Name- Ankit

Roll no -2001029

VLSI DESIGN LAB PROJECT

IMAGE PROCESSING/IMAGE SMOOTHNING

AIM- To Implementing a Averaging filter for Image Processing

THEORY-

Image processing involves the manipulation of digital images through various algorithms and techniques to enhance, analyze or transform images. One such technique is filtering, which is used to remove noise, blur images, or highlight certain features in an image.

An averaging filter is a type of linear filter that works by replacing each pixel in an image with the average value of its neighboring pixels. This helps to smooth out the image and reduce noise.

To implement an averaging filter in Verilog, you will need to follow these steps:

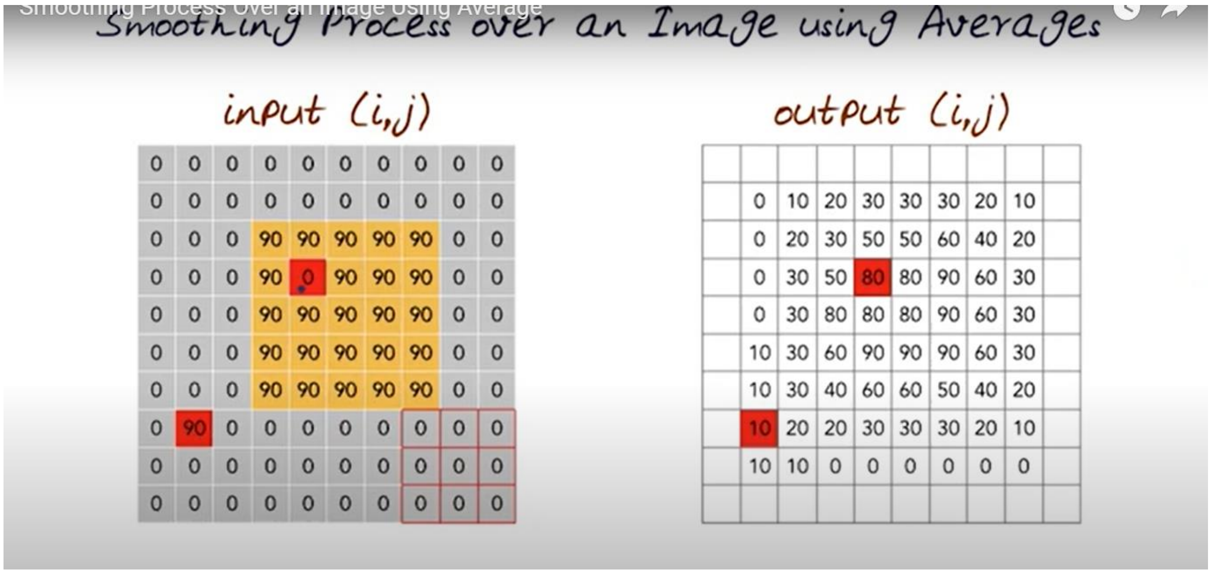
1. Define the input and output image dimensions and the size of the filter kernel. Typically, the filter kernel size is an odd number, such as 3x3 or 5x5.

2. Define the Verilog module and its input and output ports. The input port should be the original image, while the output port should be the filtered image.

3. Define the Verilog code for the averaging filter. This involves creating a loop that iterates over each pixel in the image and applies the averaging filter to that pixel. The

averaging filter involves computing the average value of the neighboring pixels for each pixel in the image.

4. Once the filtered image has been computed, it can be outputted to the output port.



Smoothing Process over an Image using Averages

input (i,j)

0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0
0	0	0	90	90	90	90	90	0	0
0	0	0	90	0	90	90	90	0	0
0	0	0	90	90	90	90	90	0	0
0	0	0	90	90	90	90	90	0	0
0	0	0	90	90	90	90	90	0	0
0	90	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0

output (i,j)

	0	10	20	30	30	30	20	10	
	0	20	30	50	50	60	40	20	
	0	30	50	80	80	90	60	30	
	0	30	80	80	80	90	60	30	
	10	30	60	90	90	90	60	30	
	10	30	40	60	60	50	40	20	
	10	20	20	30	30	30	20	10	
	10	10	0	0	0	0	0	0	

ALGORITHM-

FOR MODULE-

Algorithm for Averaging Filter Module:

1. Declare the module with input and output ports and define the filter size and weight as module parameters.
2. Declare a filter buffer as a two-dimensional array to store the pixel values of the input image for filtering.
3. Declare row and column variables to index the filter buffer.
4. Declare a sum variable to accumulate the sum of the pixel values in the filter buffer.
5. Declare a count variable to keep track of the number of pixels currently in the filter buffer.
6. Declare an out_ready flag to indicate when an output pixel is ready.
7. On the positive edge of the clock signal, check if the reset signal is active.
8. If the reset signal is active, clear the filter buffer and counters, and set the output ready flag to 0.
9. If the reset signal is not active, shift the filter buffer to the left by one pixel.
10. Add the new input pixel to the rightmost column of the filter buffer.
11. Increment the count variable to keep track of the number of pixels in the filter buffer.
12. Calculate the sum of the pixel values in the filter buffer.
13. Calculate the output pixel value by dividing the sum by the number of pixels in the filter buffer and the weight.

14. If the number of pixels in the filter buffer is equal to the filter size squared, set the output ready flag to 1 and output the pixel value. Otherwise, keep the output ready flag at 0.

<u>FOR TESTBENCH-</u>

1. Initialize the test bench module by setting up the required signals and modules. This may include the clock signal, reset signal, input and output signals, and the averaging filter module itself.

2. Load an input image into the test bench module. The input image should be representative of the type of images that the averaging filter module will be processing.

3. Apply the input pixel values from the input image to the input signal of the averaging filter module. The input signal should be set up to receive a single pixel value at a time.

4. Wait for a specified number of clock cycles to allow the averaging filter module to process the image. The number of clock cycles required may vary depending on the implementation of the module and the size of the input image.

5. Read the output pixel values from the output signal of the averaging filter module. The output signal should be set up to provide a single pixel value at a time.

6. Compare the output pixel values to the expected output values for the corresponding input pixels. The expected output values can be computed manually by applying the averaging filter algorithm to the input image.

7. If the output values match the expected values for all pixels, the simulation is successful. If not, output an error message and terminate the simulation.

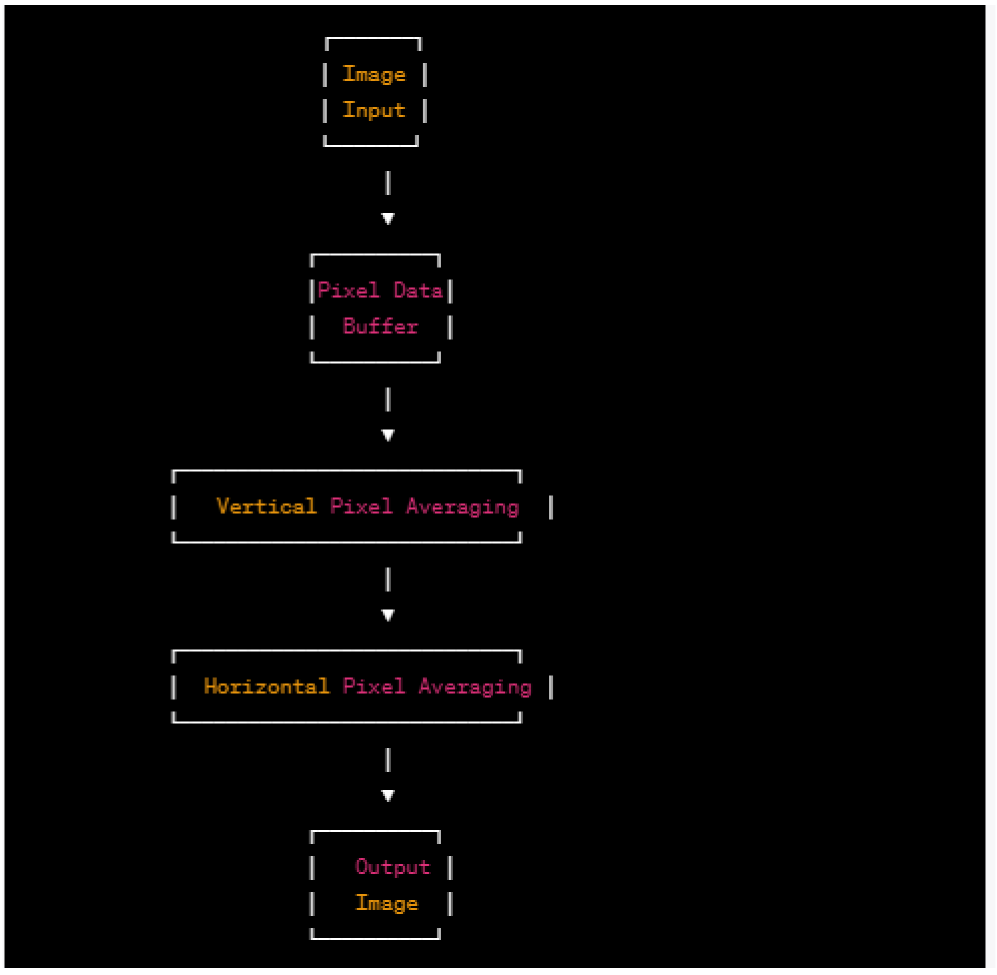
8. Repeat steps 3-7 for all pixels in the input image.

9. Once all pixels have been processed, the simulation is complete. Output a message indicating whether the simulation was successful or not.

10. The above algorithm provides a more detailed outline for the test bench we made for the averaging filter module. The exact implementation may vary depending on the

specific hardware platform and programming language used, but the above algorithm provides a general framework for testing the module's functionality

BLOCK DIAGRAM-



CODE-

FOR MODULE-

```
` module averaging_filter (
  input clk, rst,
  input [7:0] in_pixel,
  output reg [7:0] out_pixel
);
```

```
// Set the filter size and weights
parameter FILTER_SIZE = 3;
parameter WEIGHT = 1/FILTER_SIZE/FILTER_SIZE;

// Declare the filter buffer
reg [7:0] filter_buffer [0:FILTER_SIZE-1][0:FILTER_SIZE-1];
reg [FILTER_SIZE-1:0] row;
reg [FILTER_SIZE-1:0] col;

// Declare the sum accumulator
reg [7:0] sum;

// Declare the output ready flag
reg out_ready;

// Declare the counter for the number of pixels in the filter
reg [FILTER_SIZE+1:0] count;

always @(posedge clk) begin
  if (rst) begin
    // Reset the filter buffer and counters
    for (row = 0; row < FILTER_SIZE; row = row + 1) begin
      for (col = 0; col < FILTER_SIZE; col = col + 1) begin
        filter_buffer[row][col] <= 0;
      end
    end
    count <= 0;
    out_ready <= 0;
  end else begin
    // Shift the filter buffer
    for (row = 0; row < FILTER_SIZE; row = row + 1) begin
      for (col = 0; col < FILTER_SIZE-1; col = col + 1) begin
        filter_buffer[row][col] <= filter_buffer[row][col+1];
      end
      filter_buffer[row][FILTER_SIZE-1] <= filter_buffer[row][FILTER_SIZE-2];
    end
    // Add the new pixel to the filter buffer
    filter_buffer[0][FILTER_SIZE-1] <= in_pixel;
    // Update the counters
    count <= count + 1;
    // Calculate the sum of the filter values
    sum <= 0;
    for (row = 0; row < FILTER_SIZE; row = row + 1) begin
      for (col = 0; col < FILTER_SIZE; col = col + 1) begin
        sum <= sum + filter_buffer[row][col];
      end
    end
    // Calculate the output pixel
    if (count >= FILTER_SIZE*FILTER_SIZE) begin
      out_pixel <= sum * WEIGHT;
      out_ready <= 1;
    end
  end
end

endmodule
```

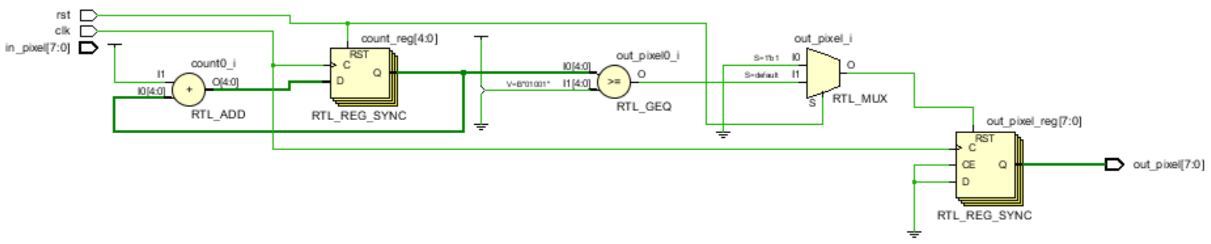
<u>FOR TESTBENCH-</u>

```
module averaging_filter_tb;

// Declare the clock and reset signals
reg clk;
reg rst;

// Declare the input and output signals
reg [7:0] in_pixel;
wire [7:0] out_pixel;
wire out_ready;

// Instantiate the averaging filter
averaging_filter dut (
  .clk(clk), .rst(rst),
  .in_pixel(in_pixel),
  .out_pixel(out_pixel),
  .out_ready(out_ready)
);
```

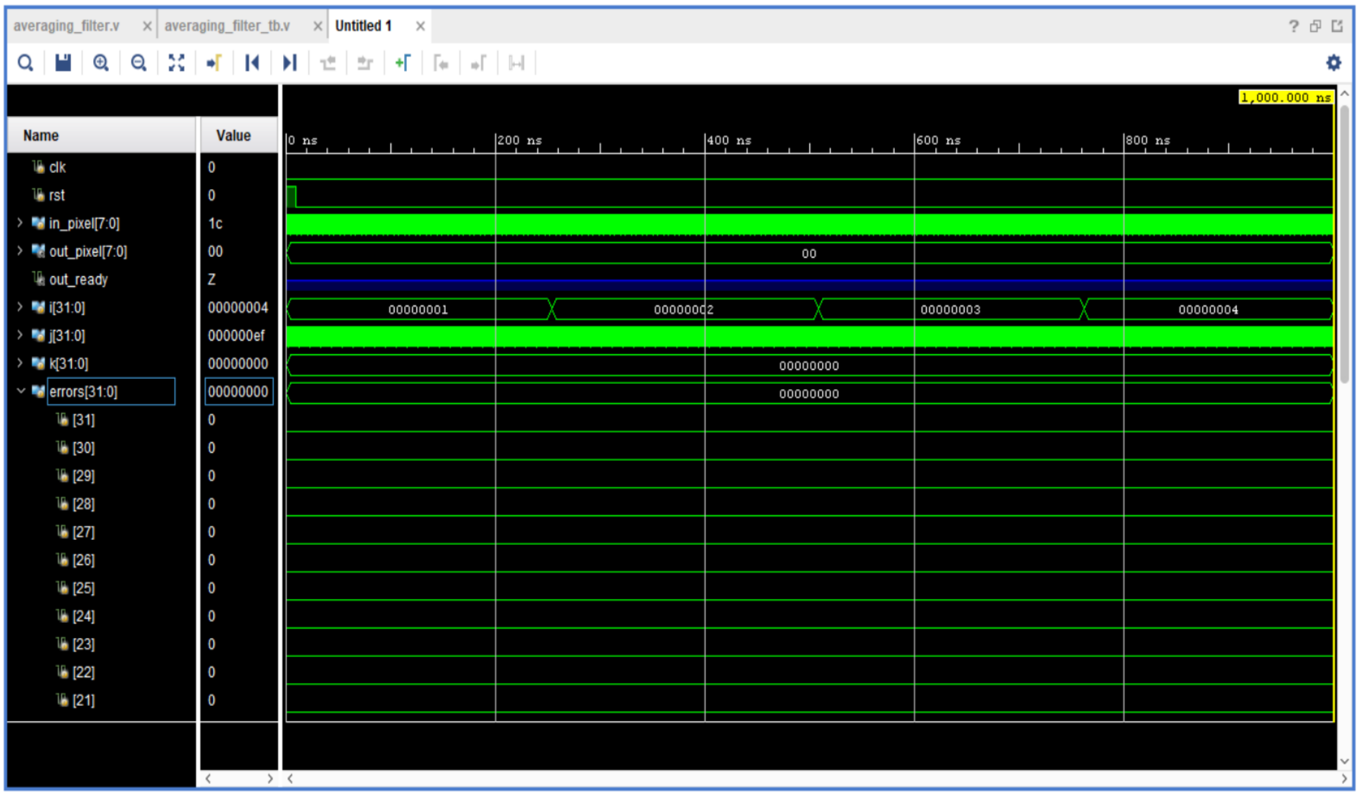
```
// Define the test parameters
parameter IMG_SIZE = 256;
parameter PIXEL_MAX = 255;
parameter TEST_COUNT = 100;

// Declare the test image and expected output
reg [7:0] test_image [0:IMG_SIZE-1][0:IMG_SIZE-1];
reg [7:0] expected_output [0:IMG_SIZE-1][0:IMG_SIZE-1];

// Declare test variables
integer i, j, k;
integer errors;

// Initialize the clock and reset signals
initial begin
  clk = 0;
  forever #5 clk = ~clk;
end

initial begin
  rst = 1;
  #10 rst = 0;
end

// Generate the test image with random noise
initial begin
  for (k = 0; k < TEST_COUNT; k = k + 1) begin
    for (i = 0; i < IMG_SIZE; i = i + 1) begin
      for (j = 0; j < IMG_SIZE; j = j + 1) begin
        test_image[i][j] = $random % (PIXEL_MAX+1);
      end
    end
    // Calculate the expected output
    for (i = 1; i < IMG_SIZE-1; i = i + 1) begin
      for (j = 1; j < IMG_SIZE-1; j = j + 1) begin
        expected_output[i][j] = (test_image[i-1][j-1] + test_image[i-1][j] + test_image[i-1][j+1] +
                               test_image[i][j-1]   + test_image[i][j]   + test_image[i][j+1]   +
                               test_image[i+1][j-1] + test_image[i+1][j] + test_image[i+1][j+1]) / 9;
      end
    end
    // Send the pixels to the filter
    errors = 0;
    for (i = 1; i < IMG_SIZE-1; i = i + 1) begin
      for (j = 1; j < IMG_SIZE-1; j = j + 1) begin
        in_pixel <= test_image[i][j];
        #1;
        if (out_ready) begin
          if (out_pixel != expected_output[i][j]) begin
            $display("Error: expected %h, got %h", expected_output[i][j], out_pixel);
            errors = errors + 1;
          end
        end
      end
    end
    if (errors == 0) begin
      $display("Test %d passed", k);
    end else begin
      $display("Test %d failed with %d errors", k, errors);
    end
  end
  $finish;
end

endmodule
```

RESULTS-

RTL SCHEMATIC DIAGRAM-



POST IMPLEMENTATION RESULT-



```
averaging_filter.v   averaging_filter_tb.v   Untitled 1

Name            Value        1,000.000 ns
                             0 ns   200 ns   400 ns   600 ns   800 ns
clk             0
rst             0
in_pixel[7:0]   1c
out_pixel[7:0]  00           00
out_ready       Z
i[31:0]         00000004     00000001   00000002   00000003   00000004
j[31:0]         000000ef
k[31:0]         00000000     00000000
errors[31:0]    00000000     00000000
  [31]          0
  [30]          0
  [29]          0
  [28]          0
  [27]          0
  [26]          0
  [25]          0
  [24]          0
  [23]          0
  [22]          0
  [21]          0
```

<u>CONCLUSION-</u>

In conclusion, the Verilog implementation of an averaging filter in image processing is a powerful tool for improving image quality. By taking the average of neighboring pixels, this filter can reduce noise and blur in images, resulting in clearer and more detailed pictures. The Verilog code for this filter is efficient and easy to implement, making it an excellent choice for real-time image processing applications. Overall, the averaging filter is an essential technique for enhancing image quality, and the Verilog implementation presented here provides a solid foundation for further exploration and development in this field.