Hindawi Publishing Corporation
EURASIP Journal on Embedded Systems
Volume 2009, Article ID 318654, 19 pages
doi:10.1155/2009/318654

Research Article

Efficient Processing of a Rainfall Simulation Watershed on an FPGA-Based Architecture with Fast Access to Neighbourhood Pixels

Lee Seng Yeong, Christopher Wing Hong Ngau, Li-Minn Ang, and Kah Phooi Seng

School of Electrical and Electronics Engineering, The University of Nottingham, 43500 Selangor, Malaysia

Correspondence should be addressed to Lee Seng Yeong, yls@tm.net.my

Received 15 March 2009; Accepted 9 August 2009

Recommended by Ahmet T. Erdogan

This paper describes a hardware architecture to implement the watershed algorithm using rainfall simulation. The speed of the architecture is increased by utilizing a multiple memory bank approach to allow parallel access to the neighbourhood pixel values. In a single read cycle, the architecture is able to obtain all five values of the centre and four neighbours for a 4-connectivity watershed transform. The storage requirement of the multiple bank implementation is the same as a single bank implementation by using a graph-based memory bank addressing scheme. The proposed rainfall watershed architecture consists of two parts. The first part performs the arrowing operation and the second part assigns each pixel to its associated catchment basin. The paper describes the architecture datapath and control logic in detail and concludes with an implementation on a Xilinx Spartan-3 FPGA.

1. Introduction

Image segmentation is often used as one of the main stages in object-based image processing. For example, it is often used as a preceding stage in object classification [1–3] and object-based image compression [4–6]. In both these examples, image segmentation precedes the classification or compression stage and is used to obtain object boundaries. This leads to an important reason for using the watershed transform for segmentation as it results in the detection of closed boundary regions. In contrast, boundary-based methods such as edge detection detect places where there is a difference in intensity. The disadvantage of this method is that there may be gaps in the boundary where the gradient intensity is weak. By using a gradient image as input into the watershed transform, qualities of both the region-based and boundary-based methods can be obtained.

This paper describes a watershed transform implemented on an FPGA for image segmentation. The watershed algorithm chosen for implementation is based on the rainfall simulation method described in [7–9]. There is an implementation of a rainfall-based watershed algorithm on hardware proposed in [10], using a combination of a DSP and an FPGA. Unfortunately, the authors do not give much details on the hardware part and their architecture. Other sources have implemented a watershed transform on reconfigurable hardware based on the immersion watershed techniques [11, 12]. There are two advantages of using a rainfall-based watershed algorithm over the immersion-based techniques. The first advantage is that the watershed lines are formed in-between the pixels (zero-width watershed). The second advantage is that every pixel would belong to a segmented region. In immersion-based watershed techniques, the pixels themselves form the watershed lines. A common problem that arises from this is that these watershed lines may have a width greater than one pixel (i.e., the minimum resolution in an image). Also, pixels that form part of the watershed line do not belong to a region. Other than leading to inaccuracies in the image segmentation, this also slows down the region merging process that usually follows the calculation of the watershed transform. Other researchers have proposed using a hill-climbing technique for their watershed architecture [13]. This technique is similar to that of rainfall simulation except that it starts from the minima and climbs by the steepest slope. With suitable modifications, the techniques

proposed in this paper can also be applied for implementing a hill-climbing watershed transform.

This paper describes a hardware architecture to implement the watershed algorithm using rainfall simulation. The speed of the architecture is increased by utilizing a multiple memory bank approach to allow parallel access to the neighbourhood pixel values. This approach has the advantage of allowing the centre and neighbouring pixel values to be obtained in a single clock cycle without the need for storing multiple copies of the pixel values. Compared to the memory architecture proposed in [14], our proposed architecture is able to obtain all five values required for the watershed transform in a single read cycle. The method described in [14] requires two read cycles, one read cycle for the centre pixel value using the Centre Access Module (CAM) and another read cycle for the neighbouring pixels using the Neighbourhood Access Module (NAM).

The paper is structured as follows. Section 2 will describe the implemented watershed algorithm. Section 3 will describe a multiple bank memory storage method based on graph analysis. This is used in the watershed architecture to increase processing speed by allowing multiple values (i.e., the centre and neighbouring values) to be read in a single clock cycle. This multiple bank storage method has the same memory requirement as methods which store the pixel values in a single bank. The watershed architecture is described in two parts, each with their respective examples. The parts are split up based on their functions in the watershed transform as shown in Figure 1. Section 4 describes the first part of the architecture, called "Architecture-Arrowing" which is followed by an example of its operation in Section 5. Similarly, Section 6 describes the second part of the architecture, called "Architecture-Labelling" which is followed by an example of its operation in Section 7. Section 8 describes the synthesis and implementation on a Xilinx Spartan-3 FPGA. Section 9 summarizes this paper.

2. The Watershed Algorithm Based on Rainfall Simulation

The watershed transformation is based on visualizing an image in three dimensions: two spatial coordinates versus grey levels. The watershed transform used is based on the rainfall simulation method proposed in [7]. This method simulates how falling rain water flows from higher level regions called peaks to lower level regions called valleys. The rain drops that fall over a point will flow along the path of the steepest descent until reaching a minimum point.

The general processes involved in calculating the watershed transform is shown in Figure 1. Generally, a gradient image is used as input to the watershed algorithm. By using a gradient image the catchment basins should correspond to the homogeneous grey level regions of the image. A common problem to the watershed transform is that it tends to oversegment the image due to noise or local irregularities in the gradient image. This can be corrected using a region merging algorithm or by preprocessing the image prior to the application of the watershed transform.

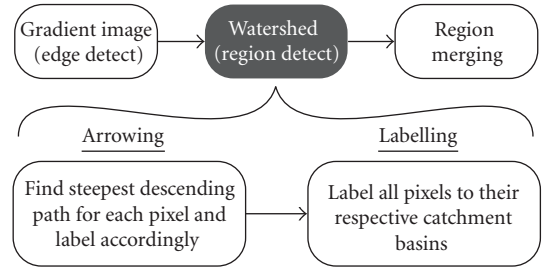


Figure 1: General preprocessing and postprocessing steps involved when using the watershed. Also it shows the two main steps involved in the watershed transform. Firstly find the direction of steepest descending path and label the pixels to point in that direction. Using the direction labels, the pixels will be relabelled to match the label of their corresponding catchment basin.

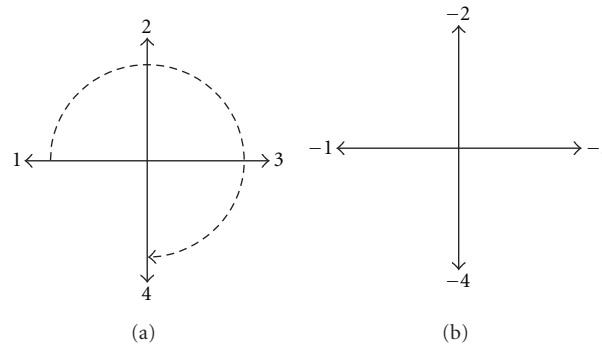


Figure 2: The steepest descending path direction priority and naming convention used to label the direction of the steepest descending path. (a) shows the criterion used when determining order of steepest descendent path when there is more than one possible path; that is, the pixel has two or more lower neighbours with equivalent values. Paths are numbered in increasing priority from the left moving in a clockwise direction towards the right and to the bottom. Shown here is the path with the highest priority labelled as 1 to the lowest priority, labelled as 4. (b) shows labels used to indicate direction of the steepest descent path. The labels shown correspond with the direction of the arrows.

The watershed transform starts by labelling each input pixel to indicate the direction of the steepest descent. In other words, each pixel points to its neighbour with the smallest value. There are two neighbour connectivity approaches that can be used. The first approach called 8-connectivity considers all eight neighbours surrounding the pixel and the second approach called 4-connectivity only considers the neighbours to its immediate north, south, east, and west. In this paper, we use the 4-connectivity approach. The direction labels are chosen to be negative values from $-1 \rightarrow -4$ so that it will not overlap with the catchment basin labelling which will start from 1. These direction labels are shown in Figure 2. There are four different possible direction labels for each pixel for neighbours in the vertical and horizontal directions. This process of finding the steepest descending path is repeated for all pixels so that every pixel will point

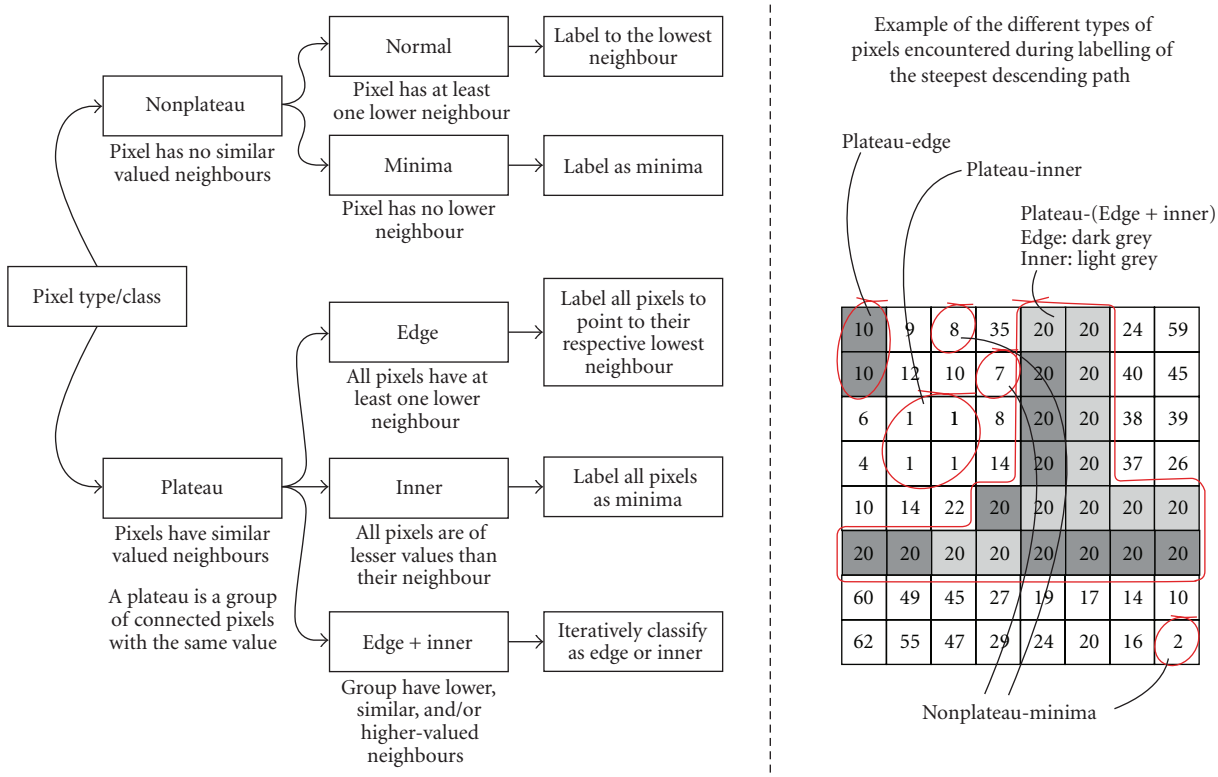


Figure 3: Various arrowing conditions that occur.

to the direction of steepest descent. If a pixel or a group of similar valued pixels which are connected has no neighbours with a lower value, it becomes a regional minima. Following the steepest descending paths for each pixel will lead to a minimum (or regional minima). All pixels along the steepest descending path will be assigned the label of that minimum to form a catchment basin. Catchment basins are formed by the minimum and all pixels leading to it. Using this method, the region boundary lines are formed by the edges of the pixels that separate the different catchment basins.

The earlier description assumed that there will always be only one lower-valued neighbour or none at all. However, this is often not the case. There are two other conditions that can occur during the pixel labelling operation: (1) when there is more than one steepest descending paths because two or more lowest-valued neighbours have the same value, and (2) when the current pixel value is the same as any of its neighbours. The second condition is called a plateau condition and increases the complexity in determining the steepest descending path.

These two conditions are handled as follows.

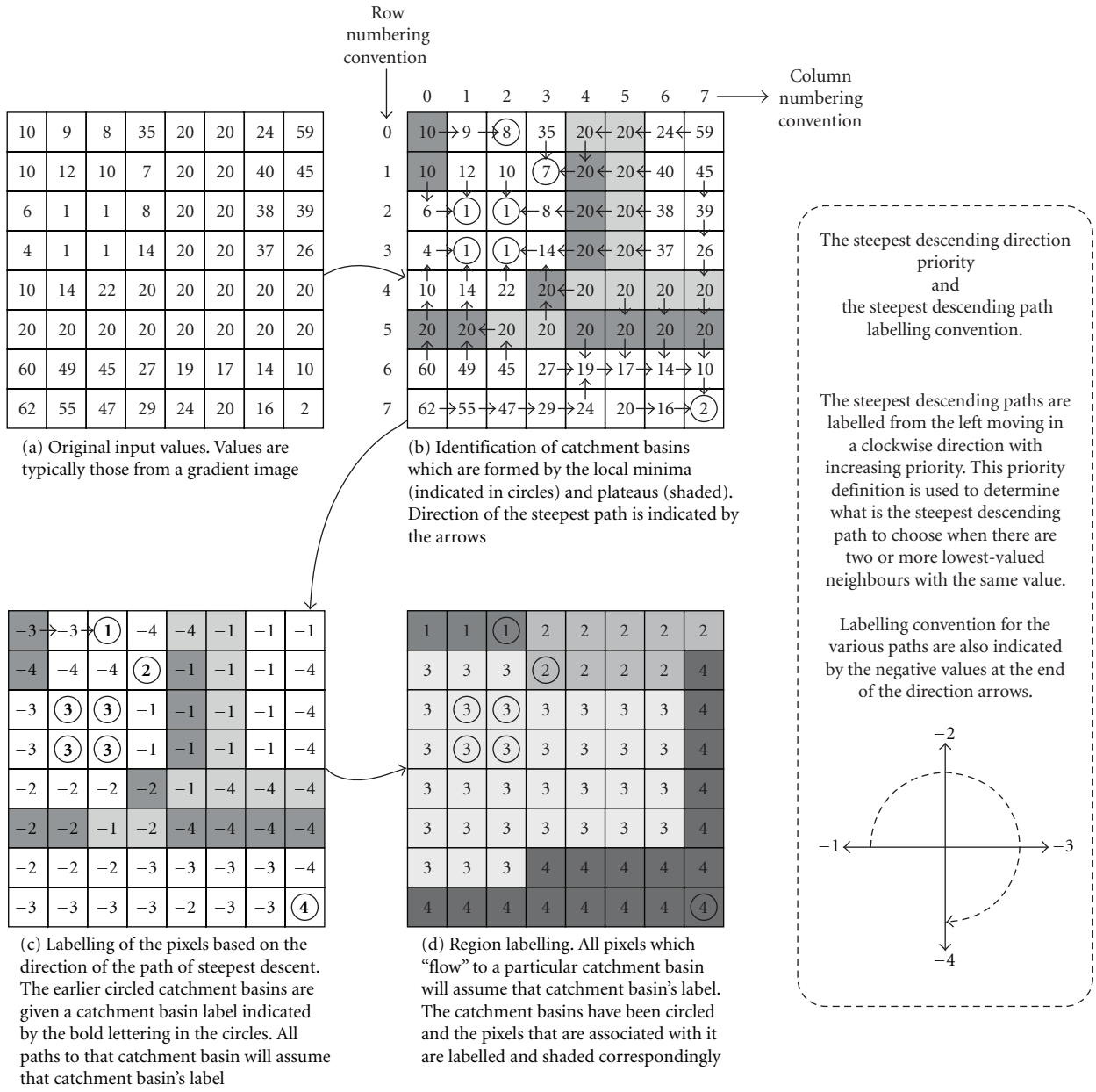
(1) If a pixel has more than one steepest descending path, the steepest descending path is simply selected based on a predefined priority criterion. In the proposed algorithm, the highest priority is given to those going up from the left and decreases as we move to the right and down. The order of priority is shown in Figure 2.

(2) If the image has regions where the pixels have the same value and are not a regional minimum, they are called nonminima plateaus. The nonminima plateaus are a group of pixels which can be divided into two groups.

(i) *Descending edge pixels of the plateau*. This group consists of every pixel in the plateau which has a neighbour with a lower value. These pixels simply labelled with the direction to their lower-valued neighbour.

(ii) *Inner pixels*. This group consists of every pixel whose neighbours have equal or higher values than its own value.

Figure 3 shows a summary of the various arrowing conditions that may occur. Normally, the geodesic distances from the inner points to the descending edge are determined to obtain the shortest path. In our watershed transform this step has been simplified by eliminating the need to explicitly calculate and store the geodesic distance. The method used can be thought of as a shrinking plateau. Once the edges of a plateau has been labelled with the direction of the steepest descent, the inner pixels neighbouring these edge pixels will point to those edges. These edges will be "stripped" and the neighbouring inners will become the new edges. This is performed until all the pixels in the plateau have been labelled with the path of steepest descent (see Section 4.7 for more information).



(a) Original input values. Values are typically those from a gradient image

(b) Identification of catchment basins which are formed by the local minima (indicated in circles) and plateaus (shaded). Direction of the steepest path is indicated by the arrows

(c) Labelling of the pixels based on the direction of the path of steepest descent. The earlier circled catchment basins are given a catchment basin label indicated by the bold lettering in the circles. All paths to that catchment basin will assume that catchment basin's label

(d) Region labelling. All pixels which "flow" to a particular catchment basin will assume that catchment basin's label. The catchment basins have been circled and the pixels that are associated with it are labelled and shaded correspondingly

Figure 4: Example of four-connectivity watershed performed on an 8×8 sample data. (a) shows the original gradient image values. (b) shows the direction of the steepest descending path for each pixel. Minima are highlighted with circles. (c) shows pixels where the steepest descending paths and minima have been labelled. The labels used for the direction of the steepest descending path are shown on the right side of the figure. (d) shows the 8×8 data fully labelled. The pixels have been assigned to the label of their respective minima forming a catchment basin.

The final step once all the pixels have been labelled with the direction of steepest descent is to assign them labels that correspond to the label of their respective minimum/minima. This is done by scanning each pixel and to follow the path indicated by each pixel to the next pixel. This is performed repeatedly until a minimum/minima is reached. All the pixel in the path are then assigned to the label of that minimum/minima. An example of all the algorithm steps is shown in Figure 4. The operational flowchart of the watershed algorithm is shown in Figure 5.

3. Graph-Based Memory Implementation

Before going into the details of our architecture, we will discuss a multiple bank memory storage scheme based on graph analysis. This is used to speed up operations by allowing all five pixel values required for the watershed transform to be read in a single clock cycle with the same memory storage requirement as a single bank implementation. A similar method has been proposed in [14]. However, their method requires twice the number of read cycles compared

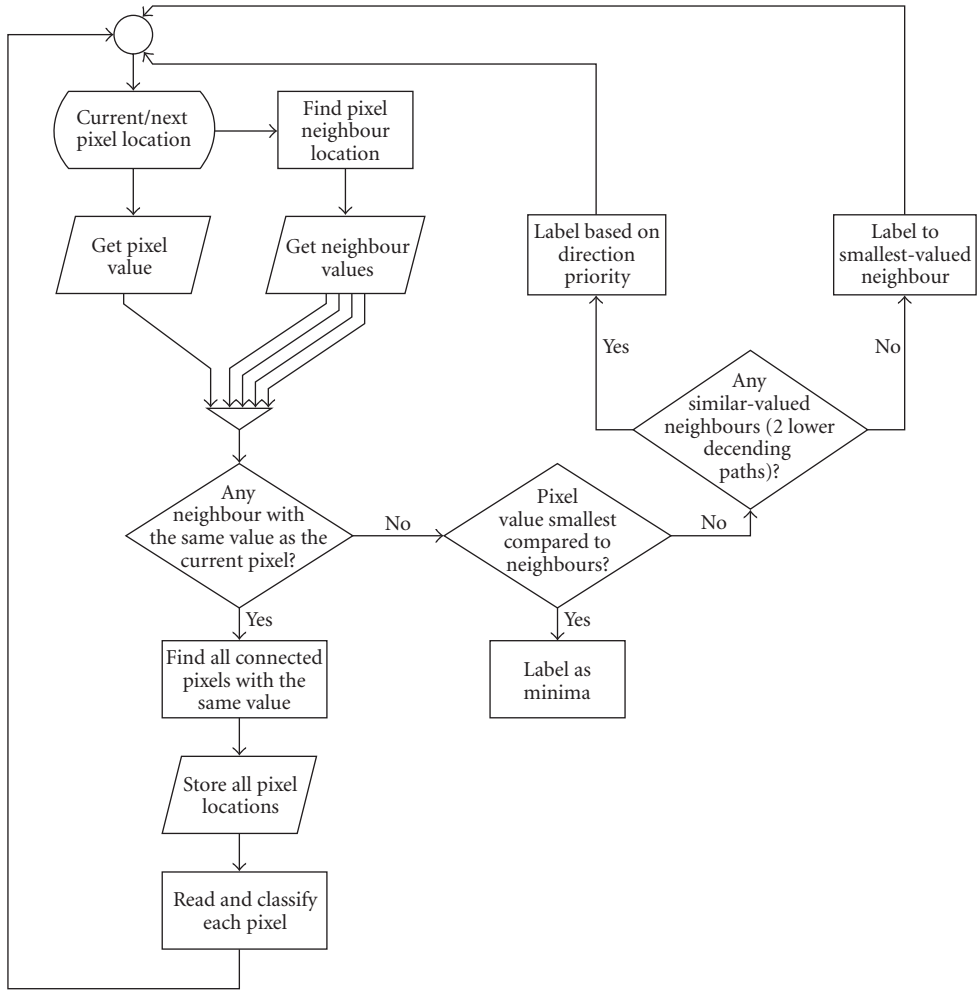


Figure 5: Watershed algorithm flowchart.

to our proposed method. Their proposed method requires two read cycles, one to obtain the centre value and another to obtain the neighbourhood values. This effectively doubles the number of clock cycles required for reading the pixel values.

To understand why this is important, recall that one of the main procedures of the watershed transform was to find the path of the steepest descent. This required the values of the current and neighbouring pixels. Traditionally, these values can be obtained using

(1) sequential reads: a single memory bank the size of the image is read five times, requiring five clock cycles,

(2) parallel read: it reads five replicated memory banks each size of the image. This requires five times more memory required to store a single image but all required values can be obtained in a single clock cycle.

Using this multiple bank method, we can obtain the speed advantage of the parallel read with the nonreplicating storage required by the sequential reading method. The advantages of using this multiple bank method are to

(1) reduce the memory space required for storing the image by up to five times,

(2) obtain all values for the current pixel and its neighbours in a single read cycle, eliminating the need for a five clock cycle read.

This multiple bank memory storage stores the image in separate memory banks. This is not a straightforward division of the image pixels by the number of memory banks, but a special arrangement is required that will not overlap and that will support the access to five banks simultaneously to obtain the five pixel values (Centre, East, North, South, West). The problem now is to

(1) determine the number of banks required to store the image,

(2) fill the banks with the image data,

(3) access the data in these banks.

All of these steps shall be addressed in the following sections in the order listed above.

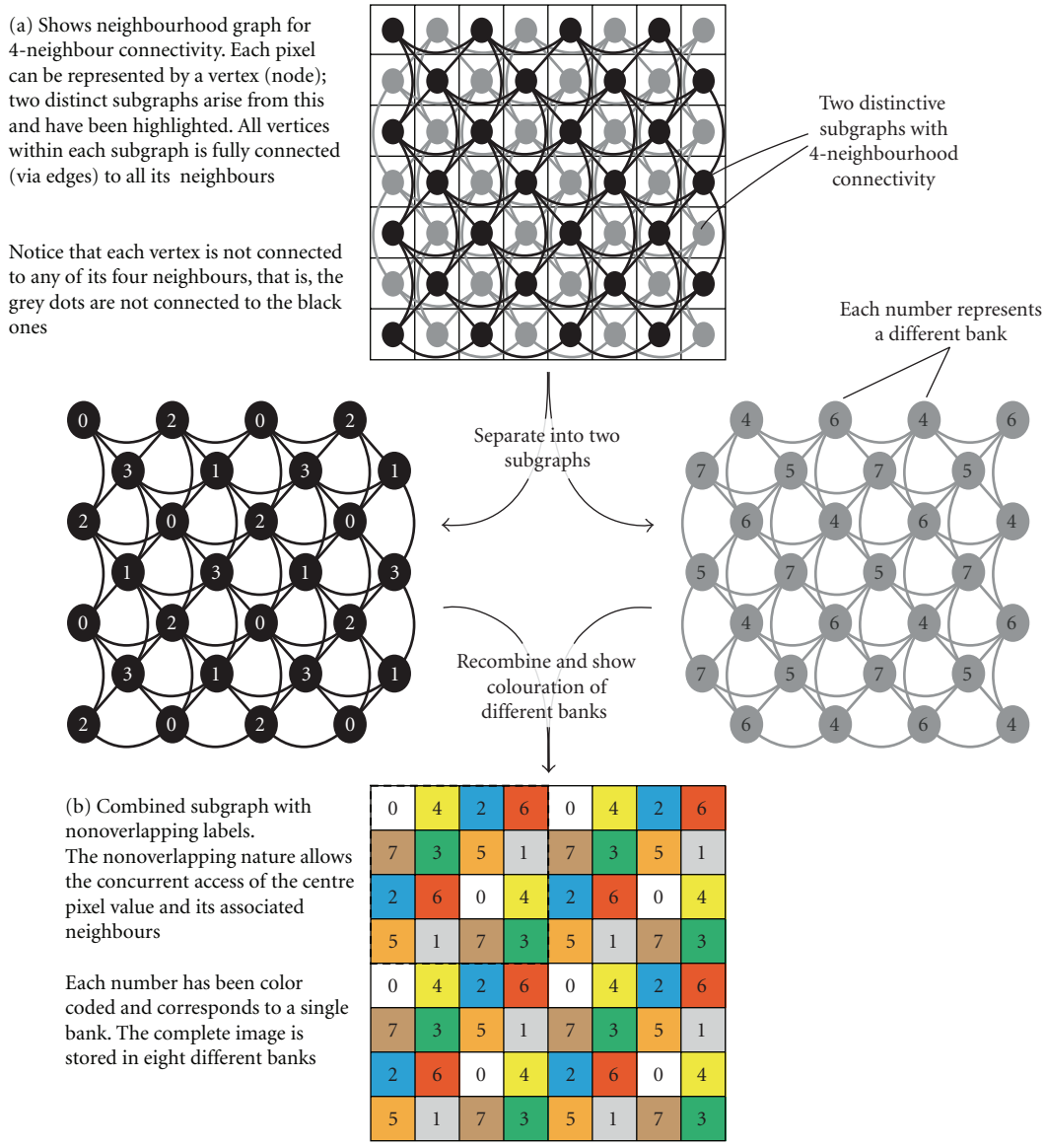


FIGURE 6: N_4 connectivity graph. Two sub-graphs combined to produce an 8-bank structure allowing five values to be obtained concurrently.

3.1. Determining How Many Banks Are Needed. This section will describe how the number of banks needed to allow simultaneous access is determined. This depends on (1) the number of neighbour connectivity and (2) the number of values to be obtained in one read cycle. Here, graph theory is used to determine the minimum number of databanks required to satisfy the following:

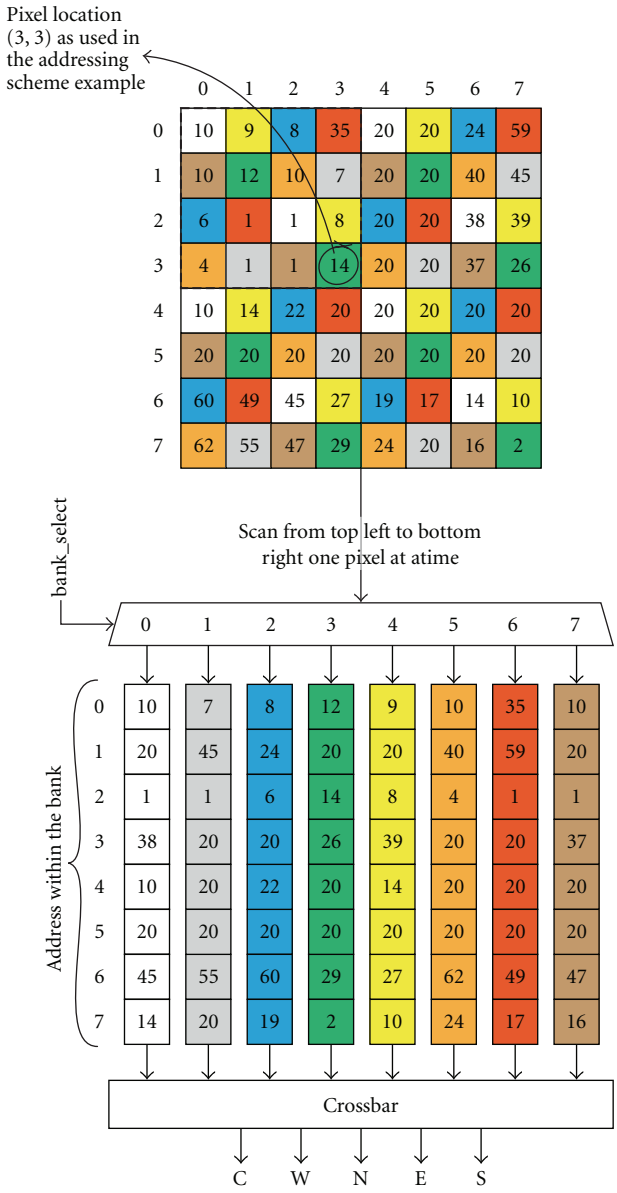
(1) any of the values that we want cannot be from the same bank;

(2) none of the image pixels are stored twice (i.e., no redundancy).

Satisfying these criteria results in the minimum number of banks required with no additional memory needed compared to a standard single bank storage scheme.

Imagine every pixel in an image as a region and a vertex (node) will be added to each pixel. For 4-neighbour connectivity (N_4), the connectivity graph is shown in Figure 6. To determine the number of banks for parallel access can be viewed as a graph colouration problem, whereby any of the parallel values cannot be from the same bank. We ensure that each of the nodes will have a neighbour of a different colour, or in our case number. Each of these colours (or numbers) corresponds to a different bank. The same method can be applied for different connectivity schemes such as 8-neighbour connectivity.

In our implementation of 4-neighbourhood connectivity and five concurrent memory access (for five concurrent values), we require eight banks. In the discussion and examples to follow, we will use these implementation criteria.



(a) Using cardinal directions, CWNES are the centre, west, north, east, and south values, respectively. These correspond to the current pixel, left, top, right, and bottom neighbour values

(b) Any filling order is possible. For any filling order, the bank and address within the bank is determined by the same logic in the address bar (see Figure 8) Using a traditional raster scan pattern as an example. The order of bank_select is

$0 \to 4 \to 2 \to 6 \to 0 \to 4 \to 2 \to 6 \to 7 \to 3 \to 5 \to 1 \to 7 \to 3 \to 5 \to 1 \to 2 \to 6 \to 0 \to 4 \ \ldots 5 \to 1 \to 7 \to 3$

Figure 7: Block diagram of graph-based memory storage and retrieval.

3.2. Filling the Banks. After determining how many banks are needed, we will need to fill the banks. This is done by writing the individual values one at a time into the respective banks. During the determination of the number of required banks, a pattern emerges from the connectivity graph. An example of this pattern is highlighted with a detached bounding box in Figures 6 and 7.

The eight banks are filled with one value at a time. This can be done in any order. The bank number and bank address is calculated using some logic. The same logic is used to determine the bank and bank address during reading (See Section 3.3 for more details on this). For the ease of explanation, we shall adopt a raster scan type of sequence. Using this convention, the order of filling is simply the order of the bank number as it appears from top-left to bottom-right. An example of this is shown in Figure 7.

The group of banks replicates itself every four pixels in either direction (i.e., right and down). Hence, to determine how many times the pattern is replicated, the image size is simply divided by sixteen. Alternatively, any one of its sides can be divided by four since all images are square. This is important as the addressing for filling the banks (and reading) holds true for square images whose sizes are to the power of two (i.e. $2^2, 2^3, 2^4$). Image sizes which are not square are simply padded.

3.3. Accessing Data in the Banks. To access the data from this multiple bank scheme, we need to know (1) which bank and (2) location within that bank. The addressing scheme is a simple addressing scheme based on the pixel location. A hardware unit called the Address Processor (AP) handles the memory addressing. By providing the AP with the pixel location, it will calculate the address to retrieve that pixel value. This address will tell us which bank and location within that bank the pixel value is stored in.

To understand how the AP works, consider a pixel coordinate which consists of a row and column value with the origin located at the upper left corner. These two values are represented in their binary form and the lowest significant bits for the column and row are used to determine the bank. The number of bits required to represent the number of banks is dependent on the total number of banks in this multiple bank scheme. In our case of eight banks, three bits from the address are needed to determine in which bank the value for that particular pixel location is stored in. These binary values go through some logic as shown in Figure 8 or in equation form:

$$\begin{aligned} B[2] &= r[0]c[0]' + c[0]r[0]', \\ B[1] &= r[1]'r[0]'c[1] + r[1]r[0]'c[0]' \\ &\quad + r[1]'r[0]c[0]' + r[1]r[0]c[0], \\ B[0] &= r[0], \end{aligned} \tag{1}$$

where $B[0 \to 2]$ represent the three bits that determine the bank number (from $0 \to 7$). $r[0]$ and $r[1]$ represent the first two bits of the row value in binary while $c[0]$ and $c[1]$ represent the first two bits of the column value in binary.

Now that we have determined which bank the value is in; the remainder of the bits is used to determine the location of the value within the bank. An example is given in Figure 8(a).

For an image of size y-rows and x-columns, the number of bits required for addressing will simply be the number of bits required to store the largest value of the row and column in binary, that is, $no\_of\_address\_bits = \log_2(x) + \log_2(y)$.

This addressing scheme is shown in Figure 8. (Note that the steps described here assume an image with a minimum size of 4×4 and increase in powers of 2).

3.4. Sorting the Data from the Banks. After obtaining the five values from the banks, they need to be sorted according to the expected neighbour location output to ensure that values of a particular direction is sent to the right output position. This sorting is handled by another hardware unit called the Crossbar (CB). In addition, the CB also tags invalid values from invalid neighbour conditions which occur at the corners and edges of the image. This tagging is part of the output multiplexer control.

The complete structure for reading from the banks is shown in Figure 9. In this figure, five pixel locations are fed into the AP which generates five addressees, for the centre and its four neighbours. These five addresses are fed into all eight banks. However, only the address corresponding to the correct bank is chosen by the add_sel_x, where $x = 0 \rightarrow 7$. The addresses fed into the banks will generate eight values however, only five will be chosen by the CB. These values are also sorted using the CB to ensure that the values corresponding to the centre pixel and a particular neighbour are output onto the correct data lines. The mux control, CB_sel_x, is controlled by the same logic that selects the add_sel_x.

4. Arrowing Architecture

This section will provide the details on the architecture that performs the arrowing function of the algorithm. This part of the architecture will describe how we get from Figure 4(a) to Figure 4(c) in hardware. As mentioned in the previous description of the algorithm, things are simple when every pixel has a lower neighbour and gets more complicated due to plateau conditions. Similarly, this plateau condition complicates the architecture. Adding to this complexity is the fact that all neighbour values are obtained simultaneously, and instead of processing one value at a time, we have to process five values, the centre and its four neighbours. This part of the architecture that performs the arrowing is shown in Figure 10.

When a pixel location is fed into the system, it enters the "Centre and Neighbour Coordinates" block. From this, the coordinates of the centre and its four neighbours are output and fed into the "Multibank Memory" block to obtain all the pixel values and the pixel status (PS) from the "Pixel Status" block.

Assuming the normal state, the input pixel will have a lower neighbour and no neighbours of the same value, that is, inner = 0 and plat = 0. The pixel will just be arrowed to the

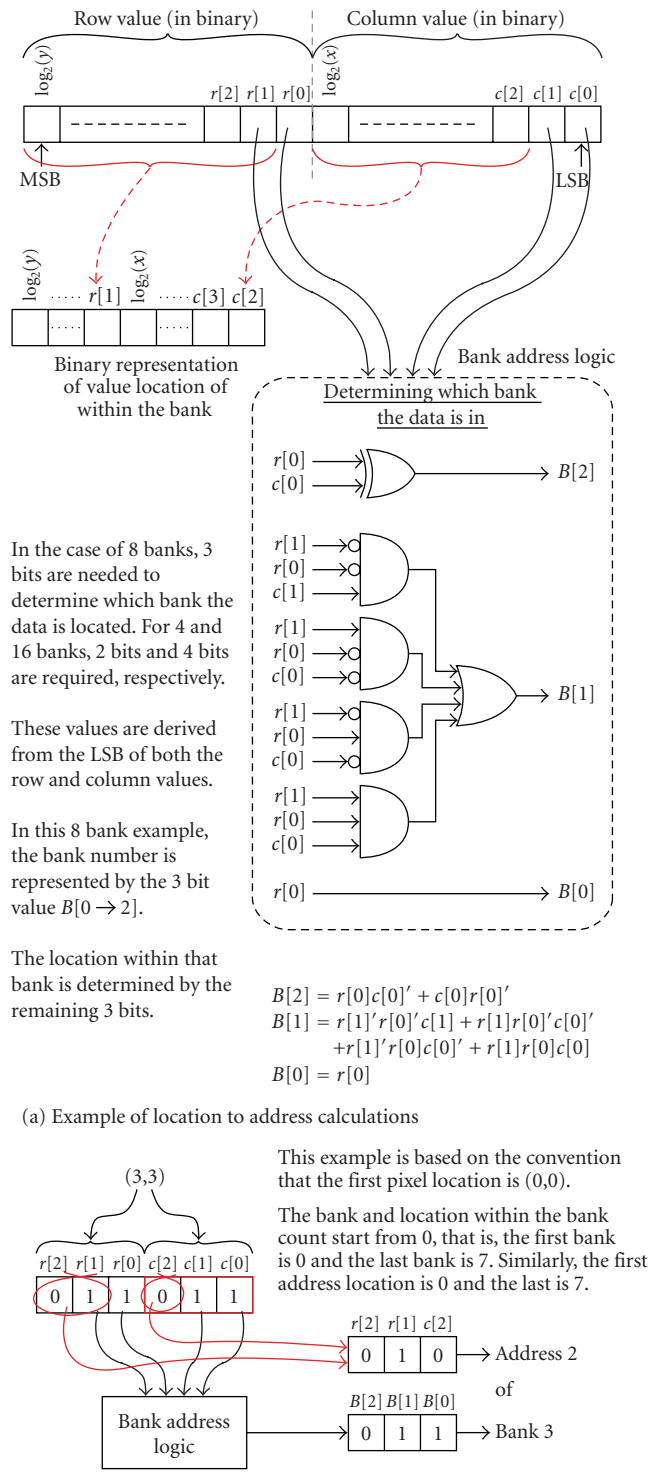


FIGURE 8: The addressing scheme for the multiple bank graph-based memory storage.

nearest neighbour. The Pixel Status (PS) for that pixel will be changed from $0 \rightarrow 6$ (See Figure 19).

However, if the pixel has a similar valued neighbour, plat = 1 and plateau processing will start. Plateau processing starts off by finding all the current pixel neighbours of similar value and writes them to Q1. Q1 is predefined to be the first

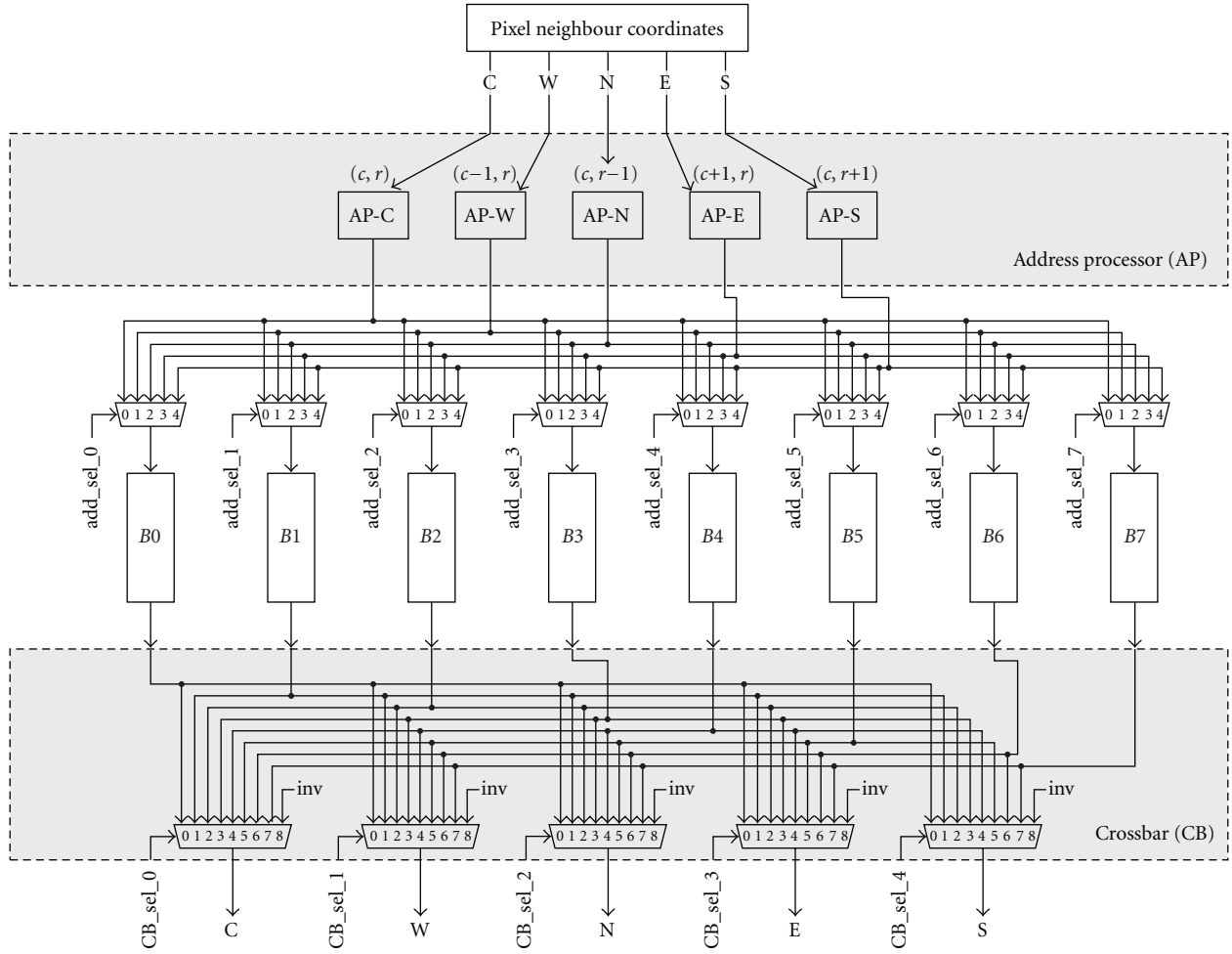


Figure 9: 8 Bank memory architecture.

queue to be used. After writing to the queue, the PS of the pixels is changed from 0 → 1. This is to indicate which pixel locations have been written to queue to avoid duplicate entries in the queue. At the end of this process, all the pixel locations belonging to the plateau will have been written to Q1.

To keep track of the number of elements in Q1_WNES, two sets of memory counters are used. These two sets of counters consist of mc1 → mc4 in one set and mc6 → mc9 in another. When writing to Q1_WNES, both sets of counters are incremented in parallel but when reading from Q1_WNES to obtain the neighbouring plateau pixels, only mc1–4 is decremented while mc6–9 remains unchanged. This means that, at the end of the Stage 1 processing, mc1–4 = 0 and mc6–9 will contain the count of the number of pixel locations which are contained within Q1_WNES. This is needed to handle the case of a lower complete minima (i.e., a plateau with all inner pixels). When this type of plateau is encountered, mc1–5 = 0, and Q1_WNES will be read once again using mc6–9, this time not to obtain the same valued neighbours but to label all the pixel locations within Q1_WNES with the current value stored in the minima register. Otherwise, mc5 > 0 and values will be read from Q1_C and subsequently from Q2_WNES and Q1_WNES until all the locations in the plateau have been visited and classified. The plateau processing steps and the associated conditions are shown in Figure 11.

There are other parts which are not shown in the main diagram but warrants a discussion. These are

(1) memory counters—to determine the number of unprocessed elements in a queue,

(2) priority encoder—to determine the controls for Q1_sel and Q2_sel.

The rest of the architecture consists of a few main parts shown in Figure 10 and are

(1) centre and neighbour coordinates—to obtain the centre and neighbour locations,

(2) multibank memory—to obtain the five required pixel values,

(3) smallest-valued neighbour—to determine which neighbour has the smallest value,

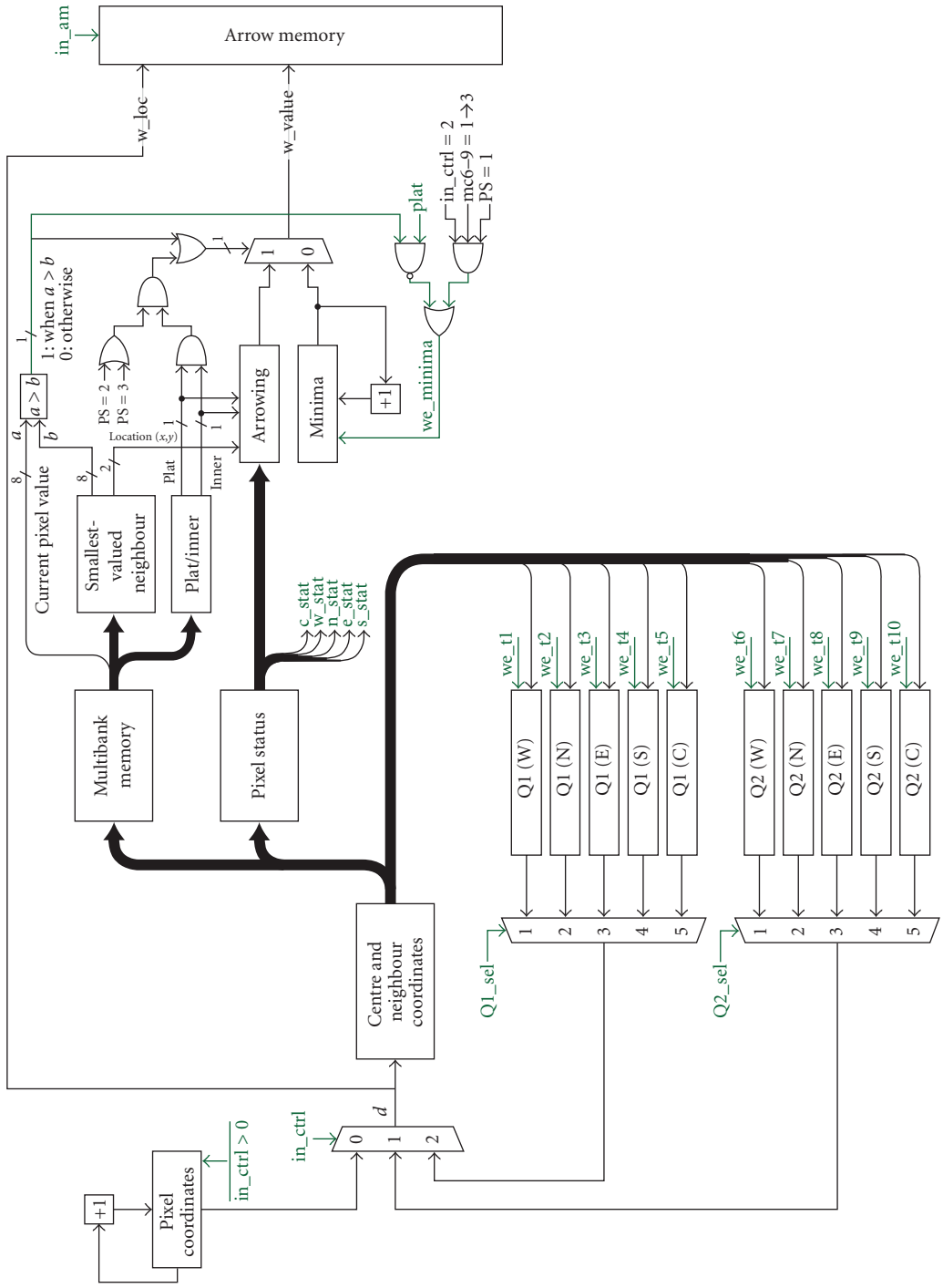


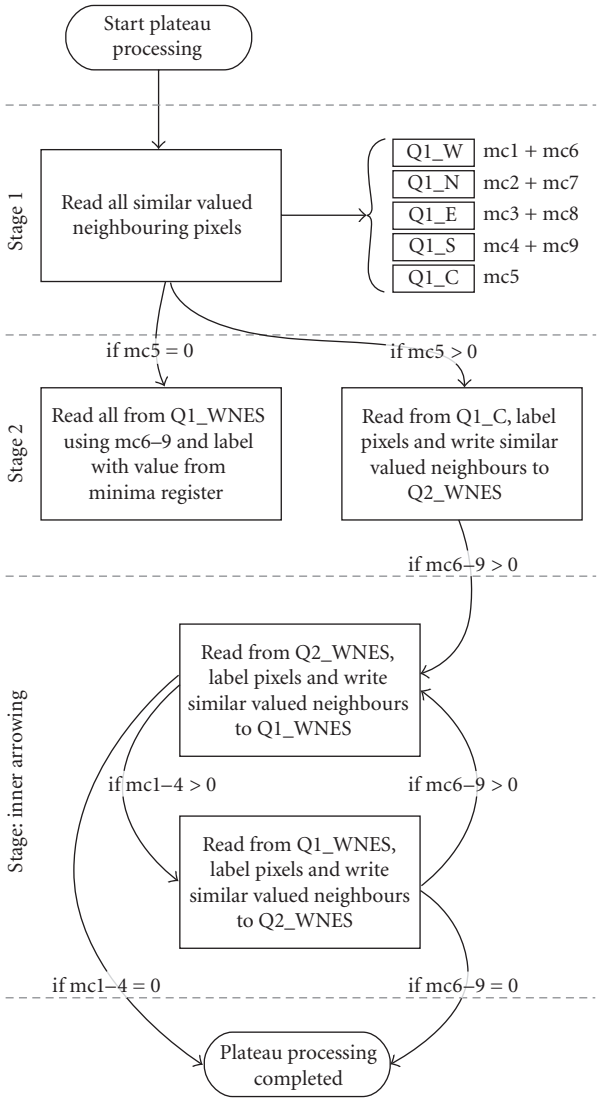
Figure 10: Watershed architecture based on rainfall simulation. Shown here is the arrowing architecture. This architecture starts from pixel memory and ends up with an arrow memory with labels to indicate the steepest descending paths.

(4) plat/inner—to determine if the current pixel is part of a plateau and whether it is an edge or inner plateau pixel,

(5) arrowing—to determine the direction of the steepest descent. This direction is to be written to the "Arrow Memory",

(6) pixel status—to determine the status of the pixels, that is, whether they have been read before, put into queue before, or have been labelled.

The next subsections will begin to describe the parts listed above in the same order.



Notes:
1. In stage 1 of the processing, mc6–9 is used as a secondary counter for Q1_WNES and incremented as mc1–4 increments but does not decrement when mc1-4 is decremented. In stage 2, if mc5 = 0 (i.e., complete lower minima), mc6–9 is used as the counter to track the number of elements in Q1_WNES. In this state, mc6-9 is decremented when Q1_WNES is read from. However, if mc5 > 0, mc6–9 is reset and resumes the role of memory counter for Q2_WNES.
2. Q1_C is only ever used once and that is during stage 2 of the processing.

Figure 11: Stages of Plateau Processing and their various conditions.

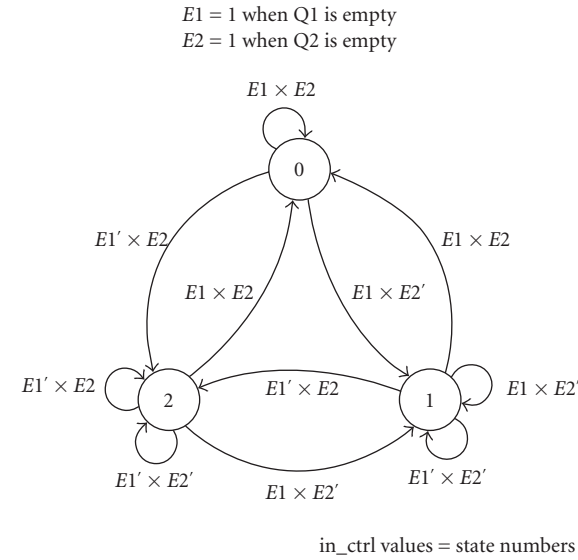


Figure 12: State diagram of the architecture-ARROWING.

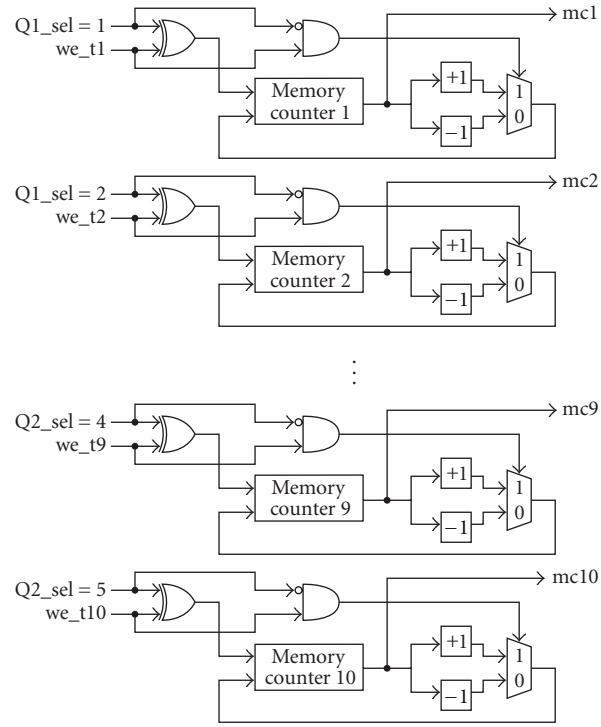


Figure 13: Memory counter for Queue C, W, N, E, and S. The memory counter is used to determine the number of elements in the various queues for the directions of Centre, West, North, East, and South.

4.1. Memory Counter. The architecture is a tristate system whose state is determined by the condition of whether the queues, Q1 and Q2, are empty or otherwise. This is shown in Figure 12. These states in turn determine the control of the main multiplexer, in_ctrl, which is the control of the data input into the system.

To determine the initial queue states, Memory Counters (MCs) are used to keep track of how many elements are pending processing in each of the West, North, East, South, and Centre queues. There are five MCs for Q1 and another five for Q2, one counter for each of the queue directions. These MCs are named mc1–5 for Q1_W, Q1_N, Q1_E, Q1_S,

Table 1: Comparison of the number of clock cycles required for reading all five required values and the memory requirements for the three different methods.

	Sequential	Parallel	Graph-based
Clock cycles	5	1	1
Memory Req.	1x image size	5x image size	1x image size

and Q1_C, respectively, and similarly mc6–10 for Q2_W, Q2_N, Q2_E, Q2_S, and Q2_C respectively. This is shown in Figure 13.

The MCs increase by one count each time an element is written to the queue. Similarly, the MCs decrease by one count every time an element is read from the queue. This increment is determined by tracking the write enable we_tx where $x = 1 - 10$ while the decrement is determined by tracking the values of Q1_sel and Q2_sel.

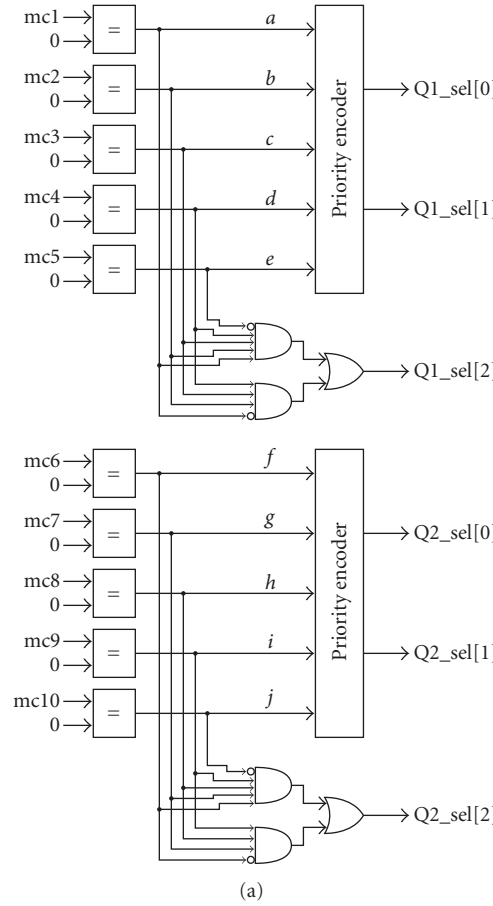
A special case occurs during the stage one of plateau processing, whereby mc6–9 is used to count the number of elements in Q1_W, Q1_N, Q1_E, and Q1_S, respectively. In this stage, mc6–9 is incremented when the queues are written to but are only decremented when Q1_WNES is read again in the stage two for complete lower minima labelling.

The MC primarily consists of a register and a multiplexer which selects between a $(+1)$ increment or a (-1) decrement of the current register value. Selecting between these two values and writing these new values to the register effectively count up and down. The update of the MC register value is controlled by a write enable, which is an output of a 2-input XOR. This XOR gate ensures that the MC register is updated when only one of its inputs is active.

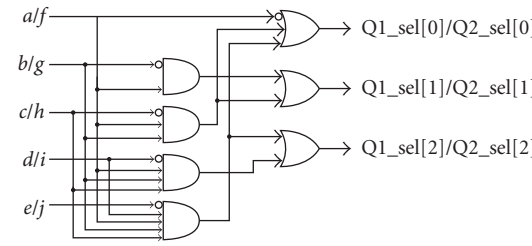
4.2. The Priority Encoder. The priority encoder is used to determine the output of Q1_sel and Q2_sel by comparing the outputs of the MC to zero. It selects the output from the queues in the order it is stored, that is, from queue Qx_W to Qx_C, $x = 1$ or 2. Together with the state of in_ctrl, Q1_sel and Q2_sel will determine the data input into the system. The logic to determine the control bits for Q1_sel and Q2_sel is shown in Figure 14.

4.3. Centre and Neighbour Coordinate. The centre and neighbourhood block is used to determine the coordinates of the pixel's neighbours and to pass through the centre coordinate. These coordinates are used to address the various queues and multibank memory. It performs an addition and subtraction by one unit on both the row and column coordinates. This is rearranged and grouped into their respective outputs. The outputs from the block are five pixel locations, corresponding to the centre pixel location and the four neighbours, West (W), North (N), East (E), and South (S). This is shown in Figure 15.

4.4. The Smallest-Valued Neighbour Block. This block is to determine the smallest-valued neighbour (SVN) and its position in relation to the current pixel. This is used to determine if the current pixel has a lower minima and to find the steepest descending path to that minima (arrowing).



(a)



$Q1\_sel[0] = a' + abc' + abcde'$ $\quad Q2\_sel[0] = f' + fgh' + fghij'$

$Q1\_sel[1] = ab' + abc'$ $\quad Q2\_sel[1] = fg' + fgh'$

$Q1\_sel[2] = abcd' + a\text{bcde}'$ $\quad Q2\_sel[2] = fghi' + fghij'$

a/f	b/g	c/h	d/i	e/j	[2]	[1]	[0]	Qx_sel
1	1	1	1	1	0	0	0	Disable
0	x	x	x	x	0	0	1	1
1	0	x	x	x	0	1	0	2
1	1	0	x	x	0	1	1	3
1	1	1	0	x	1	0	0	4
1	1	1	1	0	1	0	1	5

(b)

Figure 14: The priority encoder. (a) shows the controls for Q1_sel and Q2_sel using the priority encoders. The output of memory counters determines the multiplexer control of Q1_sel and Q2_sel. (b) shows the logic of the priority encoders used. There is a special "disable" condition for the multiplexers of Q1 and Q2. This is used so that the Q1_sel and Q2_sel can have an initial condition and will not interfere with the memory counters.

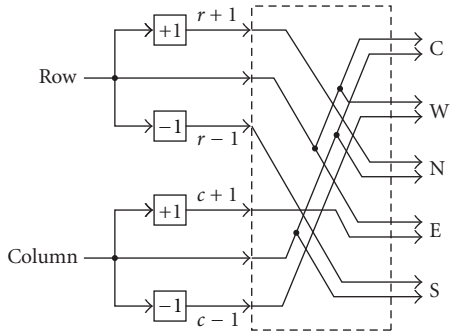


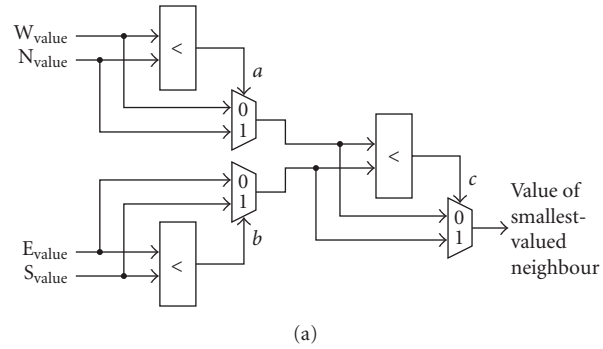
Figure 15: Inside the Pixel Neighbour Coordinate.

To determine the smallest value pixel, the values of the neighbours are compared two at a time, and the result of the comparator is used to select the smaller value of the two. The last two values are compared once again and the value of the smallest value neighbour will be obtained. As for the direction of the SVN, the outputs from the 3 stages of comparison are used and compared to a truth table. This is shown in Figure 16. This output is passed to the arrowing block to determine the direction of the steepest descent (when there is a lower neighbour).

4.5. The Plateau-Inner Block. This block is to determine whether the current pixel is part of a plateau and which type of plateau pixel it is. The current pixel type will determine what is done to the pixel and its neighbours, that is, whether they are put back into a queue or otherwise. Essentially, together with the Pixel Status, it helps to determine if a pixel or one of its neighbours should be put back into the queues for further processing. When the system is in State 0 (i.e., processing pixel locations from the PC), the block determines if the current pixel is part of a plateau. The value of the current pixel is compared to all its neighbours. If any one of the neighbours has a similar value to the current pixel, it is part of a plateau and plat = 1. The respective similar valued neighbours are put into the different queue locations based on sv_W, sv_N, sv_E, and sv_S and the value of pixel status. The logic for this is shown in Figure 17(a).

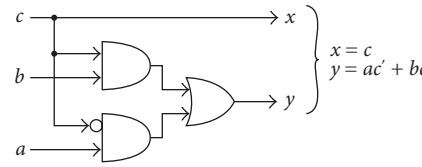
In any other state, this block is used to determine if the current pixel is an inner (i.e., equal to or smaller than its neighbours). If the current pixel is an inner, inner = 1. This is shown in Figure 17(b). Whether the pixel is an inner or not will determine the arrowing part of the system. If it is an inner, it will point to the nearest edge.

4.6. The Arrowing Block. This block is to determine the steepest descending path label for the "Arrow Memory." The steepest path is calculated based on whether the pixel is an inner or otherwise. When processing non-inner pixels the arrowing block generates a direction output based on the location of the lowest neighbour obtained from the block "Smallest Valued Neighbour." If the pixel is an inner, the arrow will simply point to the nearest edge. When there is more than one possible path to the nearest edge, a priority



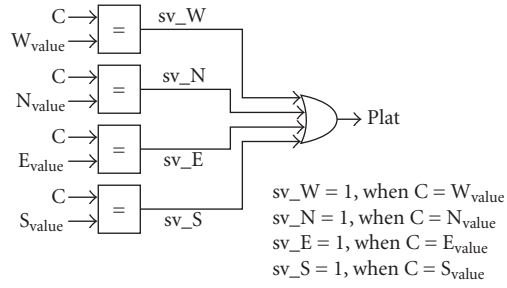
(a)

a	b	c	x	y	Direction
0	x	0	0	0	W
1	x	0	0	1	N
x	0	1	1	0	E
x	1	1	1	1	S

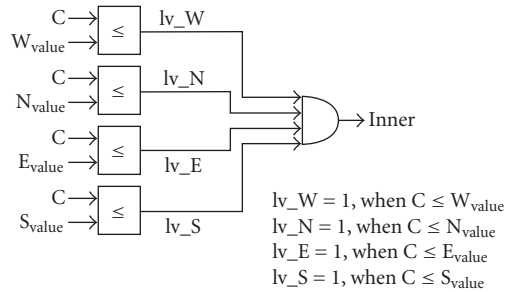


(b)

Figure 16: Inside the Smallest Value Neighbour (SVN) block. (a) The smallest-valued neighbour is determined and selected using a set of comparators and multiplexers. (b) The location of the smallest-valued neighbour is determined by the selections of each multiplexer. This location information used to determine the steepest descending path and is fed into the arrowing block.



(a)



(b)

Figure 17: Inside the Plateau-Inner Block.

encoder in the block is used to select the predefined direction of the highest priority. This is shown in Figure 18 when the system is in State = 0, and in any other state where the pixel is not an inner, this arrowing block uses the information from the SVN block and passes it through directly to its own main multiplexer, selecting the appropriate value to be written into "Arrow Memory."

If the current pixel is found to be an inner, the arrowing direction is towards the highest priority neighbour with the same value which has been previously labelled. This is possible because we are labelling the plateau pixels from the edge pixels going in, one pixel at a time, ensuring that the inners will always point in the direction of the shortest geodesic distance.

4.7. Pixel Status. One of the most important parts of this system is the pixel status (PS) registers. Since six states are used to flag the pixel, this register requires a 3-bit representation for each pixel location of the image. Thus the PS registers have as many registers as there are pixels in the input image. In the system, values from the PS help determine what processes a particular pixel location has gone through and whether it has been successfully labelled into the "Arrow Memory." The six states and their transitions are shown in Figure 19. The six states are as follows:

(i) 0 : unvisited—nothing has been done to the pixel,

(ii) 1 : queued : initial,

(iii) 2 : queued in Q2,

(iv) 3 : queued in Q1,

(v) 4 : completed when plat = 0,

(vi) 5 : completed when plat = 1 and reading from Q2,

(vii) 6 : completed when plat = 1 and reading from Q1.

To ease understanding of how the plateau conditions are handled and how the PS is used, we shall introduce the concept of the "Unlabelled pixel (UP)" and "Labelled pixel (LP)." The UP is defined as the "outermost pixel which has yet to be labelled." Using this definition, the arrowing procedure for the plateau pixels are

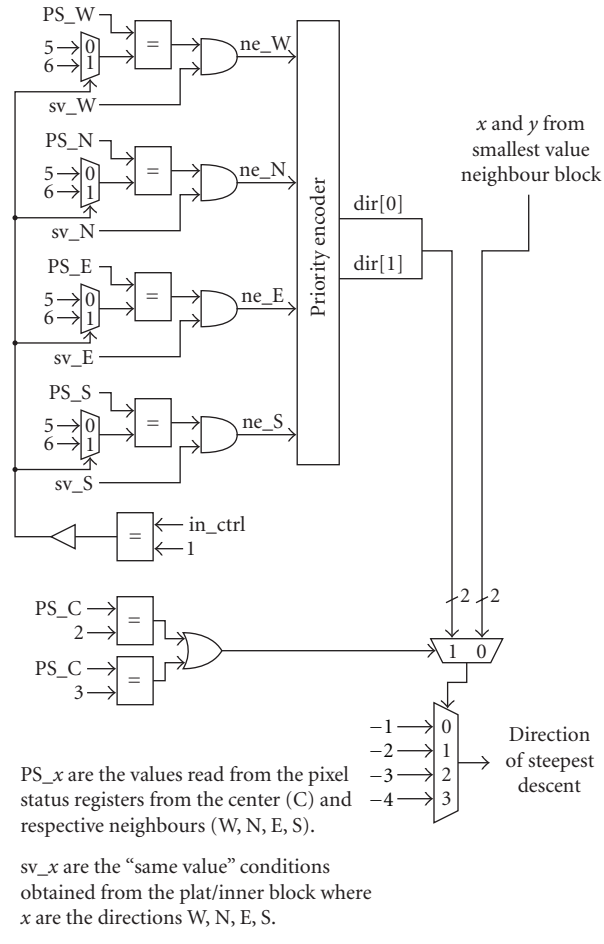
(1) arrow to lower-valued neighbour (applicable only if inner = 0)

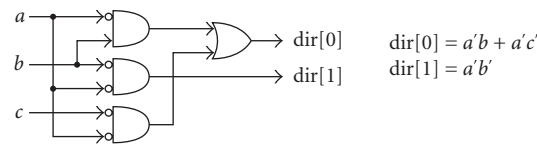
(2) arrow to neighbour with PS = 5 according to predefined arrowing priority.

With reference to Figure 20, the PS is used to determine which neighbours to the UPs have not been put into the other queue, UPs of the same label and LPs.

5. Example for the Arrowing Architecture

This example will illustrate the states and various controls of the watershed architecture for an 8×8 sample data. It is the same sample data shown in Figures 6 and 7. A table with the various controls, status, and queues for the first 14 clock cycles is shown in Table 2.





a	b	c	d	x	y	mux_ctrl
1	x	x	x	0	0	0
0	1	x	x	0	1	1
0	0	1	x	1	0	2
0	0	0	1	1	1	3

Figure 18: Inside arrowing block.

The initial condition for the system is as follows. The Program Counter (PC) starts with the first pixel and generates a $(0, 0)$ output representing the first pixel in an (x, y) format.

With both the Q1 and Q2 queues being empty, that is, mc1 → mc10 = 0, the system is in State 0. This sets in_ctrl = 0 that controls *mux*1 to select the PC value (in this case $(0, 0)$. This value is incremented on the next clock cycle.

The First Few Steps. This PC coordinate is then fed into the Pixel Neighbour Coordinate block. The outputs of this block

																			(1,0)[2]
3	2	Q1 = 5	(0,0)	1	0	S	6	inv	inv	0	0	—	0	—	0[1]	—	—	—	—	(0,0)[0] (1,0)[1]
4	2	Q1 = 5	(1,0)	1	0	N	6	inv	6	0	0	—	0	—	0[1]	—	—	—	—	(0,0)[0] (1,0)[0]
5	0	0	(0,1)	0	0	—	0 → 6	6	inv	0	0	—	1	−3	0[1]	—	—	—	—	—
6	0	0	(0,2)	0	0	—	0 → 6	6	inv	0	0	—	1	1	1[1 → 2]	—	—	—	—	—
7	0	0	(0,3)	0	0	—	0 → 6	6	inv	0	0	—	1	−4	0[2]	—	—	—	—	—
8	0	0	(0,4)	1	1	E,S	0	6	inv	0 → 1	0 → 1	t3, t4	0	0	0[2]	—	—	(0,5)[1][1]	(1,4)[1][1]	—
9	2	Q1 = 3	(0,5)	1	1	W,S	0	0 → 1	inv	0	0 → 1	t1, t4	0	0	0[2]	(0,4)[1][1]	—	(0,5)[0][1]	(1,4)[1][1]	—
																			(1,5)[2][2]	—
10	2	Q1 = 1	(0,4)	1	1	E,S	1	1	inv	1	1	—	0	0	0[2]	(0,4)[0][1]	—	(0,5)[0][1]	(1,4)[1][1]	—
																			(1,5)[2][2]	—
11	2	Q1 = 4	(1,4)	1	0	N,E,S	0 → 6	1	inv	1	0 → 1	t4, t5	1	−1	0[2]	(0,4)[0][1]	—	(0,5)[0][1]	(1,4)[0][1]	(1,4)[1]
																			(1,5)[1][2]	—
																			(2,4)[2][3]	—
12	2	Q1 = 4	(1,5)	1	1	W,N,S	1	6	1	0	0 → 1	t4, t5	1	−1	0[2]	(0,4)[0][1]	—	(0,5)[0][1]	(1,4)[0][1]	(1,4)[1]
																			(1,5)[0][2]	—
																			(2,4)[1][3]	—
																			(2,5)[2][4]	—
13	2	Q1 = 4	(2,4)	1	0	N,E,S	0 → 6	0	6	1	0 → 1	t4, t5	1	−1	0[2]	(0,4)[0][1]	—	(0,5)[0][1]	(1,4)[0][1]	(1,4)[1]
																			(1,5)[0][2]	(2,4)[2]
																			(2,4)[0][3]	—
																			(2,5)[1][4]	—
																			(3,4)[2][5]	—
14	2	Q1 = 4	(2,5)	1	1	W,N,S	1	6	1	0	0 → 1	t4	0	—	0[2]	(0,4)[0][1]	—	(0,5)[0][1]	(1,4)[0][1]	(1,4)[1]
																			(1,5)[0][2]	(2,4)[2]
																			(2,4)[0][3]	—
																			(2,5)[0][4]	—
																			(3,4)[1][5]	—
																			(3,5)[2][6]	—

The pixel status is a 3-bit register. It is used to tag the status of pixels. The various tags are as follows:

0: Never visited
1: Queued-initial (all plat pixel locations into Q1)
2: Queued in Q2
3: Queued in Q1
4: Completed when plat = 0
5: Completed when plat = 1 and reading from Q2
6: Completed when plat = 1 and reading from Q1

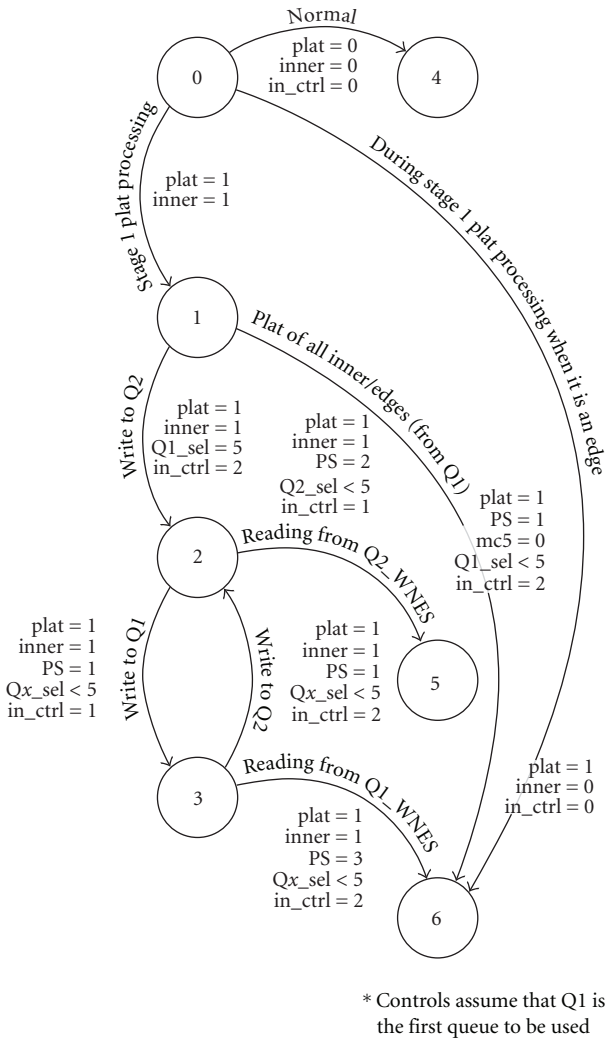


FIGURE 19: The pixel status block is a set of 3-bit registers used to store the state of the various pixels.

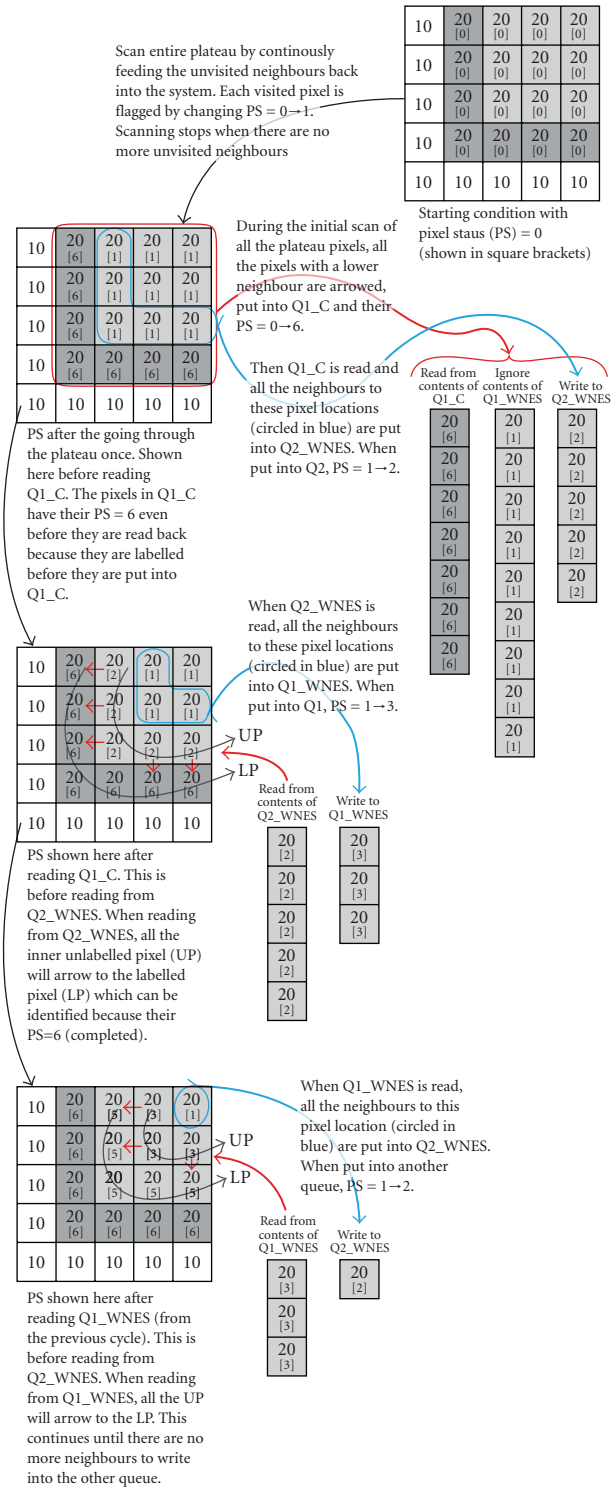


FIGURE 20: An example of how Pixel Status is used in the system.

(the pixel locations) are $(0,0)$, $(0,1) \rightarrow E$, $(1,0) \rightarrow W$, $(-1,0) \rightarrow INVALID$, and $(0,-1) \rightarrow INVALID$. The valid addresses are then used to obtain the current pixel value, 10(C), and neighbour values, 9(W) and 10(S). The invalid pixel locations are set to output an INVALID value through the CB mux. This value has been predefined to be 255.

The pixel locations are also used to determine address locations within the 3-bit pixel status registers. When read, the values are $(0,0) = 0$, $(0,1) = 0$, and $(1,0) = 0$. The neighbours with similar value are put into the queue. In the example used, only the sound neighbour has a similar value and is put into queue Q1_S. Next, the pixel status for $(1,0)$ is changed from $0 \rightarrow 1$. This tells the system that the coordinate $(1,0)$ has been put into the queue and will avoid an infinite loop once its similar valued neighbour to

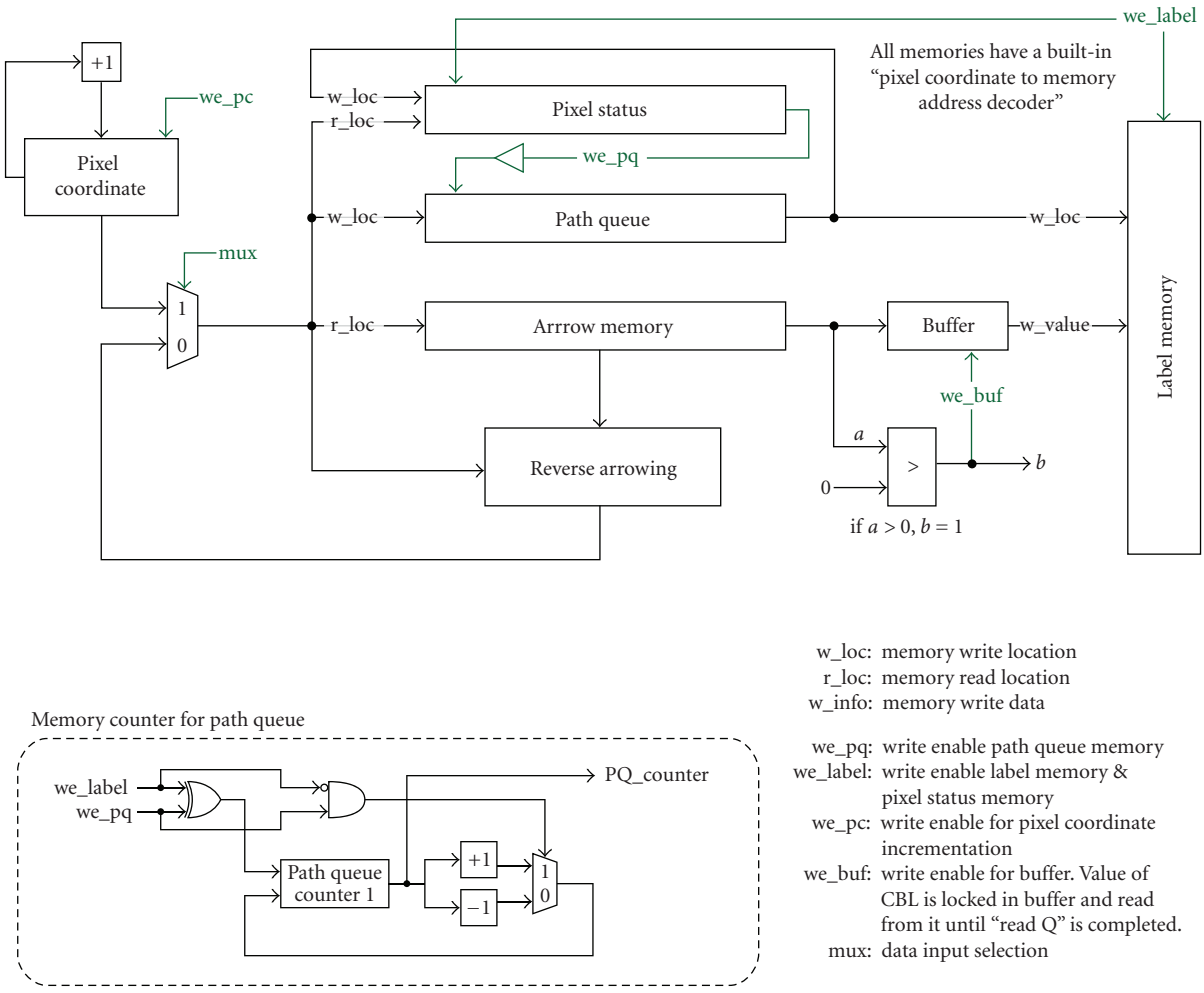


Figure 21: The watershed architecture: Labelling.

the north $(0,0)$ finds $(1,0)$ again. The current pixel location $(0,0)$ on the other hand is written to Q1_C because it is a plateau pixel but not an inner (i.e., an edge) and is immediately arrowed. The status for this location $(0,0)$ is changed from $0 \rightarrow 6$. Q1_S will contain the pixel location $(1,0)$. This is read back into the system and $\text{mc4} = 1 \rightarrow 0$ indicating Q1_S to be empty. The pixel location $(1,0)$ is arrowed and written into Q1_C. With $\text{mc1} - 4 = 0$ and $\text{mc5} > 0$, the pixel locations $(0,0)$ and $(1,0)$ is reread into the system but nothing is performed because both their PSsequal 6 (i.e., completed).

6. Labelling Architecture

This second part of the architecture will describe how we get from Figure 4(c) to Figure 4(d) in hardware. Compared to the arrowing architecture, the labelling architecture is considerably simpler as there are no parallel memory reads. In fact, everything runs in a fairly sequential manner. Part 2 of the architecture is shown in Figure 21.

The architecture for Part 2 is very similar to Part 1. Both are tristate systems whose state depends on the condition

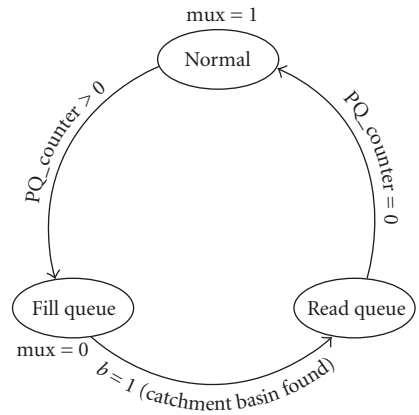


Figure 22: The 3 states in Architecture:Labelling.

of the queues and uses pixel state memory and queues for storing pixel locations. The difference is that Part 2 architecture only requires a single queue and a single bit pixel status register. The three states for the system are shown in Figure 22.

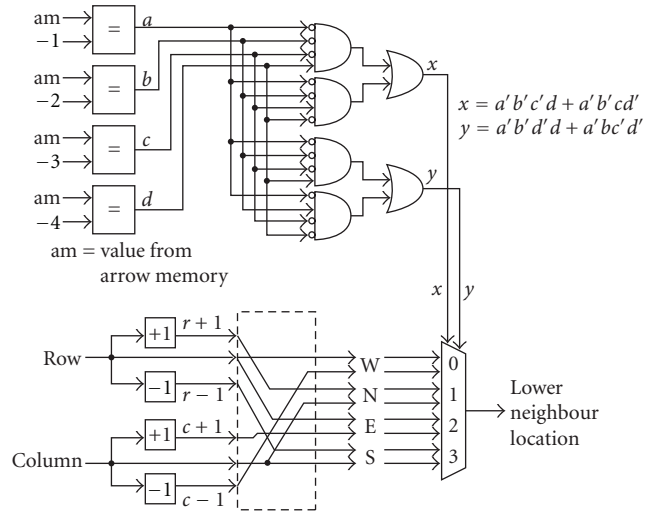
Values are initially read in from the pixel coordinate register. Whether this pixel location had been processed before is checked against the pixel status (PS) register. If it has not been processed before (i.e., was never part of any steepest descending path), it will be written to the Path Queue (PQ). Once PQ is not empty, the system will process the next pixel along the current steepest descending path. This is calculated by the "Reverse Arrowing Block" (RAB) using the current pixel location and direction information obtained from the "Arrow Memory." This process continues until a non-negative value is read from "Arrow Memory." This non-negative value is called the "Catchment Basin Label" (CBL). Reading a CBL tells that the system a minimum has been reached and all the pixel locations stored in PQ will be labelled with that CBL and written to "Label Memory." At the same time, the pixel status for the corresponding pixel locations will be updated accordingly from $0 \rightarrow 1$. Now that PQ is empty; the next value will be obtained from the pixel coordinate register.

6.1. The Reverse Arrowing Block. This block calculates the neighbour pixel location in the path of the steepest descent given the current location and arrowing label. In other words, it simply finds the location of the pixel pointed to by the current pixel.

The output of this block is a simple case of selecting the appropriate neighbouring coordinate. Firstly the neighbouring coordinates are calculated and are fed into a 4-input multiplexer. Invalid neighbours are automatically ignored as they will never be selected. The values in "Arrow Memory" only point to valid pixels. Hence, no special consideration is required to handle these cases.

The bulk of the block's complexity lies in the control of the multiplexer. The control is determined by translating the value from the "Arrow Memory" into proper control logic. Using a bank of four comparators, the value from "Arrow Memory" is determined by comparing it to four possible valid direction labels (i.e., $-4 \rightarrow -1$). For each of these values, only one of the comparators will produce a positive outcome (see truth table in Figure 23). Any other values outside the valid range will simply be ignored.

The comparator output is then passed through some logic that will produce a 2-bit output corresponding to the multiplexer control. If the value from "Arrow Memory" is -1, the control logic will be $(x = 0, y = 0)$ corresponding to the West neighbour location. Similarly, if the value from "Arrow Memory" is -2, -3, or -4, the control logic will be $(x = 0, y = 1)$, $(x = 1, y = 0)$, or $(x = 1, y = 1)$ corresponding to the North, East, or South neighbour locations, respectively. This is shown in Figure 23.



a	b	c	d	x	y	mux_ctrl
1	0	0	0	0	0	0
0	1	0	0	0	1	1
0	0	1	0	1	0	2
0	0	0	1	1	1	3

Figure 23: Inside the reverse arrowing block.

7. Example for the Labelling Architecture

This example will pick up where the previous example had stopped. In the previous part, the resulting output was written to the "Arrow Memory." It contains the directions of the steepest descent (negative values from $-1 \rightarrow -4$) and numbered minima (positive values from $0 \rightarrow$ total number of minima) as seen in Figure 4(c). In this part, we will use the information stored in "Arrow Memory" to label each pixel with the label of its respective minimum. Once all associated pixels to a minimum/minima have been labelled accordingly, a catchment basin is formed.

The system starts off in the normal state and the initial conditions are as follows. PQ_counter = 0, mux = 1. In the first clock cycle, the first pixel location $(0, 0)$ is read from the pixel location register. Once this has been read in, the pixel location register will increment to the next pixel location $(0, 1)$. The PS for the first location $(0, 0)$ is 0. This enables the write enable for the PQ and the first location is written to queue. At the same time, the location $(0, 0)$ and direction -3 obtained from "Arrow Memory" are used to find the next coordinate $(0, 1)$ in the steepest descending path.

Since PQ is not empty, the system enters the "Fill Queue" state and mux = 0. The next input into the system is the value from the reverse arrowing block, $(0, 1)$, and since PS = 0, it is put into PQ. The next location processed is $(0, 2)$. For $(0, 2)$, PS = 0 and is also written to PQ. However, for this location, the value obtained from "Arrow Memory" is 1. This is a CBL and is buffered for the process of the next state. Once a non-negative value from "Arrow Memory" is read (i.e., b = 1), the system enters the next state which is the "Read Queue" state. In this state, all the pixel locations stored in PQ is read one at a time and the memory locations in "Label Memory" corresponding to these locations are written with the buffered CBL. At the same time, PS is also updated from $0 \rightarrow 1$ to reflect the changes made to "Label Memory." It tells the system that the locations from PQ have been processed so that it will not be rewritten when it is encountered again.

Table 3: Results of the implemented architecture on a Xilinx Spartan-3 FPGA.

64×64 image size,	
Arrowing	
Slice flip flops	423 out of 26,624 (1%)
Occupied slices	2,658 out of 13,312 (19%)
Labelling	
Slice flip flops	39 out of 26,624 (1%)
Occupied slices	37 out of 13,312 (1%)

With each read from PQ, PQ_counter is decremented. When PQ is empty, PQ_counter = 0 and the system will return to the normal state.

In the next clock cycle, $(0, 1)$ is read from the pixel coordinate register. For $(0, 1)$, PS = 1 and nothing gets written to PQ and PQ_counter remains at 0. The same goes for $(0, 2)$. When the coordinate $(0, 3)$ is read from the pixel coordinate register, the whole processes of filling up PQ and reading from PQ and writing to "Label Memory" start again.

8. Synthesis and Implementation

The rainfall watershed architecture was designed in Handel-C and implemented on a Celoxica RC10 board containing a Xilinx Spartan-3 FGPA. Place and route were completed to obtain a bitstream which was downloaded into the FPGA for testing. The watershed transform was computed by the FPGA architecture, and the arrowing and labelling results were verified to have the same values as software simulations in Matlab. The Spartan-3 FPGA contains a total of 13312 slices. The implementation results of the architecture are given in Table 3 for an image size of 64×64 pixels. An image resolution of 64×64 required 2658 and 37 slices for the arrowing and labelling architecture, respectively. This represents about 20% of the chip area on the Spartan-3 FPGA.

9. Summary

This paper proposed a fast method of implementing the watershed transform based on rainfall simulation with a multiple bank memory addressing scheme to allow parallel access to the centre and neighbourhood pixel values. In a single read cycle, the architecture is able to obtain all five values of the centre and four neighbours for a 4-connectivity watershed transform. This multiple bank memory has the same footprint as a single bank design. The datapath and control architecture for the arrowing and labelling hardware have been described in detail, and an implemented architecture on a Xilinx Spartan-3 FGPA has been reported. The work can be extended to implement an 8-connectivity watershed transform by increasing the number of memory banks and working out its addressing. The multiple bank memory approach can also be applied to other watershed architectures such as those proposed in [10–13, 15].

References

[1] S. E. Hernandez and K. E. Barner, "Tactile imaging using watershed-based image segmentation," in *Proceedings of the Annual Conference on Assistive Technologies (ASSETS '00)*, pp. 26–33, ACM, New York, NY, USA, 2000.

[2] M. Fussenegger, A. Opelt, A. Pjnz, and P. Auer, "Object recognition using segmentation for feature detection," in *Proceedings of the 17th International Conference on Pattern Recognition (ICPR '04)*, vol. 3, pp. 41–44, IEEE Computer Society, Washington, DC, USA, 2004.

[3] W. Zhang, H. Deng, T. G. Dietterich, and E. N. Mortensen, "A hierarchical object recognition system based on multi-scale principal curvature regions," in *Proceedings of the 18th International Conference on Pattern Recognition (ICPR '06)*, vol. 1, pp. 778–782, IEEE Computer Society, Washington, DC, USA, 2006.

[4] M. S. Schmalz, "Recent advances in object-based image compression," in *Proceedings of the Data Compression Conference (DCC '05)*, p. 478, March 2005.

[5] S. Han and N. Vasconcelos, "Object-based regions of interest for image compression," in *Proceedings of the Data Compression Conference (DCC '05)*, pp. 132–141, 2008.

[6] T. Acharya and P.-S. Tsai, *JPEG2000 Standard for Image Compression: Concepts, Algorithms and VLSl Architecturcs*, John Wiley & Sons, New York, NY, USA, 2005.

[7] V. Osma-Ruiz, J. I. Godino-Llorente, N. Saáenz-Lechón, and P. Gómez-Vilda, "An improved watershed algorithm based on efficient computation of shortest paths," *Pattern Recognition*, vol. 40, no. 3, pp. 1078–1090, 2007.

[8] A. Bieniek and A. Moga, "An efficient watershed algorithm based on connected components," *Pattern Recognition*, vol. 33, no. 6, pp. 907–916, 2000.

[9] H. Sun, J. Yang, and M. Ren, "A fast watershed algorithm based on chain code and its application in image segmentation," *Pattern Recognition Letters*, vol. 26, no. 9, pp. 1266–1274, 2005.

[10] M. Neuenhahn, H. Blume, and T. G. Noll, "Pareto optimal design of an FPGA-based real-time watershed image segmentation," in *Proceedings of the Conference on Program for Research on Integrated Systems and Circuits (ProRISC '04)*, 2004.

[11] C. Rambabu and I. Chakrabarti, "An efficient immersion-based watershed transform method and its prototype architecture," *Journal of Systems Architecture*, vol. 53, no. 4, pp. 210–226, 2007.

[12] C. Rambabu, I. Chakrabarti, and A. Mahanta, "Flooding-based watershed algorithm and its prototype hardware architecture," *IEE Proceedings: Vision, Image and Signal Processing*, vol. 151, no. 3, pp. 224–234, 2004.

[13] C. Rambabu and I. Chakrabarti, "An efficient hillclimbing-based watershed algorithm and its prototype hardware architecture," *Journal of Signal Processing Systems*, vol. 52, no. 3, pp. 281–295, 2008.

[14] D. Noguet and M. Ollivier, "New hardware memory management architecture for fast neighborhood access based on graph analysis," *Journal of Electronic Imaging*, vol. 11, no. 1, pp. 96–103, 2002.

[15] C. J. Kuo, S. F. Odeh, and M. C. Huang, "Image segmentation with improved watershed algorithm and its FPGA implementation," in *Proceedingsof the IEEE International Symposium on Circuits and Systems (ISCAS '01)*, vol. 2, pp. 753–756, Sydney, Australia, May 2001.