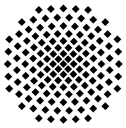


Universität Stuttgart

Institute of Parallel and Distributed Systems
Department of Parallel Systems
Universitätsstraße 38
D–70569 Stuttgart

Master Thesis Nr. 3256

Implementation of Watershed Based Image Segmentation Algorithm in FPGA

Sameer Ruparelia

Course of Study:	M.Sc. Information Technology Specialization : Embedded Systems
Examiner:	Prof.Dr.-Ing. Sven Simon
Supervisor:	Dr.-Ing. Marek Wroblewski Dipl.-Ing(FH). Arne Zender
Commenced:	September 16, 2011
Completed:	March 15, 2012
CR-Classification:	B.5.1, B.7.1, I.4.6

Acknowlegement

I would like to express sincere gratitude to Prof.Dr.-Ing. Sven Simon, Head of the Parallel Systems Department at the Institute for Parallel and Distributed Systems, University of Stuttgart for his guidance and providing me a opportunity to work on this topic.

I would like to thank Dr.-Ing. Marek Wroblewski, who has been very helpful and has assisted me in numerous ways during my thesis.

Special thanks and appreciation to Dipl.-Ing.(FH) Arne Zender. His valuable inputs and support were vital for successful completion of my master thesis. I am very grateful to the members of "Corporate Research Centre of Robert Bosch GmbH" for their kind co-operation and support which also helped me in completion of my thesis.

I would also like to thank my family and friends for all their advice and encouragement during the thesis.

Sameer Ruparelia

Abstract

The watershed algorithm is a commonly used method of solving the image segmentation problem. However, of the many variants of the watershed algorithm not all are equally well suited for hardware implementation. Different algorithms are studied and the watershed algorithm based on connected components is selected for the implementation, as it exhibits least computational complexity, good segmentation quality and can be implemented in the FPGA. It has simplified memory access compared to all other watershed based image segmentation algorithms. This thesis proposes a new hardware implementation of the selected watershed algorithm. The main aim of the thesis is to implement image segmentation algorithm in a FPGA which requires minimum hardware resources, low execution time and is suitable for use in real time applications.

A pipelined architecture of algorithm is designed, implemented in VHDL and synthesized for Xilinx Virtex-4 FPGA. In the implementation, image is loaded to external memory and algorithm is repeatedly applied to the image. To overcome the problem of over-segmentation, pre-processing step is used before the segmentation and implemented in the pipelined architecture. The pipelined architecture of pre-processing stage can be operated at up to 228 MHz. The computation time for a 512 x 512 image is about 35 to 45 ms using one pipelined segmentation unit. A proposal of parallel architecture is discussed which uses multiple segmentation units and is fast enough for the real time applications. The implemented and proposed architectures are excellent candidates to use for different applications where high speed performance is needed.

Contents

1. Introduction **1**

2. Image Segmentation and Watershed Based Approach **3**
2.1. Image Segmentation 3
2.1.1. Threshold Based Segmentation 5
2.1.2. Watershed Based Image Segmentation 6
2.2. Watershed Algorithm Based on Connected Components 12
2.2.1. Details of the Algorithm 13
2.3. Pre-Processing Stage 19
2.3.1. Median Filter 19
2.3.2. Morphological Gradient 21
2.3.3. Thresholding 26
2.4. Post-Processing Stage 27

3. Hardware Implementation **28**
3.1. Stream Cache Module 28
3.2. Pre-processing Module 30
3.2.1. Serializer 33
3.2.2. 3x3 Moving Window Architecture 34
3.2.3. Median Filter Module 36
3.2.4. Morphological Gradient and Thresholding Module 37
3.2.5. Serial-In-Parallel-Out (SIPO) Shift Register 39
3.2.6. State Machine for Pre-processing Module 39
3.3. Image Segmentation Algorithm Implementation 41
3.3.1. State Machine for Segmentation Module 47
3.4. Verification Methodology 51
3.4.1. Simulation 52

4. Performance Measurements and Synthesis Results **53**
4.1. Performance Measurements 53
4.2. Synthesis Results 59

5. Parallel Architecture Proposal **63**

6. Conclusion **66**

A. Appendix **68**

List of Figures

2.1. Image segmentation 4
2.2. Adaptive threshold based image segmentation 6
2.3. Block diagram of watershed based image segmentation 7
2.4. Watershed line and catchment basin 8
2.5. Basic concept of watershed based image segmentation 9
2.5. Basic concept of watershed based image segmentation 10
2.6. Rainfalling watershed approach 12
2.7. Basic concept of connected components approach 13
2.8. Step 1 16
2.9. Step 1 17
2.10. Step 3 18
2.11. Block diagram of pre-processing stage 19
2.12. Concept of median filter 20
2.13. Median and mean filtering 22
2.14. Concept of morphological gradient 23
2.15. Median filtered image and its morphological gradient 23
2.16. Bitonic sequence example 24
2.17. Bitonic sort - Eight elements input 25
2.18. Bitonic sort - Nine elements input 26

3.1. Block Diagram of Hardware implementation 28
3.2. Stream cache - initialization, read, write and flush operations 31
3.3. Pre-processing module - data and control signal flow 32
3.4. Pre-processing module schematic 32
3.5. Serializer module schematic 33
3.6. Moving Window 34
3.7. Implementation of 3 x 3 Window 35
3.8. 3 x 3 window module schematic 35
3.9. Median filter schematic 36
3.10. Comparator and register module schematic 37
3.11. Morphological gradient and thresholding calculation 38
3.12. Morphological gradient and thresholding module schematic 38

3.13. SIPO module schematic 39
3.14. State diagram of memory read operation for Pre-processing 40
3.15. State diagram of pre-processing module and memory write operation . 41
3.16. Block diagram of segmentation hardware architecture 42
3.17. Watershed controller module schematic 43
3.18. Segmentation module schematic 44
3.19. First approach of pipeline implementation 45
3.20. Window split diagram 46
3.21. Window generator architecture 47
3.22. State diagram of memory read operation (only for first scan using serializer) for segmentation module 48
3.23. State diagram of memory read operation (after first scan) for segmentation module 49
3.24. State diagram of memory write operation for segmentation module . . 50
3.25. Second approach of pipeline implementation 51
3.26. Verification methodology 52

4.1. Image segmentation results for pepper image 55
4.2. Image segmentation results for Lena image 56
4.3. Image segmentation for different images 57
4.3. Image segmentation for different images 58

5.1. Block diagram of parallel architecture) 64

List of Tables

3.1. Ports description of stream cache module 29

4.1. Different threshold values for pre processing stage and total number of scans for sequential implementation 54
4.2. Total number of image scans with pipeline implementation 59
4.3. Pre-processing module synthesis result 60
4.4. Image processing time for segmentation module in pipeline implementation 60
4.5. Segmentation module synthesis result (First approach of pipeline implementation) 61
4.6. Segmentation module synthesis result (second approach of pipeline implementation) 61

5.1. Segmentation units vs. Total segmentation time 64

1. Introduction

Image segmentation is process of partitioning the image into multiple segments. It is first important step in many image processing applications like image analysis, image description and recognition, image visualization and object based image compression. Image segmentation means assigning a label to each pixel in the image such that pixels with same labels share common visual characteristics. It makes an image easier to analyse in the image processing tasks. There are many different techniques available to perform image segmentation. The algorithm used in this thesis is watershed based image segmentation. It is a hybrid technique because it is the result of threshold based, edge and region based techniques using morphological watershed transform. The watershed transformation [1] is popular image segmentation technique for gray scale images. An efficient watershed algorithm based on connected components [2] shows very good results compare to other watershed based image segmentation algorithms.

In connected components based watershed image segmentation algorithm, image is scanned from top left to bottom right and from bottom right to top left. During each scan, unique labels are given to each detected regional minima. If the labels have already been given to their neighbour pixels during previous scan then those labels are copied to the pixels. Finally each component (pixel) is connected to its local minima and all components connected to same local minima make a segment.

Watershed based image segmentation produces over-segmentation based on different properties of the image. In this thesis, pre-processing of the image before image segmentation is considered to reduce the over-segmentation problem of watershed based image segmentation.

The aim of this thesis is to implement watershed based image segmentation algorithm with pre-processing in hardware for real time image processing applications. All algorithms are initially implemented in MATLAB to realize the segmentation results of connected components based watershed image segmentation with pre-processing. The pipelined architecture of connected components based watershed image segmentation is implemented in a Field Programmable Gate Array (FPGA) with pre-processing stage. Image pixels are processed by the pipeline architecture which makes possible to

give best performance on the FPGA. The key computational performance parameters are significant speed-up compared to a sequential implementation, minimum processing latency and minimum logic resources utilization. The proposed architecture is coded in VHDL and synthesized for implementation on Xilinx Virtex-4 FPGA.

The structure of the thesis is as follows: Chapter two provides basic of the image segmentation, description of the flooding based and rainfalling based watershed image segmentation approaches, and connected components based watershed image segmentation algorithm. Chapter three describes hardware implementation of pre-processing stage and connected components based watershed image segmentation algorithm. Chapter four represents performance measurement of the algorithm and synthesis results for the pre-processing and segmentation module. Proposal of parallel architecture is given in chapter five. Conclusion of the thesis is presented in the last chapter.

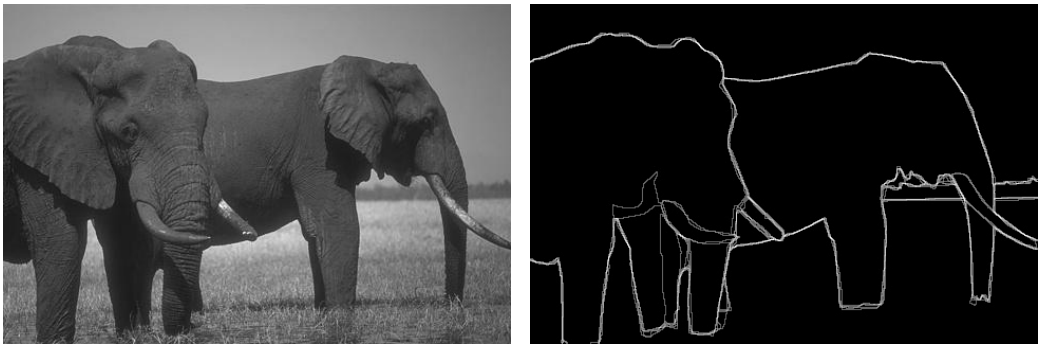
2. Image Segmentation and Watershed Based Approach

2.1. Image Segmentation

Image segmentation means division of an image into meaningful structures. It is process of extracting and representing information from the image to group pixels together with region of similarity [11]. Sonka et al. define the goal of segmentation as "to divide an image into parts that have a strong correlation with objects or areas of the real world contained in the image" [7]. Figure 2.1 shows a basic example of the image segmentation where Figure 2.1a is an original gray scale image and Figure 2.1b is a segmented image [19]. All the objects of the original image can be identified in segmented image with their boundaries. There are many techniques available for the image segmentation. Examples are, threshold based segmentation, edge based segmentation, region based segmentation, clustering based image segmentation, markov random field based segmentation and hybrid techniques. These segmentation methods differ from their computation complexity and segmentation quality. The main aim of the thesis is to find a segmentation algorithm which is feasible to implement in hardware with the minima use of hardware resources (slices/gates), gives best segmentation quality and has possibility to be used for real time image processing applications.

Computation complexity is one of the important criteria for image segmentation which should be considered carefully when real time image segmentation is required. Computational complexity is defined as number of arithmetical operations required for processing single image frame. If the segmentation algorithm is computationally more complex then it needs more computational hardware resources. The best approach should be less computational complexity, less input parameter dependency, minimum segmentation time and provide efficient segmentation output for real time applications.

Mean-shift, a clustering based segmentation gives very good segmentation results compared to all other available methods. It considers the probability density of feature



(a) Original image (b) Segmented image

Figure 2.1.: Image segmentation

vectors obtained from a given image. In mean shift segmentation approach, first mean shift filtering of the original image data and then subsequent clustering of filtered image data are performed. It is computationally very expensive method because it requires to apply mean shift to each single pixel [8].

Normalized cut is a graph partitioning approach for image segmentation. The image is considered as a weighted undirected graph. This method uses similarity among data elements of a group and dissimilarity among different groups for image segmentation. The eigenvalue method used to compute normalized cut is also computational complex [13] and inefficient in hardware implementation.

Threshold based image segmentation algorithms are mathematically less complex and one of the algorithmic approach is discussed in the section 2.1.1 with advantages and disadvantages.

Watershed based image segmentation algorithms are less computational complex and provide very good segmentation results. It is possible to implement in the hardware using pipelined and/or parallel architecture for real time applications because of the independent mathematical computations flow of the algorithms. This method is explained in detail in section 2.1.2.

2.1.1. Threshold Based Segmentation

Thresholding method for the image segmentation is based on partitioning the image into different regions according to the intensity value of the image pixels and their local properties. It is the simplest procedure to perform image segmentation. The basic principle of thresholding technique is to choose optimal threshold value to divide pixels in different classes and differentiate the object from background. When there is only a single threshold value for segmentation then any point on image for which pixel value greater than this threshold value is called as object point, otherwise called as background point.

There are mainly three types of thresholding techniques known as global, local and dynamic thresholding. Threshold value depends only on gray level value of the pixel in global thresholding where as in local thresholding, threshold value depends on gray level value of the pixel as well as local properties of the pixel like average gray value of the neighbourhood. Threshold value is calculated by local properties, pixel intensity and spatial coordinates in dynamic or adaptive thresholding. Thresholding based image segmentation is influenced by the image illumination and noise. Dynamic or adaptive thresholding gives very good segmentation results compared to two other methods.

Adaptive threshold changes threshold value dynamically over the image. The threshold value is calculated for each pixel and if this calculated value is lower than threshold value then given pixel is part of the background object otherwise it is part of the foreground object. The best method to compute adaptive threshold value is to consider neighbourhood pixels for a given pixel. Threshold value is calculated for each pixel based on relationship with its neighbourhood pixels. Mean of the intensity values of the neighbourhood pixel is used to find the threshold value for each pixel which is used to decide, whether to consider a given pixel as a background or a foreground point [12].

This method is very easy to implement in the software as well as in the hardware because it has very less computational complexity, but drawback of this method is bad segmentation quality. It is implemented in MATLAB to check image segmentation quality. One image segmentation example is shown in Figure 2.2 based on adaptive thresholding method. Images with even illumination give good segmentation results with this technique, but this segmentation approach does not give good segmentation results for images with uneven illumination. It also depends on the window size and threshold value for each image. It uses very less spatial information for image segmentation and does not give guarantee of object coherency [12]. Another disadvantage is

(a) Orignal pepper image (256 x 256) (b) Segmented image

Figure 2.2.: Adaptive threshold based image segmentation

threshold parameter which is not easy to adjust automatically for different types of the image.

2.1.2. Watershed Based Image Segmentation

Watershed transformation also called, as watershed method is a powerful mathematical morphological tool for the image segmentation. It is more popular in the fields like biomedical and medical image processing, and computer vision [4]. In geography, watershed means the ridge that divides areas drained by different river systems. If image is viewed as geological landscape, the watershed lines determines boundaries which separates image regions. The watershed transform computes catchment basins and ridgelines (also known as watershed lines), where catchment basins corresponding to image regions and ridgelines relating to region boundaries [5]. Segmentation by watershed embodies many of the concepts of the three techniques such as threshold based, edge based and region based segmentation.

Watershed algorithms based on watershed transformation have mainly two classes. The first class contains the flooding based watershed algorithms and it is a traditional approach where as the second class contains rainfalling based watershed algorithms. Many algorithms have been proposed in both classes but connected components based

watershed algorithm [2] shows very good performance compared to all others. It comes under the rainfalling based watershed algorithm approach. It gives very good segmentation results, and meets the criteria of less computational complexity for hardware implementation.

Block Diagram of Watershed Based Segmentation

There are mainly three stages as indicated by Figure 2.3 for watershed based image segmentation approach. First stage is defined as pre-processing, second stage as watershed based image segmentation and last stage as post-processing. Input image is first processed by the pre-processing stage, and then given to watershed based segmentation stage. The resulting image is post processed by the final stage to get a segmented image. Pre-processing and post-processing are necessary to overcome the problem of over-segmentation in watershed based image segmentation. Pre-processing stage is discussed in detail in this chapter. Overview of the post processing stage is given in last section but it is not used for software and hardware implementation in this thesis.

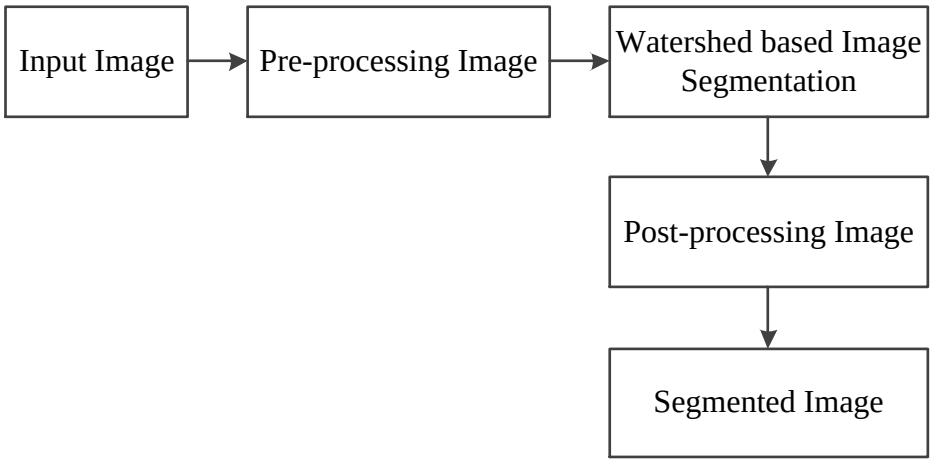


Figure 2.3.: Block diagram of watershed based image segmentation

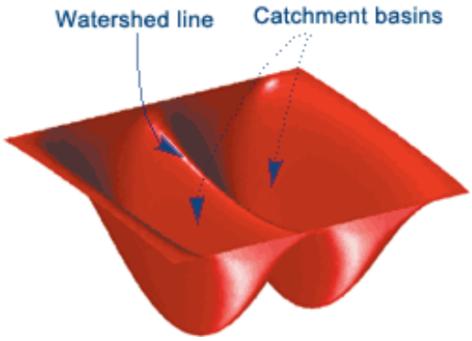


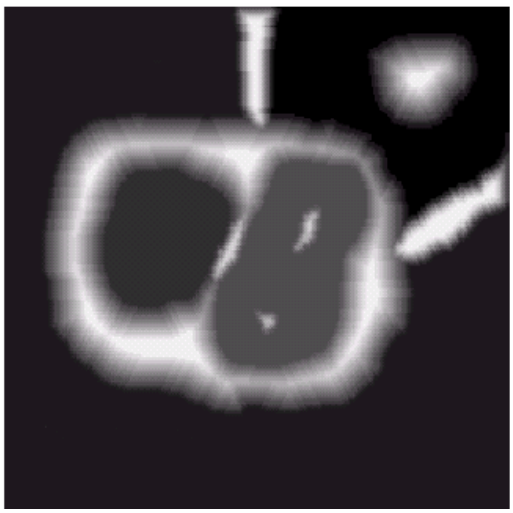
Figure 2.4.: Watershed lines and catchment basin [20]

Flooding Based Watershed Algorithms

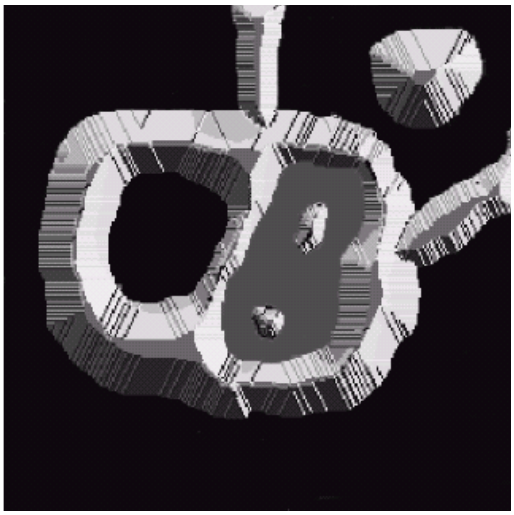
In traditional flooding based approach of watershed based image segmentation, image is considered as a topographic surface which contains three different types of points: (i) points which indicate regional minimum, (ii) points where the water falling has highest probability to fall into a single minimum region and (iii) points where the water falling has probability to fall into more than one such a minimum region. For regional minimum, the groups of points satisfy second condition called watershed or catchment basin of that minimum and the groups of point satisfy third condition makes a crest line on topographic surface termed as a watershed line. Fig. 2.4 shows an example of the watershed line and catchment basin.

Basic Concept The basic concept of watershed algorithm used for the image segmentation is to find the watershed lines. Imagine, holes at each regional minimum, and water is flooded from bottom into these holes with constant rate. Water level will rise in the topographic surface uniformly. When the rising water in different catchment basins is going to merge with nearby catchment basins then a dam is built to prevent all merging of the water. Flooding of water will reach at the point when only top of the dams are visible above water line. These continuous dam boundaries are the watershed lines.

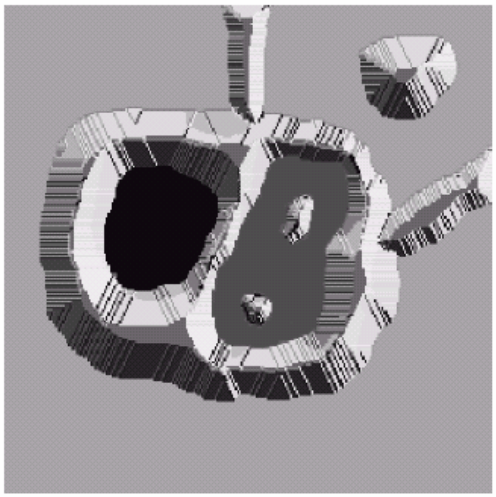
To understand this traditional concept more clearly, a simple gray scale image and its topographic surface representation are shown in Figure 2.5a and 2.5b. The height of the topographic surface is proportional to gray level values of the given image. The maximum height of the topographic mountain is similar to the maximum gray level value of the image. The perimeter of entire mountain is covered by dams which have a



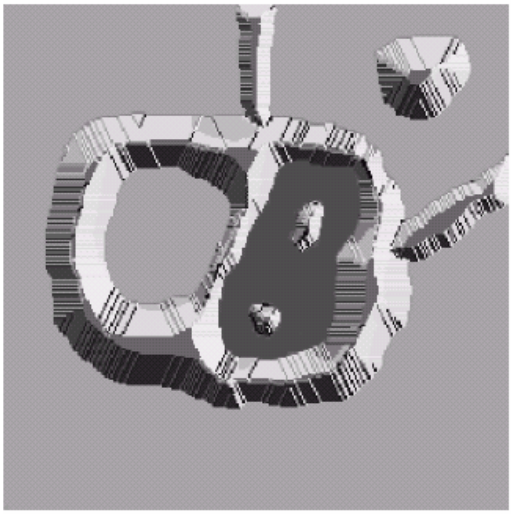
(a) Original image



(b) Topographic view

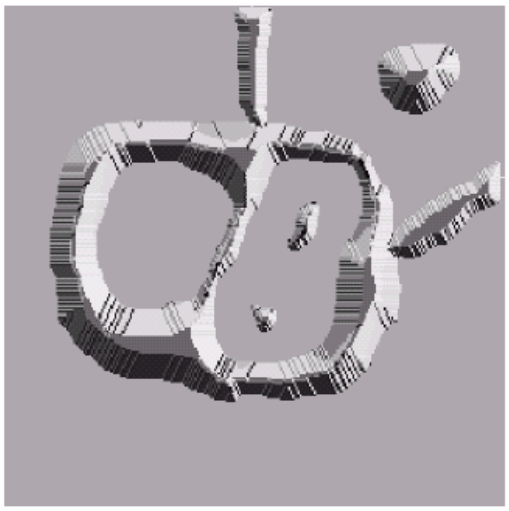


(c) First stage of flooding

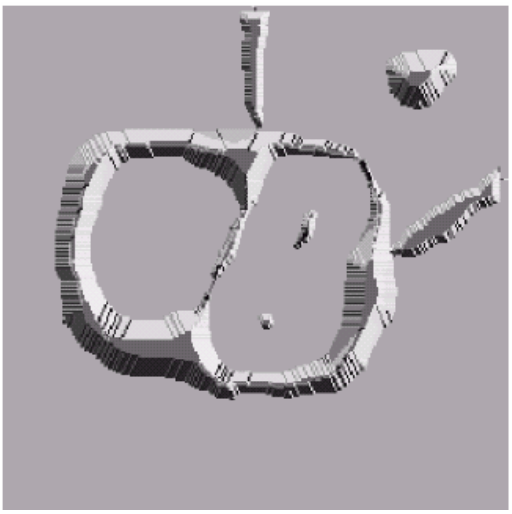


(d) Second stage of flooding

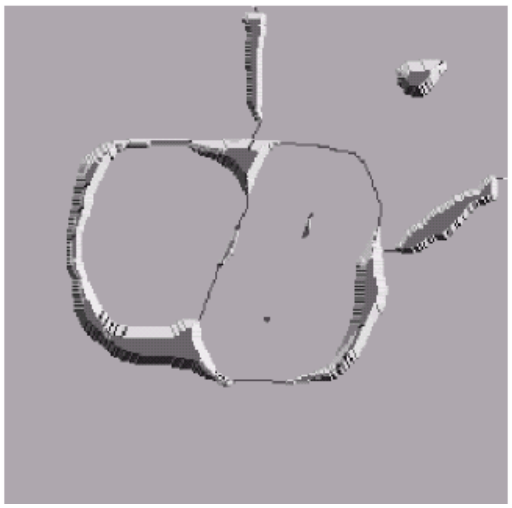
Figure 2.5.: Basic concept of watershed based image segmentation [18]



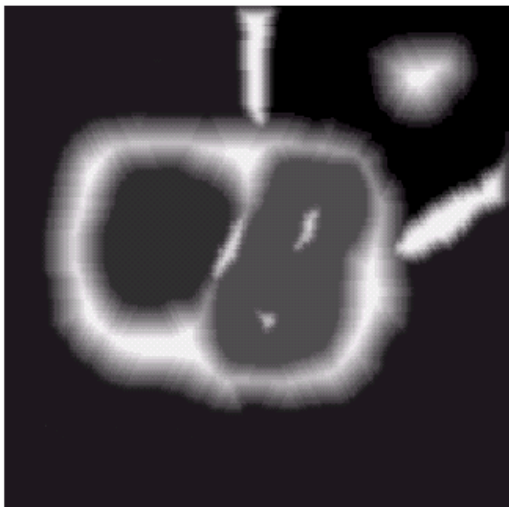
(e) Result of further flooding



(f) Beginning of merging of water from two catchment basins (a short dam was built between them)



(g) Longer dams



(h) Final watershed (segmentation lines)

Figure 2.5.: *(Continued)* Basic concept of watershed based image segmentation [18]

height greater than the maximum height of the mountain. Assume that each regional minimum have some holes and the water is flooded into each regional minimum from bottom holes and rise with constant rate towards top of the mountain.

Figure 2.5c shows that water has covered dark background of the image, and it has also reached to the surface of first catchment basin in Figure 2.5d, and the second catchment basin in Figure 2.5e, because of continuous rise of the water. The water will overflow from left catchment basin to the right because of the constant water flooding which is indicated in the Figure 2.5f. The dam is built to prevent the water flowing from one catchment basin to another. Figure 2.5g shows longer dams between the left and right catchment basin as well as another with the top right catchment basin because of uniform flooding. Flooding of water needs to stop when water level rises up to the maximum mountain height (maximum gray level value in image). Final dams indicate watershed lines and provide the image segmentation. The segmentation result is shown in Figure 2.5h with superimposed of the original image to the final dams image.

Rainfalling Watershed Algorithms

The rainfalling algorithm exploits slightly different concept to extract mountain boundaries than traditional flooding based algorithm. Rainy water drops fall on the mountain (topographic surface) and move to descending direction because of the gravity until they reach to the local minimum surface. The algorithm tracks the path of water drop for each point on the surface towards the local minimum, if rain drops pass through that point or fall on that point. All points make a segment when water drops related to them flow downwards to the same deepest location. When a point has more than one path towards the different steepest surfaces then it can be allocated to any one of the local minimum.

The drowning threshold is used to suppress the lowest mountain (the weakest edges in image). Mountains are not considered if their heights come under the drowning threshold value. The drowning threshold line can be seen on right side of the Figure 2.6 and the topographic surface of the middle mountain is not considered as a peak but as a part of the local minimum.

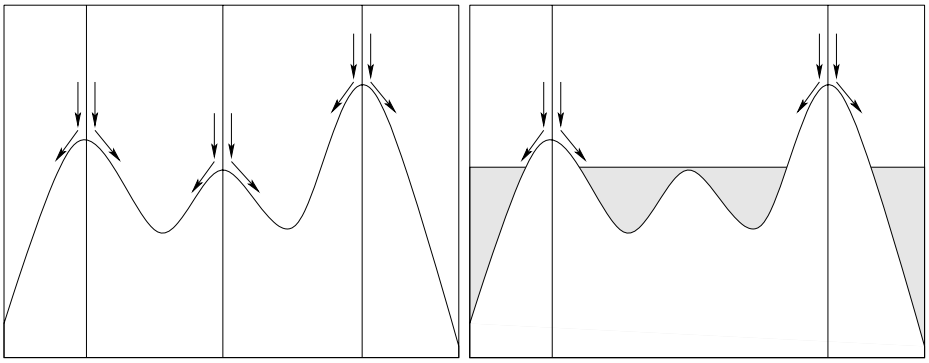


Figure 2.6.: Rainfalling watershed approach [17]

2.2. Watershed Algorithm Based on Connected Components

The traditional watershed based algorithm [1] uses hierarchical queue for flooding process. This queue requires non uniform memory access sequences and complexity of the traditional algorithm becomes very high because of the need to manage a hierarchical queue. An efficient watershed algorithm based on connected components was developed by A. Bieniek and A.Moga [2]. It does not require a hierarchical queue as a traditional algorithm implementation. This algorithm gives the same segmentation results as a traditional watershed algorithm and it has an advantage of lower complexity, simple data structure and short execution time. It connects each pixel to its lowest neighbour pixel and all pixels connect to same lowest neighbour pixel, make a segment. It uses FIFO queue and stack to perform the same functionality as a hierarchical queue of the traditional algorithm. The disadvantages of this algorithm for hardware implementations are the requirement of extra accesses of FIFO queue and stack, which are very difficult and inefficient to implement in the FPGA. This algorithm is slightly modified to simplify the memory access by T.Maruyama and D.Trieu [3] and it is used in this thesis too.

The basic concept of connected components based algorithm is explained by Figure 2.7. The original 6 x 6 image has three local minimum values indicated by gray boxes. If the component (pixel) is not a local minimum then it is connected to its lowest neighbours as shown by arrows in Figure 2.7b, where m indicates a local minimum. All components directed towards the same local minimum make a segment and are given a same label value in Figure 2.7c

7	4	8	12	11	3
7	7	8	12	11	7
13	13	15	16	16	13
19	19	18	17	15	7
20	18	17	16	15	5

(a) The original image

→	m	←	←	→	m
↗	↑	↖	←	↗	↑
↑	↑	↖	↖	↗	↑
↑	↑	↑	→	↘	↓
→	→	→	→	→	m

(b) Each pixel connect to lowest minimum

0	0	0	0	1	1
0	0	0	0	1	1
0	0	0	0	1	1
0	0	0	2	2	2
2	2	2	2	2	2

(c) The Image with labels

Figure 2.7.: Basic concept of connected components approach

2.2.1. Details of the Algorithm

The pseudo-code of connected components based watershed algorithm [3] is shown in the algorithm 1. In algorithm 1, p represents a pixel, f is the input pre-processed (processed by filter and morphological gradient) image, l is the segmented label image, *f(p)* means the gray level value of p, n is the neighbour pixel of p, *f(n)* represents gray level value of the respective neighbour pixel. The array *l[p]* is used to store the labels and *v[p]* is used to store a distance from lowest plateau or pixels. *LMAX* and *VMAX* denote maximum value for label and maximum distance in the system respectively. *VMAX* defines distance between first pixel of the first row to last pixel of the last row. *Scan_Step2* and *Scan_Step3* are used to decide whether to continue or to stop image scan for step 2 and step 3 respectively. To discuss more clearly, the same sample input image mentioned by Bieniek and Moga [2] is used for explanation of the algorithm.

Step 1

The lowest neighbourhood of each pixel is found in step 1. Initially, array *v[p]* has zero value for all array elements. As shown in Figure 2.8, Input image scan from top left to bottom right, and *v[p]* is set to '0' if it is lower than or equal to neighbourhood values otherwise set it to '1'. As indicated by gray box in Figure 2.8, gray level value '2' is the lowest in all neighbourhood and *v[p]* set to '0' for it. The gray value '5' has an equal neighbourhood so *v[p]* is set to '0' for it. The small plateau with three values of '8' is coloured with light gray in Figure 2.8a.

Algorithm 1 The pseudo-code of the algorithm

```
1: Input : f , Output : l
2: v[p] ← 0, l[p] ← 0, New_label ← 0, Scan_Step2 ← 1, Scan_Step3 ← 1 // Initial-
   ization
3: Scan from top left to bottom right : STEP1(p)
4: while Scan_Step2 = 1 do
5:     Scan image from top left to bottom right : STEP2(p)
6:     if v[p] is not changed then
7:         Scan_Step2 ← 0
8:     else
9:         Scan image from bottom right to top left : STEP2(p)
10:        if v[p] is not changed then
11:            Scan_Step2 ← 0
12:        end if
13:    end if
14: end while
15: while Scan_Step3 = 1 do
16:    Scan image from top left to bottom right : STEP3(p)
17:    if l[p] is not changed then
18:        Scan_Step3 ← 0
19:    else
20:        Scan image from bottom right to top left : STEP3(p)
21:        if l[p] is not changed then
22:            Scan_Step3 ← 0
23:        end if
24:    end if
25: end while
26: function STEP1(p)
27:    if v[p] ≠ 1 then
28:        for each n of p // n is neighbour pixel of p
29:        if f[n] < f(p) then v[p] ← 1
30:        end if
31:    end if
32: end function
```

```
33: function STEP2(p)
34:     if v[p]≠ 1  then
35:         min ← VMAX, for each n of p // n is neighbour pixel of p
36:         if f(n) = f(p) and v[n] > 0 and v[n] < min then min ← v[n]
37:         end if
38:         if min ≠ VMAX and v[p] ≠ (min+1) then v[p] ← min+1
39:         end if
40:     end if
41: end function
42: function STEP3(p)
43:     lmin ← LMAX, fmin ← f(p)
44:     if v[p] = 0  then
45:         for each n of p
46:         if f(n) = f(p) and l[n] > 0 and l[n] < lmin  then lmin ← l[n]
47:         end if
48:         if lmin = LMAX and l[p] = 0 then lmin ← New_label + 1
49:         end if
50:     else if v[p] = 1  then
51:         for each n of p
52:         if f(n) < fmin then fmin ← f[n]
53:         end if
54:         for each n of p
55:         if f(n) = fmin and l[n] > 0 and l[n] < lmin then lmin ← l[n]
56:         end if
57:     else
58:         for each n of p
59:         if f(n) = f(p) and v[n] = v[p] − 1 and l[n] > 0 and l[n] < lmin then
60:             lmin ← l[n]
61:         end if
62:     end if
63:     if lmin ≠ LMAX and l(n) ≠ lmin then l[p] ← lmin
64:     end if
65: end function
```

3	5	5	2	8	8	8	11	10	10
5	5	11	11	8	11	11	8	10	10
11	5	11	11	9	9	9	9	8	10
11	11	11	7	7	7	7	9	9	8
11	11	11	11	11	9	7	10	8	10
11	10	11	9	7	7	9	9	10	8
11	10	11	9	11	9	10	10	8	10
11	11	11	8	8	8	8	8	10	10
11	11	11	11	10	10	10	10	10	10
10	10	10	10	10	10	10	10	10	10
11	11	11	11	10	10	10	10	10	10

(a) Input image

0	1	1	0	1	0	0	1	1	0
1	1	1	1	1	1	1	0	1	1
1	0	1	1	1	1	1	1	0	1
1	1	1	0	0	0	0	1	1	0
1	1	1	1	1	1	0	1	0	1
1	0	1	1	0	0	1	1	1	0
1	0	1	1	1	1	1	1	0	1
1	1	1	0	0	0	0	0	1	1
1	1	1	1	1	1	1	1	1	0
0	0	0	0	0	0	0	0	0	0
1	1	1	1	0	0	0	0	0	0

(b) v[p]: Downward scan 1

Figure 2.8.: Step 1

Step 2

The fundamental of step 2 is that if pixel is on a plateau and its neighbouring point to one of the local minimum then the pixel points to its neighbour. To realize this fundamental, all pixels with *v[p]* not equal to 1 and have neighbour pixels on same plateau with ***v[p]* set to 1** in the step 1 are considered, then shortest distance for each pixel on plateau respect to non zero *v[p]* of neighbour plateau is calculated. In Figure 2.9a, all gray boxes are on same plateau. Here, value '2' is directed to its left neighbour '1', '3' is directed to its left neighbour '2', '4' is directed to its neighbour '3' and vice versa. During the upward scan in Figure 2.9b, value '11' directed to '10', '12' is directed to '11'and vice versa. The downward and upward scans continue until all the shortest plateau distances are calculated. There are only two scans required for this example to finish step 2. The assigned value of the distance d during the downward scan may be overwritten in the upward scan when the value of distance d of a previous scan is higher than lowest plateau located in the downward region.

Step 3

The labels are assigned to the array *l[p]* in this step. All elements of *l[p]* are initialized with zero. Labels are first given to the pixels on local minima plateau whose *v[p]* is zero, if their neighbourhood pixels with the same gray value have not been assigned

0	1	1	0	1	2	3	1	1	2
1	1	1	1	1	1	1	4	1	1
1	2	1	1	1	1	1	1	5	1
1	1	1	0	0	0	0	1	1	6
1	1	1	1	1	1	0	1	7	1
1	0	1	1	0	0	1	1	1	8
1	0	1	1	1	1	1	1	9	1
1	1	1	0	0	0	0	10	1	1
1	1	1	1	1	1	1	1	1	2
0	0	0	2	2	2	2	2	2	2
1	1	1	1	3	3	3	3	3	3

(a) v[p]: Downward scan 1

0	1	1	0	1	2	3	1	1	2
1	1	1	1	1	1	1	4	1	1
1	2	1	1	1	1	1	1	5	1
1	1	1	0	0	0	0	1	1	6
1	1	1	1	1	1	0	1	7	1
1	0	1	1	0	0	1	1	1	8
1	0	1	1	1	1	1	1	9	1
1	1	1	14	13	12	11	10	1	1
1	1	1	1	1	1	1	1	1	2
5	4	3	2	2	2	2	2	2	2
1	1	1	1	3	3	3	3	3	3

(b) v[p]: Upward scan 1

Figure 2.9.: Step 2

any labels yet. These labels are propagated to their neighbourhood pixels according to values of *v[p]* to create a region until this region has centre of the local minima. Similar regions are created for all other local minima with the same procedure. There may be possibility for assignment of different labels on same local minimum plateau but they are overwritten in subsequent scans.

For example, Figure 2.10a refers to the first downward scan and gray boxes indicate new labels assignment. New labels propagate to the neighbourhood elements according to the values of *v[p]*. As shown by gray boxes in Figure 2.10b, labels are delivered to the upward direction. Label '3' (in bold letter) denotes that label '5' was assigned that location in previous scan but it is overwritten by the correct value in this scan. All next subsequent figures show label propagation by gray boxes. In Figure 2.10f, no labels are modified compared to the previous scan and hence no more scans are needed. Total six scans are required in the step 3 for the given sample input image.

1	1	0	2	2	2	2	2	0	0
1	1	2	2	2	2	2	2	2	0
1	1	1	0	0	0	0	0	2	2
1	1	1	3	3	3	3	3	2	2
0	0	3	3	3	3	3	3	2	2
0	4	0	0	5	3	3	3	2	2
4	4	0	5	3	3	3	0	2	2
4	4	0	0	0	0	0	2	2	2
0	0	0	0	0	0	2	2	2	2
0	0	0	0	0	2	2	2	2	2
0	0	0	0	2	2	2	2	2	2

(a) Scan 1 of l[p]: Downward scan 1

1	1	2	2	1	2	2	2	2	2
1	1	2	2	2	2	2	2	2	2
1	1	1	3	3	3	3	3	2	2
1	1	1	3	3	3	3	3	2	2
4	4	3	3	3	3	3	3	2	2
4	4	3	3	**3**	3	3	3	2	2
4	4	2	5	3	3	3	2	2	2
4	4	2	2	2	2	2	2	2	2
0	0	0	0	0	0	2	2	2	2
0	0	0	0	0	2	2	2	2	2
0	0	0	2	2	2	2	2	2	2

(b) l[p]: Upward scan 1

1	1	2	2	1	2	2	2	2	2
1	1	2	2	2	2	2	2	2	2
1	1	1	3	3	3	3	3	2	2
1	1	1	3	3	3	3	3	2	2
4	4	3	3	3	3	3	3	2	2
4	4	3	3	3	3	3	3	2	2
4	4	2	3	3	3	3	2	2	2
4	4	2	2	2	2	2	2	2	2
0	0	2	2	2	2	2	2	2	2
0	0	0	2	2	2	2	2	2	2
0	0	2	2	2	2	2	2	2	2

(c) l[p]: Downward scan 2

1	1	2	2	1	2	2	2	2	2
1	1	2	2	2	2	2	2	2	2
1	1	1	3	3	3	3	3	2	2
1	1	1	3	3	3	3	3	2	2
4	4	3	3	3	3	3	3	2	2
4	4	3	3	3	3	3	3	2	2
4	4	2	3	3	3	3	2	2	2
4	4	2	2	2	2	2	2	2	2
2	2	2	2	2	2	2	2	2	2
2	2	2	2	2	2	2	2	2	2
0	0	2	2	2	2	2	2	2	2

(d) l[p]: Upward scan 2

1	1	2	2	1	2	2	2	2	2
1	1	2	2	2	2	2	2	2	2
1	1	1	3	3	3	3	3	2	2
1	1	1	3	3	3	3	3	2	2
4	4	3	3	3	3	3	3	2	2
4	4	3	3	3	3	3	3	2	2
4	4	2	3	3	3	3	2	2	2
4	4	2	2	2	2	2	2	2	2
2	2	2	2	2	2	2	2	2	2
2	2	2	2	2	2	2	2	2	2
2	2	2	2	2	2	2	2	2	2

(e) l[p]: Downward scan 3

1	1	2	2	2	2	2	2	2	2
1	1	2	2	2	2	2	2	2	2
1	1	1	3	3	3	3	3	2	2
1	1	1	3	3	3	3	3	2	2
4	4	3	3	3	3	3	3	2	2
4	4	3	3	3	3	3	3	2	2
4	4	2	3	3	3	3	2	2	2
4	4	2	2	2	2	2	2	2	2
2	2	2	2	2	2	2	2	2	2
2	2	2	2	2	2	2	2	2	2
2	2	2	2	2	2	2	2	2	2

(f) l[p]: Upward scan 3

Figure 2.10.: Step 3

2.3. Pre-Processing Stage

The watershed based image segmentation produces mostly an over-segmentation of the image. Pre-processing and post-processing of an image is performed to overcome this problem. Pre-processing is mainly applied to the image before the watershed segmentation. As shown in Figure 2.11, pre-processing includes first stage of noise removal using median filter, second stage of morphological gradient calculation and last stage of thresholding a gradient image.

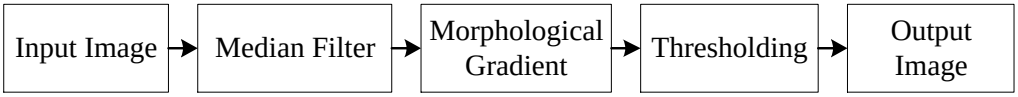


Figure 2.11.: Block diagram of pre-processing stage

In principle, Image noise is defined as distinct pixels which are not similar in appearance with the neighbourhood pixels. Over-segmentation occurs mainly due to presence of the noise and unimportant fluctuation which produces non real minima. Main objective of the pre-processing stage is to smooth the original image by removing the noise effect.

2.3.1. Median Filter

Impulse noise is most common noise in image processing. It generally occurs due to malfunctioning pixels in camera sensors, faulty memory location in hardware or error in data transmission [9]. There are mainly two types of impulse noise, one is the salt and pepper noise (also known as speckle noise) and second is the random value shot noise. Noisy pixel only takes maximum or minimum values in case of salt and pepper noise where as it takes arbitrary value in case of random value shot noise.

Median filter, also known as an edge preserving non linear filter is a simple and effective method to remove impulse noise. Median filter considers each pixel in the image and checks its nearby neighbour pixel to decide either it is representative of its neighbourhood pixels. It replaces pixel value with the median of the neighbourhood pixels. It is calculated by numerical shorting of all neighbourhood values and replace considered pixel with the middle value of nine sorted values. Figure 2.12 shows the value 200 is unrepresentative of neighbourhood and replaced with final median value

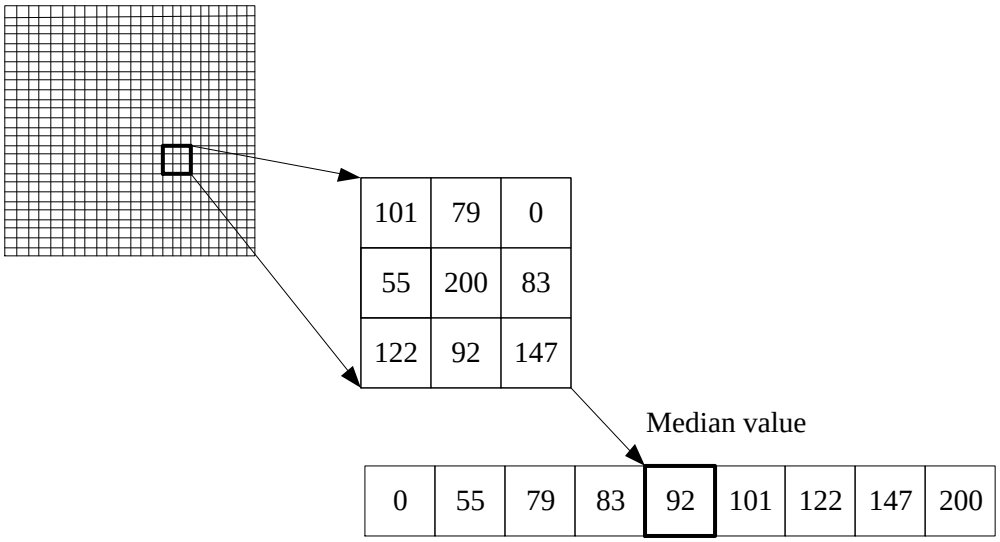


Figure 2.12.: Concept of median filter

of 92 which is middle value of the sorted neighbourhood.

Median filter is more effective and robust than mean or average filter because a single unrepresentative pixel value in neighbourhood affects very less to the median value. Median filter gives one of a neighbour value as an output pixel and hence it does not create new unrealistic values near the edges and preserves sharp edges. It is mathematically expensive to calculate the median value because it requires sorting of nine values for each pixel.

The *medfilt2* function implements median filtering in MATLAB. The filtered images

```
I = imread('p256.png'); % Read in the image
J = imnoise(I,'salt & pepper',0.02); % Add noise to the image
K = filter2(fspecial('average',3),J)/255; % Filter the noisy
    image with Mean filter
L = medfilt2(J,[3 3]); % Filter the noisy image with Median
    filter
```

Listing 2.1: Mean and Median Filter in MATLAB

```
I = imread(`pepper.png'); % Read in the image
L = medfilt2(J,[3 3]); %  Median filtering the image
f = @(x) ((max(x(:))-min(x(:)))); % function to calculate (max-
    min) of 3x3 window
gradmag_temp = nlfilter(L,[3 3],f); % Gadient caluclation: apply
     function f on each 3x3 window
gradmag = floor(gradmag_temp); % Round to nearest integer
    towards minus infinity
```

Listing 2.2: Morphological gradient calculation in MATLAB

with median and mean filter are shown in Figure 2.13 for comparison. Both filter use 3 x 3 size of neighbourhood for filtering and their MATLAB code is given below for the reference. The salt and pepper noise is added to the noiseless pepper image. It can be easily noticed from the resulted images that median filter performs better than the mean filter, and it blurs the image edges very less.

2.3.2. Morphological Gradient

The morphological gradient is a powerful tool for an edge detection. At the second stage of the pre-processing, the morphological gradient of the filtered image is computed to overcome over-segmentation problem. When the morphological transition is applied to the gray scale image, it returns to high values when sudden transitions in gray level values (along the object edges) are detected, and returns to low values if neighbourhood pixels are similar. The Watershed transform is then applied to the gradient image so that boundaries of the catchment basin could be located on high gradient points. This operator can perform well only when noise level is effectively reduced before it is applied.

The morphological gradient is calculated by taking 3 x 3 neighbourhood window of the given pixel, then the difference between the maximum (dilation) and minimum (erosion) gray level value of the neighbourhood is calculated [6]. Calculated gradient value is rounded to the nearest integer. Figure 2.15 shows, original pepper image filtered with the median filter and its morphological gradient image. MATLAB code is given for morphological gradient calculation in Listing 2.2.

(a) Noiseless pepper image (256x256 pixels)

(b) Noisy image

(c) Mean filtered image

(d) Median filtered image

Figure 2.13.: Median and mean filtering

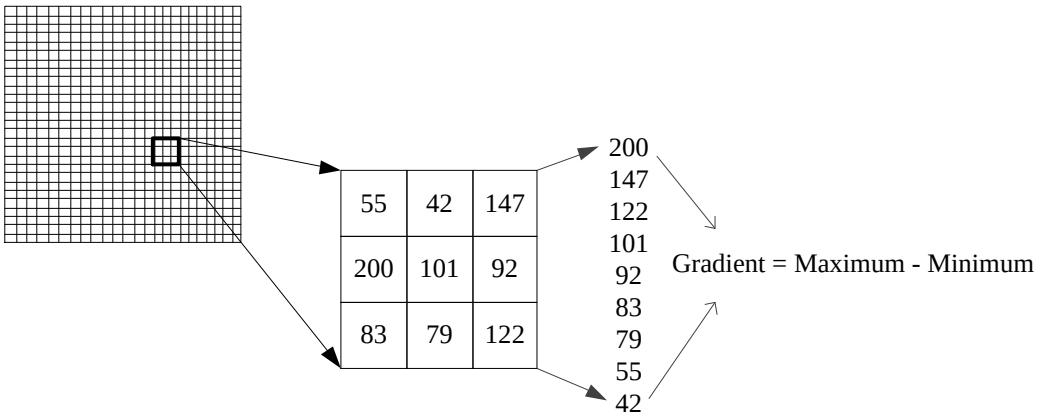


Figure 2.14.: Concept of morphological gradient

(a) Median filtered pepper image (256x256 pixels) (b) Morphological gradient image

Figure 2.15.: Median filtered image and its morphological gradient

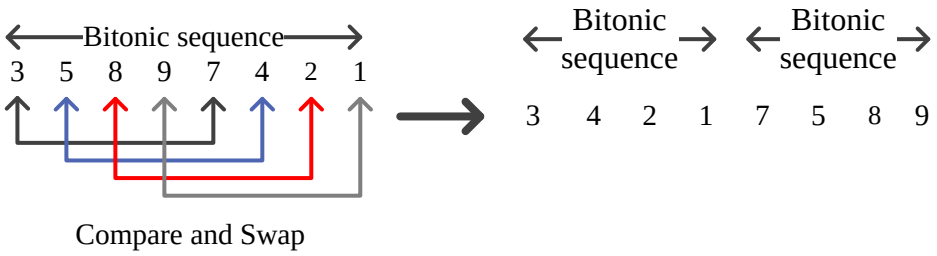


Figure 2.16.: Bitonic sequence example

Sorting Networks

Sorting networks (SN) are used to sort the numbers. For median filter and morphological gradient calculation, nine values sorting network is necessary. Sorting network is defined as a network of elementary operations denoted as *compare&swap*(CS) or comparators that sort all the elements [9]. As an example, two elements p and q are compared and exchanged if necessary for sorting sequence. Number of elements decide requirement of total number of comparators in SN.

The selection of sorting network is mainly considered for FPGA implementation. There are mainly two important criteria that needs to be considered, one is requirements of hardware resources and another one is latency. Latency of SN means the numbers sequential execution of number of CS operations, and requirement of hardware resources means number of comparators and registers required for sorting given number of elements.

Bitonic SN and odd-even merge sort are tow best sorting networks fulfil required criteria for the hardware implementation. For sorting nine elements, both of them use nearly same hardware resources and have a same computation latency [9]. Bitonic sorter is described in this section and it is used for hardware implementation in chapter three.

Bitonic sequence consists of two sub-sequences, one is monotonically increasing and another one is monotonically decreasing. The basic concept is to divide input bitonic sequence into two equal halves and compare and swap each item on the first half with respective item in the second half as shown the Figure 2.16. Smaller numbers of both pairs are moved to left and larger numbers are moved to right side. For given bitonic

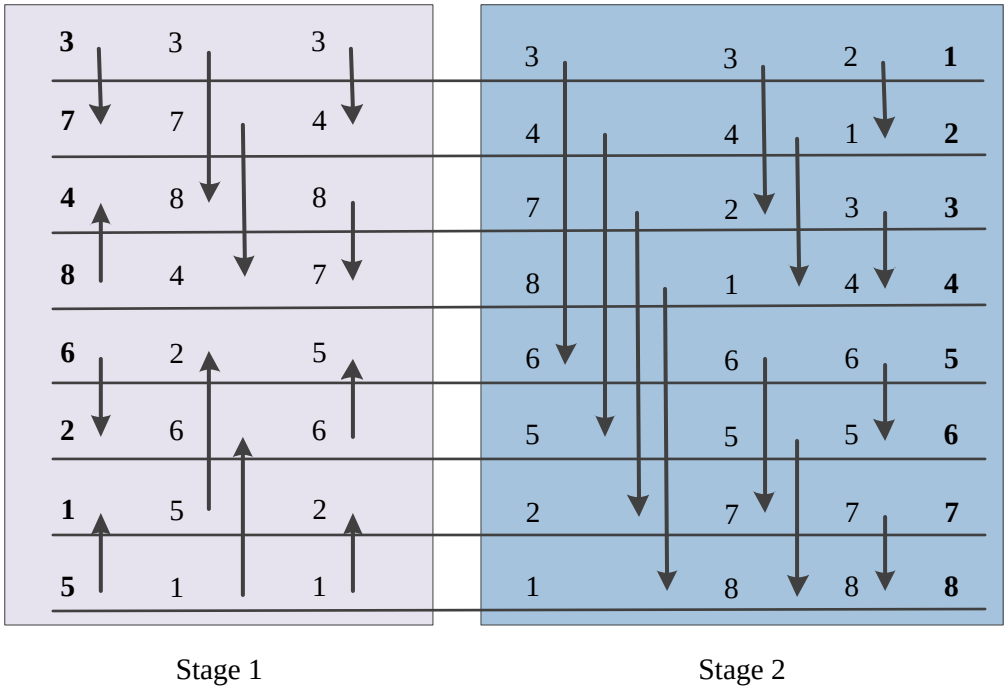


Figure 2.17.: Bitonic sort - Eight elements input

sequence, sorted sequence is obtained by recursively applying binary split as shown in stage 2 of Figure 2.17.

If the input sequence is an unordered sequence then first bitonic sequence is created by applying compare and swap operation to pairs of adjacent numbers which creates increasing and decreasing sequence. Bitonic sequence of twice size is created by merging sorted pairs as shown in the sorting example. In Figure 2.17, stage 1 is used to create bitonic sequence from unordered sequence and stage 3 describes bitonic sorting approach for sequence of eight elements.

There are total of nine elements which need to be sorted for the median and gradient calculation. Structure of nine elements bitonic sorting network [9] is shown in Figure 2.18. Stages 6 to 8 remain same as the stage 2 of eight input elements, but stages 1 to 5 are modified because of the odd numbers (nine inputs) of input elements.

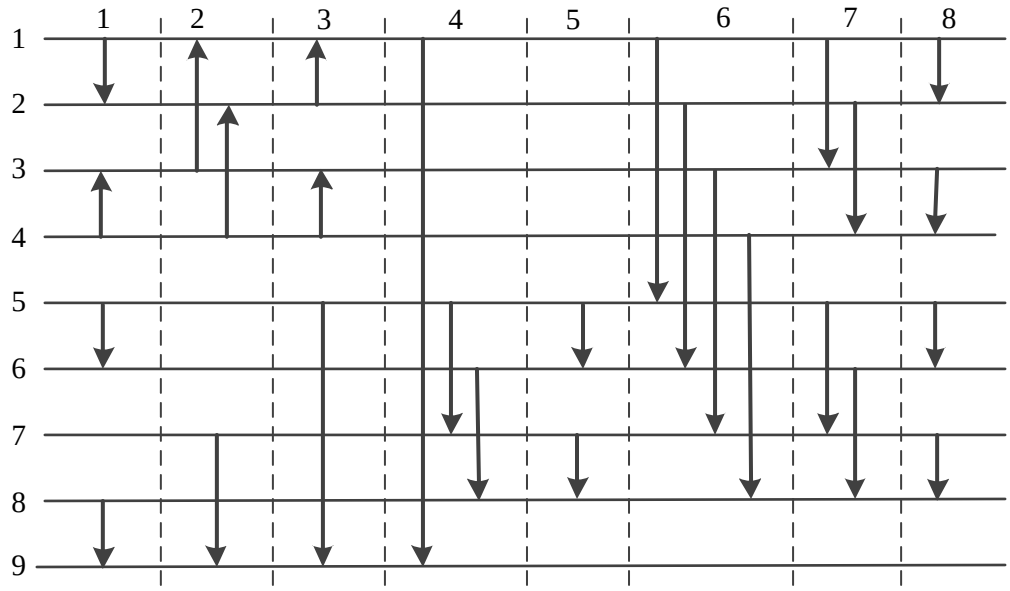


Figure 2.18.: Bitonic sort - Nine elements input and arrow indicates the position of the maximum of the two inputs

2.3.3. Thresholding

The classical approach to get an edge image is to threshold the gradient image. The main objective of third stage in the pre-processing is to reduce the over-segmentation as much as possible. The threshold value is set for the local minimum to prevent creation of large number of catchment basins. In this stage, all the gradient values lower than this threshold value are set as a local minimum and watershed starts from this local minimum. Thresholding also removes small variations within the homogeneous region

This thresholding fundamental is also explained by the drawling threshold in Figure 2.6. It is not easy to find an appropriate threshold value for each respective image. If the threshold value is very low then edges become very wide and if the threshold value is too high then edges may not be detected. The optimal threshold value can be selected by iterative method for connected components based watershed image segmentation. First some initial low threshold value is selected, and check the number of labels used by algorithm, and visualize the image segmentation results and then

threshold value is iterated for the optimal result. The number of labels are used by the algorithm depends on number of segments in the image. Number of segments can be controlled by varying the threshold value and optimum segmentation result can be obtained. For thresholding a given image, all the gradient values of the image pixels are divided by the threshold value, so all the values lower than the threshold value become zero and other values are scaled down relative to the threshold value.

2.4. Post-Processing Stage

The segmented image still has some over-segmentation after the pre-processing and segmentation. Small segments can be merged using different rules to increase the segmentation quality. The region merging method [6] is used for post processing which is developed by Haris et al. Post processing stage is not used in this thesis for hardware implementation and only short description is given in this section.

The basic concept of the region merging method is to merge most similar pair of adjacent regions. Nearest neighbour graph (NNG) is used to store the similarities between the adjacent pairs. Similarity function is given by,

$$\begin{aligned}\delta(R_i, R_j) &= \frac{A_{R_i} \times A_{R_j}}{A_{R_i} + A_{R_j}} \times |\mu_{R_i} - \mu_{R_j}|;\ If\ R_i\ and\ R_j\ are\ adjacent \\ &= \infty\ ;\ otherwise\end{aligned}$$

A_{R_i}, A_{R_i} are the areas of region R_i and R_j, μ_{R_i} and μ_{R_j} are the mean values of region R_i and R_j. Most similar pairs have smaller value of δ. Threshold value of similarity degree is used to decide the number of regions to be merged. All adjacent regions with the similarity function value lower than the threshold value are merged together.

3. Hardware Implementation

In this chapter, hardware architectures for the pre-processing stage and segmentation stage are discussed for a FPGA implementation. Initially, image is processed by the pre-processing module and then the pre-processed image is given to the segmentation module. Block digram of hardware implementation is show in Figure 3.1. The original image is loaded in the external memory for processing. Image pixels (top left to bottom right) are stored in the external memory at subsequent addresses (starting address to end address). Stream cache module is used for read and write operations with the external memory and image processing module (pre-processing or segmentation). Stream cache, pre-processing and segmentation module are discussed in detail in following sections. VHDL component descriptions of all modules are given in the appendix. Image size (height and width) is generic parameter for all modules.

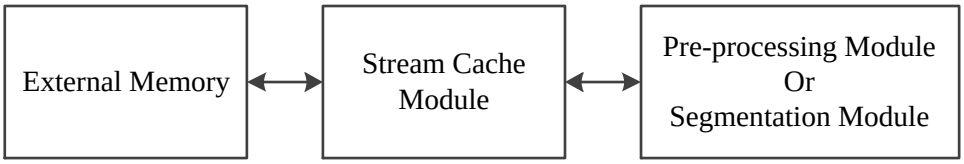


Figure 3.1.: Block Diagram of Hardware implementation

3.1. Stream Cache Module

Stream cache module is used to make image data communications with the external memory which is designed by Robert Bosch GmbH. The basic concept of the stream cache module is explained in this section.

Top level controller module (Pre_processing_Ctrl) makes interface with the stream cache module and slave controller. Pre-processing module is connected to this top

Port name	Direction	Data type
WR_Start_Addr	In	std_logic_vector(G_SYSBUS_AWIDTH-1 downto 0)
WR_Init	In	std_logic
WR_Busy	Out	std_logic
WR_Flush	In	std_logic
WR_Data	In	std_logic_vector(G_APPL_DWIDTH-1 downto 0)
WR_We	In	std_logic
RD_Start_Addr	In	std_logic_vector(G_SYSBUS_AWIDTH-1 downto 0)
RD_Init	In	std_logic
RD_Busy	Out	std_logic
RD_Data	Out	std_logic_vector(G_APPL_DWIDTH-1 downto 0)
RD_re	In	std_logic

Table 3.1.: Ports description of stream cache module

level controller module. Application ports of the stream cache module describe in the Table 3.1 which are required to connect with the pre-processing module. G_SYSBUS_AWIDTH and G_APPL_DWIDTH are generics for the system address bus and application data bus, both are set to constant value of 32. Pre-processing module gets P_ctrl_start_sl signal from the slave controller which notifies to start reading of the data from external memory using the stream cache.

In read data operations, stream cache takes 64 bytes burst from the external memory during initialization. This packet is stored in the internal memory (block RAMs) of FPGA. When RD_re signal is given then data is read from this internal memory. When the number of bytes in the internal memory are lower than constant value (e.g. 32), then the stream cache reads new packet from the external memory. In write data operations, data is written to internal memory when WR_we signal is activated. When number of bytes are 64, and if data bus is available then all 64 bytes are written to the external memory. If data bus is busy then output data can be written to the internal memory buffer until buffer is not full. If internal memory buffer gets full and data bus is still busy then it is necessary to deactivate the WR_we signal.

All signals of the stream cache module are initialized with the zero. First, starting address of memory read is set in RD_Start_Addr signal. After getting P_ctrl_start_sl signal high for one clock cycle, RD_Init signal is set to high for one clock cycle. RD_Busy signal defines busy status of the memory data bus or not enough space in

the internal memory buffer and it is high in busy mode. RD_Busy signal goes high after the initialization. When RD_Busy signal goes low again which indicate that initialization is finished. When data is required to read, RD_re signal is set to high for given number of clock cycles. If RD_re signal is set to high only for one clock cycle then it reads only one data and load it into RD_Data signal. It is needed to check RD_Busy signal every time before RD_re signal is given and RD_Busy should be low when RD_re signal is asserted high.

Write data back to the external memory using stream cache is similar process as the read data operations. First, starting address of the write memory location is set in WR_Start_Addr signal. WR_Init signal is set to high for one clock cycle which initialize write address. WR_Busy defines busy status of memory data bus and it is high in busy mode. Write data are loaded in the WR_Data signal during memory write operation. When data is required to write, WR_we signal is set to high for given number of clock cycles. If WR_we signal is set to high only for one clock cycle then it writes only one data from WR_Data signal. It is needed to check WR_busy signal every time before WR_we signal is given and WR_busy signal should be low when WR_we signal is asserted high. WR_Flush signal is used to empty cache. When it is asserted for one clock cycle, all bytes of cache are written to external memory until cache becomes empty. WR_Busy signal goes high after WR_Flush signal is given. Write data flush is completed when WR_Busy signal goes to low again. Figure 3.2 explains write initialization, write data, read data and flush write data operations. Read initialization is same as write initialization in Figure 3.2.

3.2. Pre-processing Module

The input gray scale image is given to pre-processing module from the external memory. The pre-processing module gets one new pixel on every clock cycle from the external memory, processes it and writes back a processed pixel to the external memory. Pre-processing module is designed using a pipeline concept in such a way that it processes a new pixel on each clock cycle. This module has components like serializer, 3 x 3 moving window module, median filter, morphological gradient, thresholding and Serial-In-Parallel-Out (SIPO) shift register module. Flow of the input data and controller signals from one module to another is shown in below figure.

The pixel size is 8 bit in the gray scale image but only 32 bit data read or write operations are performed with the external memory on every clock cycle. There are total four pixels available on each clock cycle. The reason behind 32 bit data reading

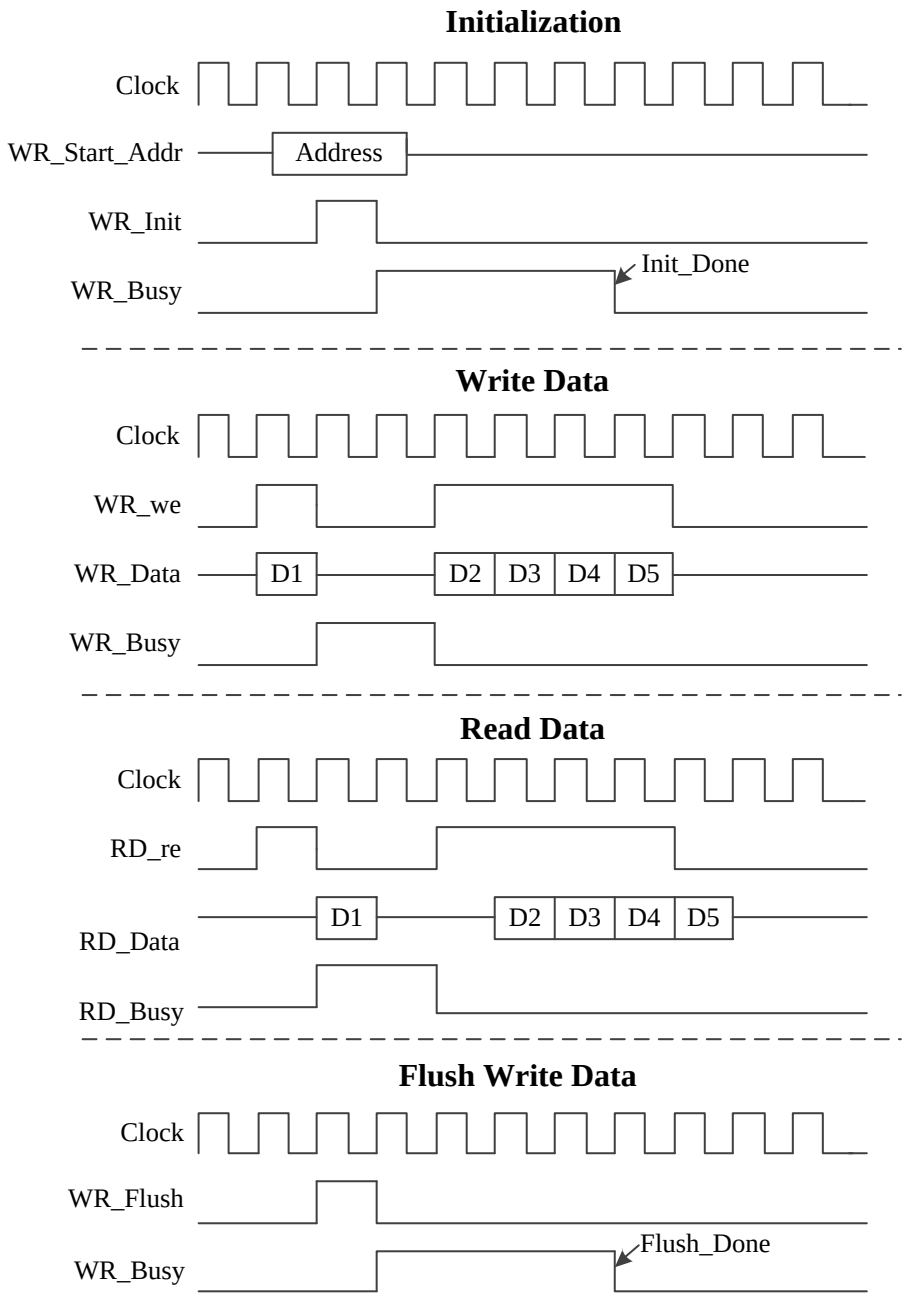


Figure 3.2.: Stream cache - initialization, read, write and flush operations

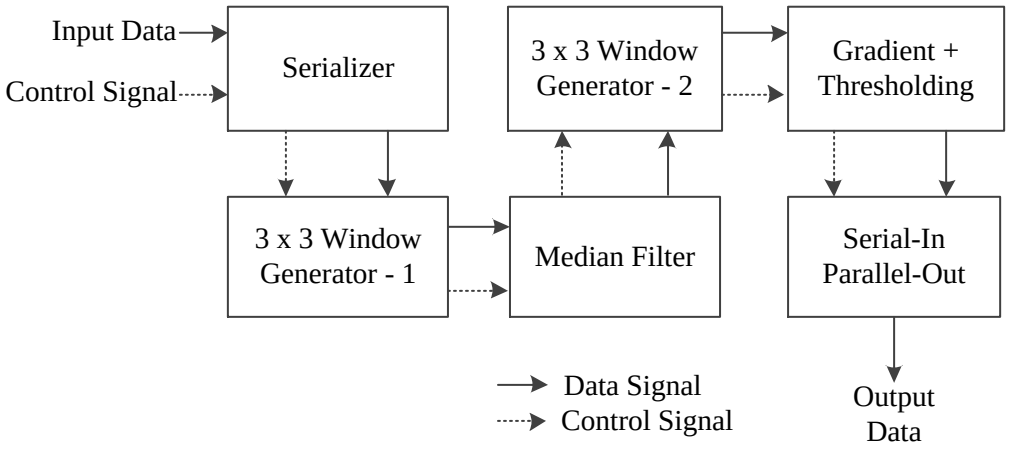


Figure 3.3.: Pre-processing module - data and control signal flow

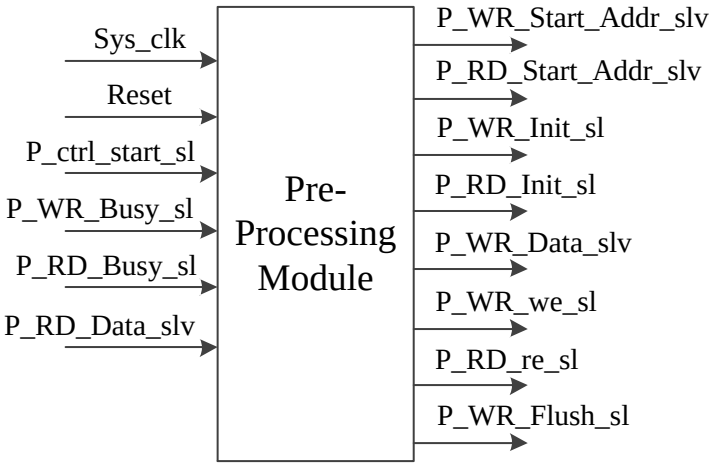


Figure 3.4.: Pre-processing module schematic

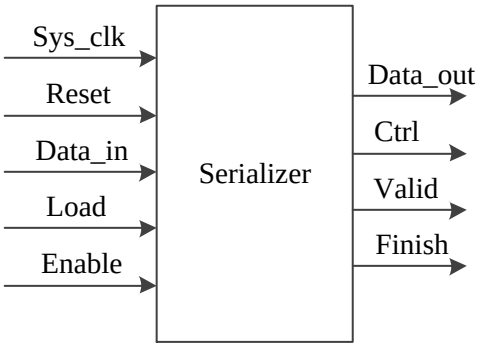


Figure 3.5.: Serializer module schematic

or writing on each clock cycle is to make read and write operations compatible with the segmentation module because segmentation module is designed to process 32 bit data on each clock cycle. Stream cache module can not switch to different data width format for read and write operations during the run time, so either it needs to set for 8 bits, 16 bits or 32 bits data width.

The schematic diagram of the pre-processing module is shown in Figure 3.4. It has control and data signals for memory read and write operations, which are connected to the stream cache module. Each signal of the pre-processing module has a same as functionality a stream cache module.

3.2.1. Serializer

As the pre-processing module is designed to process only single gray image pixel (8 bit) per clock cycle, serializer is needed to serialize the data. It loads 32 bit data in one clock cycle and provides same data as a stream of four 8 bit data in four clock cycles (one clock cycle for one 8 bit data). In 32 bit data, eight most significant bits (MSB) are part of the first pixel. Eight bits are shifted left on each clock cycle, and a new pixel is shifted to the MSB position and taken as an output data.

The schematic diagram is shown in Figure 3.5. Control and data inputs are captured and new output data is formed on rising-edge of the clock signal. Reset signal is active low synchronous reset which reset the serializer. When enable is deasserted (Low),

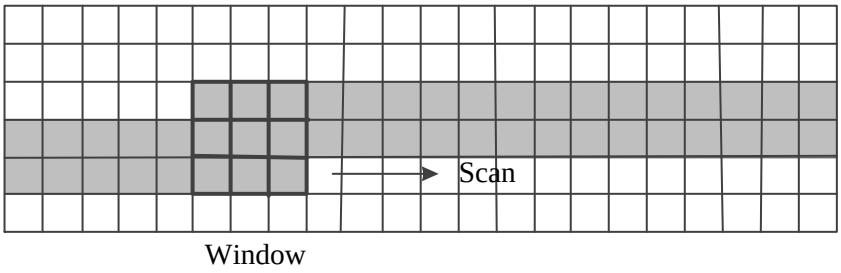


Figure 3.6.: Moving Window

all the synchronous inputs are ignored and internal data of serializer are not changed. Input data (32 bit width) is given to data_in signal and 8 bit output data is provided by data_out signal. Valid signal is asserted high when first 8 bit output data of the given 32 bit input data is ready. Finish signal goes high when total number of output data from the serializer are equal to the image size.

3.2.2. 3x3 Moving Window Architecture

Each output of the median filter or morphological gradient is a function of nine pixel values within the 3 x 3 window neighbourhood. Nine pixels are necessary to read for each window position if caching is not used, and each pixel is needed to read nine times during the image scanning. First-In-First-Out (FIFO) buffer method is well known window generation approach for the hardware implementation and very easy to implement in the VHDL. The FIFO buffer caches pixel values of previous rows, so pixel values do not need to be read again.

For 3 x 3 window operation, two FIFO buffers are necessary to cache the pixel values of two previous rows. It also requires nine registers to store the window values before they are given to the processing module. Two block RAMs are used to create two FIFO buffers. Read and write operations are performed on same clock cycle for them. The depth of the FIFO buffer should be three less than the image width. FIFO buffer module is generated by Xilinx core generator tool. Window architecture reads pixels row by row in a raster scan which are given by the serializer module. As shown in Figure 3.7, pixel is given to w33 register and it is shifted to the left in every clock cycle. Pixels are shifted to left registers continuously, initially FIFO buffer one gets

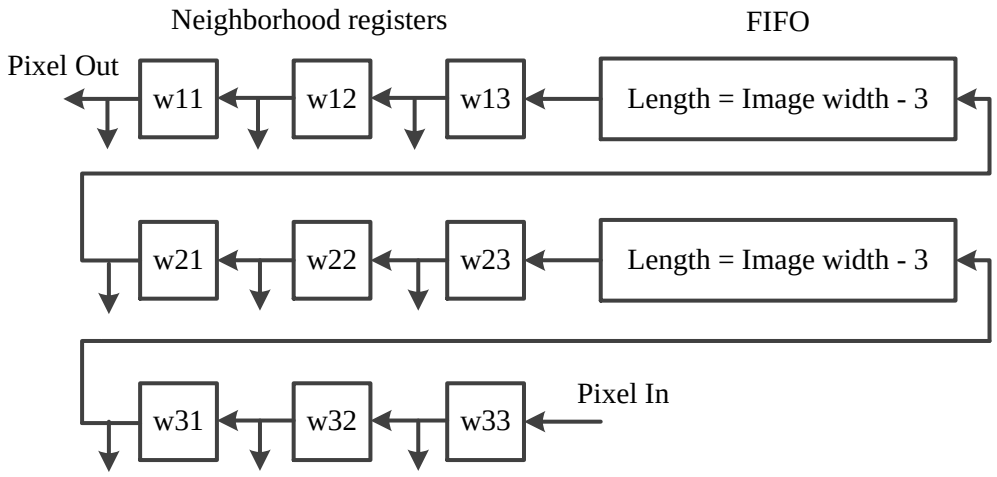


Figure 3.7.: Implementation of 3 x 3 Window

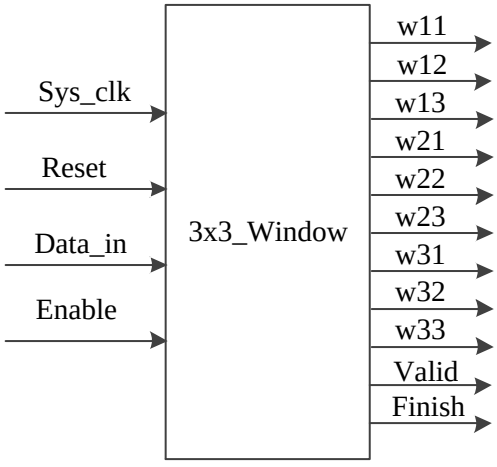


Figure 3.8.: 3 x 3 window module schematic

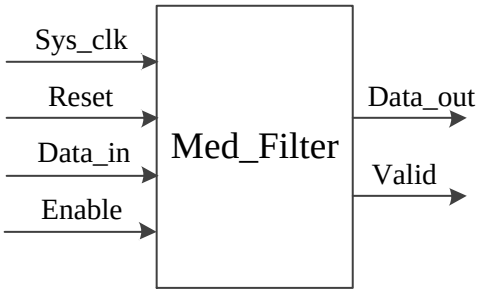


Figure 3.9.: Median filter schematic

full, then FIFO buffer two gets full and when first image pixel reaches at w11 register, first 3 x 3 window is available to process.

Figure 3.8 shows schematic diagram of this module. Functionality of clock, enable and reset signals are same as the serializer component. Input pixel of 8 bit is given to the data_in signal and then it is shifted to the internal registers. The w11, w12, w13, w21, w22, w23, w31, w32 and w33 are shift registers of window structure. Valid signal is asserted high when the first 3 x 3 window is available, and finish signal is activated when the total numbers of generated windows are same as total number of image pixels. Pixel bit width and image row size are generic parameters for this module.

3.2.3. Median Filter Module

Median filter module gets nine data inputs from the moving window module. As discussed in the chapter 2, bitonic sorter is used to find out the median value of given window. Bitonic sorter is easy to implement in the pipeline and parallel architecture using VHDL. It requires less hardware resources and provides minimum latency for sorting nine elements.

The schematic diagram is shown in Figure 3.9. System clock, enable and reset signal have same definition as given in the serializer module. Total number of input data, window width and height are constant generic parameters. Nine input data are given

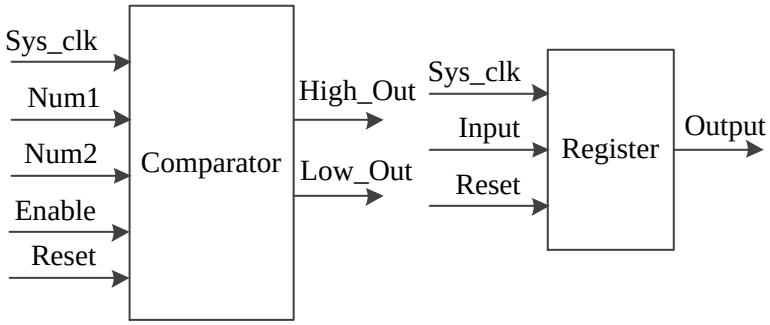


Figure 3.10.: Comparator and register module schematic

to data_in signal and data_out signal provides the median value of them.

Compare&Swap (comparator) and register modules are required to implement bitonic sorter. In Figure 3.10, schematic diagram of comparator and register modules are shown. Both modules require system clock and enable signal. Two input data are given to num1 and num2 signal of the comparator. It compares and swaps both data to High_out and Low_Out signal. 8 bit registers are used to save intermediate data in bitonic sorter to make pipeline operation possible.

Each arrow of Figure 2.18 replaces with one comparator. There are four comparators in the stage eight but only one comparator between line five and six is required in this stage to calculate the median value. Total 23 comparators and 12 registers are required to calculate median value using bitonic sorting network in the pipeline implementation. This implementation is pipelined in eight stages and each stage has some parallel compare&swap operations. Pipeline latency is eight clock cycles and pipeline throughput is one clock cycle. Pipeline implementation is eight times faster than sequential bitonic sorter for nine input elements.

3.2.4. Morphological Gradient and Thresholding Module

Morphological gradient and thresholding are designed in single module for easier hardware implementation. Output data of the median filter module is given to input of this module. Nine inputs bitonic sorter gives maximum and minimum value of the input elements, and one 8-bit subtracter is needed to calculate the difference between

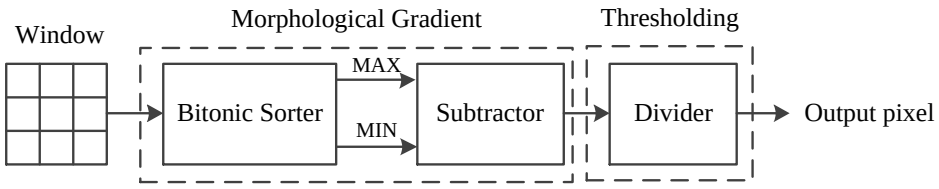


Figure 3.11.: Morphological gradient and thresholding calculation

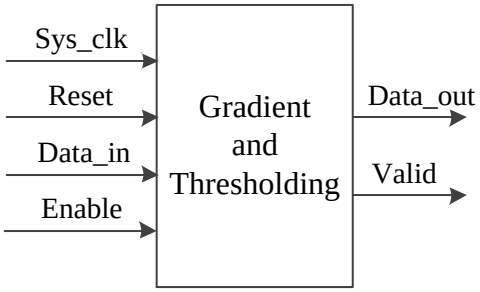


Figure 3.12.: Morphological gradient and thresholding module schematic

them as shown in Figure 3.11. Compare&swap and register modules are used for bitonic sorter, and they are already described in the median filter section.

Each arrow of Figure 2.18 replaces with one comparator. Bitonic sorter gives maximum value at line 9 and minimum value at line 1. Only comparator between line 1 and 2 is required for minimum value and all other comparators of stage eight are not required.

8-bit by 5-bit pipeline divider is used to perform thresholding operation on the calculated gradient value. It is generated using Xilinx core generator tool, which has a pipeline latency of ten clock cycles and a throughput of one clock cycle.

Schematic symbol is shown in Figure 3.12. Nine input data are given to data_in signal of module and then bitonic sorter find out maximum and minimum values which are given to the subtracter module, the difference calculated by the subtracter module is

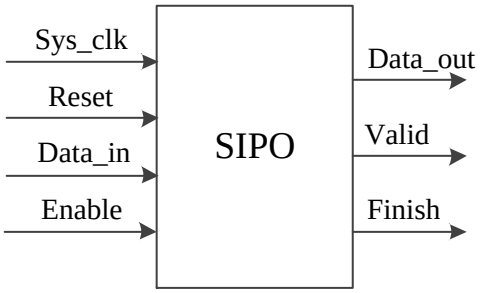


Figure 3.13.: SIPO module schematic

given as a dividend to divider module. The divisor value of the divider is constant threshold value which is generic parameter. Quotient value of the divider module is given to the output of gradient and thresholding module.

3.2.5. Serial-In-Parallel-Out (SIPO) Shift Register

This module is used to combine four serial data of 8 bit to 32 bit parallel data for memory write operations. It is necessary because stream cache module is set for 32 bit read and write operations and threshold module generates an eight bit output. The reading data width is 32 bit and writing data width should be 32 bit, because it is not possible to set different read and write data width in stream cache module. As shown in schematic diagram, it gets 8 bit input data from threshold module in Data_in signal and generates 32 bit parallel data at Data_out signal which is written to the external memory.

3.2.6. State Machine for Pre-processing Module

There are two state controllers which are used for control and data signals flow in the pre-processing module implementation. State diagram for memory read operation is shown in figure 3.14. It consists of four states, Idle, RD_Init, Serializer and RD_Stop. Initially, it is in Idle state and starting address for memory read operation is set

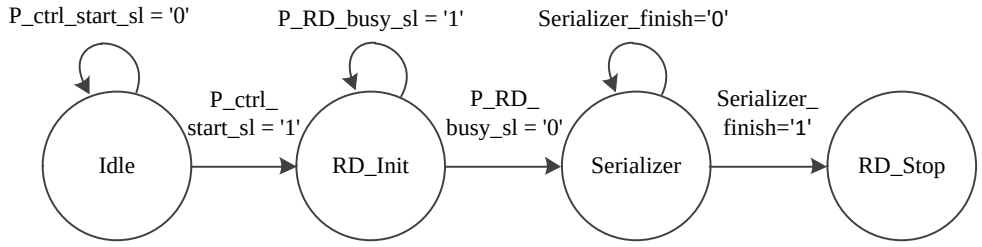


Figure 3.14.: State diagram of memory read operation for Pre-processing

in this state. When P_ctrl_start signal is asserted, state is changed to RD_Init state. RD_Busy signal is high when state enters in RD_Init state. When RD_Busy signal goes low, first read request is given by asserting (high) RD_re signal and state is changed serializer state. Serializer state reads data from external memory continuously and serializes them. When serializer reads all necessary input data, Serializer_finish signal is activated and state moves to RD_stop state. RD_stop state deactivates RD_re signal, and also indicates that reading process is completed.

State diagram for pre-processing operation is shown in figure 3.15. It consists of eight states, Idle, WinOne, MedianFilter, WinTwo, Gradient+Thresholding, WR_Init, Continue and Finish. WinOne and WinTwo indicate two 3 x 3 moving window modules. Starting state is Idle state and write start address is set in this state. When Serializer_valid signal is activated, Enable signal of 3 x 3 window is asserted and state moves to WinOne state. When first 3 x 3 window is ready by window module, it asserts WinOne_valid signal. When WinOn_valid signal is activated, state is changed to Median filter state. Median filter module activates Med_valid signal when first output data is ready from it. State moves to WinTwo state when Med_valid signal is asserted, and changes to Grad+Thresh state when WinTwo_valid signal is activated. Thresholding module gives output data to SIPO module to convert data from serial to parallel. When first output data is ready by SIPO module, it asserts Write_valid signal. WR_Init is necessary only first time to initialize write address and to activate WR_we signal for first write data. When there is no write request in Grad+Thresh state, it waits for Write_valid signal and after Write_valid is asserted, state is changed to WR_Init state. WR_busy is high when state enters in WR_Init state. It waits until WR_Busy goes low and then state is changed to Continue state. Continue state monitors mainly WinOne_Finish, WinTwo_Finish and Write_Finish signal. State is changed to one of the previous state based on high or low value of these three signals. Write_finish signal is asserted high when numbers of output data from SIPO are

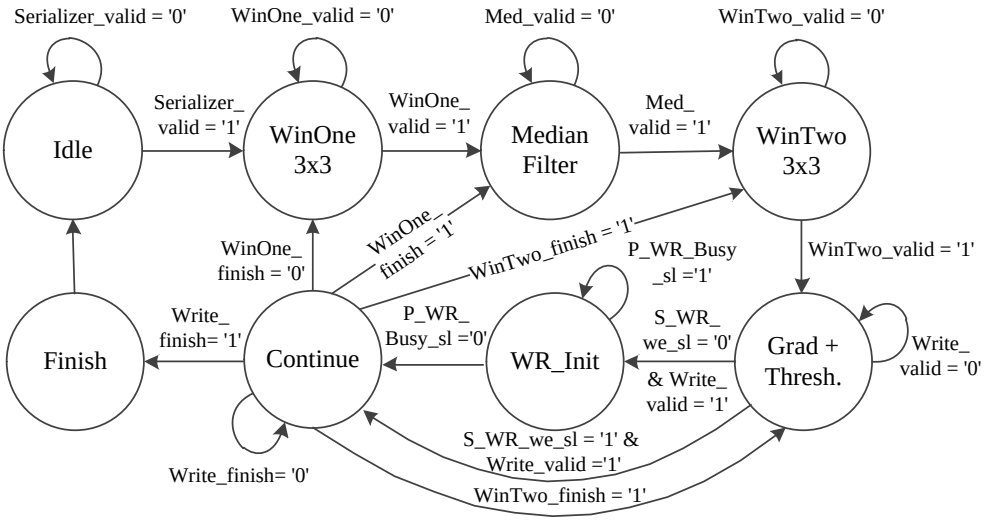


Figure 3.15.: State diagram of pre-processing module and memory write operation

counted to same as image size, and then state moves to Finish state. WR_We signal is deactivated in Finish state and state is moved back to initial Idle state.

3.3. Image Segmentation Algorithm Implementation

Implementation of connected components based watershed image segmentation is discussed in this section. The segmentation module uses pre-processed image calculated by the pre-processing module. Stream cache module is used for pixel data read and write operations with the external memory. As discussed in chapter 2, this segmentation algorithm needs image scans from top left to bottom right (forward) and from bottom right to top left (backward). Stream cache module has two more control signals (RD_Backwards and WR_Backwards) to perform forward or backward, read and write operations with the external memory. When they are set to high, read and write operations occur in the backward direction, otherwise in the forward direction.

The pre-processed image has a 8 bit pixel bit width. As discussed in the chapter 2, two arrays v[p] and l[p] are used for the segmentation, and VMAX and LMAX are maximum possible element value for them respectively. Maximum value of VMAX

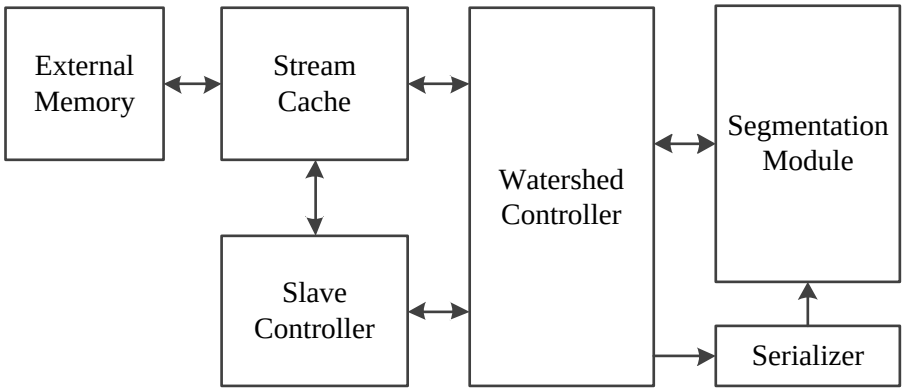


Figure 3.16.: Block diagram of segmentation hardware architecture

is image width plus image height, and addition of them always fits in 8 bit width. Maximum value of LMAX depends on number of segments in the image. Array l[p] is used mainly to store label values of the segments. If 8 bit width is selected for each element of array l[p] then label value has a range between 0 to 255. Some images may have more segments than 256 and hence label value range should be selected higher. If 16 bit width is selected for the label value then label value has a range between 0 to 65535 which is more than sufficient for the image which does not have the segments more than 65536. Original input image (or pre-processed image), array v[p] and l[p] have element data width of 8, 8 and 16 bit respectively.

The Xilinx virtex 4 FPGA (device XC4VFX60) has 232 blocks of RAM. One block has size of 18 Kb (Kilo bits). A 512 x 512 image is considered then total amount of bits are $32 \times 512 \times 512$ (32 bit is addition of one pixel of the input image, one element of array v[p] and l[p]). It requires 456 blocks of RAM to store image pixels, array v[p] and l[p] which are almost double than available in the device. It is only possible with the external memory (e.g. DDR RAM) to store this big amount of data. So, all required processing data are stored and processed using the external memory. For $32 \times 512 \times 512$ bits, 8.5 Mb (Mega bits) size is required in the external memory. The minimum requirement of the external memory size depends on the image size and size of array l[p] and v[p].

Each pixel has respective element in array v[p] and l[p]. Calculations of the algorithm require each pixel and its neighbourhood pixels, respective element of the array v[p] and its neighbourhood, and respective element of the array l[p] and its neighbourhood

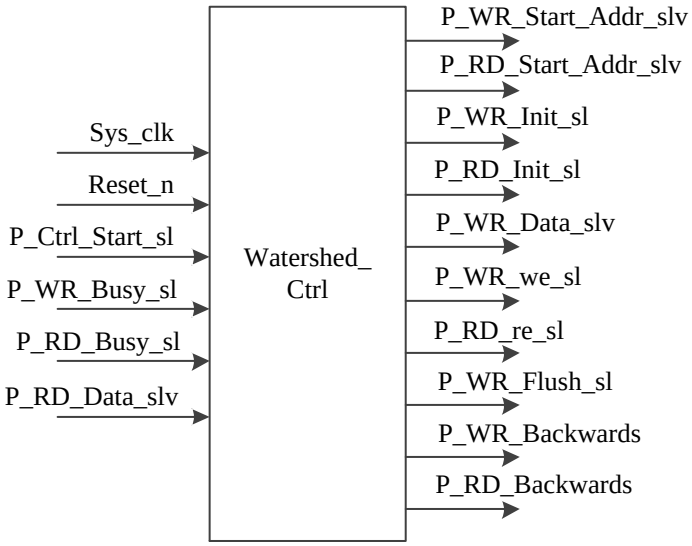


Figure 3.17.: Watershed controller module schematic

pixels. The basic idea is to read 32 bit data which has an image pixel value of 8 bit, corresponding 8 bit element of v[p] and corresponding 16 bit element of l[p]. As all nine neighbourhood pixels are used for calculations for the segmentation, 3 x 3 moving window approach is also used in this implementation.

All elements initialize with zero in both array l[p] and v[p]. As stream cache module is only possible to set for one of possible data width (8, 16 or 32 bit), and 32 bit data width read and write operations are necessary for this implementation, data width of stream cache is set to 32 bit. Original image is saved as 8 bit per pixel and if read operations for 32 bit data are performed in the first scan, serializer is necessary to break down it in four 8 bit data. The serializer module is discussed in the previous section but it is slightly modified for the segmentation module. The modification is to append high 24 bit (towards the MSB) with zero to each serial 8 bit data (lower eight bits), so the serializer gives output of 32 bit. It gets 32 bit input data, breaks down in four serial values, appends 24 most significant bit as a zero to each of them and provides 32 bit output on each clock cycle. This 32 bit output value contains pixel value (0 to 7 bit), element of v[p] value (8 to 15 bit) and element of l[p] value (16 to 31 bit) as shown in Figure 3.20. Initial values are required to set as zero in array v[p] and l[p], so 24 MSB are appended to each pixel in serializer module with zero. The

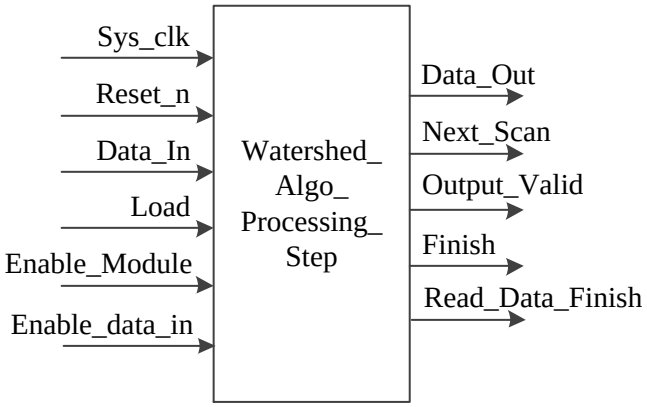


Figure 3.18.: Segmentation module schematic

same format of data is written back to the external memory. So, 32 bit data can be used directly for the subsequent image scans without use of the serializer.

As shown in the block diagram (Figure 3.16), the top module of implementation is watershed controller module (Watershed_Ctrl) which is connected to the stream cache module and slave controller. The serializer gets data only during the first image scan. Segmentation module (Watershed_Algo_ Processing_Step) performs segmentation of the image. It gets image data from serializer module in the first scan, and direct from the external memory during all subsequent scans. The Schematic diagram of watershed controller (or segmentation controller) module is shown in the figure 3.17. It has same ports functionality as described in the stream cache module.

Segmentation Module

Segmentation module performs computation for the image segmentation. RD_Busy signal and WR_Busy signal indicate whether memory data bus is free or busy for data processing with external memory. When segmentation module can not get the input data or can not write the output data because of the memory data bus is busy with data processing of other modules, then this module needs to be halt. When data bus is available again for the segmentation module, then it can resume computation from the last halted state.

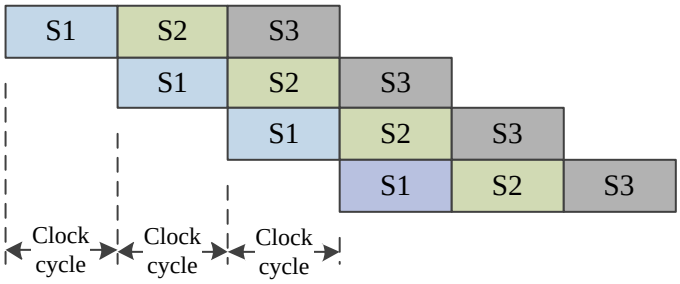


Figure 3.19.: First approach of pipeline implementation

Figure 3.18 represents schematic diagram of the segmentation module. It has pixel width, image row width, image height, VMAX and LMAX as generic parameters. Enable_module signal is used to enable all ports and internal digital logics. Input data is given to Data_In signal and output data is provided by Data_Out signal. Enable_data_in signal indicates that when it is asserted, input data is given to internal register from the Data_In signal. Output_Valid signal is activated when first valid data is available on the Data_Out signal. When one scan of the image read, segmentation computation and write back to the external memory are completed, then finish signal is asserted for one clock cycle. Next_Scan represents that subsequent image scans are required or not for the image segmentation. When Next_Scan is active high after the finish signal, then subsequent image scans are necessary for the segmentation. When it is low, then the subsequent image scans are not required for the segmentation and segmentation of the given image is completed.

If step 1, 2 and 3 process sequentially as discussed in chapter 2, total 10 scans are needed. One for Step 1 (S1), three for step 2 (S2) and six for step 3 (S3). If this algorithm is implemented in the hardware, then S1, S2 and S3 execute at a same time in pipeline architecture as shown in Figure 3.19. The number of scans are reduced by pipeline implementation and their reasons are discussed in the next chapter. The wrong value may assign to elements of v[p] and l[p], because elements of v[p] and l[p] may be calculated by neighbourhood, whose values have not been calculated yet. But, these wrong values are overwritten by correct values in the subsequent scans [3]. S1, S2 and S3 take one clock cycle each. S3 provides one output data on each clock cycle which is main objective of this pipeline implementation. So, pipeline throughput becomes one clock cycle.

3 x 3 moving window architecture is used to perform the image scan. This architecture

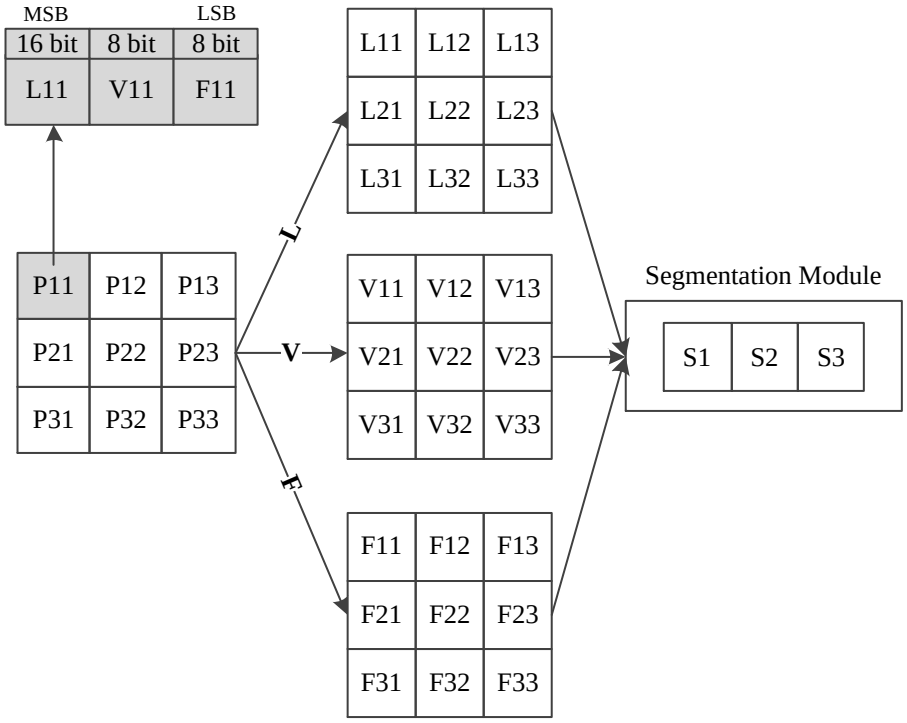


Figure 3.20.: Window split diagram

is slightly modified compared to the one which is used in the pre-processing module. The shift registers and FIFO are size of 32 bit data width instead of 8 bit. One window (32 bit of each element) has three sub window as shown in Figure 3.20. Window F represents nine pixels of the input image, window V indicates nine elements of array v and window L indicates nine elements of array l. These three windows are given to pipeline input stage S1. Stage S2 and S3 modify element V22 and L22 of window V and L respectively.

Three more shift registers w24, w25 and w26 are used as shown in Figure 3.20. Data from w22 register is shifted to three of them before it is written to the FIFO buffer. Intermediate registers are used between each stages to design pipeline behaviour. Stage S1 and S2 write data to the intermediate registers which provide input data for S2 and S3 respectively. So, new data is given to S1 on each clock cycle. One window processes in three clock cycles (S1, S2 and S3). When one window is finished

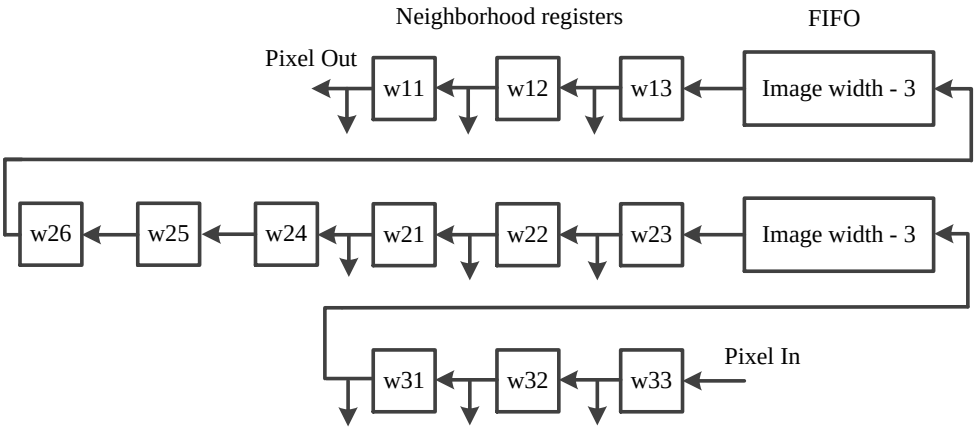


Figure 3.21.: Window generator architecture

in the last stage S3, data of w22 register in the window at stage S1 is shifted to register w25. Element V22 and L22 are calculated for each window in the stage S2 and S3 respectively. Element L22 and V22 are needed to update in respective shift register for each calculated window, because next subsequent rows use only updated neighbourhood values. It means that new window is generated by nine elements on each clock cycle and its result is updated in the register w25 because of continuous shifting of data.

FIFO buffer is used to synchronize data write to the external memory. Segmentation module gives output data to the FIFO buffer and it writes data to the external memory whenever memory data bus is free. The width of the FIFO buffer is same as the image row width.

3.3.1. State Machine for Segmentation Module

Three state machines are used for data and control signals flow. First state machine is needed for the memory read operations only during the first image scan, second one is needed for the memory read operations for all subsequent image scans except the first image scan, and third one is needed for the memory write operations for all image scans. S_First_read signal (type std_logic) is used to notify whether it is a first image scan or not. It initializes with high and after the first image scan is finished,

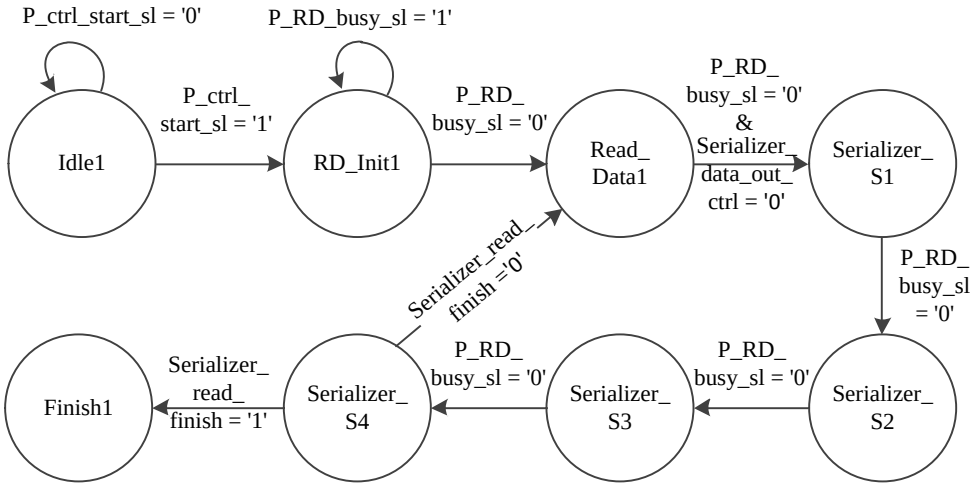


Figure 3.22.: State diagram of memory read operation (only for first scan using serializer) for segmentation module

it is set to low. Segmentation module gets valid input data in read data state and writes back valid output data in write data state.

Figure 3.22 shows state machine diagram for the memory read operations during the first image scan. Serializer module is used in the first image scan. Initially, state is in Idle1 state. Starting address location of the memory for read operations and memory read in forward direction signal are set in the Idle1 state. When slave controller asserts P_ctrl_start_sl signal, state is changed to RD_Init1 state. RD_Init1 state waits until P_RD_busy signal goes low, and after that state moves to the Read_Data state. Read request is given to the external memory in the Read_Data state. Serializer module asserts Serializer_data_out_ctrl signal which indicates that new input data (32 bit) is possible to read from the external memory and load into the serializer module. Serializer module gives one output data (8 bit) on each subsequent states, Serializer_S1, Serializer_S2, Serializer_S3 and Serializer_S4. Serializer_data_out_ctrl signal is asserted high after each last 8 bit output data for the given 32 bit input data. Serializer module activates the valid data signal with each output data which notifies the segmentation module to load new 32 bit input data. Serializer module counts number of output data and when this number is equal to the image size, it asserts Serializer_read_finish signal. After the Serializer_read_finish signal is activated for a one clock cycle, data read from the memory is stopped and state moves to Finish1

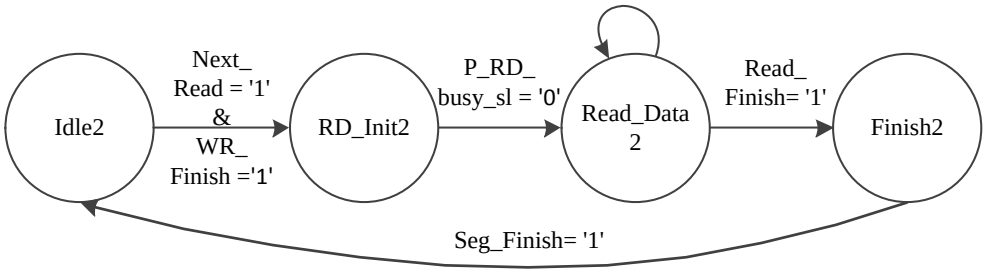


Figure 3.23.: State diagram of memory read operation (after first scan) for segmentation module

state. S_First_read signal is set to low in Finish1 state, so data is given directly to the segmentation module in next subsequent scans without using the serializer module.

Figure 3.23 shows state machine diagram for the memory read operations for all subsequent scans after the first scan. Initially, state is in Idle2 state. Starting address location of the memory read operations and memory read in backward direction signals are set in the Idle2 state. Next_Read signal is activated or deactivated based on the Next_Scan signal of the segmentation module. If Next_Read signal is activated and write operations of previous image scan is completed, then read operations of subsequent scan can be started. So, when Next_read and WR_Finish signals are activated, state is changed to RD_Init2 state. After read initialization is completed, state is moved to Read_Data2 state. Read data request is given to the external memory on each clock cycle in the Read_Data2 state if data bus of the external memory is free. Segmentation module gets each read input data from the RD_Data signal, processes the data and asserts valid signal on each valid output data. Read input data are counted in the Read_Data2 state and when this counted number is same as the image size, then state is changed to Finish2 state. When Seg_finish signal is activated by the segmentation module in the Finish2 state, state is moved to the Idle2 state again. Same procedure performs again from the Idle2 state, if the subsequent scans are required.

Figure 3.24 shows state diagram of the memory write operations for the segmentation module. Initially, state is Idle3 state. Starting address location of the memory write operations is set in the Idle3 state. When segmentation module asserts S_Output_valid signal then state is changed to WR_Init3 state. When write initialization is completed

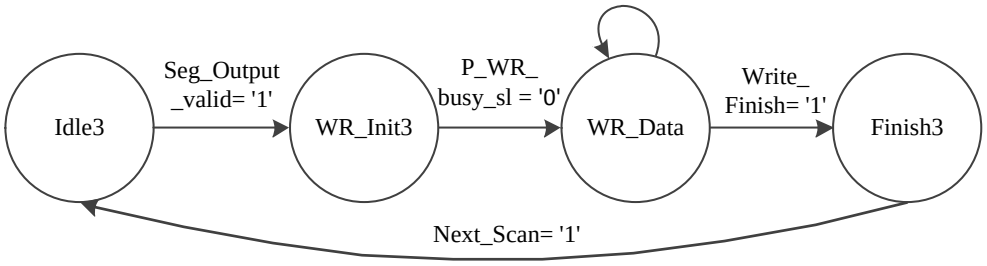


Figure 3.24.: State diagram of memory write operation for segmentation module

then state is changed to Write_Data state. When segmentation module loads output data in WR_Data signal and asserts valid signal with each output data, then output data is written to the external memory. Write data is counted in the Write_Data state, and when count value equals to the image size, then Write_finish signal is asserted high. State is changed to Finish3 state after Write_finish signal is activated. Next_read signal is activated in the Finish3 state if the Next_Scan signal is activated. State is changed to the Idle3 state, so write operations can be performed for the subsequent scans. If Next_Read signal is not asserted in the Finish3 state then subsequent scans will not be performed for the segmentation.

As shown in Figure 3.19, step 1, 2 and 3 are implemented in such a way that each of them is finished in a single clock cycle. The above implementation provides maximum clock frequency of 65.27 MHz for Virtex-4 FPGA device because of more mathematical computations in a single clock cycle. If higher system clock frequency is required then step 1, 2 and 3 are needed to split up in more stages. So, each stage performs less mathematical computations in a single clock cycle.

The algorithm is also implemented using second pipeline approach as shown in Figure 3.25. Step 1 has very less computational complexity compared to the step 2 and step 3, so it does not need to split up in the more stages. Step 2 and step 3 split up in two stages each. As per description of the algorithm, step 2 or step 3 can not process the next input data until they finish their current data respectively. So, Step 2 needs two clock cycles and then next input data is given to the step 2. Step 3 also needs two clock cycles. It means that throughput of the pipeline architecture is two clock cycles where as it is one clock cycle in the first approach of the pipeline implementation. The state machine implementation is slightly modified for this second approach of

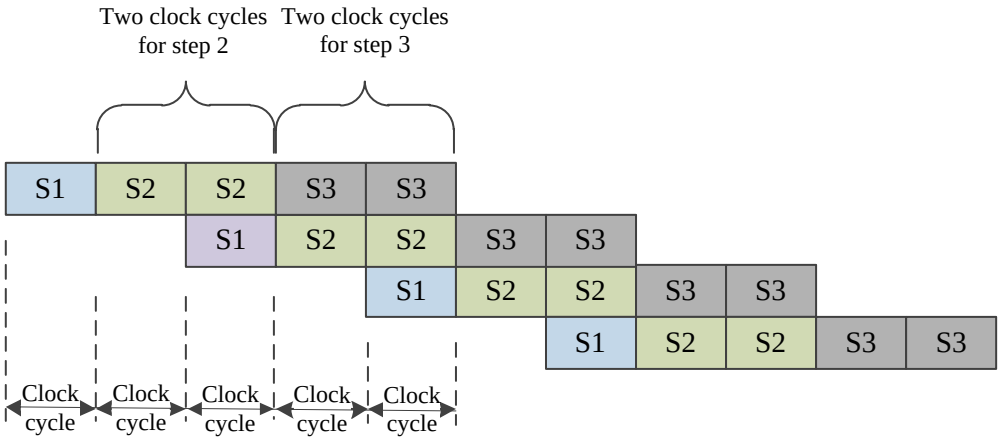


Figure 3.25.: Second approach of pipeline implementation

the pipeline implementation, but it is implemented in similar way as described in the first approach. The data read and write operations are performed every two clock cycles with the external memory. The results are discussed for the both pipeline implementation in chapter four.

3.4. Verification Methodology

Verification of the design is necessary to find out whether design meets required specifications. Functional verification and timing verification are necessary to check before synthesis the design for hardware implementation.

The pre-processing and segmentation modules are sequentially implemented in MATLAB, and they are implemented using pipeline approach using VHDL. MATLAB implementation is considered as a reference model. The ideal way to verify final segmented image is to compare it with the reference segmented image as shown in Figure 3.26. The output image of both implementations are compared using MATLAB function *isequal* which is represented as a checker in the diagram. Function *isequal* gets two matrices as inputs and provides comparison result as a output.

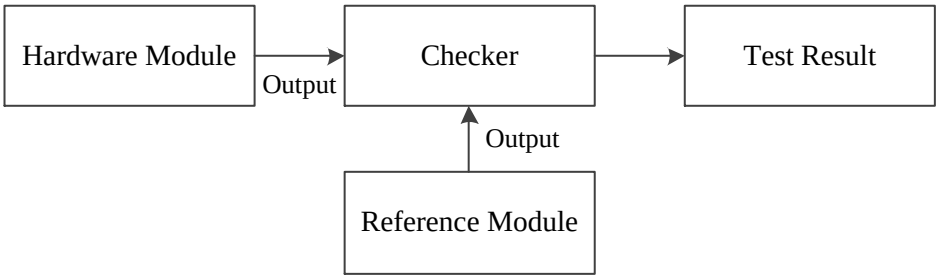


Figure 3.26.: Verification methodology

The results of pre-processing stage for median filter and morphological gradient are discussed in chapter 2 for MATLAB implementation. The final output image data are equal in sequential and pipeline implementation for pre-processing stage and segmentation stage.

3.4.1. Simulation

The simulation of designed hardware is performed using Modelsim simulator tool (version 10.0a) from the mentor graphics. The system clock frequency of 100 MHz (mega hertz) is considered for the simulation purpose.

Pre-processing module has many sub-modules as discussed in section 3.2. Each sub module is tested for the functional and timing verification using testbench. Input image is converted to text file using MATLAB code. This input data is given to the testbench from the input text file. Final results data are also written to the text file which are again converted into the image of same size as of the input image using MATLAB code. Top level pre-processing module is simulated and verified after the verification of each sub-modules. Same procedure is also performed to verify top level module followed by the sub modules for the segmentation implementation.

12 x 12 sample image is used to verify results of different scans in the segmentation implementation. It is easier to verify data during the simulation for small image size. The 12 x 12 input image is given in Figure 2.8a of chapter two.

4. Performance Measurements and Synthesis Results

This section describes comparison between MATLAB and hardware implementation, synthesis results of hardware implementation for Xilinx virtex 4 FPGA.

4.1. Performance Measurements

Connected components based watershed image segmentation algorithm gives over-segmentation without using the pre-processing stage. Watershed based image segmentation is only applied to the gradient image. Original pepper image is converted to the gradient image which is segmented without applying median filter and thresholding, and output image is over-segmented as shown in Figure 4.1b.

Image segmentation results and amount of over-segmentation depend on the threshold value of the pre-processing stage. It is necessary to find out optimal threshold value for different types of images. Threshold value also decides number of segments in the given image. A set of different images are tested for segmentation qualities by changing the threshold value. Images are taken from the berkeley segmentation dataset [19]. Some standard images are also used for performance measurement (e.g. Lena and pepper image).

Table 4.1 shows change in number of labels and scans by varying the threshold value. Number of scans and labels are based on sequential implementation as shown in the pseudo code in chapter two. It means that step 1 is finished first, followed by the step 2, and then the step 3 is finished for all necessary image scans. It is found by testing different images that starting threshold value of 7 is a good choice for the threshold iterations. The optimal visual segmentation results can be obtained by increasing the threshold value. The threshold value iteration is shown only for some images in Table 4.1. Different images (elephants, pepper, bird and aeroplane) give good segmentation

	Threshold value	Total labels	Scans			
			S1	S2	S3	Total
Pepper (256 x 256)	12	660	1	4	8	13
	14	534	1	4	7	12
	15	505	1	5	8	14
House (256 x 256)	9	440	1	5	11	17
Bird (481 x 321)	14	722	1	5	10	16
	18	609	1	5	9	15
	24	494	1	3	11	15
Elephants (481 x 321)	14	1307	1	4	11	16
Aeroplane (481 x 321)	14	202	1	4	10	15
Boat (512 x 512)	14	1300	1	4	9	14
Lena (512 x 512)	8	2997	1	7	12	20
	12	1951	1	6	16	23
	14	1671	1	6	12	19
	18	1344	1	5	10	16

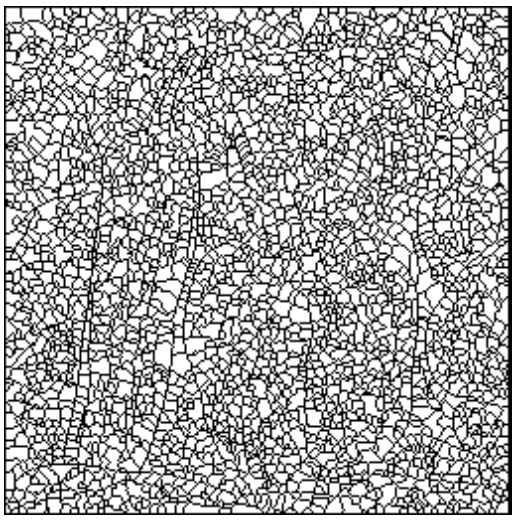
Table 4.1.: Different threshold values for pre processing stage and total number of scans for sequential implementation (S1: Step 1, S2: Step 2, S3: Step 3)

results with threshold value of 14. The selection of optimal threshold value depends on image pixels intensity and number of segments in the image.

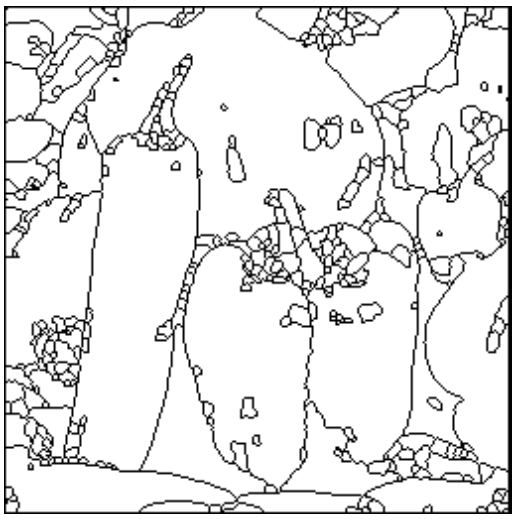
Total number of labels decrease as the threshold value is increased. If threshold value is selected very high, then some boundaries of segments are not preserved. Figure 4.1 shows segmented images with different threshold value for the pepper image (256 x 256). Total numbers of labels in over-segmented image of Figure 4.1b are 2987 whereas after using median filter and the threshold value of 12, number of labels are reduced to 660. The Lena image (512 x 512) gives good segmentation result with the threshold value of 12 as shown in Figure 4.2. The segmented Lena image with the threshold value of 8, preserved all the boundaries of original image compared to the segmented image with the threshold value of 18 as shown in Figure 4.2.

Total number of image scans are reduced in pipeline implementation because step 1, step 2 and step 3 are processing on a same time as shown in Figure 3.19. Total number of scans required by pipeline implementation is influenced by number of scans needed by step 3 of the algorithm, because step 3 dominates major scans for the segmentation. Total number of scans require in the pipeline implementation are same as number of scans needed by step 3 of sequential implementation (step 3 in Table 4.1).

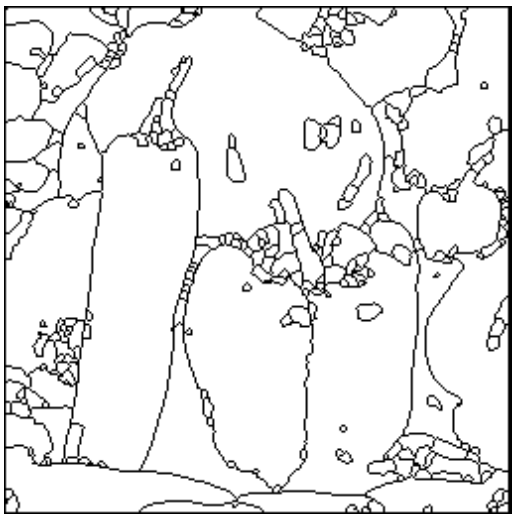
(a) Original image (256 x 256)



(b) Segmentation of gradient image without median filter and thresholding

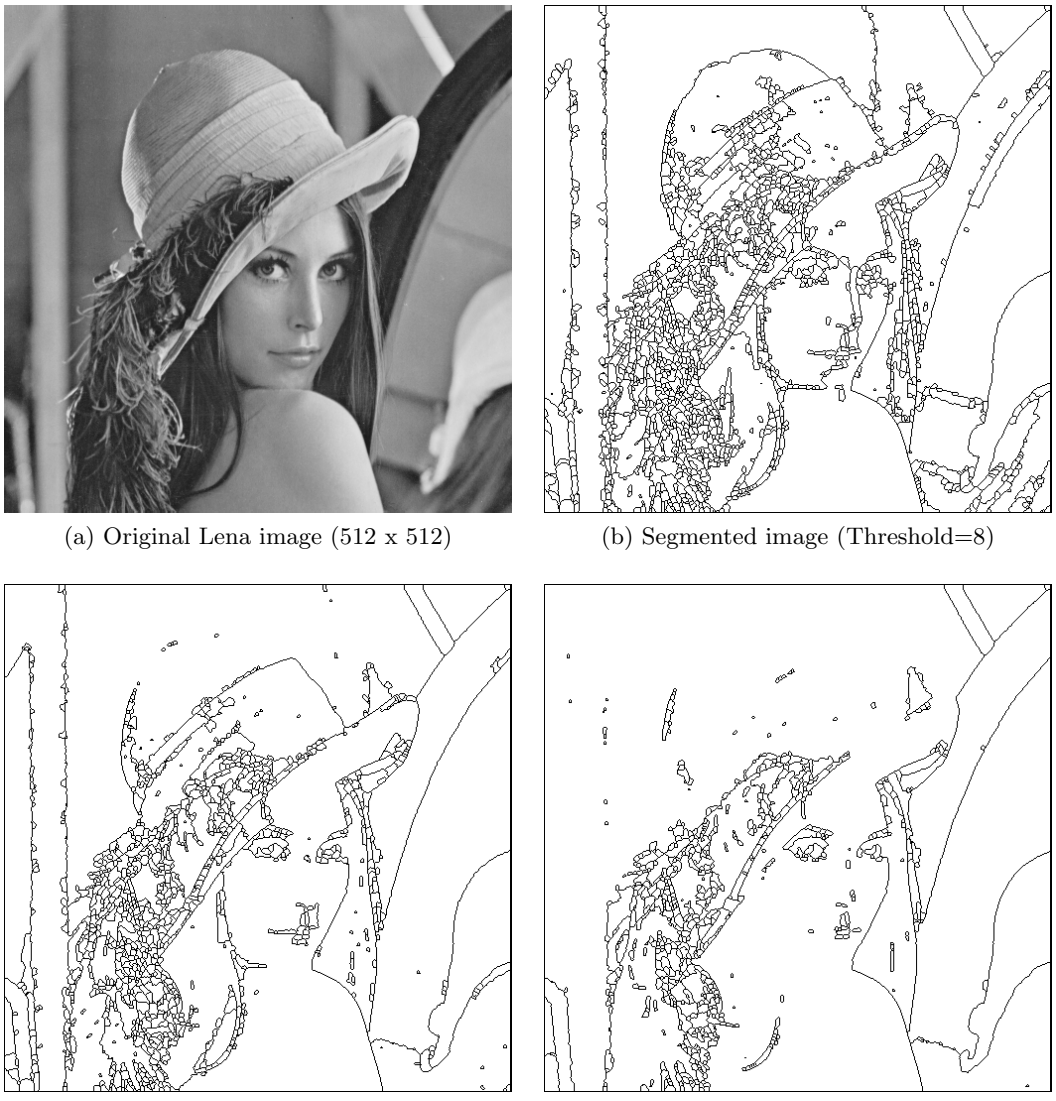


(c) Segmented image (Threshold=12)



(d) Segmented image (Threshold=14)

Figure 4.1.: Segmentation results with different threshold values for pepper image (256 x 256)



(a) Original Lena image (512 x 512)

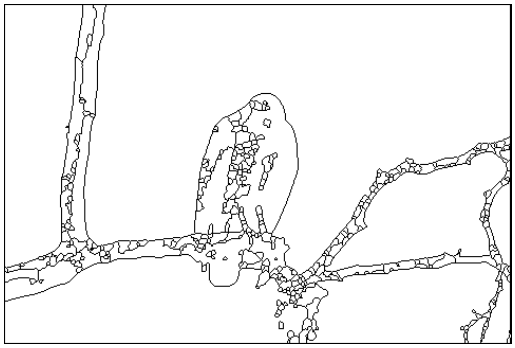
(b) Segmented image (Threshold=8)

(c) Segmented image (Threshold=12)

(d) Segmented image (Threshold=18)

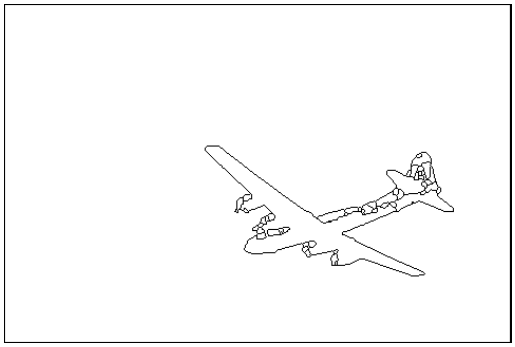
Figure 4.2.: Segmentation results with different threshold values for Lena image (512 x 512)

(a) Bird image (481 x 321)



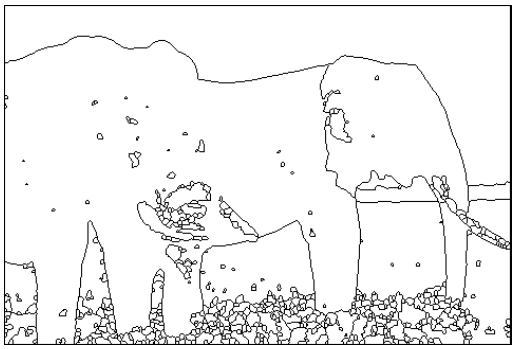
(b) Segmented image (Threshold=14)

(c) Aeroplane image (481 x 321)



(d) Segmented image (Threshold=14)

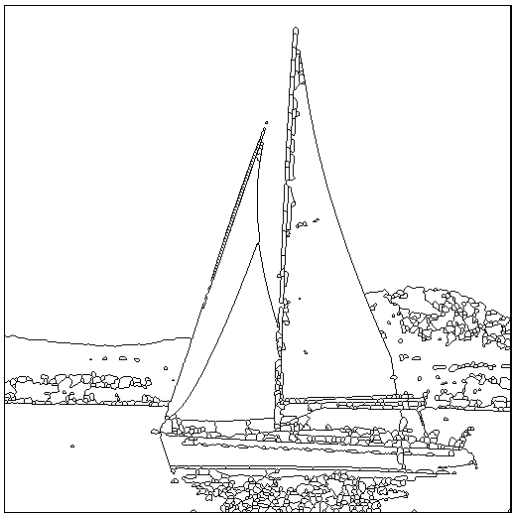
(e) Elephants Image (481 x 321)



(f) Segmented image (Threshold=14)

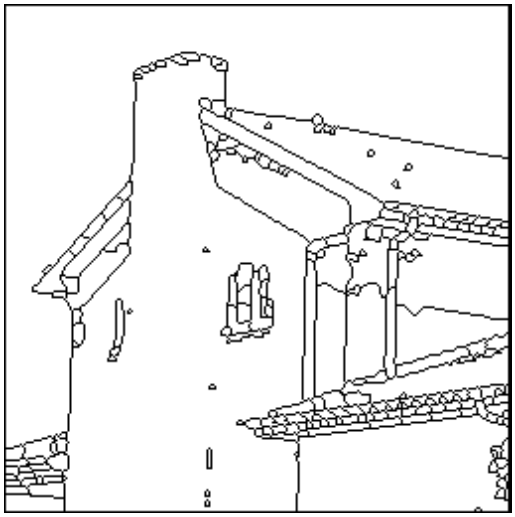
Figure 4.3.: Image segmentation for different images

(g) Boat image (512 x 512)



(h) Segmented image (Threshold=14)

(i) House Image (256 x 256)



(j) Segmented image (Threshold=9)

Figure 4.3.: *(Continued)* Image segmentation for different images

	Threshold value	**Total labels**	**Total scans**
Pepper (256 x 256)	14	534	7
House (256 x 256)	9	440	11
Bird (481 x 321)	14	772	10
Elephants (481 x 321)	14	1307	11
Aeroplane (481 x 321)	14	202	10
Boat (512 x 512)	14	1300	9
Lena (512 x 512)	8	2997	12

Table 4.2.: Total number of image scans with pipeline implementation

Threshold values used in Table 4.2 are optimal for respective images and give optimal segmentation results as shown in Figure 4.3. Total number of scans are decreased in pipeline implementation as shown in the Table 4.2 compared to the sequential implementation in Table 4.1.

4.2. Synthesis Results

Synthesis results of the pre-processing and segmentation modules are discussed under this section. The target FPGA device for synthesis is Xilinx "Virtex-4" FPGA family with device id "xc4vfx60", package id "12ff672 " and speed grade "-12". The size of input image is 512 x 512 pixels. The frequency unit is MHz (Mega Hertz) and time unit is ms (mili seconds) or ns (nano seconds).

Table 4.3 describes synthesis results for the pre-processing stage. Maximum operating frequency of the pre-processing design is 228.59 MHz. In pre-processing module, one pixel is processed in single clock cycle. An image (512 x 512) takes 1.15 msec, if the system runs on the maximum operational frequency.

$$\begin{aligned} \text{Total image processing time} &= \text{Number of pixels} \times \text{Minimum period} \\ &= 512 \times 512 \times 4.375 \text{ ns} \\ &= 1.15 \text{ ms} \end{aligned}$$

Table 4.5 presents synthesis results for the segmentation module. The first approach of the pipeline implementation (refer Figure 3.19) takes one clock cycle to process the single pixel. Maximum operating frequency of segmentation module for first approach

Device Utilization Summary (estimated values)			
Logic Utilization	**Used**	**Available**	**Utilization**
Number of Slices	1530	25280	6 %
Number of Slice Flip Flops	2229	50560	4 %
Number of 4 input LUTs	2580	50560	5 %
Number of FIFO16/RAMB16s	5	232	2 %
Number of GCLKs	1	32	3 %
Minimum period: 4.375ns (Maximum Frequency: 228.595MHz)			
Minimum input arrival time before clock: 2.633ns			
Maximum output required time after clock: 4.130ns			
Maximum combinational path delay: 2.038ns			

Table 4.3.: Pre-processing module synthesis result

of the pipeline architecture is 65.27 MHz. Total segmentation time is given in the column "First approach" of Table 4.4 for respective images. Number of scans are given in Table 4.2 for respective images. It is considered that system operates with maximum frequency for total segmentation time computation. For the boat image (512 x 512),

$$\begin{aligned}\text{Total Segmentation time} &= \text{Number of pixels} \times \text{Number of scans} \times \text{Minimum period} \\ &\quad \times \text{Number of clocks for processing one pixel} \\ &= 512 \times 512 \times 9 \times \ 15.321 \text{ ns} \times 1 \\ &= 36.15 \text{ ms}\end{aligned}$$

	Total segmentation time (ms)	
	First approach	**Second approach**
Pepper (256 x 256)	7.03	9.69
House (256 x 256)	11.05	15.23
Bird (481 x 321)	23.66	32.62
Elephants (481 x 321)	26.02	35.88
Aeroplane (481 x 321)	23.66	32.62
Boat (512 x 512)	36.15	49.84
Lena (512 x 512)	48.19	66.69

Table 4.4.: Image processing time for segmentation module in pipeline implementation

Device Utilization Summary (estimated values)			
Logic Utilization	**Used**	**Available**	**Utilization**
Number of Slices	1938	25280	7 %
Number of Slice Flip Flops	1176	50560	2 %
Number of 4 input LUTs	3413	50560	6 %
Number of FIFO16/RAMB16s	3	232	1 %
Number of GCLKs	1	32	3 %
Minimum period: 15.321ns (Maximum Frequency: 65.269MHz)			
Minimum input arrival time before clock: 7.565ns			
Maximum output required time after clock: 7.283ns			
Maximum combinational path delay: 7.718ns			

Table 4.5.: Segmentation module synthesis result (First approach of pipeline implementation)

Device Utilization Summary (estimated values)			
Logic Utilization	**Used**	**Available**	**Utilization**
Number of Slices	2038	25280	8 %
Number of Slice Flip Flops	1644	50560	3 %
Number of 4 input LUTs	3155	50560	6 %
Number of FIFO16/RAMB16s	3	232	1 %
Number of GCLKs	1	32	3 %
Minimum period: 10.564ns (Maximum Frequency: 94.658MHz)			
Minimum input arrival time before clock: 7.237ns			
Maximum output required time after clock: 7.353ns			
Maximum combinational path delay: 7.473ns			

Table 4.6.: Segmentation module synthesis result (second approach of pipeline implementation)

The synthesis results for the pipeline implementation of second approach (refer Figure 3.25) is given in Table 4.6. The frequency is increased because pipeline stages are split up into two clock cycles. Ideally, frequency should be double than the first approach. Because of the complex routing between different pipeline stages, maximum operation frequency is 94.658 MHz. This implementation approach needs two clock cycles to process a single pixel. Total segmentation time is given in the column "Second approach"of Table 4.4 for respective images. For the boat image (512 x 512),

$$\begin{aligned}\text{Total Segmentation time} &= \text{Number of pixels} \times \text{Number of scans} \times \text{Minimum period} \\ &\quad \times \text{Number of clocks for processing one pixel} \\ &= 512 \times 512 \times 9 \times \ 10.564 \text{ ns} \times 2 \\ &= 49.84 \text{ ms}\end{aligned}$$

First approach of pipeline implementation takes less total segmentation time compared to the second approach. It is possible to design more faster parallel architecture using multiple segmentation units and wider memory data bus interface with the external memory. Proposal of parallel architecture is given in next chapter for the real time image segmentation.

5. Parallel Architecture Proposal

Parallel architecture proposal is given in this section. It uses wider data bus interface with the external memory and higher hardware resources compared to the pipeline implementation. But, it reduces total time required to perform segmentation.

Table 4.2 shows that total segmentation time is between 35 to 50 ms for 512 x 512 image using the first approach of pipeline implementation. External memory interfaces of 144 or 288 bits (data bus) are very common now a days. So, if higher memory data width and multiple segmentation units are used in parallel then high performance can be achieved. In block diagram of parallel architecture, SU represents segmentation unit, n denotes number of segmentation units or P blocks, and each P block represents image segmentation data in the external memory for a single image. It is possible to select number of segmentation units based on availability of the hardware resources and maximum data bus width in FPGA.

Functional explanation of the parallel architecture is given for 10 parallel segmentation units (n =10). So, there are total 10 memory blocks for different images data. If 512 x 512 image needs 10 scan to perform segmentation then it is possible to process 10 different images of the same size at a same time by different segmentation units. Ten different images data are loaded to the external memory in block P1 to P10. One segmentation unit takes 4.01 ms to perform memory read operations, segmentation computations and memory write operations for a single scan. First image data from block P1 is given to the unit SU1 which performs first scan. After the first scan is finished, data of block P1 is given to the unit SU2. When unit SU2 performs second scan on data of block P1, SU1 gets new image data from block P2. When unit SU2 is finished data of block P1 then third scan performs by unit SU3 for data of block P1. Data of each P block is given to SU1 to SU10 sequentially. After unit SU10 finishes computation for data of block P1 for tenth scan then segmentation is finished for image in block P1. Next segmented image available after a one scan processing time of the segmentation unit which is 4.01 ms. After an each scan, one image segmentation is finished for respective P block. It means that each segmentation unit performs one scan for each image and new image data is continuously loaded after each scan to respective P block. Images are processed in pipeline by proposed parallel

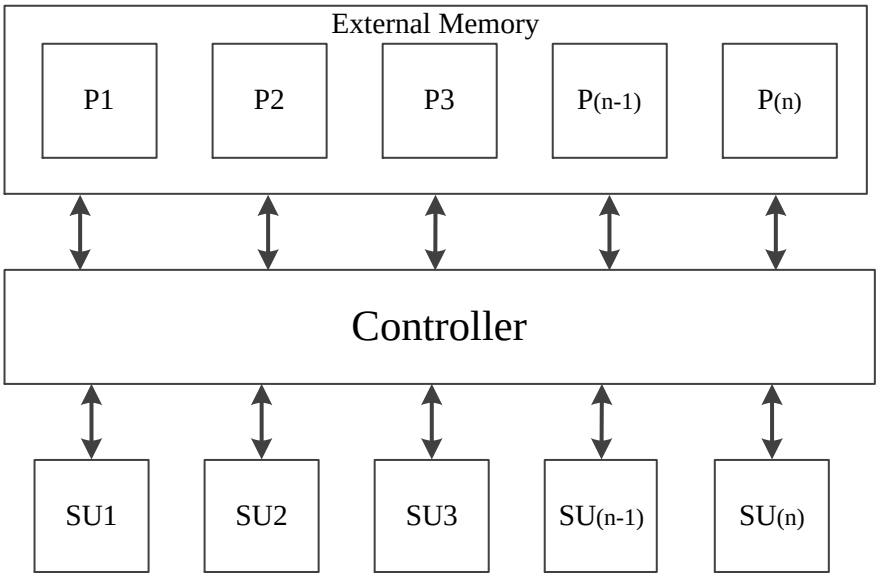


Figure 5.1.: Block diagram of parallel architecture)

architecture.

Controller module is needed to design which control all data communications between external memory and segmentation units. It is also possible to use less number of segmentation units and split number of scans to each units. As an example, if number of scans are ten and total number of segmentation units are five, then each segmentation units performs two scans for each image. In this case, controller module also needs to control the scans by each segmentation unit.

Segmentation Units	**Total Segmentation Time (ms)**	**Device Utilization (approx.)**
1	40.10	7%
5	8.02	35%
10	4.01	70%

Table 5.1.: Total segmentation time for 512 x 512 image with 10 scans and device utilization for Virtex-4 FPGA device

Table shows total segmentation time for 512 x 512 image with single, five or ten

segmentation units. It is considered that each image needs 10 scans to compute the segmentation. Device utilization is given for Virtex-4 FPGA device. The device utilization is approximated based on the first approach of pipeline implementation (Tabel 4.5). As shown by total segmentation time in Table 5.1, it is possible to perform real time image segmentation using the proposed parallel architecture.

6. Conclusion

The starting point of the thesis is to build image segmentation architecture for an FPGA implementation that provides good segmentation quality, requires few hardware resources and short execution time, and is suitable for use in real time applications. A watershed algorithm based on connected components is selected for the implementation, because it has simplified memory access, least computational complexity and good segmentation results compared to the other image segmentation algorithms. A pre-processing step is required to overcome the problem of over-segmentation by watershed based image segmentation. Pipelined architectures of both the segmentation algorithm and pre-processing step are designed, implemented in VHDL and synthesized for Xilinx Virtex-4 FPGA.

An external memory is needed to store an input image and segmentation data because of the limited internal memory in the targeted device. Two different pipelined architectures are designed for the segmentation algorithm and both architectures use very few hardware resources. The first architecture provides throughput of one pixel per clock cycle with the limitation of maximum operation frequency of 65.27 MHz whereas second architecture provides throughput of one pixel per two clock cycles with the maximum operation frequency of 94.66 MHz. The bitonic sorting network is selected to sort the nine elements in the pre-processing step as it requires less compare and swap operations compared to the other sorting networks and is possible to implement using pipeline approach. The pipelined architecture of the pre-processing step has throughput of one pixel per clock cycle and it can operate up to 228 MHz system frequency.

Different images are tested for segmentation quality and execution time. Segmentation results are visually acceptable and almost identical segmentation results can always be obtained using the given implementation. The computation time for a 512 x 512 image is about 35 to 45 milliseconds with the implemented segmentation architecture.

The use of external memory can also be avoided by using the FPGA device with the larger internal memory. Single pipelined segmentation unit uses very few hardware resources, so it is possible to use multiple segmentation units to achieve higher perfor-

mance. The proposal of parallel architecture given in the chapter five uses multiple pipelined segmentation units and is fast enough to use for real time applications. There is still some over-segmentation left after the pre-processing and segmentation steps and hence a post processing step is necessary to implement which will remove the over-segmentation. The implemented and proposed architectures of watershed algorithm based on connected components are excellent candidates to use for different image processing applications where high speed performance is needed.

A. Appendix

```
component Pre_Processing_Module is
generic( G_APPL_AWIDTH    : natural:=16;
         G_APPL_DWIDTH    : natural:=32;
         G_SYSBUS_AWIDTH  : natural:=32);
port(
    sys_clk                         : in  std_logic;
    reset_n                         : in  std_logic;
    P_ctrl_start_sl                 : in  std_logic;
    P_WR_Start_Addr_slv             : out std_logic_vector(
       G_SYSBUS_AWIDTH-1 downto 0);
    P_WR_Init_sl                    : out std_logic;
    P_WR_Busy_sl                    : in  std_logic;
    P_WR_Flush_sl                   : out std_logic;
    P_WR_Data_slv                   : out std_logic_vector(
       G_APPL_DWIDTH-1 downto 0);
    P_WR_we_sl                      : out std_logic;
    P_RD_Start_Addr_slv             : out std_logic_vector(
       G_SYSBUS_AWIDTH-1 downto 0);
    P_RD_Init_sl                    : out std_logic;
    P_RD_Busy_sl                    : in  std_logic;
    P_RD_Data_slv                   : in  std_logic_vector(
       G_APPL_DWIDTH-1 downto 0);
    P_RD_re_sl                      : out std_logic);
end component Pre_Processing_Module;
```

Listing A.1: Pre-processing component module in VHDL

```
component serializer is
generic ( image_row_width: integer);
port(
  sys_clk     :  in  std_logic;
  reset       :  in  std_logic;
  data_in     :  in  std_logic_vector(31 downto 0);
  load        :  in  std_logic;
  enable      :  in  std_logic;
```

```
  data_out    :  out std_logic_vector(7 downto 0);
  valid       :  out std_logic;
  ctrl        :  out std_logic;
  finish      :  out std_logic
  );
end component  serializer;
```

Listing A.2: Serializer component module in VHDL

```
component window_3x3 is
generic ( pixel_width       : integer:=8;
          image_row_width   : integer;
          valid_adj         : integer:=0);
port(
  sys_clk     : in  std_logic;
  reset       : in  std_logic;
  valid       : out std_logic;
  finish      : out std_logic;
  enable      : in  std_logic;
  data_in     : in  std_logic_vector(pixel_width-1 downto 0);
  w11         : out std_logic_vector(pixel_width-1 downto 0);
  w12         : out std_logic_vector(pixel_width-1 downto 0);
  w13         : out std_logic_vector(pixel_width-1 downto 0);
  w21         : out std_logic_vector(pixel_width-1 downto 0);
  w22         : out std_logic_vector(pixel_width-1 downto 0);
  w23         : out std_logic_vector(pixel_width-1 downto 0);
  w31         : out std_logic_vector(pixel_width-1 downto 0);
  w32         : out std_logic_vector(pixel_width-1 downto 0);
  w33         : out std_logic_vector(pixel_width-1 downto 0));
end component window_3x3;
```

Listing A.3: 3 x 3 Moving window component module in VHDL

```
type Med_data_in is array (8 downto 0) of std_logic_vector(7
   downto 0);

component Med_Filter is
generic( win_elements   : natural := 9;
         win_width      : natural := 3;
         win_height     : natural := 3);
port(
  sys_clk     : in  std_logic;
  reset_n     : in  std_logic;
  enable      : in  std_logic;
```

```
  valid      : out std_logic;
  data_in    : in  Med_data_in;
  data_out   : out std_logic_vector(7 downto 0));
end component Med_Filter;
```

Listing A.4: Median filter component module in VHDL

```
type Grad_data_in is array (8 downto 0) of std_logic_vector(7
   downto 0);

component Gradient_and_thresholding is
generic( win_elements       : natural := 9;
         win_width          : natural := 3;
         win_height         : natural := 3;
         divider_threshold  : natural :=14;
         image_row_width    : natural);
port(
  sys_clk    :  in  std_logic;
  reset      :  in  std_logic;
  enable     :  in  std_logic;
  valid      :  out std_logic;
  Finish     :  out std_logic;
  data_in    :  in  Grad_data_in;
  data_out   :  out std_logic_vector(7 downto 0));
end component Gradient_and_thresholding;
```

Listing A.5: Morphological gradient and thresholding component module in VHDL

```
component SIPO is
generic( image_row_width    : integer);
port(
    sys_clk       : in  std_logic;
    reset         : in  std_logic;
    valid         : out std_logic;
    finish        : out std_logic;
    enable        : in  std_logic;
    data_in       : in  std_logic_vector(7 downto 0);
    data_out      : out std_logic_vector(31 downto 0));
end component SIPO ;
```

Listing A.6: Serial-In-Parallel-Out (SIPO) component module in VHDL

```
component Comparator
port(
  sys_clk     :  std_logic;
  enable      :  std_logic;
  num1        :  in  std_logic_vector(7 downto 0); --input 1
  num2        :  in  std_logic_vector(7 downto 0); --input 2
  High_Out    :  out std_logic_vector(7 downto 0);
  Low_Out     :  out std_logic_vector(7 downto 0));
end component Comparator;

component Register_8bit
port (
  sys_clk     : in  std_logic;
  reset_n     : in  std_logic ;
  Reg_IN      : in  std_logic_vector (7 downto 0) ;
  Reg_OUT     : out std_logic_vector (7 downto 0));
end component Register_8bit;
```

Listing A.7: Comparator and Register module in VHDL

```
Component Watershed_Ctrl is
generic( G_APPL_AWIDTH    : natural:=16;
         G_APPL_DWIDTH    : natural:=32;
         G_SYSBUS_AWIDTH  : natural:=32);
port(
  sys_clk                 : in  std_logic;
  reset_n                 : in  std_logic;
  P_ctrl_start_sl         : in  std_logic;
  P_WR_Start_Addr_slv     : out std_logic_vector(G_SYSBUS_AWIDTH
     -1 downto 0);
  P_WR_Init_sl            : out std_logic;
  P_WR_Busy_sl            : in  std_logic;
  P_WR_Data_slv           : out std_logic_vector(G_APPL_DWIDTH-1
     downto 0);
  P_WR_we_sl              : out std_logic;
  P_WR_Backwards          : out std_logic;
  P_WR_Flush              : out std_logic;
  P_RD_Start_Addr_slv     : out std_logic_vector(G_SYSBUS_AWIDTH
     -1 downto 0);
  P_RD_Init_sl            : out std_logic;
  P_RD_Busy_sl            : in  std_logic;
  P_RD_Data_slv           : in  std_logic_vector(G_APPL_DWIDTH-1
     downto 0);
  P_RD_re_sl              : out std_logic;
```

```
  P_RD_Backwards          : out std_logic);
end component Watershed_Ctrl;
```

Listing A.8: Watershed_Ctrl module (top level) in VHDL

```
entity Watershed_Algo_Processing_Step is
generic( pixel_width         : natural:=32;
         image_row_width     : natural:=512;
         image_height        : natural:=512;
         VMAX                : natural:=255;
         LMAX                : natural:=8000);
port(
    sys_clk            : in  std_logic;
    reset_n            : in  std_logic;
    Enable_Module      : in  std_logic;
    Enable_Data_In     : in  std_logic;
    Read_Data_Finish   : out std_logic;
    Output_Valid       : out std_logic;
    Finish             : out std_logic;
    Next_Scan          : out std_logic;
    Data_In            : in  std_logic_vector(pixel_width downto
       0);
    Data_Out           : out std_logic_vector(pixel_width downto
       0));
end Watershed_Algo_Processing_Step;
```

Listing A.9: Watershed_Algo_Processing_Step module in VHDL

Bibliography

[1] S. Beucher, F. Meyer, *The morphological approach to segmentation: The watershed transformation*, Mathematical Morphology in Image processing, Marcel Dekker Inc, New York, pp.433-481, 1993.

[2] A. Bieniek, A. Moga, *An efficient watershed algorithm based on connected components*, Pattern Recognition 33, pp.907-916, 2000.

[3] Dang Ba Khac Trieu and Tsutomu Maruyama, *An implementation of parallel and pipelined watershed algorithm in FPGA*, International Conference on Field-Programmable Logic and Applications, pp.561-566, 2006.

[4] Md. Shakowat Zaman Sarker, Tan Wooi Haw and Rajasvaran Logeswaran, *Morphological based technique for image segmentation*, International Journal of Information Technology, Vol. 14, No. 1.

[5] Manisha Bhagwat, R. K. Krishna and Vivek Pise, *Simplified Watershed Transformation*, International Journal of Computer Science and Communication, Vol. 1, No. 1., pp. 175-177, 2010

[6] K. Haris and S. Efstratiadis, *Hybrid Image Segmentation using Watersheds and Fast Region Merging*, IEEE Transaction on Image Processing, 1998.

[7] M. Sonka, V. Hlavac, and R. Boyle, *Image Processing, Analysis, and Machine Vision*, PWS Publishing, 1999.

[8] D. Comaniciu, *Mean Shift: A robust Approach Toward Feature Space Analysis*, IEEE Transaction on PMAI, Vol. 24, No. 5, pp. 609-619, 2002

[9] Z. Vasicek and L. Sekanina, *Novel Hardware Implementation of Adaptive Median Filters*, Design and Diagnostics of Electronic Circuits and Systems, 11th IEEE Workshop, 2008.

[10] Prof. R. Szeliski, *Lecture notes, Universiteit Utrecht, Department of Information and Computing Sciences*, http://www.cs.uu.nl/docs/vakken/ibv/reader/chapter10.pdf

[11] Dr. Sukhendu Das, *Lecture notes, IIT Madras, India*, http://vplab.iitm.ac.in/courses/CV_DIP/PDF/lect-Segmen.pdf

[12] Robert Fisher, Simon Perkins, Ashley Walker and Erik Wolfart, *Hypermedia Image Processing Reference (HIPR2)*, http://homepages.inf.ed.ac.uk/rbf/HIPR2/hipr_top.htm

[13] Jianbo Shi and Jitendra Malik, *Normalized Cuts and Image Segmentation*, Proc. IEEE Conf. Computer Vision and Pattern Recognition

[14] Gonzalez and Woods, *Digital Image Processing, 2nd ed.*, Prentice hall publication, ISBN number 0201180758.

[15] Donald G. Bailey, *Design for Embedded Image Processing on FPGAs*, John Wiley & Sons (Asia) Pte Ltd, ISBN number 9780470828496.

[16] Wen-Xiong Kang, Qing-Qiang Yang, Run-Peng Liang, *The Comparative Research on Image Segmentation Algorithms*, First International Workshop on Education Technology and Computer Science, pp.703-707, 2009.

[17] Johan De Bock and Wilfried Philips, *Line Segment Based Watershed Segmentation*, MIRAGE 2007, LNCS 4418, pp. 579586, 2007.

[18] Edward R. Dougherty and Roberto A. Lotufo, *Hands-on morphological image processing*, SPIE Press, 2003.

[19] *The Berkeley Segmentation Dataset and Benchmark*, http://www.eecs.berkeley.edu/Research/Projects/CS/vision/bsds/

[20] *MATLAB Notes*, http://www.mathworks.de/company/newsletters/news_notes/win02/watershed.html

[21] *User Guide*, Xilinx LogiCORE IP FIFO Generator v4.2 and Divider Generator v3.0.

[22] *User Guide*, Virtex-4 FPGA by Xilinx.

Declaration

All the work contained within this thesis, except where otherwise acknowledged, was solely the effort of the author. At no stage was any collaboration entered into with any other party.

(Sameer Ruparelia)