Team 2: Image Editing Toolbox

Final Report

Xavier Palomo
Shan Kuan
Yuting Fu
Muhammad Ikram Ul Haq
Zeeshan Hayat

Abstract

Existing image processing applications like photoshop, MS Paint etc. usually process images on the software level. The general idea of our project, however, is to implement some of the basic image processing algorithms on the hardware level, and see how fast our application can process as compared to some software applications.

1. Introduction

We developed an Android interface for the user to upload the image and choose the editing method they want to implement on the image. The uploaded image will then be sent to our FPGA board using FTP protocol. There is a Linux system running on the board. We use XML-RPC to communicate between the client APP and the server on the board. The image received by the board will be processed by a python program which converts the image to a matrix of binary numbers. The binary matrix will then be mapped into the physical memory on the FPGA board using mmap method. After storing the image in DDR memory on the board, we will use DMA controller to take the stored image from DDR memory, and then process it and store it back to DDR memory.

At the moment, we have two algorithms available for users to choose from: Grayscale and Binary Image Conversion. The final processed image will be transferred back to our Android app via FTP.

The block diagram shown in **figure 1** illustrates the overall workflow of our system.

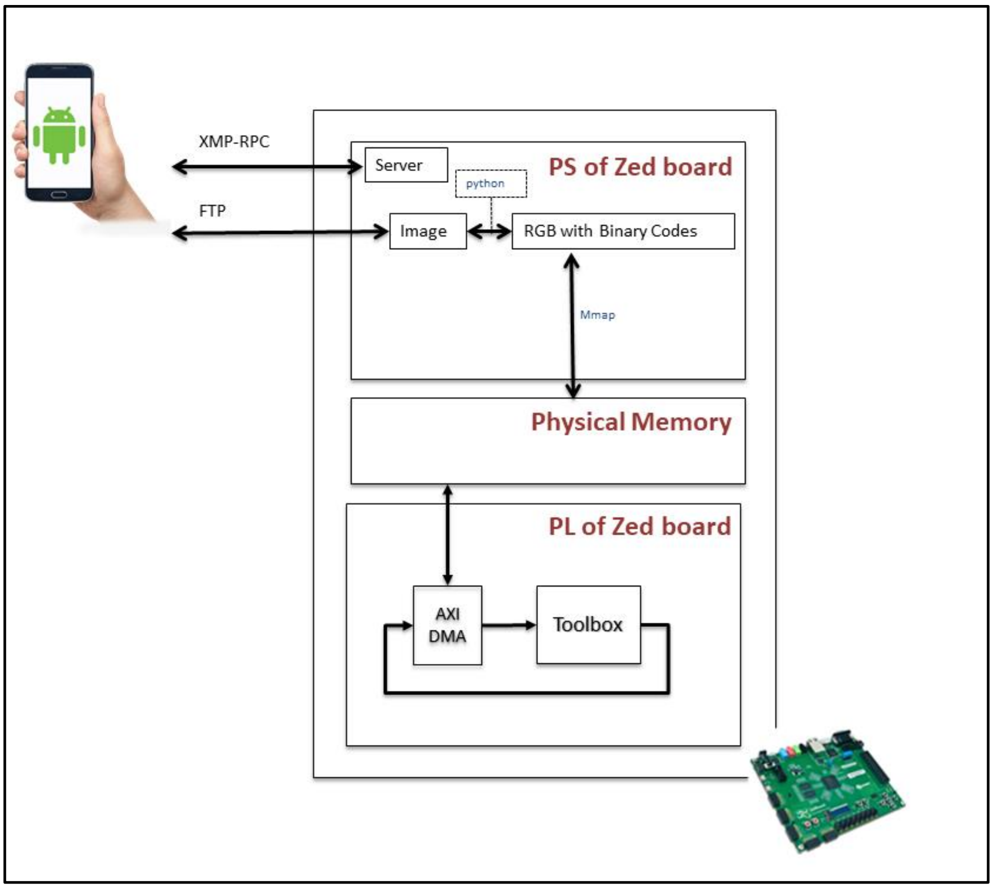


Figure 1: System Workflow

2. Proposed Solution

This portion will explain how our proposed system works and how different parts interact with each other in order to make communication and processing possible.

2.1 Image Upload

We developed a simple Android application for the user to upload the image that needed to be edited. Then the user needs to choose the way they want to edit the image. At the moment, Binary Image conversion and Grayscale algorithms are available. The user could also choose to take a picture from the phone or upload an image from the phone camera roll. The interface of uploading an image and choosing the algorithm is shown in the **figure 2** below.

2.2 Image conversion

To implement the image processing algorithms on the hardware level, we need to first convert the PNG image into the binary format. The processing algorithms will deal with image in RGB model, which means each pixel is represented by three numbers indicating the brightness intensities of red, green and blue channels respectively.

Our front-end user will upload a PNG image to the Android application, or take a photo from the camera of the phone. Then the image will be sent to the board via FTP protocol. In order to enable transferring of the image file we need to make sure communication system is established and running properly between Android application and Zedboard. For this purpose, we use XML-RPC protocol to enable communication between the client app, Andriod app in our case, and the server on the FPGA board.

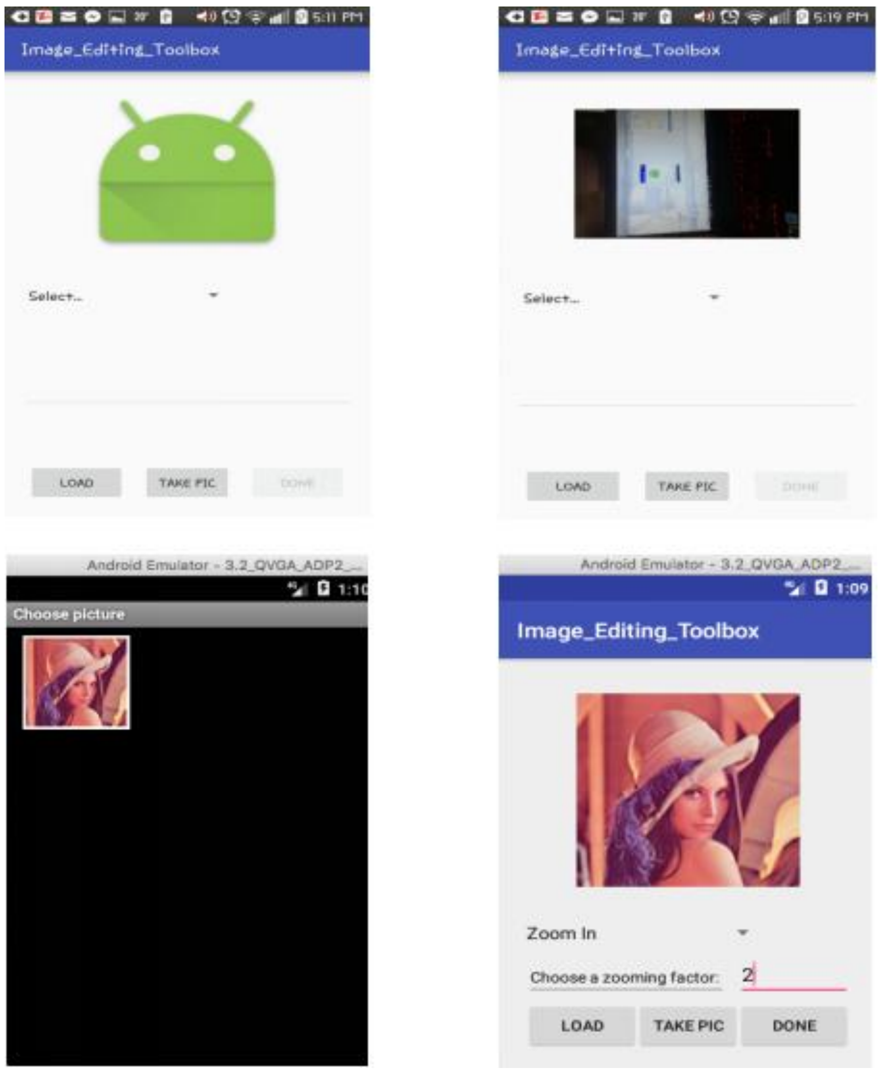


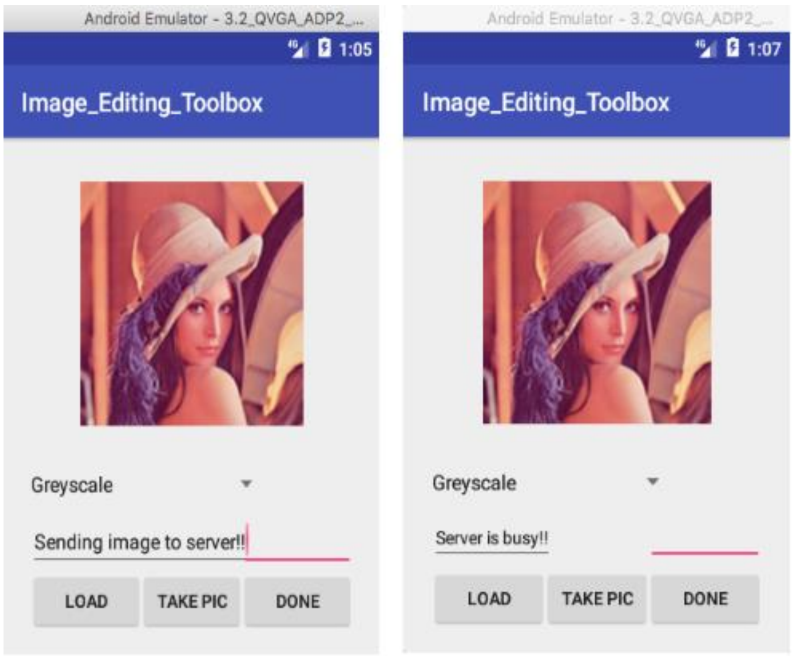
Figure 2: User Interface for image uploading and algorithm selection

XML-RPC works by sending an HTTP request to the FPGA board where the protocol is implemented. Multiple input parameters can be passed to this remote client, and one value is returned, which indicates whether the server is available or not as shown in **figure 3**. The parameter types allow us to nest parameters into maps and lists. Therefore, XML-RPC can be used to transport objects or structures as input and as output parameters as well.

After identifying that the server is available, the client-server connection is established, and the image file is sent to the board via FTP. A Python program on the board will then convert the received image into a binary file.

The File Transfer Protocol (FTP) is built on a client-server model architecture and uses separate control and data connections between the client and the server. FTP users may authenticate themselves with a clear-text sign-in protocol, normally in the form of a username and password, but can connect anonymously if the server is configured to allow it. Identification of clients for authorization purposes can be achieved using popular HTTP security methods. Basic access authentication is used for identification. HTTPS is used when identification (via certificates) and encrypted messages are needed.

The Python program will first parse the image into RGB model format, then further convert the numbers into a matrix of binary numbers that can later be mapped into the physical memory using memmap.



Sending image to server!!

Server is busy!!

Figure 3: Check the Server

2.3 Access to the Memory

The binary matrix of the image will first be mapped into the physical memory on the FPGA board via mmap. Mmap is a unix system call that maps files or devices into memory. It is a method of memory-mapped file I/O. It naturally implements demand paging, because file contents are not read from disk initially and the physical RAM is not used at all. The actual reads from the disk are performed in a "lazy" manner after a specific location is accessed.

The memory reserved on the FPGA board is 256MB, and we use 32MB of it to store the image.

2.4 DDR

To access the DDR memory from the PL part of zedboard, there is no direct connection available. Therefore, we used the high performance AXI port of zynq processor that is connected to the memory controller. To enable the communication from DDR to PL as well as from PL to DDR, we used the AXI DMA controller. AXI DMA refers to traditional FPGA direct memory access which roughly corresponds to transferring arbitrary streams of bytes from FPGA to a slice of DDR memory, and vice versa. AXI DMA distinguishes two channels:

- MM2S (memory-mapped to stream) transports data from DDR memory to FPGA.
- S2MM (stream to memory-mapped) transports arbitrary data stream to DDR memory.

The **figure 4** below explains the internal working of AXI DMA controller.

AXI DMA has different configurations and modes which we can enable. In our project we are using simple mode which sends and receives data in a linear way. We get 32 bits of data from DDR memory to process it and send it back to DDR memory. The configuration which we used for AXI DMA in our project is shown below in **figure 5.**

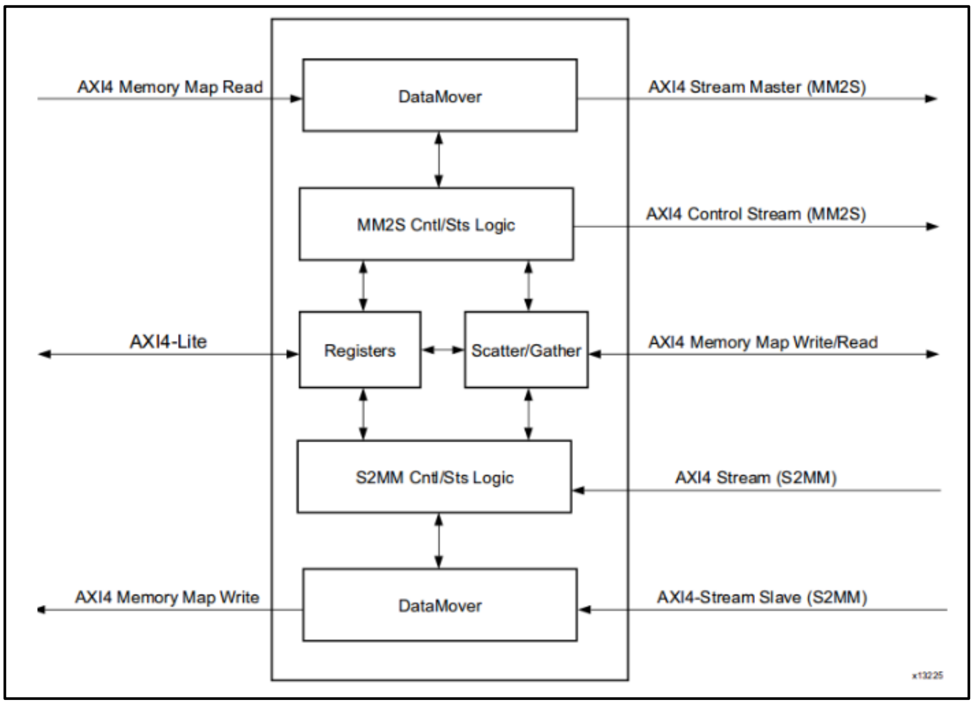
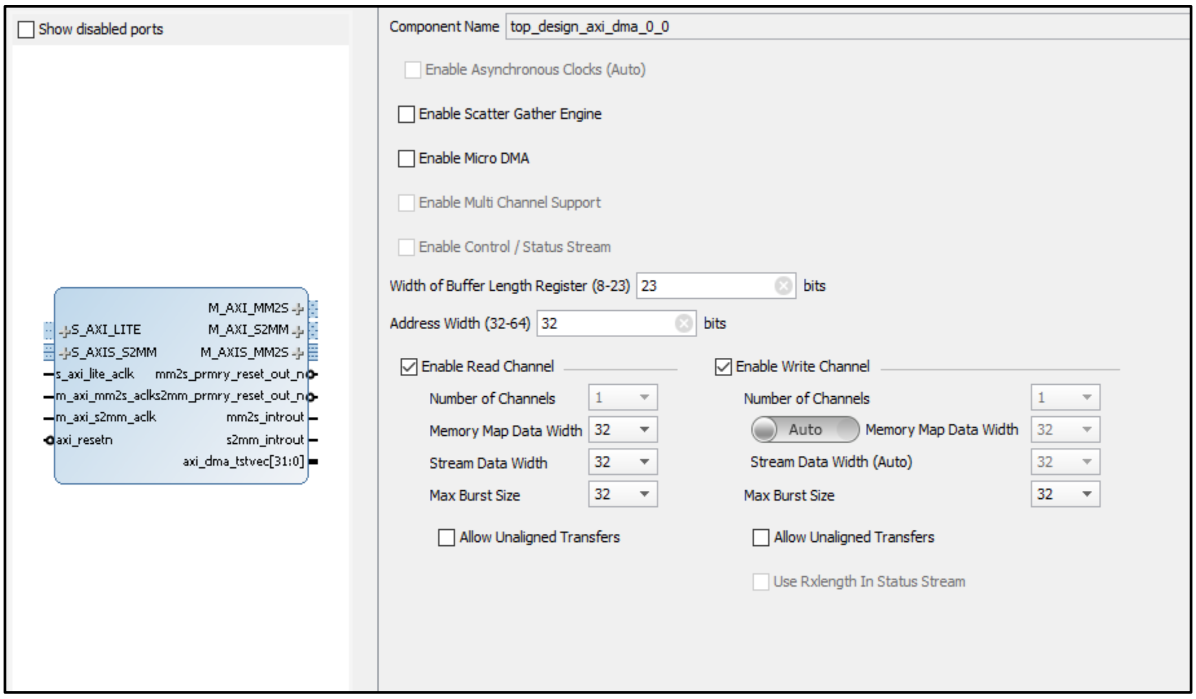


Figure 4: AXI DMA controller



☐ Show disabled ports

Component Name: top_design_axi_dma_0_0

☐ Enable Asynchronous Clocks (Auto)

☐ Enable Scatter Gather Engine

☐ Enable Micro DMA

☐ Enable Multi Channel Support

☐ Enable Control / Status Stream

Width of Buffer Length Register (8-23): 23 bits

Address Width (32-64): 32 bits

✓ Enable Read Channel		✓ Enable Write Channel	
Number of Channels	1	Number of Channels	1
Memory Map Data Width	32	Auto Memory Map Data Width	32
Stream Data Width	32	Stream Data Width (Auto)	32
Max Burst Size	32	Max Burst Size	32
☐ Allow Unaligned Transfers		☐ Allow Unaligned Transfers	
		☐ Use Rxlength In Status Stream	

Figure 5: AXI DMA Configuration for Our Project

2.5 Algorithm Implementation

2.5.1 Grayscale Algorithm

A grayscale or grayscale digital image is an image in which the value of each pixel is a single sample that carries only intensity information. Images of this sort are composed exclusively of shades of gray, varying from black at the weakest intensity to white at the strongest.

There are different ways to convert a color image to grayscale image. In our algorithm, to simplify the process, we choose the average method which simply averages the values: (R + G + B) / 3.

Our grayscale algorithm takes 32 bits from the binary image matrix as an input, which consists of three 8-bit RGB values of one pixel and another 8-bit ASCII code indicating a row number after the three values. The output of this algorithm would be 32-bit data of the same format, which consists of three same grayscale value for RGB channels and same row number at the end. The general circuit structure to implement this grayscale algorithm is shown in **figure 6** below.

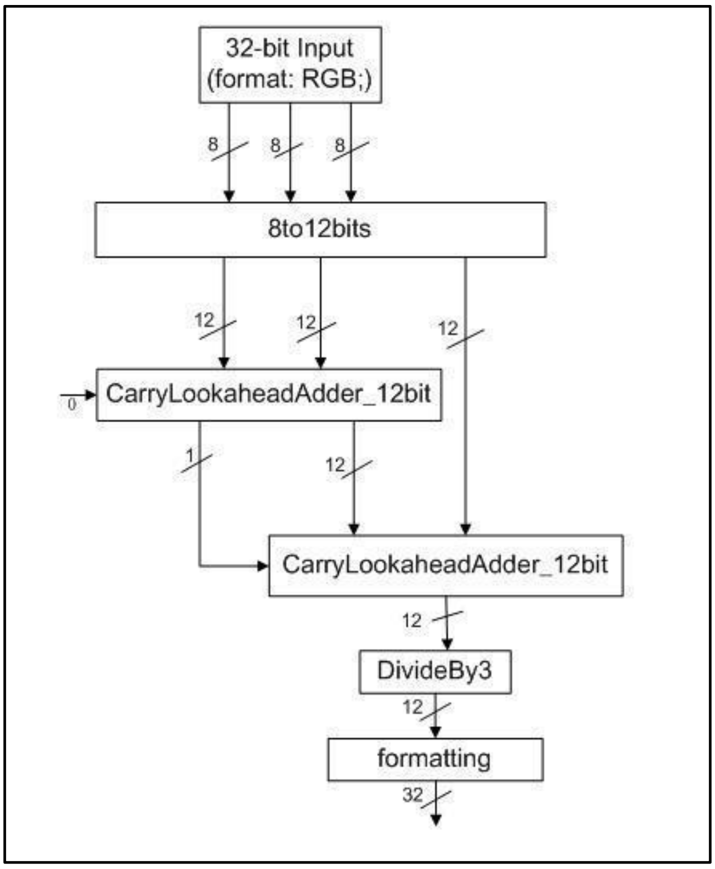
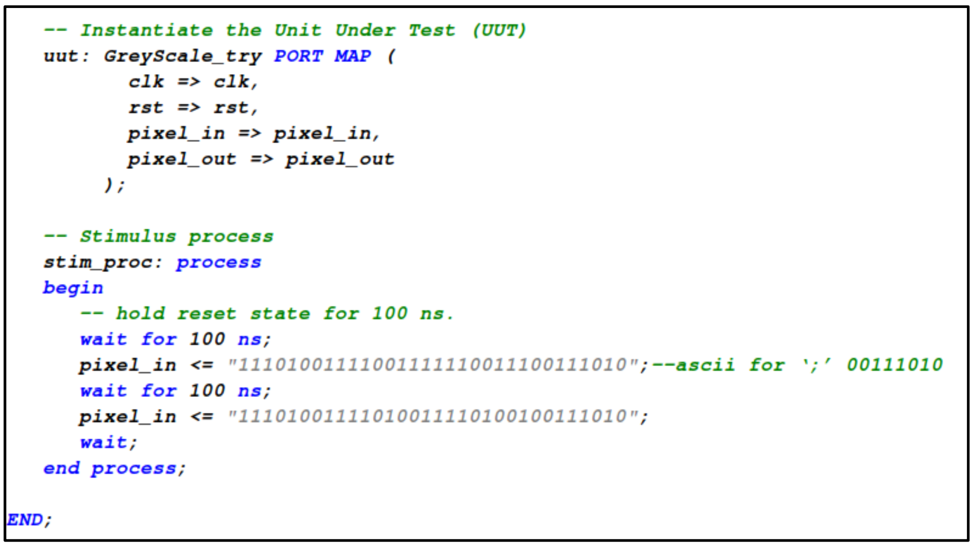


Figure 6: GrayScale Algorithm Block Diagram

Before implementing the algorithm on the board, we created a simple testbench to simulate the circuit and got a desired result as shown in the **figure 7** below.

Two different input strings: “11101001111001111110011100111010” and “11101001111010011110100100111010” are provided as inputs in the testbench, which are the binary format of “233 231 231;” and “233 233 233; ”. And the simulation outputs are “231 231 231;” and “233 233 233;”. The simulation results are shown in **figure 8**.



```
   -- Instantiate the Unit Under Test (UUT)
   uut: GreyScale_try PORT MAP (
          clk => clk,
          rst => rst,
          pixel_in => pixel_in,
          pixel_out => pixel_out
        );

   -- Stimulus process
   stim_proc: process
   begin
      -- hold reset state for 100 ns.
      wait for 100 ns;
      pixel_in <= "11101001111001111110011100111010";--ascii for ';' 00111010
      wait for 100 ns;
      pixel_in <= "11101001111010011110100100111010";
      wait;
   end process;

END;
```

Figure 7: GrayScale Algorithm Testbench

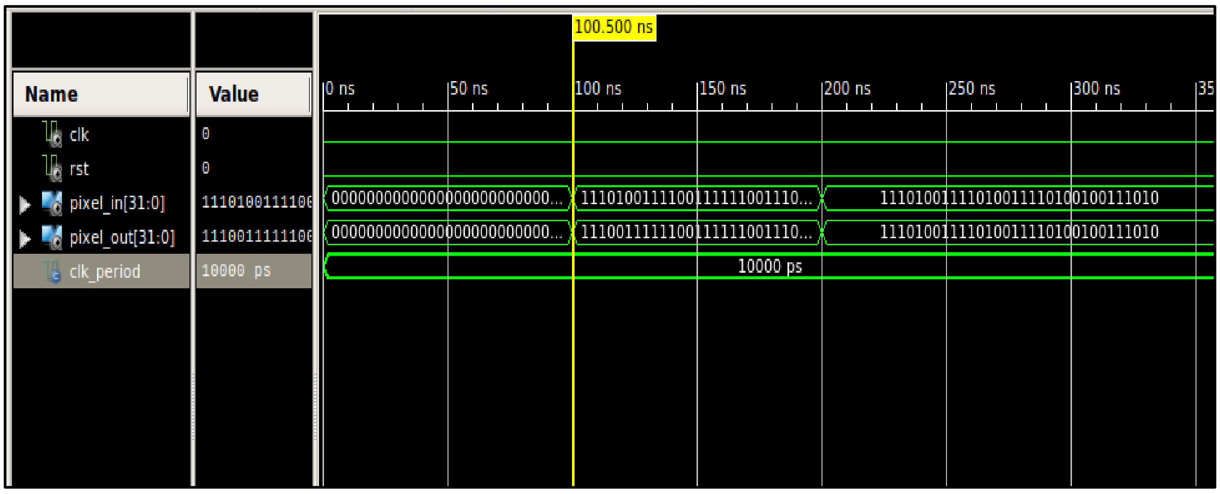


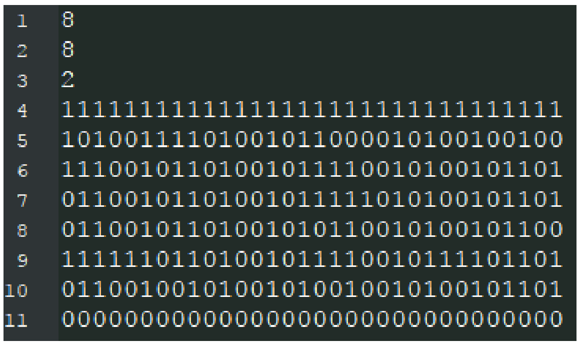
Figure 8: GrayScale Algorithm Simulation Results

2.5.2 Zoom in Algorithm

In order to zoom in an image, there are many existing algorithms such as bilinear, bicubic interpolation, or nearest neighbor (pixel replication). For this project, we have opted for that last one.

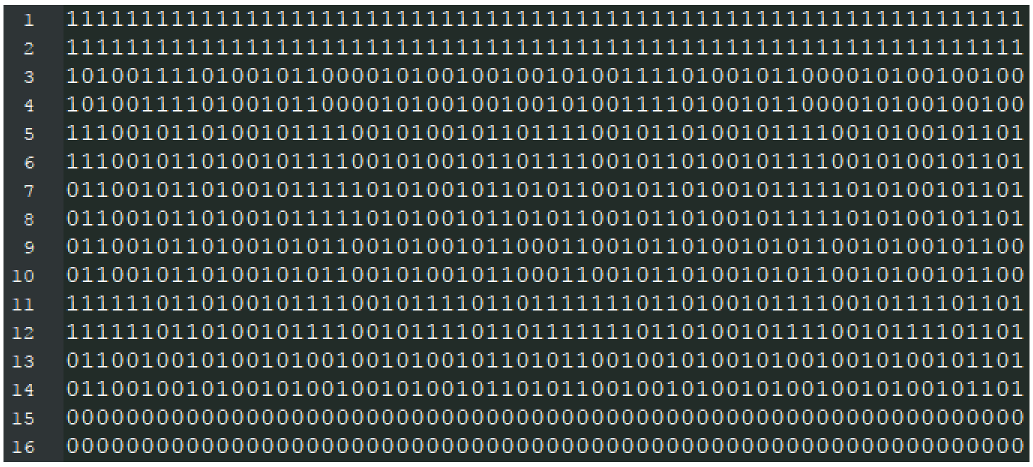
The main idea of this algorithm is to, given a zooming parameter, replicate the value of each pixel both vertically and horizontally. We achieve that by means of two loops which will write back the read value. The number of iterations will depend on the zooming parameter we want to apply.

Initial input - 8x8 matrix with a zooming factor of 2 is shown below in **figure 9**. The Output-16-16 matrix as shown in **figure 10**. We can appreciate the replication logic as described in **figure 11**.



```
8
8
2
11111111111111111111111111111111
10100111101001011000010100100100
11100101101001011110010100101101
01100101101001011111010100101101
01100101101001010110010100101100
11111101101001011110010111101101
01100100101001010010010100101101
00000000000000000000000000000000
```

Figure 9: Zoom In Algorithm Input Matrix



```
1111111111111111111111111111111111111111111111111111111111111111
1111111111111111111111111111111111111111111111111111111111111111
1010011110100101100001010010010010100111101001011000010100100100
1010011110100101100001010010010010100111101001011000010100100100
1110010110100101111001010010110111100101101001011110010100101101
1110010110100101111001010010110111100101101001011110010100101101
0110010110100101111101010010110101100101101001011111010100101101
0110010110100101111101010010110101100101101001011111010100101101
0110010110100101011001010010110001100101101001010110010100101100
0110010110100101011001010010110001100101101001010110010100101100
1111110110100101111001011110110111111101101001011110010111101101
1111110110100101111001011110110111111101101001011110010111101101
0110010010100101001001010010110101100100101001010010010100101101
0110010010100101001001010010110101100100101001010010010100101101
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
```

Figure 10: Zoom In Algorithm Output Matrix

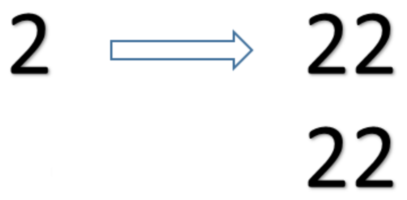


Figure 11: Replication Logic Illustration

2.5.3 Binary Conversion

The binary image algorithm outputs the input image as a black & white image which only contains two pixel values, either 0 or 255. The pixel value 0 refers to black and the pixel value 1 refers to white. In order to convert an RGB image into binary image we firstly need to convert it into a grayscale image. After that we need to set a threshold level and based on that, we need to convert our image into binary. The threshold which we set in our project is 127. If the grayscale pixel value is greater than 127, then we will set that pixel value to 1 in our output image, otherwise, to 0. The block diagram in **figure 12** describes how the binary Image conversion algorithm works.

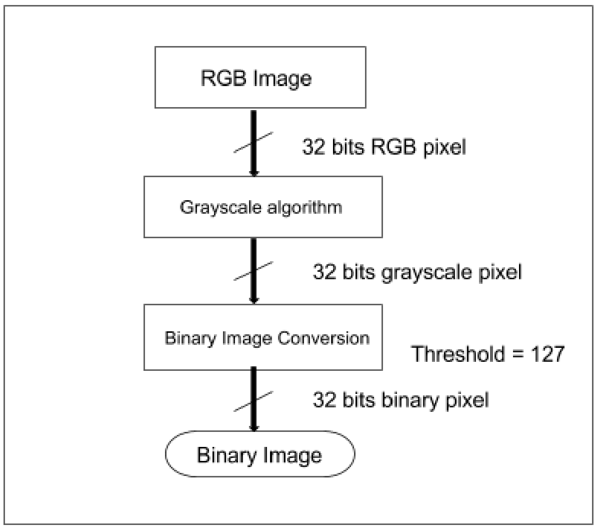
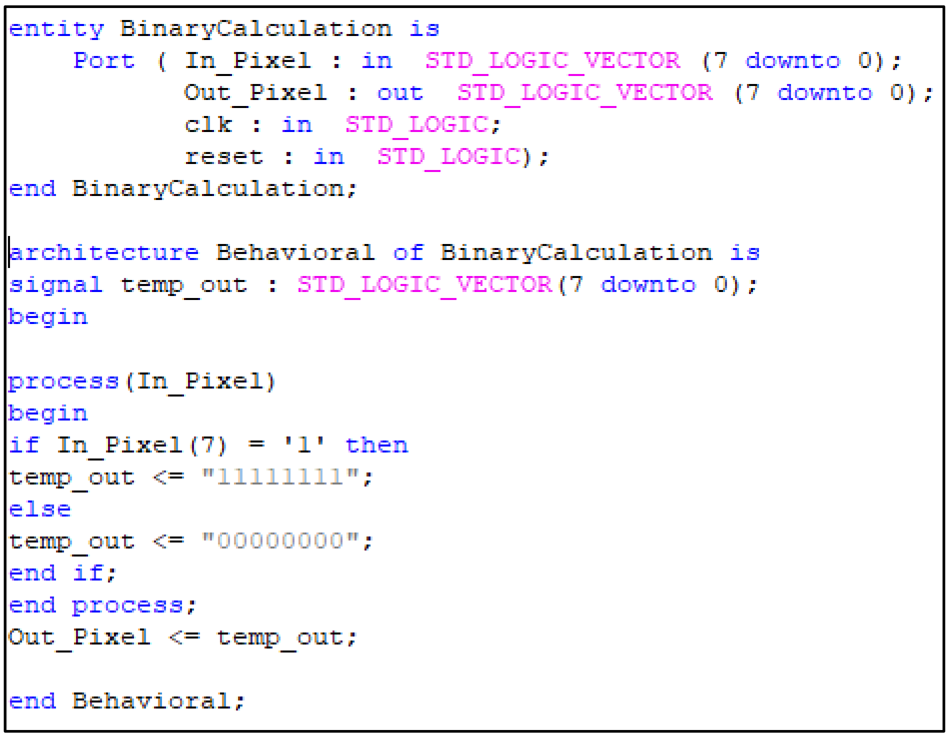


Figure 12: Binary Image Algorithm Block Diagram

The code logic for binary image conversion algorithm is shown below in **figure 13**. Some sample inputs along with their simulation results for binary image conversion algorithm are shown in **figure 14.**



```
entity BinaryCalculation is
    Port ( In_Pixel : in  STD_LOGIC_VECTOR (7 downto 0);
           Out_Pixel : out  STD_LOGIC_VECTOR (7 downto 0);
           clk : in  STD_LOGIC;
           reset : in  STD_LOGIC);
end BinaryCalculation;

architecture Behavioral of BinaryCalculation is
signal temp_out : STD_LOGIC_VECTOR(7 downto 0);
begin

process(In_Pixel)
begin
if In_Pixel(7) = '1' then
temp_out <= "11111111";
else
temp_out <= "00000000";
end if;
end process;
Out_Pixel <= temp_out;

end Behavioral;
```

Figure 13: Binary Image Code Snippet

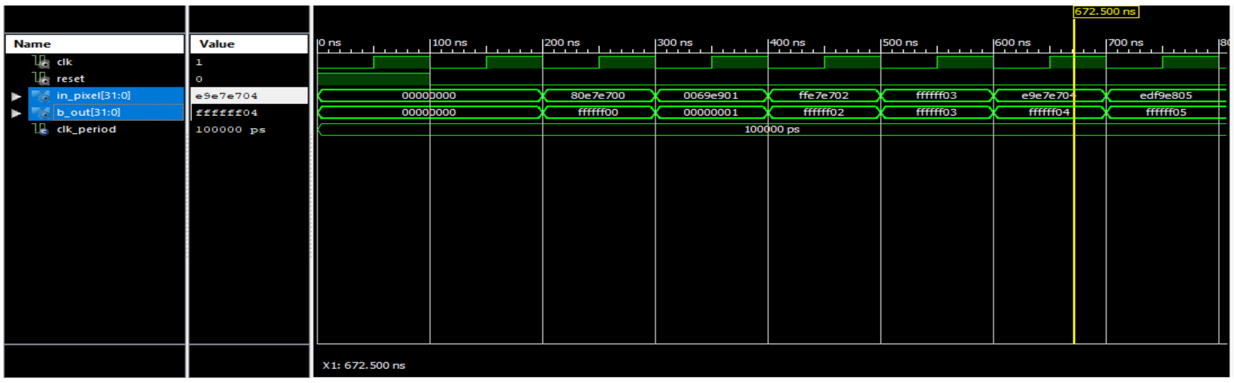


Figure 14: Binary Image Algorithm Simulation Result

In the above **figure 14** it can be seen that the second input is “80e7e700” which is a 32 bits RGB pixel value. If we convert this pixel into a grayscale pixel using 80 + e7 + e7 / 3 , we will get the answer “196” or C4 in Hex which is greater than 127. Therefore, we set the output pixel as “ffffff00” in the three channel RGB. The last two “00” represent row number in this case.

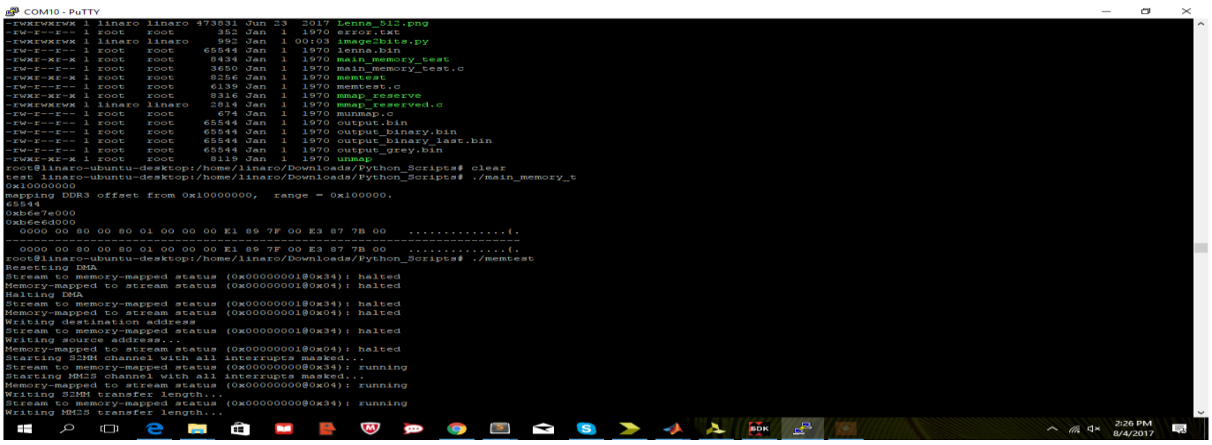
2.6 Send Image Back to the APP

The final results are converted back to a PNG image format and sent back to the APP via the FTP protocol. The processed results after applying the Grayscale algorithm are stored back to physical memory. Then we read the results and write it as a .bin file. As a next step we convert the matrix back to decimal RGB matrix, then to numpy.array which we can later convert back to the actual processed image.

3. Results

We have captured some snapshots during the whole process of writing the data to memory, read the data from memory, process data and write back to memory again.The following figures show the details of the overall process along with the final output image. Furthermore, we also made a video to show this process which can be found in our project files.

Figure 15 illustrates firstly how we used the script “main_memory_test” to map the Lenna_128.bin file, which is our input image, into the memory. Afterwards the script “memtest” was run to store the image into the memory, then read it, process it and send it back to the same destination addresses using DMA. Also, the processed data is saved in the output.bin file.

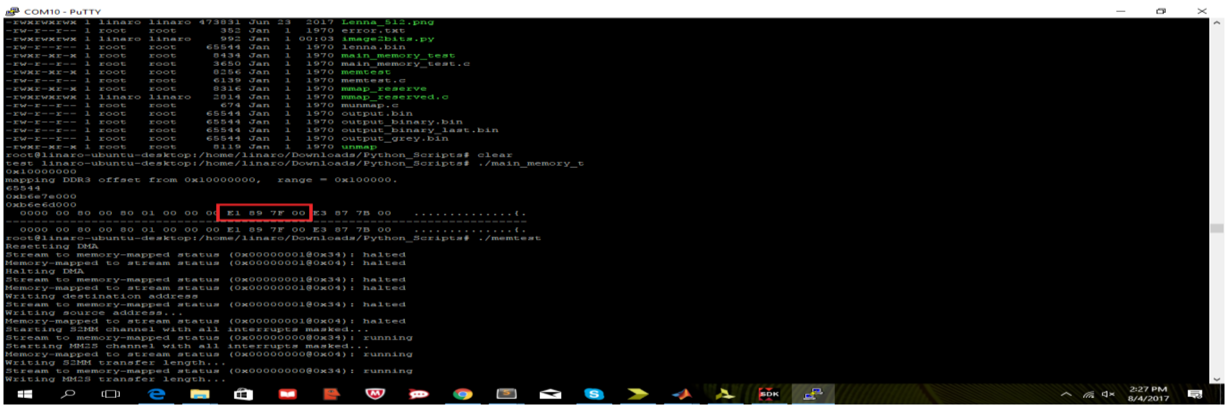


```
-rwxrwxrwx 1 linaro linaro 473831 Jun 23  2017 Lenna_512.png
-rw-r--r-- 1 root   root      352 Jan  1  1970 error.txt
-rwxrwxrwx 1 linaro linaro    992 Jan  1 00:03 image2bits.py
-rw-r--r-- 1 root   root    65544 Jan  1  1970 lenna.bin
-rwxr-xr-x 1 root   root     8434 Jan  1  1970 main_memory_test
-rw-r--r-- 1 root   root     3650 Jan  1  1970 main_memory_test.c
-rwxr-xr-x 1 root   root     8256 Jan  1  1970 memtest
-rw-r--r-- 1 root   root     6139 Jan  1  1970 memtest.c
-rwxr-xr-x 1 root   root     8316 Jan  1  1970 mmap_reserve
-rwxrwxrwx 1 linaro linaro   2814 Jan  1  1970 mmap_reserved.c
-rw-r--r-- 1 root   root      674 Jan  1  1970 munmap.c
-rw-r--r-- 1 root   root    65544 Jan  1  1970 output.bin
-rw-r--r-- 1 root   root    65544 Jan  1  1970 output_binary.bin
-rw-r--r-- 1 root   root    65544 Jan  1  1970 output_binary_last.bin
-rw-r--r-- 1 root   root    65544 Jan  1  1970 output_grey.bin
-rwxr-xr-x 1 root   root     8119 Jan  1  1970 unmap
root@linaro-ubuntu-desktop:/home/linaro/Downloads/Python_Scripts# clear
test linaro-ubuntu-desktop:/home/linaro/Downloads/Python_Scripts# ./main_memory_t
0x10000000
mapping DDR3 offset from 0x10000000,  range = 0x100000.
65544
0xb6e7e000
0xb6e6d000
  0000 00 80 00 80 01 00 00 00 E1 89 7F 00 E3 87 7B 00   ...............(.
-------------------------------------------------------------------------------
  0000 00 80 00 80 01 00 00 00 E1 89 7F 00 E3 87 7B 00   ...............(.
root@linaro-ubuntu-desktop:/home/linaro/Downloads/Python_Scripts# ./memtest
Resetting DMA
Stream to memory-mapped status (0x00000001@0x34): halted
Memory-mapped to stream status (0x00000001@0x04): halted
Halting DMA
Stream to memory-mapped status (0x00000001@0x34): halted
Memory-mapped to stream status (0x00000001@0x04): halted
Writing destination address
Stream to memory-mapped status (0x00000001@0x34): halted
Writing source address...
Memory-mapped to stream status (0x00000001@0x04): halted
Starting S2MM channel with all interrupts masked...
Stream to memory-mapped status (0x00000000@0x34): running
Starting MM2S channel with all interrupts masked...
Memory-mapped to stream status (0x00000000@0x04): running
Writing S2MM transfer length...
Stream to memory-mapped status (0x00000000@0x34): running
Writing MM2S transfer length...
```

Figure 15: Processing of Source Image

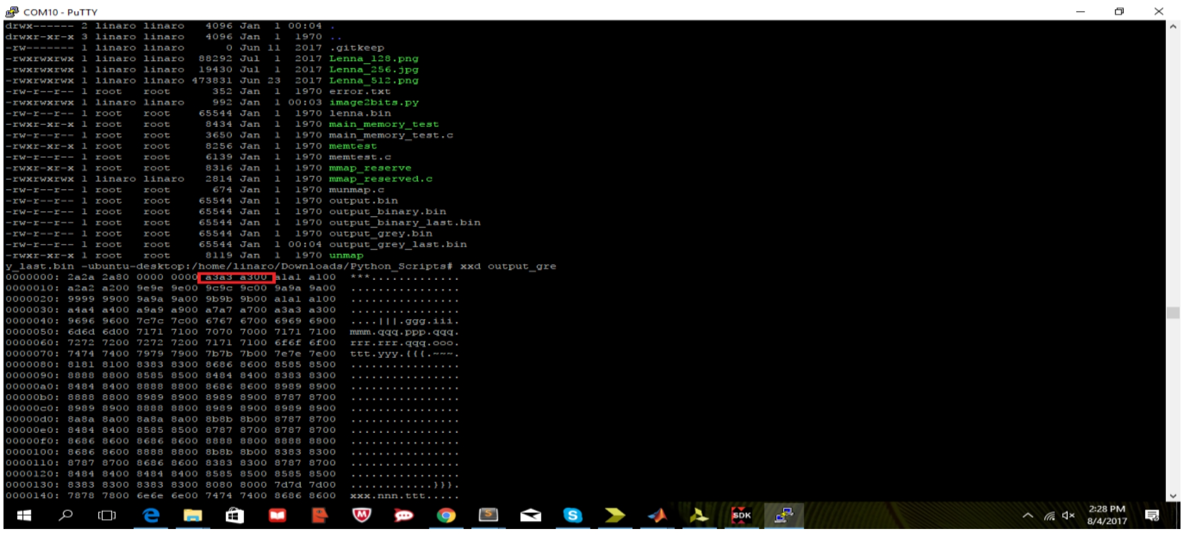
Figure 16 shows that the source image pixel "E1 89 7f 00" which is stored in memory will be taken as a 32 bits input to AXI DMA, which will then be sent to Image Editing Toolbox IP core for further processing. It will process the data and store the results back to the same destination address later.

Figure 17 shows the output results after applying Grayscale algorithm on the source image, which is stored in memory. After processing, the input pixel value "E1 89 7f 00" produces the results " A3 A3 A3 00", which is the average of "E1 + 89 + 7f /3 " while the last 00 depicts the row number.



```
-rwxrwxrwx 1 linaro linaro 473831 Jun 23  2017 Lenna_512.png
-rw-r--r-- 1 root   root      352 Jan  1  1970 error.txt
-rwxrwxrwx 1 linaro linaro    992 Jan  1 00:03 image2bits.py
-rw-r--r-- 1 root   root    65544 Jan  1  1970 lenna.bin
-rwxr-xr-x 1 root   root     8434 Jan  1  1970 main_memory_test
-rw-r--r-- 1 root   root     3650 Jan  1  1970 main_memory_test.c
-rwxr-xr-x 1 root   root     8256 Jan  1  1970 memtest
-rw-r--r-- 1 root   root     6139 Jan  1  1970 memtest.c
-rwxr-xr-x 1 root   root     8316 Jan  1  1970 mmap_reserve
-rwxrwxrwx 1 linaro linaro   2814 Jan  1  1970 mmap_reserved.c
-rw-r--r-- 1 root   root      674 Jan  1  1970 munmap.c
-rw-r--r-- 1 root   root    65544 Jan  1  1970 output.bin
-rw-r--r-- 1 root   root    65544 Jan  1  1970 output_binary.bin
-rw-r--r-- 1 root   root    65544 Jan  1  1970 output_binary_last.bin
-rw-r--r-- 1 root   root    65544 Jan  1  1970 output_grey.bin
-rwxr-xr-x 1 root   root     8119 Jan  1  1970 unmap
root@linaro-ubuntu-desktop:/home/linaro/Downloads/Python_Scripts# clear
test linaro-ubuntu-desktop:/home/linaro/Downloads/Python_Scripts# ./main_memory_t
0x10000000
mapping DDR3 offset from 0x10000000,  range = 0x100000.
65544
0xb6e7e000
0xb6e6d000
  0000 00 80 00 80 01 00 00 00 E1 89 7F 00 E3 87 7B 00   ..............{.
-----------------------------------------------------------------------------
  0000 00 80 00 80 01 00 00 00 E1 89 7F 00 E3 87 7B 00   ..............{.
root@linaro-ubuntu-desktop:/home/linaro/Downloads/Python_Scripts# ./memtest
Resetting DMA
Stream to memory-mapped status (0x00000001@0x34): halted
Memory-mapped to stream status (0x00000001@0x04): halted
Halting DMA
Stream to memory-mapped status (0x00000001@0x34): halted
Memory-mapped to stream status (0x00000001@0x04): halted
Writing destination address
Stream to memory-mapped status (0x00000001@0x34): halted
Writing source address...
Memory-mapped to stream status (0x00000001@0x04): halted
Starting S2MM channel with all interrupts masked...
Stream to memory-mapped status (0x00000000@0x34): running
Starting MM2S channel with all interrupts masked...
Memory-mapped to stream status (0x00000000@0x04): running
Writing S2MM transfer length...
Stream to memory-mapped status (0x00000000@0x34): running
Writing MM2S transfer length...
```

Figure 16: Source Image Data Stored in Memory



```
drwx------ 2 linaro linaro   4096 Jan  1 00:04 .
drwxr-xr-x 3 linaro linaro   4096 Jan  1  1970 ..
-rw------- 1 linaro linaro      0 Jun 11  2017 .gitkeep
-rwxrwxrwx 1 linaro linaro  88292 Jul  1  2017 Lenna_128.png
-rwxrwxrwx 1 linaro linaro  19430 Jul  1  2017 Lenna_256.jpg
-rwxrwxrwx 1 linaro linaro 473831 Jun 23  2017 Lenna_512.png
-rw-r--r-- 1 root   root      352 Jan  1  1970 error.txt
-rwxrwxrwx 1 linaro linaro    992 Jan  1 00:03 image2bits.py
-rw-r--r-- 1 root   root    65544 Jan  1  1970 lenna.bin
-rwxr-xr-x 1 root   root     8434 Jan  1  1970 main_memory_test
-rw-r--r-- 1 root   root     3650 Jan  1  1970 main_memory_test.c
-rwxr-xr-x 1 root   root     8256 Jan  1  1970 memtest
-rw-r--r-- 1 root   root     6139 Jan  1  1970 memtest.c
-rwxr-xr-x 1 root   root     8316 Jan  1  1970 mmap_reserve
-rwxrwxrwx 1 linaro linaro   2814 Jan  1  1970 mmap_reserved.c
-rw-r--r-- 1 root   root      674 Jan  1  1970 munmap.c
-rw-r--r-- 1 root   root    65544 Jan  1  1970 output.bin
-rw-r--r-- 1 root   root    65544 Jan  1  1970 output_binary.bin
-rw-r--r-- 1 root   root    65544 Jan  1  1970 output_binary_last.bin
-rw-r--r-- 1 root   root    65544 Jan  1  1970 output_grey.bin
-rw-r--r-- 1 root   root    65544 Jan  1 00:04 output_grey_last.bin
-rwxr-xr-x 1 root   root     8119 Jan  1  1970 unmap
y_last.bin -ubuntu-desktop:/home/linaro/Downloads/Python_Scripts# xxd output_gre
0000000: 2a2a 2a80 0000 0000 a3a3 a300 a1a1 a100  ***.............
0000010: a2a2 a200 9e9e 9e00 9c9c 9c00 9a9a 9a00  ................
0000020: 9999 9900 9a9a 9a00 9b9b 9b00 a1a1 a100  ................
0000030: a4a4 a400 a9a9 a900 a7a7 a700 a3a3 a300  ................
0000040: 9696 9600 7c7c 7c00 6767 6700 6969 6900  ....|||.ggg.iii.
0000050: 6d6d 6d00 7171 7100 7070 7000 7171 7100  mmm.qqq.ppp.qqq.
0000060: 7272 7200 7272 7200 7171 7100 6f6f 6f00  rrr.rrr.qqq.ooo.
0000070: 7474 7400 7979 7900 7b7b 7b00 7e7e 7e00  ttt.yyy.{{{.~~~.
0000080: 8181 8100 8383 8300 8686 8600 8585 8500  ................
0000090: 8888 8800 8585 8500 8484 8400 8383 8300  ................
00000a0: 8484 8400 8888 8800 8686 8600 8989 8900  ................
00000b0: 8888 8800 8989 8900 8989 8900 8787 8700  ................
00000c0: 8989 8900 8888 8800 8989 8900 8989 8900  ................
00000d0: 8a8a 8a00 8a8a 8a00 8b8b 8b00 8787 8700  ................
00000e0: 8484 8400 8585 8500 8787 8700 8787 8700  ................
00000f0: 8686 8600 8686 8600 8888 8800 8888 8800  ................
0000100: 8686 8600 8888 8800 8b8b 8b00 8383 8300  ................
0000110: 8787 8700 8686 8600 8383 8300 8787 8700  ................
0000120: 8484 8400 8484 8400 8585 8500 8585 8500  ................
0000130: 8383 8300 8383 8300 8080 8000 7d7d 7d00  .............}}}.
0000140: 7878 7800 6e6e 6e00 7474 7400 8686 8600  xxx.nnn.ttt.....
```

Figure 17: Processed Results Stored in Memory

Figure 18 shows the final output image we got after applying the Grayscale algorithm on the input source image. The **table 1** below describes the procedure that has been taken along with their timing which they took for execution on zedboard.

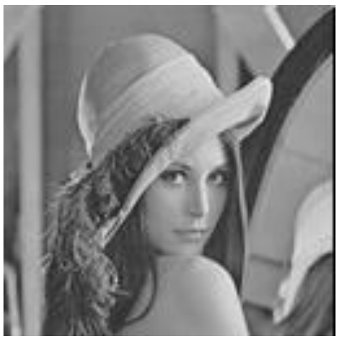


Figure 18: Grayscale Final Result Image

Sr.No	**Process / Task**	**Timing**
1	Sending File from Android to Linux (PS of Zedboard)	2 seconds
2	Parsing the PNG file into .bin file with 128 * 128 image resolution	2 seconds
3	Mapping the bin file into physical memory using memmap	2 seconds
4	Reading and writing to Memory using DMA Taking Data from memory, process it and store back to Memory. Also store in output.bin file (128 * 128 image resolution , Clock Frequency 50 MHz)	2 seconds
5	Writing back the output.bin file into PNG	Approx 2 seconds

Table 1 : Process Execution Timing on Zedboard

Reference

XML-RPC:
https://en.wikipedia.org/wiki/XML-RPC

FTP:
https://en.wikipedia.org/wiki/File_Transfer_Protocol

Mmap:
https://en.wikipedia.org/wiki/Mmap

Grayscale:
https://en.wikipedia.org/wiki/Grayscale

Linaro Setup:
https://www.youtube.com/watch?v=IH5vk8N8bl0&t=1077s

AXI DMA:
http://www.fpgadeveloper.com/2014/08/using-the-axi-dma-in-vivado.html