Displaying and Filtering Grayscale Image

Objective

The objective of the lab was to display a 256 by 256 gray-scale image onto the VGA monitor. With keyboard attached to the FPGA board, each press of the '0' key would run the image through a Low Pass Filter and display the resulting image onto the monitor. Low Pass Filter operations were to be repeatable, successive filtering would cause the image to appear more blurred. The goal of this task was to learn general principles behind storing, retrieving, and processing image data.

Procedure

The image data was treated as a 256 by 256 matrix of intensity values. Each value was 4 bits and represented one pixel. To store the image, a memory array with 65536 (256 * 256) depth was required. The yielded memory with 16-bit address space, as 2^{16} = 65536. Each address space would hold one pixel and thus each address space would be 4 bits wide.

Imaging blurring effects can be achieved through use averaging. By averaging out the differences between the pixels, inconsistences between a pixel and its neighbors can be reduced. This yields the appearance of the image blurring or softening. The method used in this lab was a weighted averaging filter. Each pixel was cross correlated with the weights shown below, the resulting value replaces the pixel. The center pixel receives the greatest weight to reserves most of its features.

1/16	2/16	1/16
2/16	4/16	2/16
1/16	2/16	1/16

$\longrightarrow$

P-257	P-256	p-254
P-1	P	P+1
P+257	P+256	P+254

The resulting pixel values will need to be stored in a separate block of memory in order to not interfere with the filter process. As the result that replaces a pixel is dependent on its neighbors, if post filter pixel values are used each successive pixel would be 'more blurred' then the last. To solve this, two RAM modules were used in this design. RAM module 1 contains the image data that is to be displayed while RAM module 2 stores the result of the blur operations. To apply the filter to each pixel stored in RAM 1 the state machine detailed below was used. Upon pressing the zero key on the keyboard a state machine follows the steps below, starting from memory element 0 in RAM 2. A counter is used to address each pixel location in memory, counter initialized as zero and increments to 0xFFFF, one value for each pixel.

1. retrieves the value of the pixel and neighbors located at counter addressed space in mem
2. computes the sum of the pixel with weights applied
3. computes average of sum of pixels
4. stores the average in the location in memory of the source pixel in RAM2

5. if counter is less than 0xFFFF, increment counter and go back to step one. Else transfer contents of RAM2 to RAM1 and wait for another keyboard input.

To implement the design the following modules, reused from previous labs include:

- PS/2 Keyboard data parser: ps2_ctrl.v
- VGA controller: sync_gen.v
- Clock divider: clk_div.v
- ASCII decoder: ascii_convert.v

The module **ps2_ctrl.v** and **clk_div.v** remained unchanged for this lab, updates to the remaining where made to meet lab objective. The VGA controller **sync_gen.v** was updated to include blanking control within the module. Previously, registers which drove VGA color output were driven low from within the top module whenever the a blanking flag from the **sync_gen.v** module was asserted. Now, as can be seen in the code below, display conditions are included within the VGA control module. The blanking signal is directly tied to the VGA color output, driving the output low when blanking conditions are met.

assign blanking = (*blanking condition*) ? 4'b1111 : 4'b0000;

assign VGA_RED = *Color Data* & ~w_blanking;

The **asc_convert.v** module was updated to return a 2 bit wide confirmation packet when the zero key is pressed. All other actions send a low signal to the top module. The confirmation signal, 1 or 01 in binary, initiates the image bluring finite state machine.

```
always @(*)
    begin
      case(r_key)
        8'h45: r_keyAddr = 2'b01;
        default: r_keyAddr = 2'b00;
      endcase
    end
```

RESULTS

Various methods were used to test the design. To verify operation of the image blurring FSM, LEDs were used as a type of hardware breakpoint to give visual confirmation that each state of the FSM was executing. To confirm that the filter algorithm functioned as required a testbench was written to simulate its function, shown on following page. Utilization of dual port memories made transfer of image data simpler. By utilizing dual port memories, image display and image storage operations could address memory using separate drivers, simultaneously. This made operation is 2 clock domains easier, display addressing could run at 25MHz while memory transfer could be drives using the 100MHz clock.

Module Under Test

```
module top(i_clk, r_sum, r_filtPicData

                                    );

  input i_clk;


  output [7:0] r_sum;

  output [3:0] r_filtPicData;


  reg [5:0] r_pixelArrayUL = 6'hF;

  reg [5:0] r_pixelArrayUC = 6'hF;

  reg [5:0] r_pixelArrayUR = 6'hF;

  reg [5:0] r_pixelArrayL = 6'hF;

  reg [5:0] r_pixelArrayC = 6'hF;

  reg [5:0] r_pixelArrayR = 6'hF;

  reg [5:0] r_pixelArrayLL = 6'hF;

  reg [5:0] r_pixelArrayLC = 6'hF;

  reg [5:0] r_pixelArrayLR = 6'hF;


  assign r_sum = r_pixelArrayUL + 2*r_pixelArrayUC + r_pixelArrayUR + 2*r_pixelArrayL)+
4*r_pixelArrayC + 2*r_pixelArrayR + r_pixelArrayLL + 2*r_pixelArrayLC + r_pixelArrayLR;


  assign r_filtPicData = r_sum >> 4;


  endmodule
```

TESTBENCH

```
module test;
  reg i_clk = 0;

  wire[3:0] r_filtPicData;
  wire[7:0] r_sum;

  //Instantiate design under test
  top DUT(.i_clk(i_clk),.r_sum(r_sum),.r_filtPicData(r_filtPicData));

  initial          begin
             i_clk = 0;
  end

  always        begin
            #10 i_clk =! i_clk;
  end

  initial           begin
            $dumpfile("dump.vcd");
            $dumpvars(1);
  end

  initial
            #100 $finish;

endmodule
```