12-17-2010

Implementation of Directional Median Filtering using Field Programmable Gate Arrays

Madhuri Gundam
University of New Orleans

Implementation of Directional Median Filtering using Field Programmable Gate Arrays

A Thesis

Submitted to the Graduate Faculty of the
University of New Orleans
in partial fulfillment of the
requirements of the degree of

Master of Science
in
Engineering
Electrical

By

Madhuri Gundam

B.Tech. Jawaharlal Nehru Technological University, 2007

December, 2010

Acknowledgements

I would like to thank my advisor Dr. Dimitrios Charalampidis for his invaluable guidance and motivation throughout this work.

I would also like to thank Dr. Rasheed M A Azzam and Dr. Vesselin P Jilkov for serving on my thesis defense committee.

A special thanks to my family and friends for their unconditional love and support.

Table of Contents

List of Figures

List of Tables

Abstract

Median filtering is a non-linear filtering technique which is effective in removing impulsive noise from data. In this thesis, directional median filtering has been implemented using cumulative histogram of samples in several directions. Different methods to implement directional median filtering have been proposed. The filtered images are smoothed along the direction of the filtering window. All implementations aimed to generate outputs in the least amount of time, while reducing the resource utilization on hardware. The implementation methods were designed for Xilinx Virtex 5 FPGA devices but were also attempted on Spartan 3E. The proposed methods used less than 30% of the resources on Virtex 5 FPGA but the resource utilization on Spartan 3E exceeded the number of available resources. After an initial delay, methods 1 and 2 generate a new output for every 5 clock cycles while method 3 generates an output for every 1.5 clock cycles.

Keywords: Histogram, Median, Sorting Network, Directional Median Filtering, FPGA, CLB, LUT, Spartan 3E, Virtex5.

Chapter 1

Introduction

1.1 Introduction to FPGAs

Programmable Logic Devices (PLDs) are semiconductor devices that can be configured by the end user. In other words, their functionality is not defined during manufacturing, and can be programmed to perform a large number of different functions [1].

Field Programmable Gate Array (FPGA) is a PLD that uses logic cells, which are made up of basic gates. FPGAs are capable of implementing any logic function as long as the resources required by the function are available. A design is implemented on FPGAs by making the connections between the logic cells. Fig. 1.1 represents the internal structure of an FPGA with logic blocks, interconnections, and input-output blocks. Nowadays other components such as the block memory, Digital Clock Managers (DCMs), and dedicated modules for some arithmetic operations are also present on the chip. It can be observed from fig. 1.1 that there are provisions for connecting different blocks on the chip. The end user can decide which connections to use.

The logic cells consist of flip-flops, multiplexers, arithmetic and carry logic, and function generators [2]. Although logic cells are used to implement the logic, they can also be used as distributed memory. Of course, this will reduce the number of cells available for implementing the desired circuit.

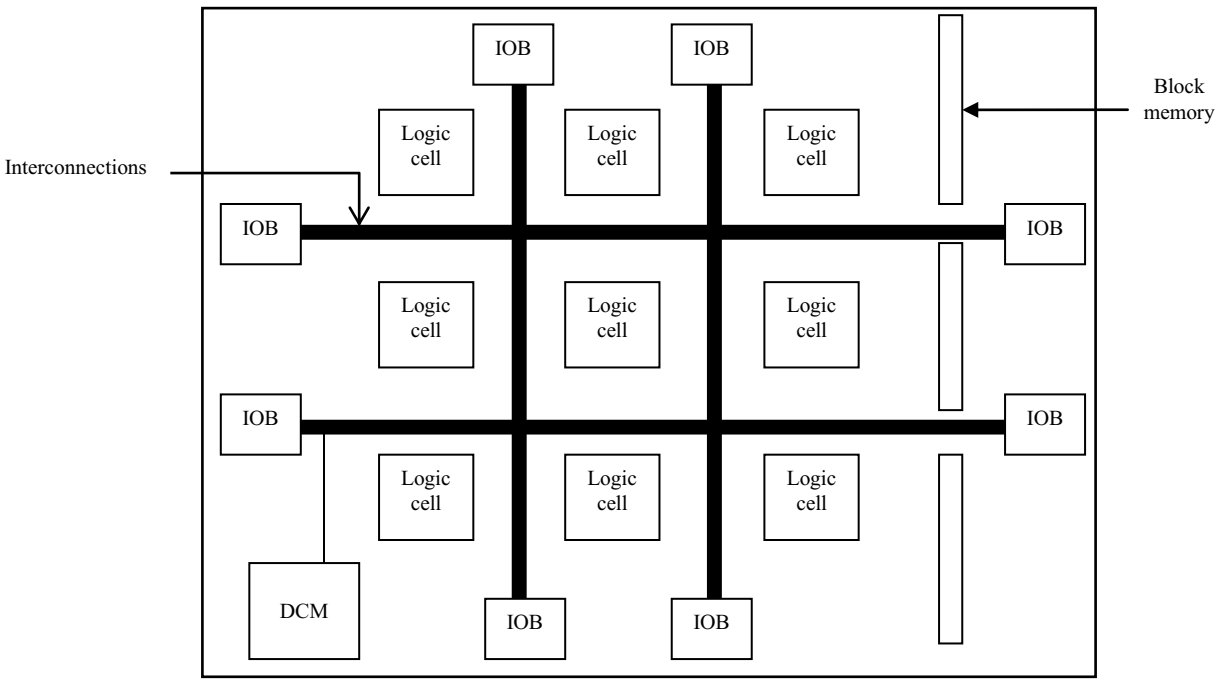


Fig. 1.1 Internal structure of FPGA

Unlike Digital Signal Processors (DSPs), which are mostly sequential devices, FPGAs are highly parallel devices - they are capable of performing multiple tasks at same time in parallel. Since DSPs are sequential devices, they require more time to perform an operation that can be carried out faster on FPGAs by exploiting their parallelism. Usually, image processing applications require the same operation to be performed repeatedly on every pixel. In such cases, FPGAs can perform the operations efficiently [3].

As the name suggests, FPGA devices are field programmable, i.e., they can be configured by the end user and are re-configurable. The re-programmable and field programmable abilities provide FPGAs with an edge over Application Specific Integrated Circuits (ASIC). Moreover, FPGA design process is a less time consuming process compared to ASIC design [1], [4].

Therefore, once an application has been designed for and verified on an FPGA, it can be migrated to an ASIC.

The rest of the thesis is organized as follows:

Chapter 2 describes median filters, which are non-linear filters used to remove impulsive noise from data. This chapter also briefly describes the methods commonly employed to implement median filtering on hardware.

Hardware description languages (HDLs), such as Very High Speed Integrated Circuit HDL (VHDL) and Verilog, are commonly high level programming languages used to describe the functionality of the circuits. After describing the circuit using HDL, the designs are simulated and synthesized. A more detailed explanation of the design steps are provided in chapter 3.VHDL and Xilinx ISE 10.1 Design Suite have been used in this thesis.

Chapter 4 gives a description of the previous work on histogram based median filtering. This method is appealing because the histogram calculation does not depends on the size of the window and depends only the gray level intensities in an image. This chapter also describes about how this method can be extended to 2D filtering.

Chapter 5 describes different implementations of cumulative histogram-based directional median filtering on FPGAs. Those implementations aimed to reduce the utilization of FPGA resources and generate outputs in as few clock cycles as possible. The implementation results are provided in chapter 6. The detailed reports on implementation are provided at the end of the chapter. Finally, chapter 7 provides some conclusions drawn from this thesis work and discusses future implementations.

Chapter 2

Median Filter

Real-time signal processing and image processing applications employ filtering to process and to manipulate the signals, or to remove noise from data. Median filter is a non-linear filter used for removing impulsive noise from data. This chapter provides a description of the median filter and median filtering techniques implemented on the hardware devices.

2.1 Median Filter

There are two types of filters: linear filters and non-linear filters. The median filter is a non-linear filter; it is a special case of rank order filters whose rank is half the length of the sequence. In image processing applications, median filter is used to remove impulsive noise from images while preserving the edges [5], [6].

One of the disadvantages of linear filters, such as the moving average filter, when used to denoise the data, is that they not only smooth the noise, but also smooth the sudden and sharp transitions that were present in the original data, such as edges in images. Moreover, they are not as efficient as the median filters in removing certain types of noise, such as impulsive noise [5]. Although median filters do not blur the edges as much as the linear filters do, because they still possess smoothing characteristics, as the size of the filter increases, there may be significant image blurring [7]. Impulsive noise can be classified into two types: (1) salt and pepper noise (2) random valued noise. Salt and pepper noise pixels take only two values, either the minimum or the maximum possible value, for example, in a gray scale image, salt and pepper noise pixels will be either 0 or 255. However, random valued noise pixels take any random value [8], which

is more difficult to remove than the salt and pepper noise. If p and q are the probabilities of occurrences of 255 and 0, where $0 \leq p, q \leq 1$ and p and q can be equal or different, a pixel may be replaced by 255 with a probability p and by 0 with a probability q.

The median of a given sequence is given by sorting the sequence and choosing the middle value from the sorted sequence. If there are odd number of elements in a sequence, then the median is the middle element in the sorted list. If there are even number of elements then the median is given by the arithmetic mean of the two middle elements in the sorted sequence. In image processing, the 2D filtering operation is performed by sliding the window along all the rows and columns of the image until all the pixels are covered by the window. The filtering is done by sliding the window across the image, sorting all the pixels in the window, which consists of a center pixel and the neighborhood pixels, and then replacing the central pixel with the median intensity of the window. Since salt and pepper noise pixels take only either the maximum or the minimum possible value and the result of a median filter excludes the extreme value, median filtering provides a good reduction of the salt and pepper noise.

2.2 Median Filtering Methods

Various algorithms have been developed to implement the median filtering on hardware. Median filtering techniques are usually based on the sorting network architectures; another approach to implement the median filter is based on the histogram [9].

2.2.1 Sorting network based median filtering

Sorting network architectures may depend on bubble sorting, quick sorting, and insertion sorting to implement the median filter on FPGAs [5], [6], [10] - [12]. These sorting networks use

compare and delay units to implement the median filter. The incoming pixels are passed through a network of comparators and swapping units - the comparator units compare two to three incoming pixels at once and then the swapping unit sorts them accordingly. The median value will be the middle value of the sorting network.

To implement a *3x3* median filter using bubble sort, *41* compare-and-swap units are required and as the size of the window increases, the number of compare-and-swap units required for implementation will also increase. Some optimizations may lead to the reduction in the number of these units required as presented in [10]. The resources required to implement the sorting network architecture on a FPGA device increases with the size of the filtering window. However, the sorting network based algorithms are independent of the size of the image and depend only on the size of the window.

2.2.2 Histogram based median filtering

Histogram is a representation of the distribution of the intensities in an image, i.e., it shows how many pixels in an image take a particular intensity value. The histogram is calculated by incrementing the value of the bin representing the corresponding intensity level. Every time a particular intensity is encountered, the value of the corresponding bin is increased by one.

Similar to any software implementation, the hardware implementation of the histogram requires as many counters as the expected number of intensity levels in an image. Each counter is associated to an intensity level and the values of these counters are incremented according to the incoming pixel intensities.

The cumulative histogram is calculated by increasing the values of all the bins greater than or equal to the incoming pixel intensity. Then, after building the cumulative histogram, the

first bin which has a value greater than or equal to the median index is taken as the median value of the window [9]. For example, the median of 3x3 window using cumulative histogram method is found as follows: 156, 89, 75, 190, 204, 89, 89, 75, 255. All the bins greater than and equal to the these intensities are increased by one every time the number is encountered, i.e., all the bins above 155^{th} bin are increased by one, when the value 89 comes-in the values of all the bins above 88^{th} bin are increased by 1(they are incremented 2 more times in this sequence) and so on for the rest of the sequence. The median index of a 3x3 window is 5, so counting the bin values to find the first bin which has the value equal to the median index gives 89 as the median value of the given sequence.

The FPGA resource utilization for the histogram based median filtering depends on the size of the window. The reason is that the size of the counters depends on the size of the window and the number of expected gray level intensities in an image - since the number of bins to be instantiated depends on the number of gray levels.

Chapter 3

Hardware and Software

This chapter describes about the software design implementation steps in the Xilinx ISE 11.1 Design Suite and some of the features of the Spartan 3E Starter board and Virtex-5 family chips.

3.1 Xilinx FPGA Features

In an FPGA, a design is implemented by making connections between the logic cells present in Configurable Logic Blocks (CLBs). CLBs are the basic logic units of FPGA in which logic is implemented. Xilinx FPGA CLBs have two slices per CLB: SLICEL and SLICEM. A slice consists of function generators or Look-up tables (LUTs), storage elements, arithmetic and carry logic and multiplexers [13]. LUTs present in the slices implement the boolean functions. Spartan family FPGAs are made of 4-input LUTs while the Virtex family devices consist of 6-input LUTs. The advantage of 6-input LUTs over 4-input LUTs is that the same design can be implemented using less resources on an FPGA.

Two types of on-chip memory are available for the user: distributed memory and block memory (block RAM or BRAM). In particular, LUTs can be configured as distributed memory. The usage of distributed memory reduces the data access time, but it also reduces the number of LUTs that will be available for design implementation. Hence, block RAM provides a better alternative. Block RAM is the embedded memory located along the columns of the FPGA chip.

Block RAM can be used as either single port or dual port memory. When used as dual port memory, each port gives an output on different clock edges, i.e., port A gives an output on

the rising edge of the clock and port B on the falling edge of the clock. Hence, in a clock period, the block memory can be read twice using dual port configuration. The block memory introduces a clock cycle delay, i.e., the BRAM output will be available in the next clock cycle after the input has been given to it. However, the block RAM can be accessed at much higher clock speeds than the on-board clock rate using Digital Clock Manager (DCM) – DCMs allow for clock multiplying, dividing, phase-shifting, and distributing operations to be performed easily. Moreover, BRAMs can be configured as quad port blocks, which let the RAM to be read 4 times in a clock cycle [14]. BRAMs can also be used as shift registers, state machines and counters [15]. For this work, the block memory has been used to store the input image and the 256-bit code that is used to update the histogram.

The Spartan family XC3S500E FPGA has 20 dual port RAM blocks that are arranged in two rows with each block providing 18Kb of memory [16]. Virtex family XC5VLX85T has 108 RAM blocks with each block providing 36Kb of single port memory block [13]. The Spartan3E FPGA has 9,312 4-input LUTs whereas the Virtex5 FPGA has 51,840 6-input LUTs. The Spartan3E FPGA has 74,496Kb of distributed RAM while Virtex5 FPGA has 840Kb of distributed RAM.

3.2 Spartan 3E Starter Board

The hardware device used for this work is the Spartan 3E Starter Board [17]. The Spartan 3E Starter board is mounted with a XC3S500E FPGA chip, 64MB external RAM, 4MB PROM, serial flash memory, 50 MHz oscillator, CoolRunner-II CPLD, A/D and D/A converters, PS/2 mouse and Keyboard port, VGA display port, RS-323 port, Ethernet interface, 2 line 16-

character LCD display, eight LEDs, four switches, four push buttons and expansion connectors that allow other boards to be connected to it. Spartan 3E starter board is shown in fig. 3.1.

Fig. 3.1 Spartan 3E Starter Board

The XC3S500E FPGA chip has 1,164 configurable logic blocks (CLB). Each CLB consists of four slices; each slice has two 4-input Look-up tables (LUTs) and two storage elements. As mentioned before, the logic is actually implemented in CLBs of the XC3S500E FPGA chip.

To display the images on CRT and LCD monitors, the VGA port of the starter board can be connected to the monitor using the standard monitor cable. The VGA port has five signals – three of which are the video signals and remaining two are the synchronization signals. Each pixel is represented using three bits, hence this board can display only eight colors on the monitor at 25 MHz pixel rate; while the clock oscillator of the Spartan 3E starter board runs at a

frequency of 50 MHz. The VGA port (enclosed in the red colour box) of Spartan 3E Starter Board is shown in the fig. 3.2.

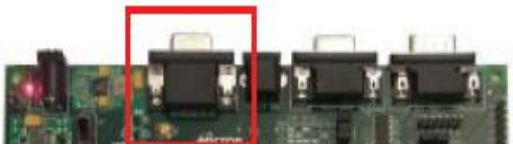


Fig. 3.2 VGA port of the Spartan 3E Starter Board

3.3 Software Design Flow

The free version of the Xilinx design suite, Integrated Software Environment 11.1, has been used to implement the design in software. The design should be created, tested and verified in the software before the hardware can be configured [18], [19].

The first step in the design flow is the HDL description of the circuit. In this step, the design files are created using one of the hardware description languages. For this thesis work, VHDL was the language used. These source files can be simulated to verify the functionality of the design in software. However, successful behavioral simulation does not guarantee successful implementation on the hardware.

The next step is to synthesize the design files that were created in the previous step. During this step, the software checks syntax errors, applies user constraints and optimizes the logic to the target device. The output files from this step will be used in the next step.

The third step is the implementation step. During this step, the software verifies whether the design can be implemented on the hardware, for example, it checks how the design will be routed on the chip and optimizes the design according to the timing specifications. The design suite provides tools such as the Floorplan editor and FPGA editor that let the designer to create

constraints, and see how the design will be placed and routed on the FPGA, and let the designer perform placing and routing manually. The software generates detailed analysis reports about the implementation.

The final step in the software design is to generate the programming file to be used configure the FPGA. The programming file thus generated is then downloaded onto the FPGA through JTAG cable. Fig. 3.3 depicts the design flow in the Xilinx ISE.

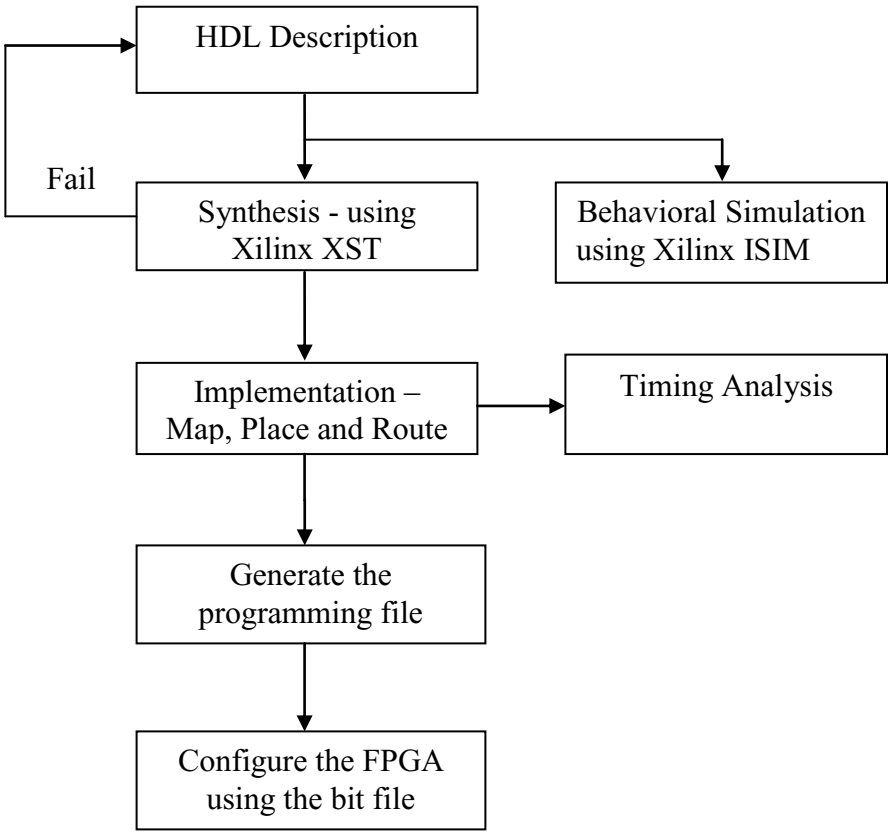


Fig. 3.3 Design Flow

Chapter 4

Cumulative Histogram Based Median Filtering

This chapter gives a brief description of the histogram based median filtering on FPGA as proposed by Fahmy et. al. in [9]. In this thesis, this concept has been extended to directional median filters.

4.1 Cumulative Histogram Based Median Filtering on FPGA

As mentioned before, histogram gives a distribution of intensity levels in an image, and hence, requires as many bins as the expected number of gray levels in the image. On an FPGA, the histogram is implemented by instantiating an array of registers, called as bin nodes, representing all the possible gray level intensities in an input image, i.e., if each pixel in the image is represented by 8-bits then 256 bins are required. A bin node is essentially a counter that keeps track of the number of times the bin is incremented. Each node consists of a register, an incrementer, and a comparator. Each node has two inputs: an enable input and a median index input, and an output, which is the comparator output. The register stores the value of the bin after every increment operation, while the comparator checks whether the register value is equal to or greater than the median index input provided. The enable input to bin node determines whether the register value has to be incremented or not. A bin value is incremented whenever the enable input is '1' otherwise it is not.

As mentioned in section 2.2.2, for cumulative histogram, all the successive bins with index greater than or equal to the incoming intensity are incremented. Since all the consecutive bins have to be incremented, the pattern in which the bins should be incremented is stored in

block memory that is accessed by input samples. The input sample acts as address input to block RAM, whose 256-bit output is then used as the enable input of the bin nodes. Each bit of the block RAM output is the enable input to only one bin node. Hence, the histogram is updated by incrementing the bins according to the 256-bit code. And then the comparator in each bin compares the value of the register with the median index and outputs a '1' if the register value is equal to or greater than the median index otherwise outputs a '0'. Finally, a priority encoder is used to find the median value, which is the index of the first bin whose output is 1.

The architecture [9] for the histogram based median filtering for a fixed window is shown in fig. 4.1.

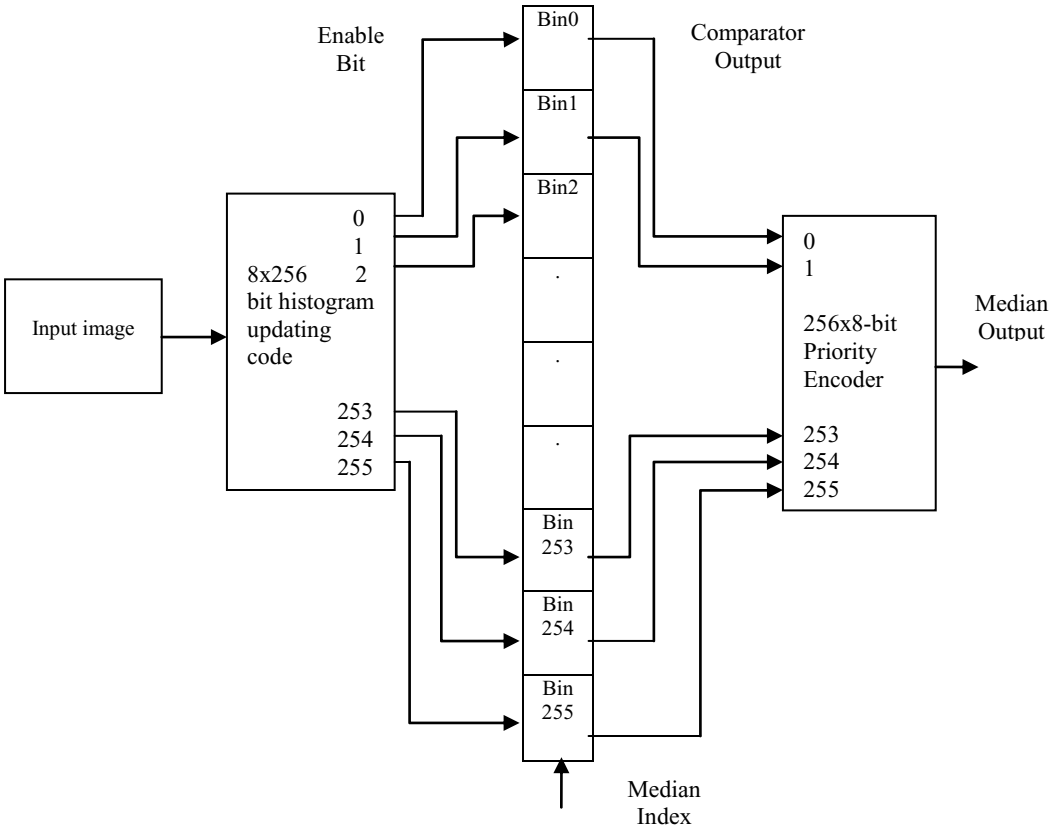


Fig. 4.1 Cumulative Histogram based Median Filter Architecture for fixed window

4.2 Cumulative Histogram for Sliding Windows

In real-time image processing applications, filtering operation is performed by sliding the filtering window over the image pixel by pixel. Thus, as the window slides over the image, a few new pixels are added while a few old pixels are removed but most of the pixels in the window are same. As some of the samples fall out of the window, the contribution of those samples on the histogram is cancelled by decrementing the values of the corresponding bins by 1. At the same time, the contribution of the new samples on the histogram is added by incrementing the values of the corresponding bins by 1. The values of the bins corresponding to the samples that do not change are not affected. This is done by using a 2-bit enable input to the bin nodes instead of 1-bit of the 2-bits of the enable input. The MSB of enable input corresponds to new sample while the LSB corresponds to old sample. Therefore, a bin is incremented if enable input equals "10", decremented if enable equals "01" and is left unchanged if the enable equals "11" or "00". If enable input equals "11", it implies that a particular intensity is present in both old and new samples. If enable input equals "00", it means that a particular intensity is present neither in new sample nor in old sample list, and hence, has no contribution to the histogram. The steps to update a bin node for the sliding window implementation are shown in fig. 4.2.

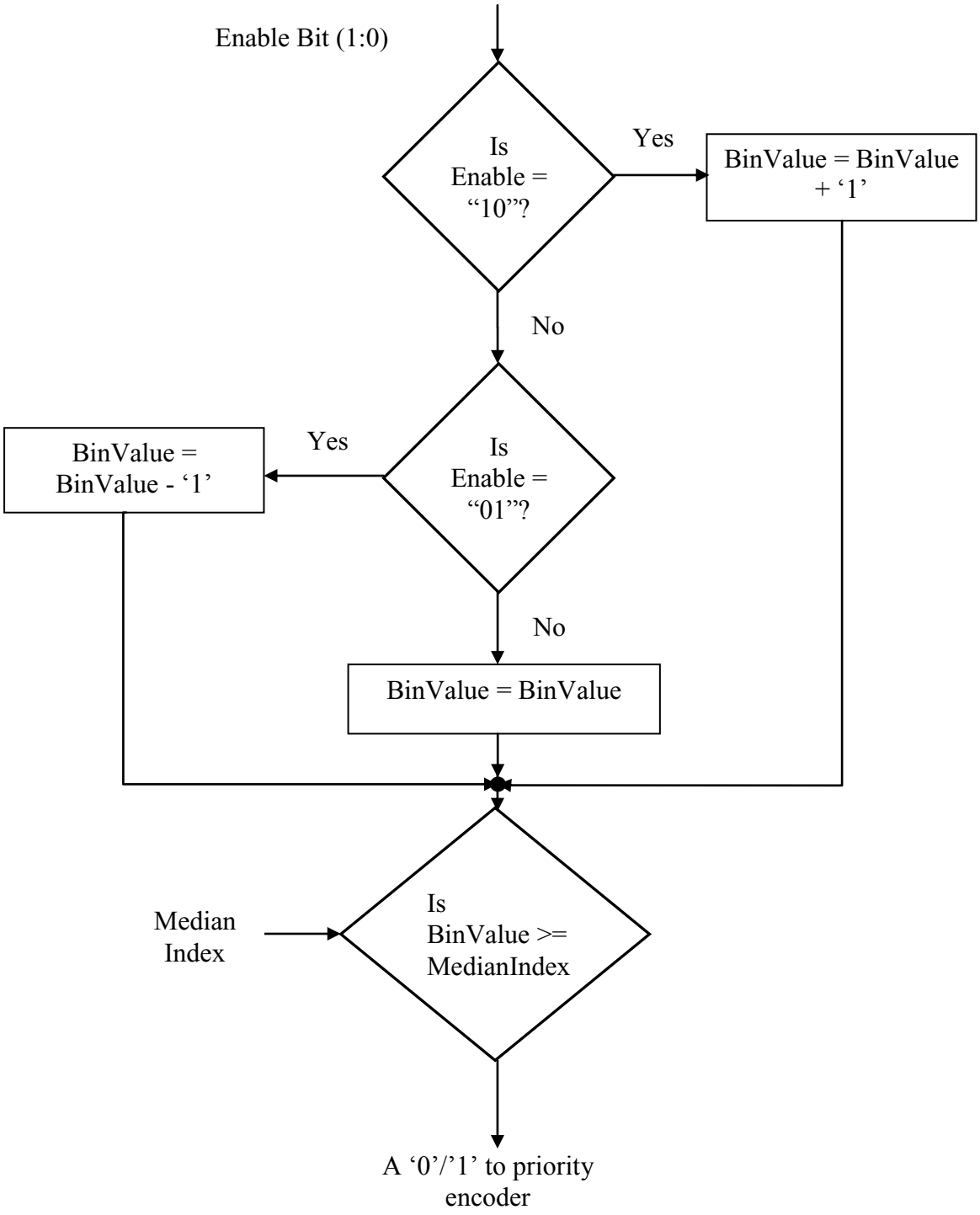


Fig. 4.2 Flow chart representing the steps to update bin nodes

4.3 Histogram Updating Codes

The histogram is updated by incrementing or decrementing the bins as the samples enter or fall out of the window in every clock cycle. All the bins are updated during the same clock cycle according to the pattern shown in Table I.

Table I. Bin node enable codes

Gray level (address input to BRAM)	Pattern to update the histogram [255:0]
0	11111111111……11111111
1	11111111111……11111110
2	11111111111……11111100
3	11111111111……11111000
4	11111111111……11110000
.	.
.	.
.	.
253	11100000000……00000000
254	11000000000……00000000
255	10000000000……00000000

It can be observed from the table I that if the input intensity equals 0, then the enable input to all bins is ‘1’, i.e., all the bins will be incremented by 1. If the intensity equals 3 then the output of BRAM from the 0^{th} bit to the 2^{nd} bit is ‘0’ and the remaining bits are ‘1’s, in other words, bins 0 to 2 are not incremented while bins 3 to 255 will be incremented by 1.

Chapter 5

Directional Median Filter

This chapter provides a description of the four directional median filter implementations. The cumulative histogram based median filtering concept that was presented in the chapter 4 has been extended to directional median implementations in this thesis. These implementations aimed to minimize the resource utilization and generate the median result in the least number of clock cycles as possible.

5.1 Implementations

A 5x5 window has been considered for all four implementations. The directional median filtering is performed along the 4 directions D1, D2, D3 and D4 shown in fig. 5.1.

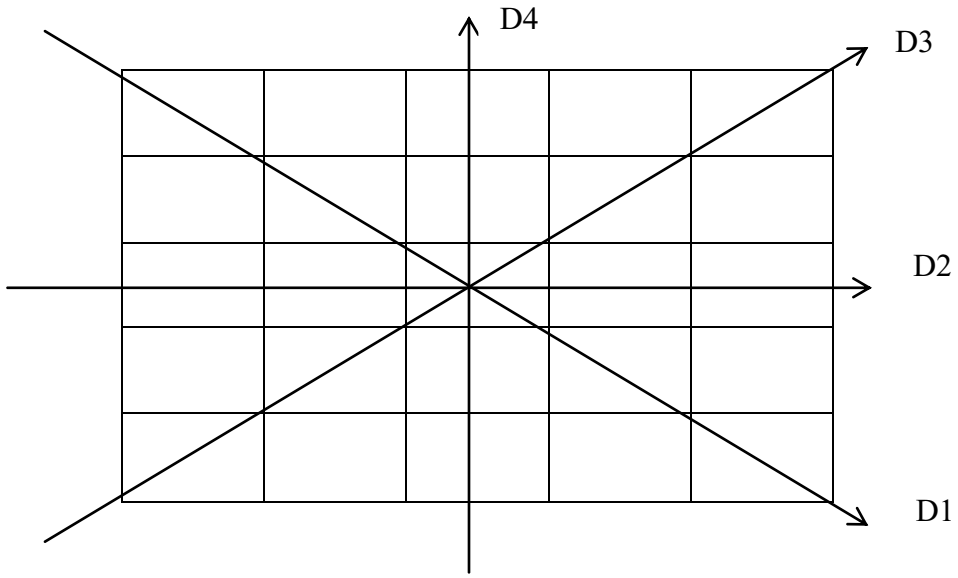


Fig. 5.1 Four directions considered for directional median filtering implementations

The pixels of original images are stored row-wise in BRAM. As the window slides to the right, the order in which the current window samples and the next window samples are read from memory is shown in fig. 5.2. Using BRAM in dual port configuration, 2 pixels in a column are read in a clock cycle.

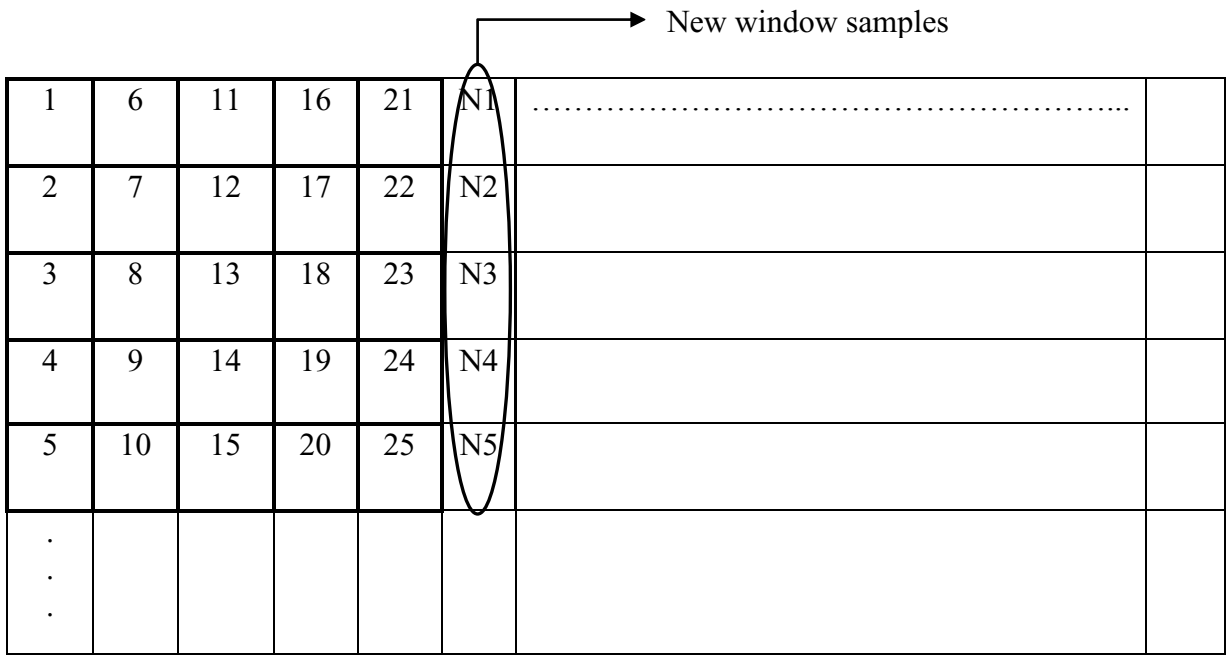


Fig. 5.2 Reading image from block memory

As the samples are read from BRAM, they are buffered until all samples associated to the current window are read. Hence, only the pixels to be processed are present in the buffers. As a new sample is being buffered, the oldest one is removed from the buffers as shown in fig. 5.3. N1, N2, N3, N4 and N5 in fig. 5.2 and fig. 5.3 represent samples associated to the next window.

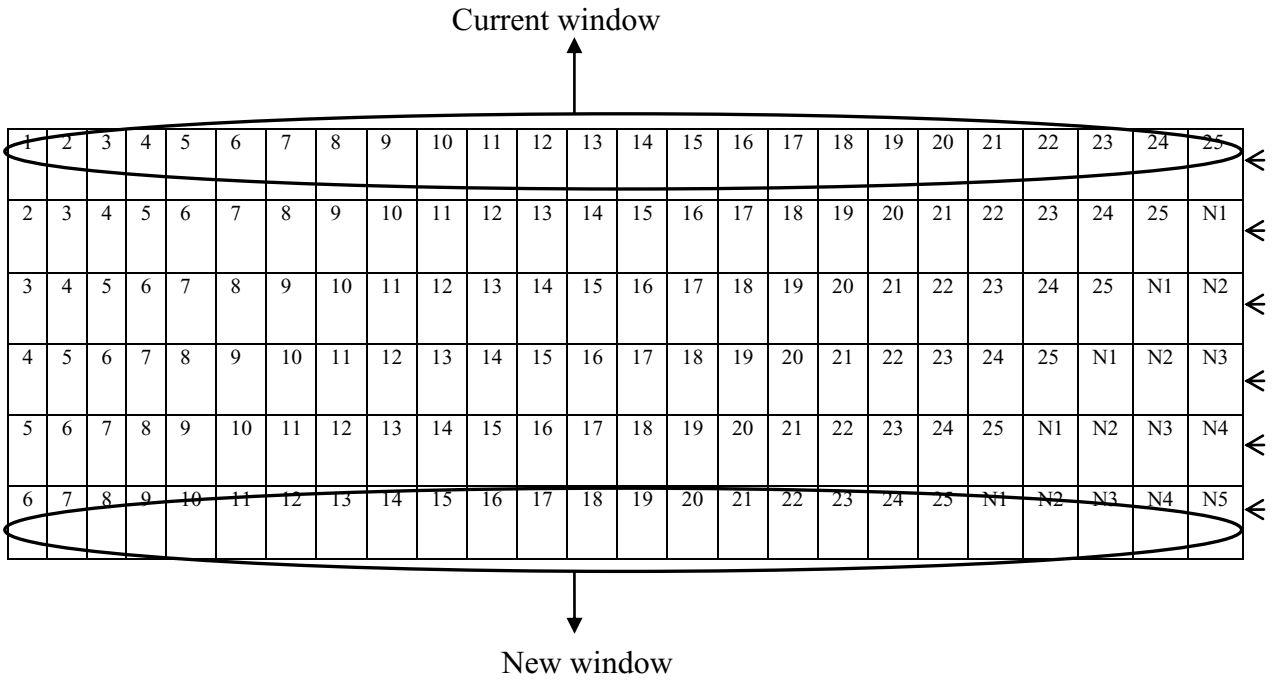


Fig. 5.3 Buffering the input samples

5.1.1 Method 1: By saving only directional codes

As the window slides along the image, for a 5x5 window, 5 new samples enter the window and 5 old samples fall out of the window. The image and the 256-bit codes required to update the histogram are stored in the embedded block memory. This introduces a delay of 2 clock cycles – 1 clock period to fetch the sample and 1 to fetch the access pattern to update the histogram.

For the first implementation, all the samples in the window are saved. Since only one new sample is read from the memory per clock cycle, it takes 5 cycles to update the window with new samples after an initial delay. Nevertheless, the 256-bit codes corresponding to the 5 samples in a particular direction are accessed from block RAM for every new window. As a result, for every window, 20 directional codes should be accessed in 5 clock cycles. This can be

performed by configuring the block RAM that stores the 256x256 codes as quad port memory module running at double the rate at which the window is updated. Therefore, four new histograms corresponding to four directions are built for every new window. This implementation requires 5 clock cycles to produce the new median output.

Fig. 5.4 illustrates the order in which directional codes are read from the memory. The numbers in the cells correspond to n^{th} sample in the direction. The codes corresponding to samples with same number are read in 1 clock cycle. For example, the codes of all the samples represented by 2 are read in 1 clock cycle.

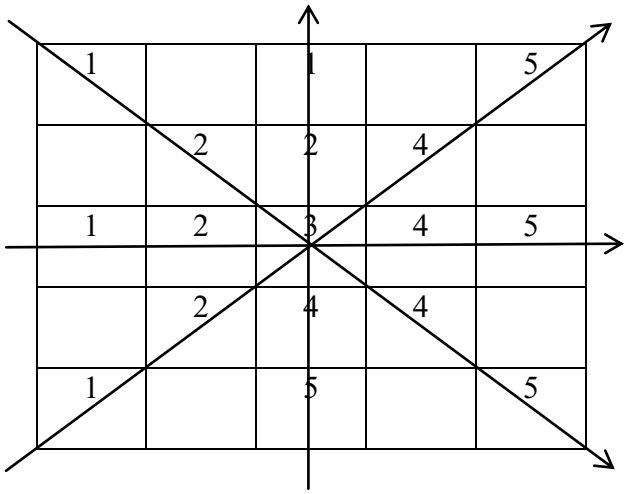


Fig. 5.4 Order in which the codes are accessed from block memory

The directional histogram is updated only after all the 5 codes corresponding to the 5 samples in the direction are accessed from memory.

As mentioned in chapter 4, each bit of the 256-bit access code corresponds to only one bin out of the 256 bin nodes. After all the 5 codes are obtained from memory, the corresponding bits are added and the result of this addition is used to update the bins, i.e., the 0^{th} bits of all 5 codes in a direction are added and the result is used to update the 0^{th} bin node and so on for the

remaining bins. Finally, the comparator compares the bin value with the median index to produce the median output. This process is illustrated in fig. 5.5 and the bin node is shown in the fig. 5.6.

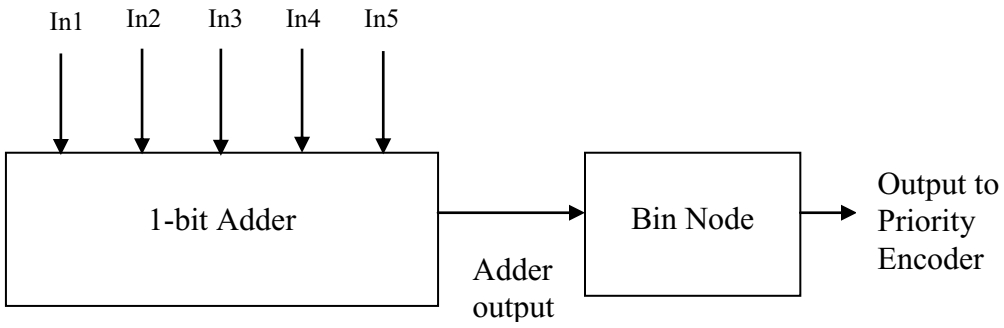


Fig. 5.5 Updating bin node for Method 1

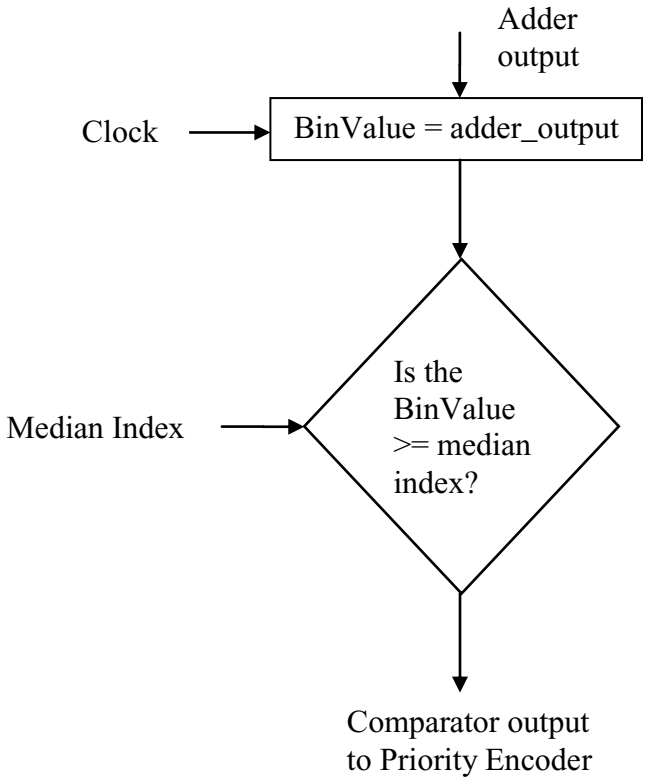


Fig. 5.6 Flow chart for method 1

Sample 1 Code	255^{th} bit	254^{th} bit	……	3^{rd} bit	2^{nd} bit	1^{st} bit	0^{th} bit
Sample 2 Code	255	254	……	3	2	1	0
Sample 3 Code	255	254	……	3	2	1	0
Sample 4 Code	255	254	……	3	2	1	0
Sample 5 Code	255	254	……	3	2	1	0
Addition Result	To 255^{th} bin	To 254^{th} bin	……	To 3^{rd} bin	To 2^{nd} bin	To 1^{st} bin	To 0^{th} bin

Fig. 5.7 Addition of corresponding bits of codes in a direction

The process has to wait until it gets all the 5 inputs of the adder and buffer the codes of first 4 samples of the direction before the codes can be added. This implementation requires 256 parallel adders to be instantiated to perform addition for all bins in parallel.

5.1.2 Method 2: Without saving directional codes

The second implementation is similar to the first method except that in the second method the 256-bit codes are not saved. The codes, corresponding to the respective samples in four directions, are read in a similar manner as described in the section 5.1.1, i.e., by using block RAM in quad port configuration and reading the codes of the samples represented with same number in fig. 5.4 in 1 clock cycle. This implementation also requires 5 clock cycles to produce a new median output.

In this case, the histogram is updated in a slightly different manner. For the first sample in any direction, bin nodes are updated by assigning the enable bit of that sample to bin register

and for the remaining 4 samples in a direction, bins are updated according to enable inputs of samples.

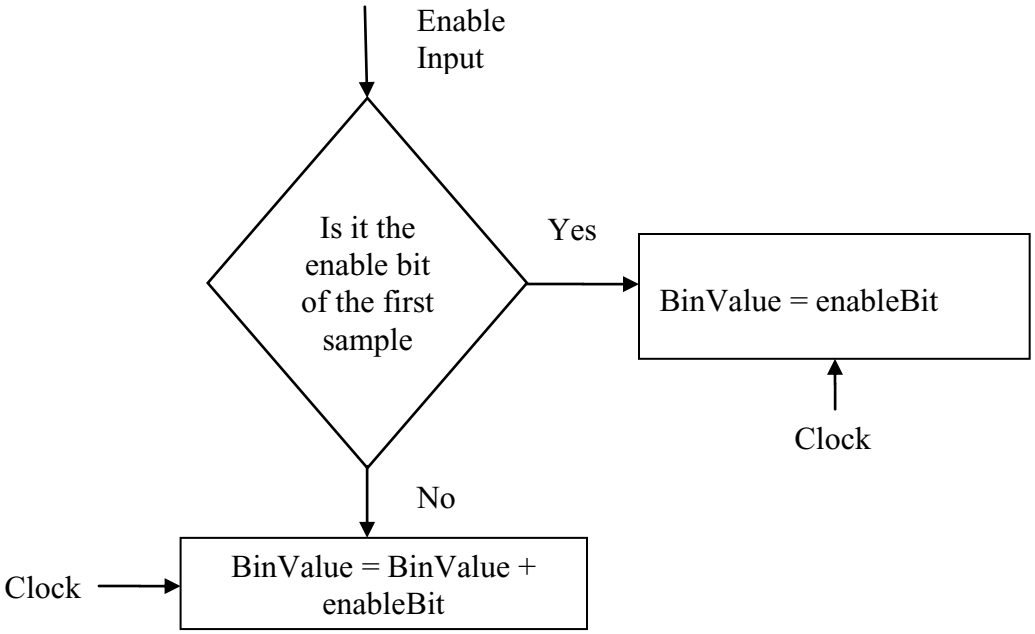


Fig. 5.8 Flow chart for second method

This implementation could also be performed by updating the histogram as the enable bits are available and then by resetting the histogram after the median has been found. Nevertheless, this requires an extra clock cycle to reset the histogram.

This algorithm would reduce the resource utilization on the FPGA as there is no need to save the 256-bit directional codes and there is no need to instantiate the 256 5-bit parallel adders. The details of the resource utilization are provided in the next chapter.

5.1.3 Method 3: By saving all codes of the window

In the first two implementations, the samples in the window had to be saved. In the third implementation, 25 codes corresponding to the 25 samples in the window are saved while the samples are not saved. Although this requires 25 256-bit registers for saving the codes, it drastically reduces the number of clock cycles needed to generate the new output. In this method, a new median output is produced every 1.5 clock cycles.

In order to generate a new output every 1.5 clock cycles, the rate at which the samples are read from the memory should be equal to the rate at which the codes are read. Hence, the block memory that stores the image has to be configured as quad port along with the block memory that stores the 256x256 codes. This is necessary because if the two rates are not equal then a few rows/columns of the image will have to be buffered before the processing could begin. Otherwise, the process reads the codes and waits for the samples to be read.

Since the codes that are required to calculate the median for a window are already buffered, the corresponding enable bits can be added, as shown in fig. 5.5, before updating the bin register.

The schematic for this implementation is the same as in the first method. In other words, the third method would also require 256 parallel adders, which will increase the resource utilization.

5.1.4 Method 4: By sliding windows in 4 directions

This method has not been implemented in this thesis but the basic idea is presented here, and will be considered in future work. Method 4 is different from the methods 1, 2 and 3 as four sliding windows of sizes 1x5, one along each direction, are considered in method 4.

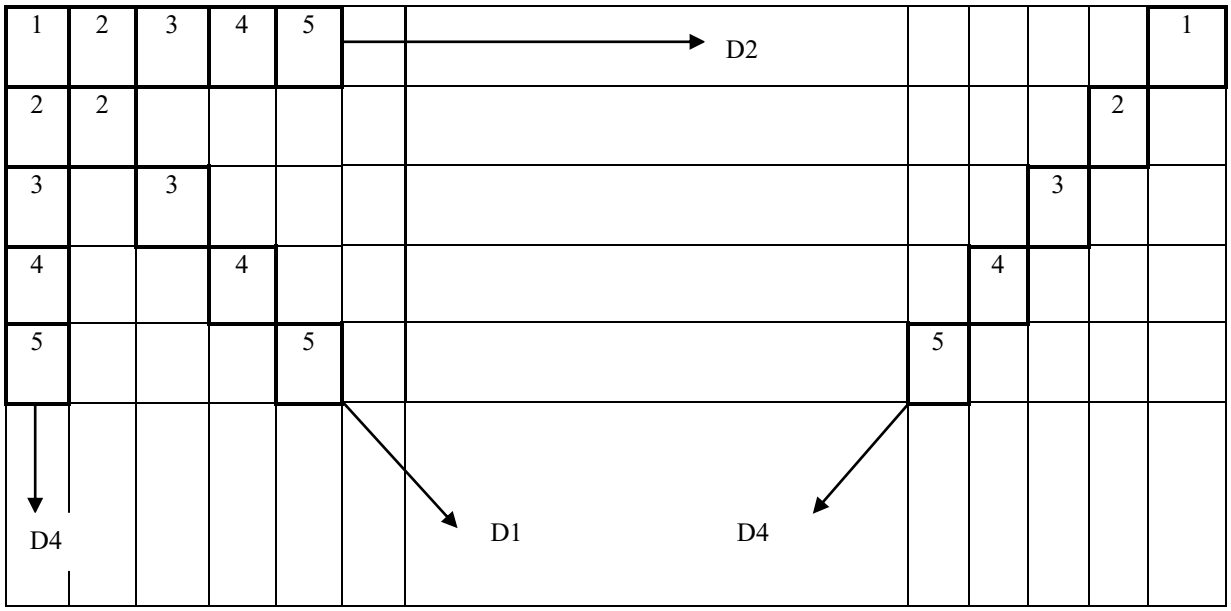


Fig. 5.9 Windows for method 4

The order in which the samples are accessed from memory for this method is shown in fig. 5.9. The four directional windows slide along the directions D1, D2, D3 and D4 and only the current window samples corresponding to the four are read from memory.

As the samples are read from memory, the corresponding codes are read simultaneously eliminating the need to buffer the samples. However, the bin enable codes have to be saved. For the first window in any direction, the histogram can be built by incrementing the bins according to the intensity levels. As a window slides along its direction, only one new sample and corresponding enable bit code are read from BRAM. The histogram can be updated by cancelling the effect of the oldest sample from the histogram and adding the effect of newest sample in the window as presented in section 4.2. This process requires only 1 clock cycle to update the histogram and generate the median output. Therefore after an initial delay of 7 clock

cycles required to generate the first median output, the new median outputs are expected to be generated every clock cycle.

In methods 1, 2 and 3, all four medians, in 4 directions, of a particular sample are produced at the same time whereas in method 4, all medians of a sample are not produced at same time. Hence, care has to be taken when these outputs are saved for further processing or for display as four median outputs correspond to four different address locations. One solution to this problem is to use BRAM for saving the median outputs. The address input of the median output BRAM will be the address of the original sample corresponding to that particular median output. Although the resource utilization is not expected to increase when compared to method 3, additional BRAMs will be required to save the median outputs.

The filtering results and resource utilization on Virtex 5 for directional median filtering methods 1, 2 and 3 which were discussed in this chapter are presented in chapter 6.

Chapter 6

Implementation Results

The results for the three implementation methods discussed in chapter 5 are presented in this chapter. The following are the simulation results for the first three filtering methods described. The simulation outputs were read into MATLAB for display purposes. These methods occupy 150% of the resources on the Spartan 3E board (with XC3S500E chip). For this reason, the methods were implemented on Virtex XC5VLX85T FPGA.

For the first and second methods, the useful output is produced every 5 clock cycles. Hence, the output is sampled every 5 clock cycles. However, for the third implementation, the output is produced every 1.5 clock cycles. Therefore, the output is sampled every 1.5 clock cycles.

The output images of the directional median filtering are smoothed along the direction in which the filtering is performed. This can be clearly observed from the results obtained for the image depicted in fig. 6.2(a). The output images in fig. 6.2 (b)-(e) preserve white pixels only along the direction of the window. For example, when filtering is performed along direction D1for fig. 6.2 (a), only the pixels in fourth quadrant retain their original intensities.

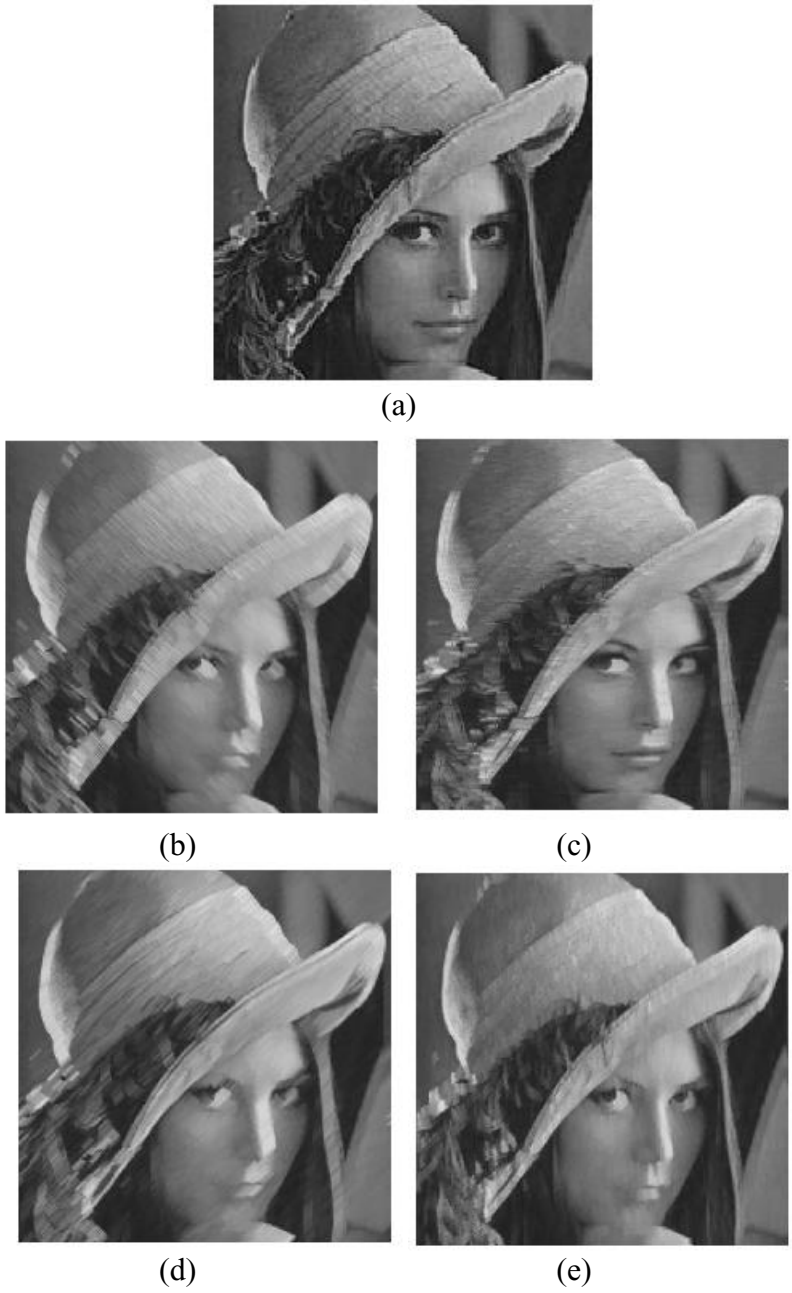
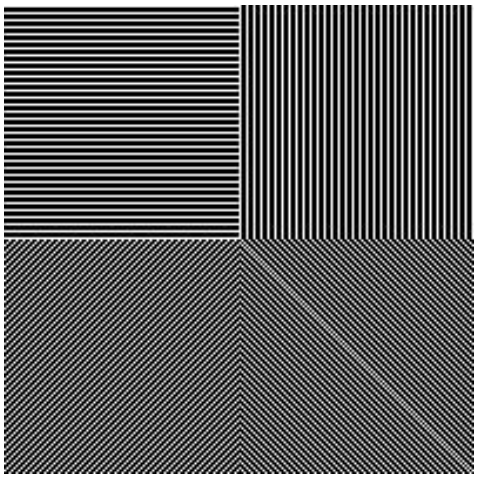
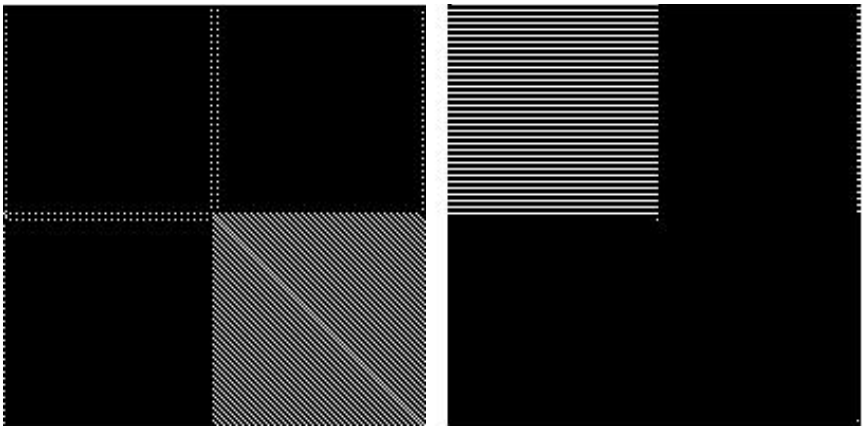


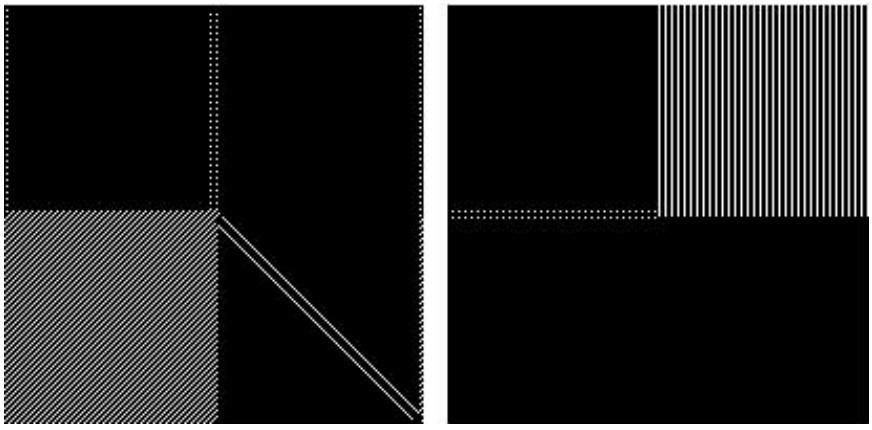
Fig. 6.1 Directional Median Filtering results for image 1: (a) Original Image (b) Direction 1 result (c) Direction 2 result (d) Direction 3 result (e) Direction 4 result



(a)



(b) (c)



(d) (e)

Fig. 6.2 Directional Median Filtering results for image 2: (a) Original Image (b) Direction 1 result (c) Direction 2 result (d) Direction 3 result (e) Direction 4 result

The following table gives the device logic utilization for the Virtex XC5VLX85T FPGA chip for a 100x100 image. A total of 10 BRAMs are required to store a 200x200 image and 8 BRAMs are used to store the 256x256 bin enable codes.

Table II. Logic Utilization summary for a 200x200 image for Virtex 5 FPGA

Implementation method	Percentage of slice occupied	Percentage of slice LUTs used	Percentage of slice registers used	Clock cycles required to generate new output
Method 1	29%	10%	18%	5
Method 2	23%	12%	10%	5
Method 3	26%	14%	16%	1.5

From Table II, it can be observed that the percentage of slices occupied for method 2 is less than that of method 1, because in method 2 the 256-bit codes are not saved and 256 parallel adders are not instantiated. Since in method 1 the samples in the window are also buffered along with the directional codes, the percentage of slices occupied in method 1 is higher than that of method 3; in method 3 only the directional codes associated to the processing window are stored. Moreover, the adders in method 1 will be implemented in the slices, hence the occupancy is higher.

The detailed implementation reports generated by Xilinx ISE for methods 1, 2 and 3 are presented next.

Method 1:

Table III. Logic Utilization summary for Method 1

Slice Logic Utilization	Number	Percentage
1. Number of Slice Registers	9,708 / 51,840	18
2. Number used as Flip Flops	9,708	
3. Number of Slice LUTs	5,581/ 51,840	10
i. Number used as logic	5,498/51,840	10
ii. Number used as Memory	64/13,440	1
iii. Number used as Shift Register	64	
Slice Logic Distribution	Number	Percentage
1. Number of occupied slices	3872/12960	29
i. Number of occupied SLICEMs	16/3,360	1
2. Number of LUT flip-flop pairs used	11,002	
i. Number of fully used LUT-FF pairs	4,287/11,002	38
ii. Number with an unused Flip Flop	1,294/11,002	11
iii. Number with an unused LUT	5,421/11,002	49
3. Number of BlockRAM/FIFO	18/108	16
i. Number of 36k BlockRAM used	17	
ii. Number of 18k BlockRAM used	1	
4. Number of DCM_ADVs	1/12	8

Method 2:

Table IV. Logic Utilization summary for Method 2

Slice Logic Utilization	Number	Percentage
1. Number of Slice Registers	5,617 / 51,840	10
2. Number used as Flip Flops	5,617	
3. Number of Slice LUTs	6,641/ 51,840	12
i. Number used as logic	65,58/51,840	12
ii. Number used as Memory	64/13,440	1
iii. Number used as Shift Register	64	
Slice Logic Distribution	Number	Percentage
1. Number of occupied slices	3,110/12,960	23
i. Number of occupied SLICEMs	16/3,360	1
2. Number of LUT flip-flop pairs used	7,596	
i. Number of fully used LUT-FF pairs	4,662/7,596	61
ii. Number with an unused Flip Flop	1,979/7,596	26
iii. Number with an unused LUT	955/7,596	12
3. Number of BlockRAM/FIFO	18/108	16
i. Number of 36k BlockRAM used	17	
ii. Number of 18k BlockRAM used	1	
4. Number of DCM_ADVs	1/12	8

Method 3:

Table V. Logic Utilization summary for Method 3

Slice Logic Utilization	Number	Percentage
1. Number of Slice Registers	8,641 / 51,840	16
2. Number used as Flip Flops	8,641	
3. Number of Slice LUTs	7,262/ 51,840	14
i. Number used as logic	5,457/51,840	10
ii. Number used as Memory	1,792/13,440	13
iii. Number used as Shift Register	1,792	
Slice Logic Distribution	Number	Percentage
1. Number of occupied slices	3,384/12,960	26
i. Number of occupied SLICEMs	448/3,360	13
2. Number of LUT flip-flop pairs used	9,903	
i. Number of fully used LUT-FF pairs	6,000/9,903	60
ii. Number with an unused Flip Flop	1,262/11,002	12
iii. Number with an unused LUT	2,641/11,002	26
3. Number of BlockRAM/FIFO	18/108	16
i. Number of 36k BlockRAM used	17	
ii. Number of 18k BlockRAM used	1	
4. Number of DCM_ADVs	1/12	8

Chapter 7

Conclusion

This thesis work deals with extending an existing cumulative histogram based median filtering technique to directional median filtering. All implementations were designed for Virtex 5 FPGAs.

The implementations were also attempted for Spartan 3E FPGA. However, the resource utilizations for all implementations exceeded the resources available on Spartan 3E. Hence the designs were modified and implemented for Virtex 5 FPGA. Moreover, the Virtex 5 device runs at 100 MHz while the Spartan 3E device runs at 50 MHz. As a result, hardware implementations on Virtex will generate results faster than Spartan. Another advantage of Virtex device is that the BRAM in Virtex FPGA can be used in Quad Port configuration while the BRAM in Spartan 3E cannot be configured as Quad Port blocks.

All three methods occupied more than 150% of the available slices on Spartan while on Virtex device the resource utilization was less than 30%. However, the designs to perform directional median filtering in only one out of four directions were synthesizable for Spartan 3E with 70% resource utilization. Moreover, since the number of BRAMs in Spartan 3E is 20, smaller images were considered for implementation.

After an initial delay of 27 clock cycles required to get the first median output, new median outputs were produced in 5, 5 and 1.5 clock cycles for methods 1, 2 and 3 respectively. For method 4, the initial delay is expected to be 7 clock cycles and the new median outputs could be generated every clock cycle. The histogram is updated by cancelling the contribution of the oldest sample and adding that of the newest sample at the same time. In VHDL, the contribution

of the oldest sample is cancelled by subtracting 1 from the bin value. While performing the subtraction operation, the bin register value should not take a negative value. One of the reasons for this is that the histogram does not take negative values. The other reason is that the binary representation of a negative number will be considered as a positive number, i.e., if the register value becomes '-1', whose binary equivalent is "1111" (4 bits), then the register value should be "1111". Nevertheless, since the length of bin registers considered for this work is 3 bits the value assigned to the bin will be "111" which is equal to 7. Hence, the bin assumes the value of 7 instead of a 0 which could result in wrong median output.

Of all the four methods discussed in this thesis, method 4 is expected to be the best as the output will be generated in 1 clock cycle. However, the drawback in the method 4 is the additional BRAMs required for saving the output results. This is not a problem when devices with many BRAMs, such as Virtex, are used.

This work concentrated on the efficient implementation of the techniques. Therefore, the denoising capability of proposed methods has not been verified. Future work would deal with applying the proposed directional median filtering methods to denoising images.

Bibliography

[1] Stephen Brown, Zvonko Vranesic, "Fundamentals of Digital Logic with VHDL Design", pg. 4-6, 300-305, McGraw Hill Publications,2000.

[2] Introduction to Field Programmable Gate Arrays: http://cas.web.cern.ch/cas/Sweden-2007/Lectures/Web-versions/Serrano-1.pdf

[3] Richard Williams, "Increase Image Processing System Performance with FPGAs", Xilinx Journals.

[4] Xilinx Inc., FPGA vs. ASIC: http://www.xilinx.com/company/gettingstarted/fpgavsasic.htm#pcs

[5] Mustafa Karaman, Abdullah Atalar, "Design and Implementation of a General-Purpose Median Filter Unit in CMOS VLSI," IEEE Journal of Solid-State Circuits, Vol. 25, No. 2, April 1990.

[6] Hyeong-Soek Yu, Joon-Yeop Lee and Jun-Dong Cho, "A Fast VLSI Implementation of Sorting Algorithm for Standard Median Filters," 12th Annual IEEE International ASIC/SOC Conference Proceedings, 1999.

[7] Alfredo Restrepo (Palacios), Liliana Chacon, "A Smoothing Property of Median Filter," IEEE Transactions on Signal Processing, Vol. 42, No. 6, June 1994.

[8] Siavash Khodambashi, Mohsen Ebrahimi Moghaddam, "An Impluse Noise Fading Technique based on Local Histogram Processing," IEEE International Symposium on Signal Processing and Information Technology, 2009.

[9] S.A.Fahmy, P.Y.K.Cheung, W.Luk, "High-throughput one-dimensional median and weighted median filters on FPGA," IET Comput. Digit. Tech., Vol. 3, Iss. 4, pp. 384-394, 2009.

[10] Miguel A. Vega-Rodriguez, Juan M. Sanchez-Perez, Juan A. Gomez-Pulido, "An FPGA-Based Implementation for Median Filtering Meeting the Real-time Requirements of Automated Visual Inspection Systems," Proceedings of the 10th Mediterranean Conference on Control and Automation, July 2002.

[11] K. Benkrid, D.Crookes, A. Benkrid, "Design and Implementation of a Novel Algorithm for General Purpose Median Filtering on FPGAs", IEEE International Symposium on Circuits and Systems, Vol. 4, pp. IV-425-IV-428, 2002.

[12] Gavin L.Bates and Saeid Nooshabadi, "FPGA Implementation of a Median Filter," Proceedings of IEEE Speech and Image Technologies for Computing and Telecommunications, Vol. 2, pp. 437-440 vol.2, 1997.

[13] Xilinx Inc., Xilinx DS202 (v5.3) Virtex-5 FPGA Family Data Sheet, 2010.

[14] Nick Sawyer, Marc Defossez, "XAPP228 (v1.0): Quad Port Memories in Virtex Devices", September, 2002.

[15] Peter Alfke, "Xilinx wp225 (v1.0), Creative Uses of Block RAM", 2008.

[16] Xilinx Inc., Xilinx DS312 (v3.8) Spartan-3E FPGA Family Data Sheet, 2009.

[17] Xilinx Inc., Xilinx UG320 (v1.1) Spartan-3E FPGA Starter Kit Board User Guide, 2008.

[18] Xilinx ISE Design Suite 11.1 Help.

[19] Xilinx Inc., Xilinx ISE XST User Guide, 2010.

[20] Nick Sawyer, Marc Defossez, "XAPP229 (v1.1): Wider Block Memories", April, 2007.

[21] Xilinx Inc., Xilinx DS512 (v3.1), "Block Memory Generator", April, 2009.

[22] "XAPP463 (v1.1.2): Using Block RAM in Spartan-3 FPGAs", July, 2003.

[23] Xilinx Inc., Xilinx UG500 (v1.0), "Programmable Logic Design – Quick Start Guide", May, 2008.

Appendices

Appendix A

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

---- Uncomment the following library declaration if instantiating
---- any Xilinx primitives in this code.
library UNISIM;
use UNISIM.VComponents.all;

entity DirectionalMedian is
port (
	clk: in std_logic;
	med_index : in std_logic_vector(2 downto 0);
	D1_out, D2_out, D3_out, D4_out: out std_logic_vector(7 downto 0)
	 );
end DirectionalMedian;

architecture Behavioral of DirectionalMedian is

component Image
		port (
		clka: IN std_logic;
		addra: IN std_logic_VECTOR(13 downto 0);
		douta: OUT std_logic_VECTOR(7 downto 0));
end component;

component Decoder ---Quad port BRAM configuration
		port (
		clka: IN std_logic;
		addra: IN std_logic_VECTOR(7 downto 0);
		douta: OUT std_logic_VECTOR(255 downto 0);
		clkb: IN std_logic;
		addrb: IN std_logic_VECTOR(7 downto 0);
		doutb: OUT std_logic_VECTOR(255 downto 0)
);
end component;

component Histogram
		Port (
		in1 : in  STD_LOGIC_VECTOR (255 downto 0);
		in2 : in  STD_LOGIC_VECTOR (255 downto 0);
		in3 : in  STD_LOGIC_VECTOR (255 downto 0);
		in4 : in  STD_LOGIC_VECTOR (255 downto 0);
		in5 : in  STD_LOGIC_VECTOR (255 downto 0);
		clk : in std_logic;
		 median_index : in std_logic_vector(2 downto 0);
		output : out std_logic_vector(255 downto 0)
);
end component;
```

```
component PriorityEncoder
        Port (
        gtmed_in : in  STD_LOGIC_VECTOR (255 downto 0);
        median : out  STD_LOGIC_VECTOR (7 downto 0)
 );
end component;


signal clk2x,DCM_LOCKED: std_logic := '0';
signal NewSample, address_decoder1,address_decoder2 : std_logic_vector(7 downto 0):=(others => '0');
signal median_out11,median_out12,median_out13,median_out14 : std_logic_vector(7 downto 0):=(others => '0');
signal cnt : std_logic_vector(13 downto 0):= (others => '0');
type array5 is array(1 to 5) of std_logic_vector(13 downto 0);
signal addr: array5 := (others => (others => '0'));
signal RAM_addr : std_logic_vector(13 downto 0):=(others => '0');
signal index : integer range 1 to 25 := 1;
constant Depth : integer := 25;
type array25 is array(1 to depth) of std_logic_vector(7 downto 0);
signal R: array25 := (others => (others => '0'));
signal i, delay : integer := 0;
signal k : integer := 5;
signal k2 : integer  range 1 to 10 := 1;
signal j: integer range 1 to 5 := 1;
signal R1_EN : STD_LOGIC_VECTOR(255 DOWNTO 0):= (OTHERS => '0');
signal R2_EN : STD_LOGIC_VECTOR(255 DOWNTO 0):= (OTHERS => '0');
signal R3_EN,R4_EN,R5_EN : STD_LOGIC_VECTOR(255 DOWNTO 0):= (OTHERS => '0');
type DirEn_Array is array(1 to 10) of STD_LOGIC_VECTOR(255 DOWNTO 0);
SIGNAL D_EN1,D_EN2 : DirEn_Array := (others => (others => '0'));
signal histogram1, histogram2, histogram3, histogram4 : STD_LOGIC_VECTOR(255 DOWNTO 0):= (OTHERS
=> '0');

begin

process(clk)
begin
if(clk'event and clk = '1')then
if(DCM_LOCKED = '1')then
if(i < 4)then
i <= i + 1;
else
i <= 0;
end if;
j <= i + 1;
RAM_addr <= addr(j);
if(cnt < 9999)then
if(delay = 5)then
cnt <= cnt + '1';
delay <= 1;
else
delay <= delay + 1;
end if;
else
cnt <= (others => '0');
end if;
addr(1) <= cnt - "11001000";
```

```
addr(2) <= cnt - "1100100";
addr(3) <= cnt;
addr(4) <= cnt + "1100100";
addr(5) <= cnt + "11001000";
end if;
end process;

process(clk)
begin
if(clk'event and clk = '1')then
R(1) <= NewSample;
R(2) <= R(1);R(3) <= R(2);R(4) <= R(3);R(5) <= R(4);R(6) <= R(5);R(7) <= R(6);
R(8) <= R(7);R(9) <= R(8);R(10) <= R(9);R(11) <= R(10);R(12) <= R(11);R(13) <= R(12);
R(14) <= R(13);R(15) <= R(14);R(16) <= R(15);R(17) <= R(16);R(18) <= R(17);R(19) <= R(18);
R(20) <= R(19);R(21) <= R(20);R(22) <= R(21);R(23) <= R(22);R(24) <= R(23);R(25) <= R(24);
end if;
end process;

process(clk,cnt, delay)
begin
if(clk'event and clk = '1')then
if(cnt >= 3)then
if(delay = 3)then
D_1(1) <= R(1);D_1(3) <= R(7);D_1(5) <= R(13);D_1(7) <= R(19);D_1(9) <= R(25);-
D_1(2) <= R(3);D_1(4) <= R(8);D_1(6) <= R(13);D_1(8) <= R(18);D_1(10) <= R(23);
D_2(1) <= R(5);D_2(3) <= R(9);D_2(5) <= R(13);D_2(7) <= R(17);D_2(9) <= R(21);-
D_2(2) <= R(11);D_2(4) <= R(12);D_2(6) <= R(13);D_2(8) <= R(14);D_2(10) <= R(15);
else
D_1 <= D_1; D_2 <= D_2;
end if;
end if;
end if;
end process;

process(clk2x)
begin
if(clk2x'event and clk2x = '1')then
address_decoder1 <= D_1(k2);
address_decoder2 <= D_2(k2);
if(k < 9)then
k <= k + 1;
else
k <= 0;
end if;
k2 <= k + 1;
end if;
end process;

process(clk2x)
begin
if(clk2x'event and clk2x = '1')then
D_En1(1) <= R1_En;D_En2(1) <= R2_En;
D_En1(2) <= D_En1(1);D_En2(2) <= D_En2(1);
D_En1(3) <= D_En1(2);D_En2(3) <= D_En2(2);
D_En1(4) <= D_En1(3);D_En2(4) <= D_En2(3);
D_En1(5) <= D_En1(4);D_En2(5) <= D_En2(4);
```

```
D_En1(6) <= D_En1(5);D_En2(6) <= D_En2(5);
D_En1(7) <= D_En1(6);D_En2(7) <= D_En2(6);
D_En1(8) <= D_En1(7);D_En2(8) <= D_En2(7);
D_En1(9) <= D_En1(8);D_En2(9) <= D_En2(8);
D_En1(10) <= D_En1(9);D_En2(10) <= D_En2(9);
end if;
end process;

u0 : DCM_SP
generic map( CLK_FEEDBACK => "2X",CLKDV_DIVIDE => 2.0,CLKFX_DIVIDE => 1,CLKFX_MULTIPLY
=> 4,CLKIN_DIVIDE_BY_2 => FALSE, CLKIN_PERIOD => 20.000, CLKOUT_PHASE_SHIFT =>
”NONE",DESKEW_ADJUST => "SYSTEM_SYNCHRONOUS",DFS_FREQUENCY_MODE => "LOW",
DLL_FREQUENCY_MODE => "LOW",DUTY_CYCLE_CORRECTION => TRUE,FACTORY_JF =>
x"C080",PHASE_SHIFT => 0, STARTUP_WAIT => FALSE)
port map (CLKFB => clk2x, CLKIN => CLK, DSSEN => '0',PSCLK => '0',PSEN => '0',PSINCDEC => '0',RST =>
'0',CLKDV => open, CLKFX => open, CLKFX180 => open,CLK0 => open,CLK2X => CLK2X, CLK2X180 =>
open,CLK90 => open,CLK180 => open, CLK270 => open, LOCKED => DCM_Locked, PSDONE => open,
STATUS => open);

U1: Image port map(clka => clk, addra => RAM_addr, douta => NewSample);
U2: Decoder port map (clka => clk2x,  addra => address_decoder1, d outa => R1_En, clkb => clk2x, addrb =>
ddress_decoder2, doutb => R2_En);
U3: Histogram  port map(clk => clk2x, in1 => D_EN1(1), IN2 => D_EN1(3), IN3 => D_EN1(5), IN4 =>
D_EN1(7), IN5 => D_EN1(9), median_index => med_index, output => histogram1);
U4: PriorityEncoder port map(gtmed_in => histogram1, median => median_out11);
U5: Histogram  port map(clk => clk2x, in1 => D_EN1(2), IN2 => D_EN1(4), IN3 => D_EN1(6), IN4 =>
D_EN1(8), IN5 => D_EN1(10), median_index => med_index, output => histogram3);
U6: PriorityEncoder port map(gtmed_in => histogram3, median => median_out13);
U7: Histogram  port map(clk => clk2x, in1 => D_EN2(1), IN2 => D_EN2(3), IN3 => D_EN2(5), IN4 =>
D_EN2(7), IN5 => D_EN2(9), median_index => med_index, output => histogram2);
U8: PriorityEncoder port map(gtmed_in => histogram2, median => median_out12);
U9: Histogram port map(clk => clk2x, in1 => D_EN2(2), IN2 => D_EN2(4), IN3 => D_EN2(6), IN4 => D_EN2(8),
IN5 => D_EN2(10), median_index => med_index, output => histogram4);
U10: PriorityEncoder port map(gtmed_in => histogram4, median => median_out14);


D1_out <= median_out11;
D2_out <= median_out12;
D3_out <= median_out13;
D4_out <= median_out14;

end Behavioral;

----------------------------------------------------------------------------------
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

---- Uncomment the following library declaration if instantiating
---- any Xilinx primitives in this code.
--library UNISIM;
--use UNISIM.VComponents.all;
```

```
entity Histogram is
Port (
in1 : in  STD_LOGIC_VECTOR (255 downto 0);
in2 : in  STD_LOGIC_VECTOR (255 downto 0);
in3 : in  STD_LOGIC_VECTOR (255 downto 0);
in4 : in  STD_LOGIC_VECTOR (255 downto 0);
in5 : in  STD_LOGIC_VECTOR (255 downto 0);
 clk : in std_logic;
median_index : in std_logic_vector(2 downto 0);
 output : out std_logic_vector(255 downto 0)
);
end Histogram;

architecture Behavioral of Histogram is

component Adder
port(
a1,a2,a3,a4,a5 : in  STD_LOGIC;
sum : out std_logic_vector(2 downto 0)
);
end component;

component BinNode
Port (
median_index : in  STD_LOGIC_VECTOR (2 downto 0);
clk : in std_logic;
-- windowFull : in std_logic;
add_value : in  STD_LOGIC_VECTOR (2 downto 0);
-- reg_val : out integer;
gtmed : out  STD_LOGIC
);
end component;

signal r1,r2,r3,r4,r5 : STD_LOGIC_VECTOR (255 downto 0) := (others => '0');
type array256 is array(1 to 256) of std_logic_vector(2 downto 0);
signal sum1: array256 := (others => (others => '0'));

begin
r1 <= in1; r2 <= in2; r3 <= in3; r4 <= in4; r5 <= in5;

u0: Adder port map(a1 => r1(255),a2 => r2(255),a3 => r3(255),a4 => r4(255),a5 => r5(255),sum => sum1(256));
u1: Adder port map(a1 => r1(254),a2 => r2(254),a3 => r3(254),a4 => r4(254),a5 => r5(254),sum => sum1(255));
u2: Adder port map(a1 => r1(253),a2 => r2(253),a3 => r3(253),a4 => r4(253),a5 => r5(253),sum => sum1(254));
u3: Adder port map(a1 => r1(252),a2 => r2(252),a3 => r3(252),a4 => r4(252),a5 => r5(252),sum => sum1(253));
u4: Adder port map(a1 => r1(251),a2 => r2(251),a3 => r3(251),a4 => r4(251),a5 => r5(251),sum => sum1(252));
u5: Adder port map(a1 => r1(250),a2 => r2(250),a3 => r3(250),a4 => r4(250),a5 => r5(250),sum => sum1(251));
u6: Adder port map(a1 => r1(249),a2 => r2(249),a3 => r3(249),a4 => r4(249),a5 => r5(249),sum => sum1(250));
u7: Adder port map(a1 => r1(248),a2 => r2(248),a3 => r3(248),a4 => r4(248),a5 => r5(248),sum => sum1(249));
u8: Adder port map(a1 => r1(247),a2 => r2(247),a3 => r3(247),a4 => r4(247),a5 => r5(247),sum => sum1(248));
u9: Adder port map(a1 => r1(246),a2 => r2(246),a3 => r3(246),a4 => r4(246),a5 => r5(246),sum => sum1(247));
u10: Adder port map(a1 => r1(245),a2 => r2(245),a3 => r3(245),a4 => r4(245),a5 => r5(245),sum => sum1(246));

<-------------------------------------------------------------------------------------------------------------------------------------->
<----------------------------------------similar to above for bins 11 to 245---------------------------------------------------------->
<-------------------------------------------------------------------------------------------------------------------------------------->
```

```
u246: Adder port map(a1 => r1(9),a2 => r2(9),a3 => r3(9),a4 => r4(9),a5 => r5(9),sum => sum1(10));
u247: Adder port map(a1 => r1(8),a2 => r2(8),a3 => r3(8),a4 => r4(8),a5 => r5(8),sum => sum1(9));
u248: Adder port map(a1 => r1(7),a2 => r2(7),a3 => r3(7),a4 => r4(7),a5 => r5(7),sum => sum1(8));
u249: Adder port map(a1 => r1(6),a2 => r2(6),a3 => r3(6),a4 => r4(6),a5 => r5(6),sum => sum1(7));
u250: Adder port map(a1 => r1(5),a2 => r2(5),a3 => r3(5),a4 => r4(5),a5 => r5(5),sum => sum1(6));
u251: Adder port map(a1 => r1(4),a2 => r2(4),a3 => r3(4),a4 => r4(4),a5 => r5(4),sum => sum1(5));
u252: Adder port map(a1 => r1(3),a2 => r2(3),a3 => r3(3),a4 => r4(3),a5 => r5(3),sum => sum1(4));
u253: Adder port map(a1 => r1(2),a2 => r2(2),a3 => r3(2),a4 => r4(2),a5 => r5(2),sum => sum1(3));
u254: Adder port map(a1 => r1(1),a2 => r2(1),a3 => r3(1),a4 => r4(1),a5 => r5(1),sum => sum1(2));
u255: Adder port map(a1 => r1(0),a2 => r2(0),a3 => r3(0),a4 => r4(0),a5 => r5(0),sum => sum1(1));
u256: BinNode port map(clk => clk, median_index => median_index, add_value => sum1(256), gtmed =>
output(255));
u257: BinNode port map(clk => clk, median_index => median_index, add_value => sum1(255), gtmed =>
output(254));
u258: BinNode port map(clk => clk, median_index => median_index, add_value => sum1(254), gtmed =>
output(253));
u259: BinNode port map(clk => clk, median_index => median_index, add_value => sum1(253), gtmed =>
output(252));
u260: BinNode port map(clk => clk, median_index => median_index, add_value => sum1(252), gtmed =>
output(251));
u261: BinNode port map(clk => clk, median_index => median_index, add_value => sum1(251), gtmed =>
output(250));
u262: BinNode port map(clk => clk, median_index => median_index, add_value => sum1(250), gtmed =>
output(249));
u263: BinNode port map(clk => clk, median_index => median_index, add_value => sum1(249), gtmed =>
output(248));
u264: BinNode port map(clk => clk, median_index => median_index, add_value => sum1(248), gtmed =>
output(247));
u265: BinNode port map(clk => clk, median_index => median_index, add_value => sum1(247), gtmed =>
output(246));

<------------------------------------------------------------------------------------------------------------------------------->
<----------------------------------------similar to above for bins 11 to 244---------------------------------------------------->
<------------------------------------------------------------------------------------------------------------------------------->


u501: BinNode port map(clk => clk, median_index => median_index, add_value => sum1(11), gtmed =>
output(10));
u502: BinNode port map(clk => clk, median_index => median_index, add_value => sum1(10), gtmed =>
output(9));
u503: BinNode port map(clk => clk, median_index => median_index, add_value => sum1(9), gtmed => output(8));
u504: BinNode port map(clk => clk, median_index => median_index, add_value => sum1(8), gtmed => output(7));
u505: BinNode port map(clk => clk, median_index => median_index, add_value => sum1(7), gtmed => output(6));
u506: BinNode port map(clk => clk, median_index => median_index, add_value => sum1(6), gtmed => output(5));
u507: BinNode port map(clk => clk, median_index => median_index, add_value => sum1(5), gtmed => output(4));
u508: BinNode port map(clk => clk, median_index => median_index, add_value => sum1(4), gtmed => output(3));
u509: BinNode port map(clk => clk, median_index => median_index, add_value => sum1(3), gtmed => output(2));
u510: BinNode port map(clk => clk, median_index => median_index, add_value => sum1(2), gtmed => output(1));
u511: BinNode port map(clk => clk, median_index => median_index, add_value => sum1(1), gtmed => output(0));
--------------------------------------------------------------------------------------------------------------

end Behavioral;


-----------------------------------------------------------------------------
```

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

---- Uncomment the following library declaration if instantiating
---- any Xilinx primitives in this code.
--library UNISIM;
--use UNISIM.VComponents.all;

entity BinNodeArray is
port(
clk : in std_logic;
rst : in std_logic;
median_index : in  STD_LOGIC_VECTOR (2 downto 0);
enBin : in std_logic_vector(255 downto 0);
output : out std_logic_vector(255 downto 0)
);
end BinNodeArray;

architecture Behavioral of BinNodeArray is

component BinNode
port(
clk : in std_logic;
rst : in std_logic;
median_index : in  STD_LOGIC_VECTOR (2 downto 0);
add_bit : in std_logic;
gtmed : out  STD_LOGIC
);
end component;

begin

b0: BinNode port map(clk => clk, rst => rst, median_index => median_index, add_Bit => enBin(255), gtmed =>
output(255));
b1: BinNode port map(clk => clk, rst => rst, median_index => median_index, add_Bit => enBin(254), gtmed =>
output(254));
b2: BinNode port map(clk => clk, rst => rst, median_index => median_index, add_Bit => enBin(253), gtmed =>
output(253));
b3: BinNode port map(clk => clk, rst => rst, median_index => median_index, add_Bit => enBin(252), gtmed =>
output(252));
b4: BinNode port map(clk => clk, rst => rst, median_index => median_index, add_Bit => enBin(251), gtmed =>
output(251));
b5: BinNode2 port map(clk => clk, rst => rst, median_index => median_index, add_Bit => enBin(250), gtmed =>
output(250));
b6: BinNode port map(clk => clk, rst => rst, median_index => median_index, add_Bit => enBin(249), gtmed =>
output(249));
b7: BinNode port map(clk => clk, rst => rst, median_index => median_index, add_Bit => enBin(248), gtmed =>
output(248));
b8: BinNode port map(clk => clk, rst => rst, median_index => median_index, add_Bit => enBin(247), gtmed =>
output(247));
b9: BinNode port map(clk => clk, rst => rst, median_index => median_index, add_Bit => enBin(246), gtmed =>
output(246));
b10: BinNode port map(clk => clk, rst => rst, median_index => median_index, add_Bit => enBin(245), gtmed =>
output(245));
```

```
<-------------------------------------------------------------------------------------------------------------------------------------------->
<----------------------------------------similar to above for bins 11 to 249---------------------------------------------------------->
<-------------------------------------------------------------------------------------------------------------------------------------------->


b250: BinNode port map(clk => clk, rst => rst, median_index => median_index, add_Bit => enBin(5), gtmed =>
output(5));
b251: BinNode port map(clk => clk, rst => rst, median_index => median_index, add_Bit => enBin(4), gtmed =>
output(4));
b252: BinNode port map(clk => clk, rst => rst, median_index => median_index, add_Bit => enBin(3), gtmed =>
output(3));
b253: BinNode port map(clk => clk, rst => rst, median_index => median_index, add_Bit => enBin(2), gtmed =>
output(2));
b254: BinNode port map(clk => clk, rst => rst, median_index => median_index, add_Bit => enBin(1), gtmed =>
output(1));
b255: BinNode port map(clk => clk, rst => rst, median_index => median_index, add_Bit => enBin(0), gtmed =>
output(0));

end Behavioral;


----------------------------------------------------------------------------------------------------

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

---- Uncomment the following library declaration if instantiating
---- any Xilinx primitives in this code.
--library UNISIM;
--use UNISIM.VComponents.all;

entity PriorityEncoder is
Port (
gtmed_in : in  STD_LOGIC_VECTOR (255 downto 0);
median : out  STD_LOGIC_VECTOR (7 downto 0)
);
end PriorityEncoder;

architecture Behavioral of PriorityEncoder is

begin
                median <="00000000" when gtmed_in(255) = '1' else
                                "00000001" when gtmed_in(254) = '1' else
                                "00000010" when gtmed_in(253) = '1' else
                                "00000011" when gtmed_in(252) = '1' else
                                "00000100" when gtmed_in(251) = '1' else
                                "00000101" when gtmed_in(250) = '1' else
                                "00000110" when gtmed_in(249) = '1' else
                                "00000111" when gtmed_in(248) = '1' else
                                "00001000" when gtmed_in(247) = '1' else

<-------------------------------------------------------------------------------------------------------------------------------------------->
<----------------------------------------similar to above for 246 to 9---------------------------------------------------------->
```

<--->

```
                                "11110101" when gtmed_in(10) = '1' else
                                "11110110" when gtmed_in(9) = '1' else
                                "11110111" when gtmed_in(8) = '1' else
                                "11111000" when gtmed_in(7) = '1' else
                                "11111001" when gtmed_in(6) = '1' else
                                "11111010" when gtmed_in(5) = '1' else
                                "11111011" when gtmed_in(4) = '1' else
                                "11111100" when gtmed_in(3) = '1' else
                                "11111101" when gtmed_in(2) = '1' else
                                "11111110" when gtmed_in(1) = '1' else
                                "11111111" when gtmed_in(0) = '1' ;


end Behavioral;


entity Adder is
port(
a1,a2,a3,a4,a5 : in  STD_LOGIC;
sum : out std_logic_vector(2 downto 0)
);
end Adder;

architecture Behavioral of Adder is
--signal temp: std_logic_vector(4 downto 0) := (others => '0');

begin
sum <= ("00"&a1)+("00"&a2)+("00"&a3)+("00"&a4)+("00"&a5);
end Behavioral;


entity BinNode is
Port (
 median_index : in  STD_LOGIC_VECTOR (2 downto 0);
clk : in std_logic;
-- windowFull : in std_logic;
add_value : in  STD_LOGIC_VECTOR (2 downto 0);
--reg_val : out integer;
 gtmed : out  STD_LOGIC
);
end BinNode;

architecture Behavioral of BinNode is

signal reg : STD_LOGIC_VECTOR(2 downto 0):=(others => '0');
begin

process(clk)
begin
if(clk'event and clk = '1')then
reg <= add_value;
if(reg >= median_index)then
gtmed <= '1';
else
```

```
gtmed <= '0';
end if;
end if;
end process;
--reg_val <= conv_integer(reg);
end Behavioral;
```

Appendix B

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

---- Uncomment the following library declaration if instantiating
---- any Xilinx primitives in this code.
--library UNISIM;
--use UNISIM.VComponents.all;

entity BinNode is
Port (
median_index : in  STD_LOGIC_VECTOR (2 downto 0);
clk : in std_logic;
--reg_out : out STD_LOGIC_VECTOR (2 downto 0);
 rst : in std_logic;
add_bit : in  STD_LOGIC;
gtmed : out  STD_LOGIC
);
end BinNode;

architecture Behavioral of BinNode is

signal reg,reg1 : STD_LOGIC_VECTOR(2 downto 0):=(others => '0');
begin

process(clk)
begin

if(clk'event and clk = '1')then
--for samples 2 to 5 in a direction
if(rst = '0')then
reg <= reg + add_bit;
--for first sample in a direction
elsif(rst = '1')then
reg <= add_bit;
end if;
if(reg >= median_index)then
gtmed <= '1';
else
gtmed <= '0';
end if;
end if;
end process;
end Behavioral;
```

```
----------------------------------------------------------------------------------
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

---- Uncomment the following library declaration if instantiating
---- any Xilinx primitives in this code.
--library UNISIM;
--use UNISIM.VComponents.all;

entity BinNodeArray is
port(
clk : in std_logic;
rst : in std_logic;
median_index : in  STD_LOGIC_VECTOR (2 downto 0);
enBin : in std_logic_vector(255 downto 0);
output : out std_logic_vector(255 downto 0)
);
end BinNodeArray;

architecture Behavioral of BinNodeArray is

component BinNode
port(
clk : in std_logic;
rst : in std_logic;
median_index : in  STD_LOGIC_VECTOR (2 downto 0);
add_bit : in std_logic;
gtmed : out  STD_LOGIC
);
end component;

begin

b0: BinNode port map(clk => clk, rst => rst, median_index => median_index, add_Bit => enBin(255), gtmed =>
output(255));
b1: BinNode port map(clk => clk, rst => rst, median_index => median_index, add_Bit => enBin(254), gtmed =>
output(254));
b2: BinNode port map(clk => clk, rst => rst, median_index => median_index, add_Bit => enBin(253), gtmed =>
output(253));
b3: BinNode port map(clk => clk, rst => rst, median_index => median_index, add_Bit => enBin(252), gtmed =>
output(252));
b4: BinNode port map(clk => clk, rst => rst, median_index => median_index, add_Bit => enBin(251), gtmed =>
output(251));
b5: BinNode port map(clk => clk, rst => rst, median_index => median_index, add_Bit => enBin(250), gtmed =>
output(250));
b6: BinNode port map(clk => clk, rst => rst, median_index => median_index, add_Bit => enBin(249), gtmed =>
output(249));
b7: BinNode port map(clk => clk, rst => rst, median_index => median_index, add_Bit => enBin(248), gtmed =>
output(248));
b8: BinNode port map(clk => clk, rst => rst, median_index => median_index, add_Bit => enBin(247), gtmed =>
output(247));
b9: BinNode port map(clk => clk, rst => rst, median_index => median_index, add_Bit => enBin(246), gtmed =>
output(246));
```

```
<----------------------------------------------------------------------------------------------------------------------------------->
<----------------------------------------similar to above for bins 11 to 245---------------------------------------------------->
<----------------------------------------------------------------------------------------------------------------------------------->

b245: BinNode port map(clk => clk, rst => rst, median_index => median_index, add_Bit => enBin(10), gtmed =>
output(10));
b246: BinNode port map(clk => clk, rst => rst, median_index => median_index, add_Bit => enBin(9), gtmed =>
output(9));
b247: BinNode port map(clk => clk, rst => rst, median_index => median_index, add_Bit => enBin(8), gtmed =>
output(8));
b248: BinNode port map(clk => clk, rst => rst, median_index => median_index, add_Bit => enBin(7), gtmed =>
output(7));
b249: BinNode port map(clk => clk, rst => rst, median_index => median_index, add_Bit => enBin(6), gtmed =>
output(6));
b250: BinNode port map(clk => clk, rst => rst, median_index => median_index, add_Bit => enBin(5), gtmed =>
output(5));
b251: BinNode port map(clk => clk, rst => rst, median_index => median_index, add_Bit => enBin(4), gtmed =>
output(4));
b252: BinNode port map(clk => clk, rst => rst, median_index => median_index, add_Bit => enBin(3), gtmed =>
output(3));
b253: BinNode port map(clk => clk, rst => rst, median_index => median_index, add_Bit => enBin(2), gtmed =>
output(2));
b254: BinNode port map(clk => clk, rst => rst, median_index => median_index, add_Bit => enBin(1), gtmed =>
output(1));
b255: BinNode port map(clk => clk, rst => rst, median_index => median_index, add_Bit => enBin(0), gtmed =>
output(0));

end Behavioral;
```

Appendix C

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

---- Uncomment the following library declaration if instantiating
---- any Xilinx primitives in this code.
library UNISIM;
use UNISIM.VComponents.all;

entity method3 is
Port(
clk_in : in  STD_LOGIC;
med_index : in std_logic_vector(2 downto 0);
D1_median_out,D2_median_out,D3_median_out,D4_median_out: out std_logic_vector(7 downto 0)
);
end method3;

architecture Behavioral of method3 is

component Image ---Quad port BRAM configuration
        port (
        clka: IN std_logic;
```

```
            addra: IN std_logic_VECTOR(13 downto 0);
            douta: OUT std_logic_VECTOR(7 downto 0);
            clkb: IN std_logic;
            addrb: IN std_logic_VECTOR(13 downto 0);
            doutb: OUT std_logic_VECTOR(7 downto 0)
);
end component;

component Decoder  ---Quad port BRAM configuration
            port (
            clka: IN std_logic;
            addra: IN std_logic_VECTOR(7 downto 0);
            douta: OUT std_logic_VECTOR(255 downto 0);
            clkb: IN std_logic;
            addrb: IN std_logic_VECTOR(7 downto 0);
            doutb: OUT std_logic_VECTOR(255 downto 0)
);
end component;

component Histogram
            Port (
            in1 : in  STD_LOGIC_VECTOR (255 downto 0);
            in2 : in  STD_LOGIC_VECTOR (255 downto 0);
            in3 : in  STD_LOGIC_VECTOR (255 downto 0);
            in4 : in  STD_LOGIC_VECTOR (255 downto 0);
            in5 : in  STD_LOGIC_VECTOR (255 downto 0);
             clk : in std_logic;
             median_index : in std_logic_vector(2 downto 0);
             output : out std_logic_vector(255 downto 0)
 );
end component;

component PriorityEncoder
            Port(
            gtmed_in : in  STD_LOGIC_VECTOR (255 downto 0);
             median : out  STD_LOGIC_VECTOR (7 downto 0)
);
end component;

signal NewSample1,NewSample2,temp : std_logic_vector(7 downto 0):=(others => '0');
signal median_out11,median_out12,median_out13,median_out14 : std_logic_vector(7 downto 0):=(others => '0');
signal clkfb, clk0, clk2x, dcm_locked : std_logic := '0';
signal cnt : std_logic_vector(13 downto 0):= (others => '0');
type array3 is array(1 to 3) of std_logic_vector(13 downto 0);
signal addr1,addr2: array3 := (others => (others => '0'));
signal RAM_addr1,RAM_addr2 : std_logic_vector(13 downto 0):=(others => '0');
signal R_EN, R1_EN, R2_EN : STD_LOGIC_VECTOR(255 DOWNTO 0):= (OTHERS => '0');
constant Depth : integer := 30;
signal index : integer range 1 to depth := 1;
type array30 is array(1 to depth) of std_logic_vector(7 downto 0);
signal R: array30 := (others => (others => '0'));
type En_Array is array(1 to depth) of STD_LOGIC_VECTOR(255 DOWNTO 0);
SIGNAL D_En : En_Array := (others => (others => '0'));
type Dir_En is array(1 to 5) of STD_LOGIC_VECTOR(255 DOWNTO 0);
signal D1,D2,D3,D4,D1_N,D2_N,D3_N,D4_N : Dir_En := (others => (others => '0'));
```

```
signal histogram1, histogram2, histogram3, histogram4 : STD_LOGIC_VECTOR(255 DOWNTO 0):= (OTHERS
=> '0');
--------------------------------------------------------------------------------
signal i,delay : integer := 0;
 signal k : integer := 10;
signal k2 : integer  range 1 to 25 := 1;
signal j : integer range 1 to 3 := 1;
--------------------------------------------------------------------------------


begin

process(clk2x)
begin
if(clk2x'event and clk2x = '1')then
if(DCM_LOCKED = '1')then
if(i < 2)then
i <= i + 1;
else
i <= 0;
end if;
j <= i + 1;
RAM_addr1 <= addr1(j);
RAM_addr2 <= addr2(j);
---------------For 100x100 image-------------------------
if(cnt < 9999)then
if(delay = 3)then
cnt <= cnt + '1';
delay <= 1;
else
delay <= delay + 1;
end if;
else
cnt <= (others => '0');
end if;
addr1(1) <= cnt -“11001000";
addr2(1) <= cnt - "1100100";
addr1(2) <= cnt;
addr2(2) <= cnt + "1100100";
addr1(3) <= cnt + "11001000";
addr2(3) <= (others => '0');
end if;
end if;
end process;

process(clk2x)
begin
if(clk2x'event and clk2x = '1')then
D_En(2) <= R1_En;D_En(1) <= R2_En;
D_EN(3) <= D_EN(1);D_EN(4) <= D_EN(2);D_EN(5) <= D_EN(3);D_EN(6) <= D_EN(4);D_EN(7) <= D_EN(5);
D_EN(8) <= D_EN(6);D_EN(9) <= D_EN(7);D_EN(10) <= D_EN(8);D_EN(11) <= D_EN(9);D_EN(12) <=
D_EN(10);D_EN(13) <= D_EN(11);
D_EN(14) <= D_EN(12);D_EN(15) <= D_EN(13);D_EN(16) <= D_EN(14);D_EN(17) <= D_EN(15);D_EN(18)
<= D_EN(16);D_EN(19) <= D_EN(17);
D_EN(20) <= D_EN(18);D_EN(21) <= D_EN(19);D_EN(22) <= D_EN(20);D_EN(23) <= D_EN(21);D_EN(24)
<= D_EN(22);D_EN(25) <= D_EN(23);
```

```
D_EN(26) <= D_EN(24);D_EN(27) <= D_EN(25);D_EN(28) <= D_EN(26);D_EN(29) <= D_EN(27);D_EN(30)
<= D_EN(28);
-----------------------------------------------------------------------------------------------------------------------
end if;
end process;

<------------------------------------------------------------------------------------------------------------------------>
<-------------------------------------------------same steps as in Appendix A-------------------------------------------------->
<------------------------------------------------------------------------------------------------------------------------>


D1_median_out <= median_out11;
D2_median_out <= median_out12;
D3_median_out <= median_out13;
D4_median_out <= median_out14;

end Behavioral;
```

Vita

Madhuri Gundam was born in Hyderabad, India. She received her Bachelor's degree in Electronics and Communications Engineering from Jawaharlal Nehru Technological University, India in 2007. She started her Master's in Engineering in the Electrical Engineering department at UNO in Fall 2007 and is expected to complete it in Fall 2010. She is currently pursuing her PhD degree in Engineering and Applied Sciences in the Electrical Engineering department at UNO.