THE UNIVERSITY OF DANANG
UNIVERSITY OF SCIENCE AND TECHNOLOGY
Faculty of Advanced Science and Technology

REPORT

Image processing on FPGA using Verilog HDL

Instructor: Nguyen Van Cuong
Class: 21ES
Group: 2
Members:

1. Phan Dinh Quy
2. Tran Dinh Bao
3. Tran Minh Anh
4. Truong Phuoc Thinh

Da Nang, 12nd June, 2024

Table of Contents

A. Introduction

I. Overview

- This project demonstrates how to perform image processing on an FPGA using Verilog HDL.
- It covers the complete process from reading a bitmap image (.bmp) in Verilog, processing the image, and then writing the processed result back to an output bitmap image in Verilog.
- The project includes full Verilog code for image reading, image processing, and image writing.

II. Targets

- Show how to read an input bitmap image in Verilog
- Demonstrate simple image processing operations such as inversion, brightness control, thresholding, black & white, negative
- Write the processed image data to an output bitmap image
- Implementation on Altera Cyclone II

B. Software

I. Convert image file

- To read the .bmp image in Verilog, the image is required to be converted from the bitmap format to the hexadecimal format. Below is a Matlab code to convert a bitmap image to a hex file. The input image size is 768x512 and the image .hex file includes R, G, B data of the bitmap image.

```
b = imread('kodim24.bmp'); % 24-bit BMP image RGB888

k = 1;
for i = 512 : - 1 : 1 % image is written from the last row to the first row
for j = 1 : 768
a(k) = b(i,j,1);
a(k + 1) = b(i,j,2);
a(k + 2) = b(i,j,3);
k = k + 3;
end
end
fid = fopen('kodim24.hex', 'wt');
fprintf(fid, '%x\n', a);
disp('Text file write done');
disp(' ');
fclose(fid);
```

II. parameter.v file

```
`define INPUTFILENAME           "kodim23.hex"     // Input file name
`define OUTPUTFILENAME          "output.bmp"      // Output file name
```

```
//`define BRIGHTNESS_OPERATION
//`define INVERT_OPERATION
//`define THRESHOLD_OPERATION
//`define BLACKandWHITE_OPERATION
//`define RED_FILTER_OPERATION
//`define EMBOSS_OPERATION
`define NEGATIVE_OPERATION
```

– This parameter.v file contains macro definitions for various image processing operations. The user can enable one of these operations by removing the // mark at the beginning of the corresponding line. In this case, the negative operation (NEGATIVE_OPERATION) is active.

III. image_read.v file

1. Initial block and image data preparation

- Initial block

```
initial begin
      $readmemh(INFILE,total_memory,0,sizeOfLengthReal-1);
end
```

- Image data preparation

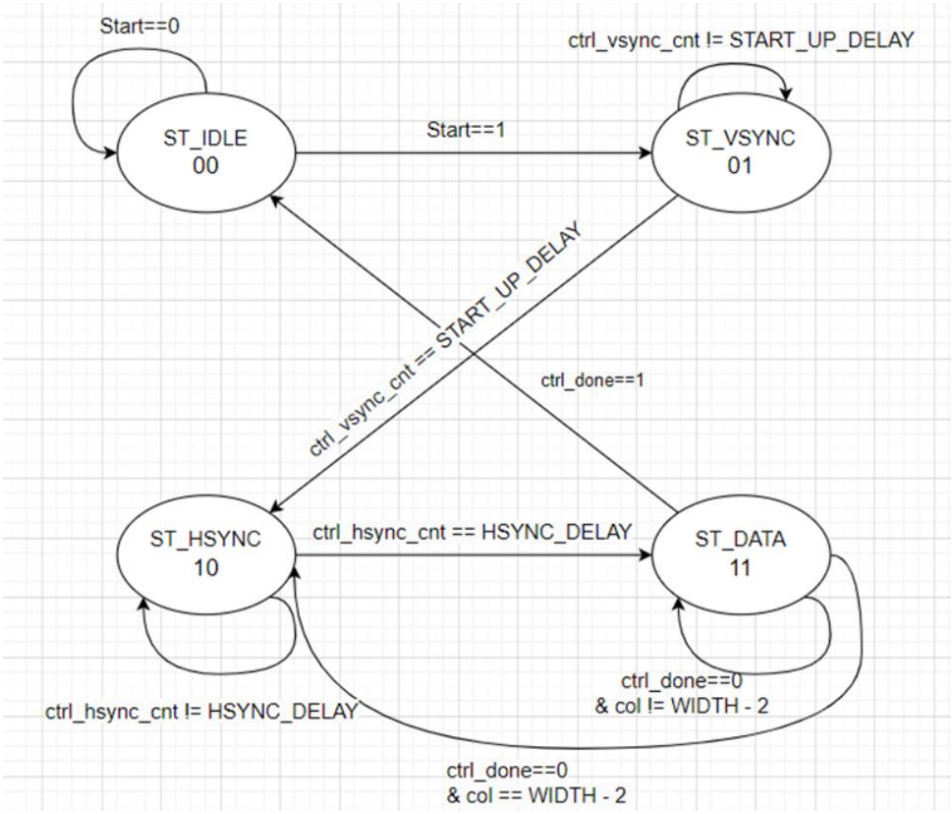
```
always@(start) begin
      if(start == 1'b1) begin
            for(i=0; i<WIDTH*HEIGHT*3 ; i=i+1) begin
                  temp_BMP[i] = total_memory[i+0][7:0];
            end

            for(i=0; i<HEIGHT; i=i+1) begin
                  for(j=0; j<WIDTH; j=j+1) begin
                        org_R[WIDTH*i+j] = temp_BMP[WIDTH*3*(HEIGHT-i-1)+3*j+0];
                        org_G[WIDTH*i+j] = temp_BMP[WIDTH*3*(HEIGHT-i-1)+3*j+1];
                        org_B[WIDTH*i+j] = temp_BMP[WIDTH*3*(HEIGHT-i-1)+3*j+2];
                  end
            end
      end
end
```

2. Start signal and FSM logic

```
always@(posedge HCLK, negedge HRESETn)
   begin
   if(!HRESETn) begin
      start <= 0;
      HRESETn_d <= 0;
   end
   else begin
      HRESETn_d <= HRESETn;
      if(HRESETn == 1'b1 && HRESETn_d == 1'b0)
            start <= 1'b1;
      else
            start <= 1'b0;
      end
```

```
end
```

- Purpose: Generate a pulse for the start signal after the reset signal is deasserted (i.e., when HRESETn transitions from low to high).
- Behavior:
  - If HRESETn is low (active low reset), both start and HRESETn_d are set to 0. On the rising edge of HCLK: HRESETn_d captures the current value of HRESETn.
  - If HRESETn is high and HRESETn_d was low (detecting a rising edge), start is set to 1. Otherwise, start is set to 0.
- Outcome: A single-cycle pulse on start when HRESETn transitions from low to high.



- FSM(Finite-state machine):
  - ST_IDLE: Idle state, waiting for the start signal.
  - ST_VSYNC: State for generating the vertical sync (VSYNC) pulse.
  - ST_HSYNC: State for generating the horizontal sync (HSYNC) pulse.
  - ST_DATA: State for processing and outputting image data.

3. Data processing

Brightness addition operation

```
// R0
tempR0 = org_R[WIDTH * row + col    ] + VALUE;
if (tempR0 > 255)
      DATA_R0 = 255;
else
      DATA_R0 = org_R[WIDTH * row + col    ] + VALUE;
```

- This code performs brightness addition for two pixels (even and odd) at the same time. The brightness value is added to each color channel (red, green, blue) and is limited to the range 0-255. If ctrl_data_run is high, the HSYNC signal is set to 1 to signal data processing.

Brightness subtraction operation

```
// R1
tempR1 = org_R[WIDTH * row + col+1    ] - VALUE;
if (tempR1 < 0)
      DATA_R1 = 0;
else
      DATA_R1 = org_R[WIDTH * row + col+1    ] - VALUE
```

- This code has the same performance as in brightness addition operation.

Black & white operation

```
value2 = (org_B[WIDTH * row + col   ] + org_R[WIDTH * row + col   ] +org_G[WIDTH *
row + col   ])/3;
DATA_R0=value2;
DATA_G0=value2;
DATA_B0=value2;
value4 = (org_B[WIDTH * row + col+1   ] + org_R[WIDTH * row + col+1   ]
+org_G[WIDTH * row + col+1   ])/3;
DATA_R1=value4;
DATA_G1=value4;
DATA_B1=value4;
```

- This code converts a color image to a black and white (or grayscale) image by calculating the average of the three color channels for each pixel. This average value is then assigned to all three color channels of the pixel, creating a black-and-white effect. This conversion is performed for two pixels (even and odd) at the same time to speed up processing.

Negative operation

```
DATA_R0 = 255 - org_R[WIDTH * row + col   ];
DATA_G0 = 255 - org_G[WIDTH * row + col   ];
DATA_B0 = 255 - org_B[WIDTH * row + col   ];
DATA_R1 = 255 - org_R[WIDTH * row + col+1   ];
DATA_G1 = 255 - org_G[WIDTH * row + col+1   ];
DATA_B1 = 255 - org_B[WIDTH * row + col+1   ];
```

- This code converts a color image into a negative image by subtracting the original value of each color channel (red, green, blue) from 255 for each pixel. This process is performed for two pixels (even and odd) at the same time to increase

processing speed. A negative image is the result of inverting the values of pixels, making bright pixels dark and vice versa.

Invert operation

```
value2 = (org_B[WIDTH * row + col  ] + org_R[WIDTH * row + col  ] +org_G[WIDTH *
row + col  ])/3;
DATA_R0=255-value2;
DATA_G0=255-value2;
DATA_B0=255-value2;
value4 = (org_B[WIDTH * row + col+1  ] + org_R[WIDTH * row + col+1  ]
+org_G[WIDTH * row + col+1  ])/3;
DATA_R1=255-value4;
DATA_G1=255-value4;
DATA_B1=255-value4;
```

– This code converts a color image to a color inverted image by calculating the average value of the color channels (red, green, blue) for each pixel, then subtracting this average value from 255. to create inverted grayscale for pixels. This process is performed for two pixels (even and odd) at the same time to increase processing speed. A color inverted image is the result of inverting the brightness values of pixels, making bright pixels dark and vice versa.

Threshold operation

```
value = (org_R[WIDTH * row + col   ]+org_G[WIDTH * row + col   ]+org_B[WIDTH *
row + col   ])/3;
if(value > THRESHOLD) begin
      DATA_R0=255;
      DATA_G0=255;
      DATA_B0=255;
end
else begin
      DATA_R0=0;
      DATA_G0=0;
      DATA_B0=0;
end
```

– This code converts a color image to black and white by applying a threshold to the average value of the three color channels (red, green, blue) for each pixel. If the average value is greater than the threshold value, the pixel will be set as white (255). Otherwise, the pixel will be set to black (0). This process is performed for two pixels (even and odd) at the same time to increase processing speed.

Red filter operation

```
DATA_R0 = org_R[WIDTH * row + col  ];
DATA_G0 = 0;
DATA_B0 = 0;
DATA_R1 = org_R[WIDTH * row + col+1  ];
DATA_G1 = 0;
DATA_B1 = 0;
```

- This block of code implements a red filter operation on the image data. When enabled (by defining RED_FILTER_OPERATION), it sets the green and blue components of the image pixels to 0, retaining only the red component. This results in an image where only the red color is preserved, and all other colors are removed, creating a "red filter" effect.

Emboss operation

```
if (row > 0 && row < HEIGHT-1 && col > 0 && col < WIDTH-1) begin
      // Get the surrounding pixel values
      pixel_tl = org_R[WIDTH * (row - 1) + (col - 1)];
      pixel_tm = org_R[WIDTH * (row - 1) + col];
      pixel_tr = org_R[WIDTH * (row - 1) + (col + 1)];
      pixel_ml = org_R[WIDTH * row + (col - 1)];
      pixel_mm = org_R[WIDTH * row + col];
      pixel_mr = org_R[WIDTH * row + (col + 1)];
      pixel_bl = org_R[WIDTH * (row + 1) + (col - 1)];
      pixel_bm = org_R[WIDTH * (row + 1) + col];
      pixel_br = org_R[WIDTH * (row + 1) + (col + 1)];

      // Apply the emboss filter
      DATA_R0 = 128 + (pixel_mm - (pixel_tl >> 1));
      DATA_R1 = 128 + (pixel_mr - (pixel_bl >> 1));

      // Ensure the output is within 0-255 range
      if (DATA_R0 > 255) DATA_R0 = 255;
      if (DATA_R0 < 0)   DATA_R0 = 0;

      if (DATA_R1 > 255) DATA_R1 = 255;
      if (DATA_R1 < 0)   DATA_R1 = 0;

      // Assuming monochrome emboss effect
      DATA_G0 = DATA_R0;
      DATA_B0 = DATA_R0;

      DATA_G1 = DATA_R1;
      DATA_B1 = DATA_R1;
end
```

- This code implements an emboss effect for an image. When EMBOSS_OPERATION is defined, it processes the image data to create a 3D-like effect by modifying pixel values based on their surrounding neighbors. The result is stored in DATA_R0, DATA_G0, DATA_B0 for the current pixel and DATA_R1, DATA_G1, DATA_B1 for the next pixel in the same row.

IV. image_write.v file

1. Function

Used to create a BMP image (Bit map picture) from a stream of data pixels. This module takes pixel data as input, processes it and creates a BMP format file, and has a signal when the data recording process is completed.

2. Code

Module Parameters

```
module image_write
     #(parameter WIDTH  = 768,             // Image width
                 HEIGHT      = 512,        // Image height
                 INFILE  = "output.bmp",   // Output image
                 BMP_HEADER_NUM = 54       // Header for bmp image
     )
```

– Here we have parameters according to the image resolution configuration, which is 768x512. Infile is a file in BMP format, the image has been recorded in this file. The size of the BMP header will be 54.

BMP Header Initialization

```
BMP_header[ 0] = 66;BMP_header[28] =24;
BMP_header[ 1] = 77;BMP_header[29] = 0;
BMP_header[ 2] = 54;BMP_header[30] = 0;
BMP_header[ 3] =  0;BMP_header[31] = 0;
BMP_header[ 4] = 18;BMP_header[32] = 0;
BMP_header[ 5] =  0;BMP_header[33] = 0;
BMP_header[ 6] =  0;BMP_header[34] = 0;
BMP_header[ 7] =  0;BMP_header[35] = 0;
BMP_header[ 8] =  0;BMP_header[36] = 0;
BMP_header[ 9] =  0;BMP_header[37] = 0;
BMP_header[10] = 54;BMP_header[38] = 0;
BMP_header[11] =  0;BMP_header[39] = 0;
BMP_header[12] =  0;BMP_header[40] = 0;
BMP_header[13] =  0;BMP_header[41] = 0;
BMP_header[14] = 40;BMP_header[42] = 0;
BMP_header[15] =  0;BMP_header[43] = 0;
BMP_header[16] =  0;BMP_header[44] = 0;
BMP_header[17] =  0;BMP_header[45] = 0;
BMP_header[18] =  0;BMP_header[46] = 0;
BMP_header[19] =  3;BMP_header[47] = 0;
BMP_header[20] =  0;BMP_header[48] = 0;
BMP_header[21] =  0;BMP_header[49] = 0;
BMP_header[22] =  0;BMP_header[50] = 0;
BMP_header[23] =  2;BMP_header[51] = 0;
BMP_header[24] =  0;BMP_header[52] = 0;
BMP_header[25] =  0;BMP_header[53] = 0;
BMP_header[26] =  1;
BMP_header[27] =  0;
```

– At offset 0-1, we have the signature, which are the two characters 'B' and 'M' in BMP according to ASCII.
– At offset 2-5, we see the size of the header, which ranges from 18 to 54 bytes.
– At offset 10-13, this marks the starting address of the pixel data, which is standard for a BMP file.
– At offset 14, we see the value 40, which is always 40 according to the standard 54-byte header of a BMP file.

- At offset 18-21, we have the Width of the image, which is 0x003 hexadecimal for 768.
- At offset 22-25, we have the Height of the image, which is 0x002 hexadecimal for 512.

Row & Column Counting

```
always@(posedge HCLK, negedge HRESETn) begin
     if(!HRESETn) begin
          l <= 0;
          m <= 0;
     end else begin
          if(hsync) begin
               if(m == WIDTH/2-1) begin
                    m <= 0;
                    l <= l + 1;
               end else begin
                    m <= m + 1;
               end
          end
     end
end
```

- This always block is used to create the necessary number of rows and columns to store the pixel values of the entire image.
- The image will be read when there is an hsync pulse. Each time this happens, it will start creating columns until it reaches the maximum number of columns in that row, then the number of rows will increase by 1. This maximum value is divided by two because we use the parallel processing method, meaning that in a certain row and column there will be two RGB values, one even and one odd.

Writing To Temp Memory

```
always@(posedge HCLK, negedge HRESETn) begin
    if(!HRESETn) begin
        for(k=0;k<WIDTH*HEIGHT*3;k=k+1) begin
            out_BMP[k] <= 0;
        end
    end else begin
        if(hsync) begin
            out_BMP[WIDTH*3*(HEIGHT-l-1)+6*m+2] <= DATA_WRITE_R0;
            out_BMP[WIDTH*3*(HEIGHT-l-1)+6*m+1] <= DATA_WRITE_G0;
            out_BMP[WIDTH*3*(HEIGHT-l-1)+6*m  ] <= DATA_WRITE_B0;
            out_BMP[WIDTH*3*(HEIGHT-l-1)+6*m+5] <= DATA_WRITE_R1;
            out_BMP[WIDTH*3*(HEIGHT-l-1)+6*m+4] <= DATA_WRITE_G1;
            out_BMP[WIDTH*3*(HEIGHT-l-1)+6*m+3] <= DATA_WRITE_B1;
        end
    end
end
```

- This always block functions to write data into temporary memory, with the table created by the always block above.

- What we need to know is that BMP files are written from bottom to top. To ensure accurate reading and writing of values, when there is an hsync pulse, 6 bytes of data will be written according to position. For example, if the data were written in a typical manner, it would appear as follows from bottom to top: R0, G0, B0; R1, G1, B1. This is incorrect for the BMP file format since the first data is written at the highest position. What we need to do is rearrange the position so it becomes B0, G0, R0; B1, G1, R1.

Generate Done Flat

```
// data counting
always@(posedge HCLK, negedge HRESETn)
begin
    if(~HRESETn) begin
        data_count <= 0;
    end
    else begin
        if(hsync)
            data_count <= data_count + 1;
    end
end
assign done = (data_count == 196607)? 1'b1: 1'b0;
always@(posedge HCLK, negedge HRESETn)
begin
    if(~HRESETn) begin
        Write_Done <= 0;
    end
    else begin
        Write_Done <= done;
    end
end
```

- The always block above is responsible for counting data and generating a 'Write done' signal when the counting process is complete.

Write .bmp File

```
initial begin
    fd = $fopen(INFILE, "wb+");
end
always@(Write_Done) begin
    if(Write_Done == 1'b1) begin
        for(i=0; i<BMP_HEADER_NUM; i=i+1) begin
            $fwrite(fd, "%c", BMP_header[i][7:0]);
        end

        for(i=0; i<WIDTH*HEIGHT*3; i=i+6) begin
            $fwrite(fd, "%c", out_BMP[i  ][7:0]);
            $fwrite(fd, "%c", out_BMP[i+1][7:0]);
            $fwrite(fd, "%c", out_BMP[i+2][7:0]);
            $fwrite(fd, "%c", out_BMP[i+3][7:0]);
            $fwrite(fd, "%c", out_BMP[i+4][7:0]);
            $fwrite(fd, "%c", out_BMP[i+5][7:0]);
```

```
        end
    end
end
```

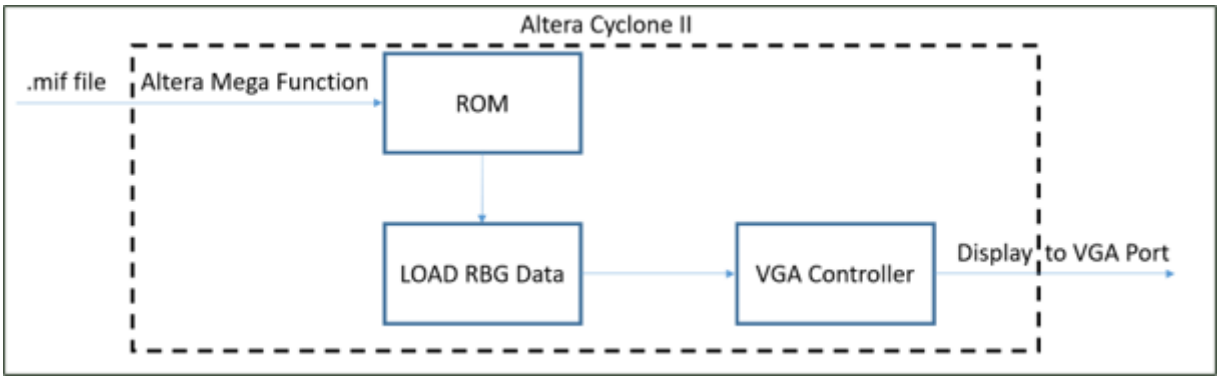
- Finally, after the 'Write done' signal, 6 bytes of data will be written in a loop, thereby creating the desired image.

V. tb_simulation.v file

- This testbench sets up a simulation environment for the image_read and image_write modules.
- It defines clock and reset signals, instantiates the modules, and sets up the clock and reset behavior.
- By modifying the macros in the included parameter.v file and enabling different operations, the testbench can be used to simulate various image processing tasks.

C. Implementation in Quartus

- There are a lot of ways to do this:
  + Using ROM in board (choosing)
  + Using SD card
  + Using UART
  + Using Ethernet
- We've tried to make an image-processing application. Here are our steps trying to do:



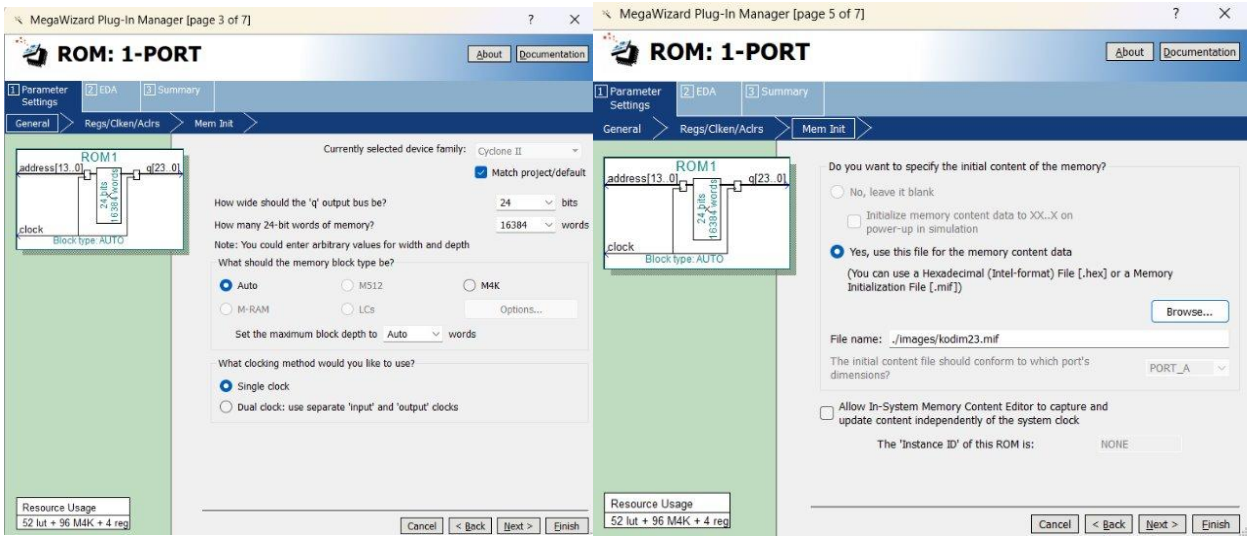
I. Generate image input

```
DEPTH=9126;
WIDTH=24;
ADDRESS_RADIX = UNS;
DATA_RADIX = HEX;
CONTENT BEGIN
        0       :       817f62;
        1       :       a0987c;
        2       :       b4a890;
        3       :       c4b49d;
        4       :       d6c6af;
        5       :       e0cfbc;
        6       :       e0cebb;
        7       :       e0cdb9;
        8       :       dbc9b4;
        9       :       d2c0ab;
```

- Convert file .bmp to .mif file (Memory Initialization File):
  + We want to display a custom image, so we first need to prepare it so FPGA can understand it.
  + The generate_mif_file.m script takes an input image and outputs a .mif file to be referenced inside the Verilog code.

II. ROM image loading

- We generate a block memory using Altera Mega Wizard Plug-in Manager and load the image data into the initial values of memory (.mif file)



III. Read image from ROM

ROM Module Instantiation

```
ROM1 #(
    .init_file(init_file),
    .data_width(24),
    .address_width(ADDRESS_WIDTH),
    .memory_size(MEMORY_SIZE)
```

```
) image (
    .address(pixel_address),
    .clock(clk),
    .q(pixel_data)
);
```

Always Block for Pixel Index and Should Draw Logic

```
always @(posedge clk) begin
    % Image is serialized column-major?
    pixel_index <= row + (column * IMAGE_HEIGHT);
    % Converting conditions into should_draw logic?
    should_draw <= (column < IMAGE_WIDTH) && (row < IMAGE_HEIGHT) && (pixel_index
< MEMORY_SIZE);
end
```

Address Conversion and Pixel Data Assignment

```
assign pixel_address = pixel_index[ADDRESS_WIDTH-1:0];
assign pixel[7:0]   = (should_draw) ? pixel_data[7:0] : 8'h00;     // Red
assign pixel[15:8]  = (should_draw) ? pixel_data[15:8] : 8'h00;    // Green
assign pixel[23:16] = (should_draw) ? pixel_data[23:16] : 8'h00;   // Blue
```

D. Result

Figure 1. Original

Figure 2. Brightness addition

Figure 3. Brightness subtraction

Figure 4. Black & white

Figure 5. Negative

Figure 6. Invert

Figure 7. Threshold

Figure 8. Red filter

Figure 9. Emboss

E. Conclusion

I. Achievements

- Designed and implemented an Image Processor using Verilog HDL, demonstrating a customizable architecture suitable for a range of image processing applications.
- Effective simulation with ModelSim, proving the design's functionality and performance
- Applying our knowlegdes about the FPGA and Verilog course

II. Future work

- Since we've not fully done in running our systems on Altera Cyclone II board, our next steps involve future development to ensure that our systems working on hardware kit:

+ Displaying image by using VGA
+ Applying with more images and techniques
+ Developing a real time image processing