3, pp. 134–168, 1956.

[27] K. Ratnayake, "An FPGA-oriented method for large-scale sorting," Tech. Rep. vidpro-TR-10-04, VidPro group, ECE, Concordia University, Montréal, Québec, Canada, Oct. 2004.

[28] V. Zabrodsky, "Album of algorithms and techniques, counting sort," Standard Rexx On-line, www.geocities.com/SiliconValley/Garage/3323/aat/a_coun.html, 2001.

[29] A. S. Kagel, "Counting sort," National Institute of Standards and Technology On-line, www.nist.gov/dads/HTML/countingsort.html, 2004.

[30] John P. Hayes, *Computer architecture and organization; (2nd ed.)*, McGraw-Hill, Inc., New York, USA, 1988.

[31] MIT, "Computation structures (course 6.004) : Handout on pipelining," Course Notes On-line, http://6004.csail.mit.edu/currentsemester/handouts/L09-1up.pdf, 2005.

[32] B. Parhami, *Computer Arithmetic, Algorithms and Hardware Design*, Oxford University Press, Inc, 2000.

[33] M. Lu, *Arithmetic and Logic in Computer Systems*, John Wiley, Inc., Publication, 2004.

Appendix A

Appendix

A.1 FPGA Implementation Tools Used

In this section the design tools utilized to compile, simulate and implement this work are summarized in the Table A.1:

Tool name	Tool version	Description
Coregen^{TM}	5.2i	Core generation tool
Synopsys VHDL Analyzer^{TM}	2001.09	VHDL code compiler
Synopsys VHDL Simulator^{TM}	2001.09	VHDL behavioral simulation tool
Synoposys Design Compiler^{TM}	V-2004.06-SP1	VHDL code synthesis tool
Xilinx Project Navigator^{TM}	5.2i	Place & route and Gate level tool

Table A.1: Input video signal specifications

A.2 Acronyms

ASIC	Application-specific integrated circuits
CCIR	International Radio Consultative Committee
CLB	Configurable logic blocks
DCM	Digital clock manager
DMA	Direct memory access
DSP	Digital signal processing
EDIF	Electronic design interchange format
FIR	Finite impulse response
FPGA	Field-programmable gate array
HDTV	High definition television
IC	Integrated circuit
IOB	Input output blocks
ITU	International Telecommunication Union
LUT	Look-up tables
PAL	Phase alternation by line
PCB	Printed circuit board
PSNR	Peak signal to noise ratio
RAM	Random access memory
RTL	Resistor transistor level
VHDL	VHSIC hardware description language
VHSIC	Very high speed integrated circuit

A.3 Symbols

$S(n)$	Original signal
$\eta(n)$	Added noise signal
$I(i,j,n)$	Pixel intensity at position (i,j) in time t
B_h	Blocks of a segmented image
ξ_{B_h}	Block's Homogeneous Measure
μ_{Bh}	Block's Sample Mean
$\sigma^2_{B_h}$	Block's Variance
σ^2_n	Final Variance
t_σ	Homogeneity threshold
$\theta(n)$	Equation Complexity