University of California, Davis
Department of Electrical and Computer Engineering

EEC180B DIGITAL SYSTEMS Winter Quarter 2017

LAB 6: Image Processing

1 Introduction

Image Processing is a common application domain for FPGAs. The advantages of FPGAs over something like a microcontroller are many fold:

- By tailoring hardware to the task performed, an FPGA can produce a result in fewer clock cycles and a microcontroller.
- Functional units in FPGAs can be replicated to exploit parallelism in the underlying task.
- Tasks can be pipelined efficiently, which is something a microcontroller cannot do.
- FPGAs can guarantee cycle accurate timing for producing results. Even an interrupt driven micro controller cannot provide this level of timing precision. This is invaluable for real-time applications.

In this lab, you will explore the first two options. First, you will design custom hardware to apply a range of simple Instagram-like filters to images using an FPGA. Second, you design a functional unit that can be used in a SIMD[1] type style to compute the pixel value histogram of an image quickly.

2 Task 1 - Establishing Data Transfer

For this task, you will use the provided Verilog modules and MATLAB scripts to transfer image data from the computer to the FPGA and back. In this lab we will be using the serial port for our communication needs. Although the serial port is very slow, it is simple to use.

To do this task, you will need the files `uart_handler.v` and `uart.v`. The file `uart_handler.v` defines a module **uart_handler** which with `uart.v` handles all of your communication needs. The port-list for **uart_handler** is shown in Figure 1 and detailed descriptions of the ports are provided in Table 1. Then general procedure for how to use this module is as follows:

1. Upon reset, **uart_handler** enters a waiting state. As data is received on the UART port, it is written into a RAM module. In this state, the inputs *addr, writeEnable* and *dataIn* are ineffective that the output *dataOut* is not valid.

[1]SIMD - Single Instruction Multiple Data

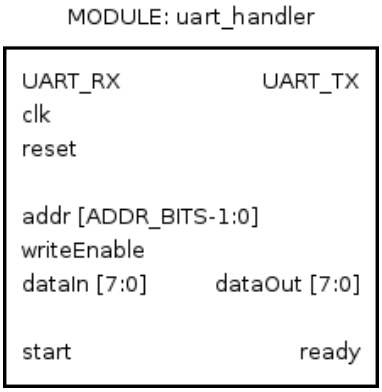

Figure 1: Block depiction for the uart_handler module. Input ports are on the left, output ports are on the right.

2. Once the RAM is full, **uart_handler** enters an idle state the output signal *ready* is set to 1. In this state, the inputs *addr, writeEnable* and *dataIn* are used to access the RAM module inside **uart_handler** and the output *dataOut* is the valid output from this RAM module.

   The contents of the RAM are the pixel data for an image. Any image processing should happen in this state.

3. Once processing is complete, the *start* signal should be set to 1. When this happens, **uart_handler** assumes control of the RAM module and sends the contents of the RAM to the computer. Once this transaction is complete, **uart_handler** is ready to receive more data from the computer.

Note that the number of entries in the RAM module inside **uart_handler** is given by the parameter RAM_SIZE. This allows you to adjust the module for different image sizes.

2.1 Create a Module to Echo Data from the Serial Port

Plug in the power and UBS Blaster cables into your DE2 board. Next, get a serial cable from your TA. Attach one end of the cable to the serial port at the back of your computer and attach the other end to the connector titled RS-232 on the DE2 board.

Create a new project in Quartus and create a new top level module. Include the files `uart_handler.v` and `uart.v`, and instantiate the module **uart_handler** in your top level module. Connect the signals *UART_TX, UART_RX, clk*, and *reset* to the connections given in Table 1.

Next, set the parameter RAM_SIZE = 28 and ADDR_BITS = 5.[2] Assign zeros to *addr, writeEnable*, and *dataIn*, and leave *dataOut* unconnected. Finally, connect the output *ready* to the input *start*.

In this configuration, *uart_handler* will immediately transmit any data it received unmodified back through the serial port. Synthesize the design and program the board. Leave the development board powered and connected.

[2]This is because we need 5 address bits to completely address a RAM of size 28.

2.2 Sending and Receiving Data using MATLAB

Open up MATLAB on the computer and set the working directory to the provided folder `lab6_matlab`. Open up the script `serialtest.m` and have a look around. There are two types of tests this script can perform, controlled by the variable *testType*.

If *testType* = 0, the script will pull a random image from the MNIST handwriting image database and transmit that image to the FPGA. Each of the images in the MNIST dataset are 28x28 pixels. The image is sent one column at a time (which is why we set the RAM_SIZE parameter to 28).

If *testType* = 1, the script will send the image pointed to by the path *imgName*. You can see all the available images in the folder `standard_test_images`. Each of the images in this folder is 512x512 pixels. When you do a test on these images, make sure you set the RAM_SIZE parameter to 512 (the size of a column of the image) and ADDR_BITS = 9.

For now, make sure *testType* = 0. Run the script. Image data should be sent one column at a time through the serial port to the FPGA and the FPGA should send the data straight back to MATLAB. If all goes well, a figure should show up in MATLAB with identical images titles "Image Sent to FPGA" and "Image Received from FPGA".

2.3 Check-off

Demonstrate the working module to your TA on the 28x28 handwriting images.

Port	Type	Bits	Description
RAM_SIZE	Paramater		Indicates how many bytes to allocate for the RAM module.
ADDR_BITS	Parameter		Number of bits to use for the address. Must be $\geq \log_2(\text{RAM\_SIZE})$.
UART_RX	Input	1	UART received data. **Top Level Connection:** UART_RXD.
UART_TX	Output	1	UART transmit data. **Top Level Connection:** UART_TXD.
clk	Input	1	Clock. **Top Level Connection:** CLOCK_50
reset	Input	1	Reset. **Top Level Connection:** KEY[0]
addr	Input	variable	Address for the RAM module inside uart_handler. Only used when *ready* is asserted.
writeEnable	Input	1	Write Enable for the RAM module inside uart_handler. Only used when *ready* is asserted.
dataIn	Input	8	Data to be written to the RAM module. RAM will only be written to when both *ready* and *writeEnable* are asserted.
dataOut	Output	8	Data from the RAM module. Only valid when *ready* is asserted. The RAM module inside **uart_handler** has a synchronous output.
start	Input	1	Indicates to uart_handler that data manipulation is complete. Initiates data transfer back to computer.
ready	Output	1	Indicates that data transfer from computer is complete and the RAM module is available for external use.

Table 1: Port list and description for the uart_handler module. The first two parameters control the attributes of the RAM module inside uart_handler. The next four signals are global signals that need to be connected to the FPGA pin listed. The following four signals are the interface to the RAM module inside uart_handler. The final two signals are control signals that indicate when to transmit data and when uart_handler is done receiving data.

Filter Name	Pixel Operation
Tint	$f(x) = \max\{x - 64, 0\}$
Invert	$f(x) = 255 - x$
Threshold	$f(x) = \begin{cases} 255 & \text{if } x > 127 \\ 0 & \text{otherwise} \end{cases}$
Increase Contrast	$f(x) = \begin{cases} x/2 & \text{if } x < 85 \\ 2x - 128 & \text{if } 85 \leq x < 171 \\ x/2 + 128 & \text{otherwise} \end{cases}$

Table 2: Mathematical operations for each filter. The variable x represents a single pixel in a gray-scale image and is an integer between 0 and 255 inclusive.

3 Task 2 - Simple Image Processor

Now that you can send data to and from the FPGA, it's time to do some simple image processing. First, it's important to know how images are stored. All the images used in this lab will be gray-scale images. Each image is made of pixels. Each pixel is an 8-bit unsigned integer. A pixel value of 0 represents pure black and pixel value of 255 is pure white. Pixel values between these extremes represent 254 shades of gray. When retrieving data from the RAM module in **uart_handler**, each byte represents a pixel in the original image.

An image with a height and width of 512 pixels requires $512 \cdot 512 = 262,144$ bytes of data, which is $2,097,152$ bits of storage. This far exceeds the block RAM capacity on the Cyclone II FPGAs included on DE2 board. Thus, images will be sent and processed one column of pixels at a time.

Four common filtering operations are described below. Each of these operations is performed on every pixel in the image. The mathematical function for each operation is shown in Table 2. You can see the effects of each of the filters in Figure 2.

3.1 Filter Descriptions

3.1.1 Tint

Tinting an image simply makes the image appear darker. It is accomplished by subtracting a set number from each pixel value and clamping the result to zero.

3.1.2 Invert

Color inverting an image essentially reflects a pixel value across the middle point in its scale. That is, it converts black pixels to white pixels and vice-versa.

3.1.3 Threshold

The threshold filter maps pixels whose value is greater than the midway point (value 127) to pure white (value 255) and pixels whose value is less than the midway point to pure black (value 0). The resulting image has a very sharp contrast.

(a) Original Image (b) Increased Contrast

(c) Tint (d) Invert (e) Threshold

Figure 2: Example of the filters applied to "cameraman", a common image used in signal processing.

3.1.4 Contrast Enhancement

In contrast enhancement, pixel values are also moved towards pure black or pure white, but in a more gradual fashion. Any monotonic function will alter the contrast of an image. For this lab, you will implement the piecewise linear function given in Table 2.

3.2 Task Description

You will design and implement a module that can perform these four operations, with the exact operation to performed being determined by a two bit signal *mode*. You module should interface with the **uart_handler** module used in the previous task. A description of the expected behavior is as follows:

1. Upon assertion of a reset signal, you module will enter an IDLE state. The processing module should stay in this state until **uart_handler** asserts its *ready* signal.

2. Once **uart_handler**'s *ready* signal is set to 1, you module must apply the operation specified by *mode* to each entry of the RAM module inside **uart_handler**. Use the following convention:

- If *mode* = 2'b00: Apply the Tint function
- If *mode* = 2'b01: Apply the Invert function
- If *mode* = 2'b10: Apply the Threshold function
- If *mode* = 2'b11: Apply the Increase Contrast function

You must manipulate the RAM contents using only the available input and output ports of **uart_handler**. You should read the contents at each address, apply the specified function, and write back the results to the same address.

3. Once the specified filtering operation has been applied to each entry in **uart_handler**'s RAM, your module should send the *start* signal to **uart_handler** to indicate that it should transmit the filtered data back to the computer. Following this, your module should then return to it's idle state to await more data.

The minimal input and output signals for your top level module are given in Table 3. To help you with your debugging, a test bench `tb_lab6.v` has been included. This testbench is designed to work on the top level module and conforms to the signal ports listed in Table 3. Feel free to modify the testbench as needed.

Port	Type	Bits	Description
CLOCK_50	Input	1	50 MHz clock.
KEY[0]	Input	1	Map this key to reset. You may have to invert it.
SW[1:0]	Input	1	Connect this to the *mode* signal in your module. This will allow you to choose the filtering operation.
UART_RXD	Input	1	Serial data from computer. Connect to UART_RX in **uart_handler**
UART_TXD	Output	1	Serial data from FPGA. Connect to UART_TX in **uart_handler**

Table 3: Minimal requirements for the top level of your design. You may add more output signals if you wish.

3.3 Checkoff Requirements

For the checkoff for this task, you will be required to demonstrate to your TA that your module successfully filters one of the large 512x512 images sent by the MATLAB script. Your TA will choose one or more of the four filtering operations at random. Remember to adjust the RAM_SIZE parameter to 512 and ADDR_BITS to 9 when working with the larger images.

3.4 Workflow Tips

- As noted in Table 1, the RAM module inside **uart_handler** has a synchronous output. This means that output data is available after one clock cycle. Take this into account when designing your control logic.
- You probably want to design your filtering units in such a way that they all take the same number of clock cycles per pixel. This will simplify your control logic.
- It is **HIGHLY** recommended that you parametrize your module according to the RAM_SIZE and ADDR_BITS parameters. This will help immensely in debugging.
- When debugging your design using the testbench, set RAM_SIZE to 8 and ADDR_BITS to 3. This allows your design to work on a small enough dataset that it can be easily verified by hand.
- When you've verified the functionality of your module using the testbench, test your design on the board using the small 28x28 handwriting images from the MATLAB script (RAM_SIZE = 28, ADDR_BITS = 5). A full transmission of the large 512x512 images takes about 45 seconds to complete using the (very slow) serial port.[3] Debugging first on the smaller images will save a lot of time.
- If you parametrized your module correctly and verified your design using the testbench and the small images, your module should work just fine with the larger images.

[3]The maximum bit rate of a typical serial port is 115200 bits per second.

4 Task 3 - Image Histogram Processor

4.1 Introduction

Image histograms are widely used to graphically show the pixel intensity distribution of an image. To construct an image histogram, the pixel value range is broken up into a number of (usually equal sized) ranges or bins. Each pixel value is then examined, the bin to which that pixel belongs is calculated, and a counter in that range is incremented. The final output is a count of the number of pixels whose numerical value falls within each bin. For the images we are using, the range of pixel values is from 0 to 255. The pseudo code to compute the histogram of an image into four equal sized bins is as follows:

```
function compute_histogram(image)
  hist = [0,0,0,0]
  for pixel in image
    if 0 <= pixel < 64
      hist[1] += 1
    elseif 64 <= pixel < 128
      hist[2] += 1
    elseif 128 <= pixel < 192
      hist[3] += 1
    else
      hist[4] += 1
    end
  end
  return hist
end
```

You may consult the included MATLAB script `histogramtest.m` and online resources for more information about histograms.

4.2 Exposing Parallelism

Histogram computation is a problem that is easily parallelized. This is done by breaking the image into multiple pieces, computing a miniature histogram on those pieces in parallel, and then summing the miniature histograms together to produce the final histogram.

In order for this to be effective, multiple pieces of the image need to be accessed simultaneously. This may be realized by increasing the width of the RAM module used to store the image. In Tasks 1 and 2, the buffering RAM module had a width of 8 bits which meant that each address stored one pixel. By increasing this width to 32 bits, four pixels may be stored at a single address and may be accessed simultaneously and processed in parallel. This style of architecture where a single operation occurs on multiple data is called SIMD, Single Instruction, Multiple Data. Most modern computer processors will have SIMD type instructions and modern GPUs are composed of many complicated SIMD-style computational units.

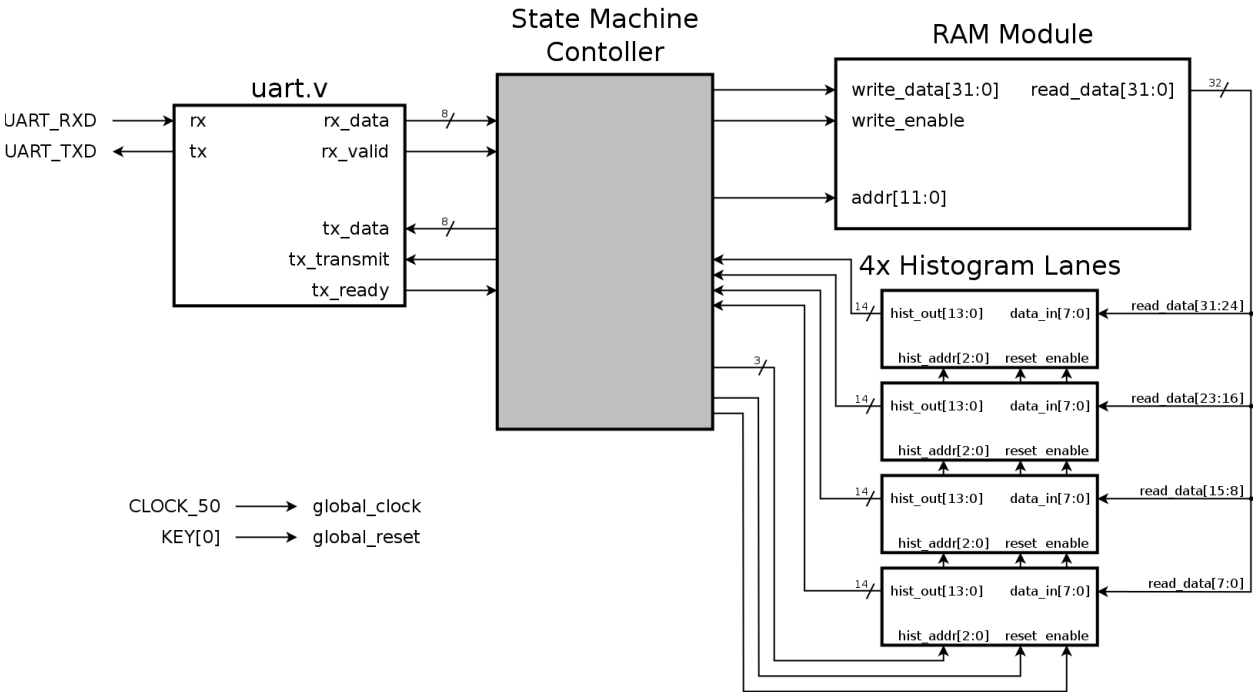

Figure 3: General block diagram for your final design.

4.3 Task Description

You will design a custom image histogram computation unit capable of reading and storing a full 128x128 image using the RS232 serial port, performing a parallel histogram computation on that image using SIMD style units, and transmitting a 8 bin histogram encoded in 16 bytes back through the serial port. You may use any of the provided Verilog source code in your design. Specifics are given below:

- Image data should be saved a RAM with a width of 32 bits. This means that each address in the RAM stores 4 pixels. The RAM should be large enough that it stores a full 128x128 image.[4] This implies that your state machine should temporarily buffer pixel data until it has four new pixels to store in RAM.

- Once the RAM has been populated with the whole 128x128 image, you module will perform a parallel histogram computation.[5] To do this, you must instantiate four histogram computational units and distribute the four pixels stored at a single memory address to the four units.

[4]The Cyclone II devices do have enough M4K blocks to do this.

[5]You may be wondering: "If each pixel is sent one at time through an absurdly lethargic serial port, why don't I just compute the histogram of the image as pixels get decoded by the UART?" To answer your question - that's a GREAT idea. However, (1) that's too easy and (2) we're only using the serial port as a means of getting data to and from the FPGA. In an embedded system where hardware acceleration is likely to be used, a host processor and accelerator may communicate through memory. That is, the image already exists in a block of memory that the host processor allows the accelerator to access. Since many processors have a 32-bit data memory, a 32-bit wide RAM is a natural choice.

Port	Bit Width	Functionality
reset	1	Clears all histogram counters in the module.
enable	1	Enables computation. Counters should not update if enable is not asserted.
data_in	8	Pixel value. If lane is enabled, the histogram counter corresponding to the bin to which the pixel belongs should be incremented by 1.
hist_addr	≥ 3	Address to retrieve histogram data. When 0, this module should send count value for histogram bin 0 to the output. Address width should be large enough to access all bins in the histogram.
hist_out	≥ 14	Histogram count for the bin associated *hist_addr*. Width should be large enough to handle the worst case scenario where all pixel values fall in the same bin.

Table 4: Suggested functionality for a single histogram computation lane.

This histogram computational unit should be implemented as module. The inputs, outputs, and functionality of the module are given in Table 4.

- Sweep through the entire RAM to compute the histogram for the full image.
- Once the entire image has been scanned, you will have to add together and the bin counts from the four computational units to get the final histogram of the image. You may assume that the incoming image is 128x128 pixels. Size your data path so that you can handle the case where all pixels in an image fall into the same bin.
- When the entire histogram has been calculated, it must be sent through the serial port where the data will be read and displayed by MATLAB. The MATLAB expects data to be sent in the following manner:
  - The final count for the bin corresponding the lowest pixel values (0 to 31) should be sent first. The count for the bin corresponding to the highest pixel values should be sent last.
  - Two bytes should be sent per count since count values will likely be larger than what can be stored in one byte. The least significant byte should be sent first and the most significant byte should be sent second.
  - In all, for 8 bins, the MATLAB script expects to receive 16 bytes of data.

Refer to Figure 3 for a general block diagram of your final design.

4.4 Checkoff Requirements

For checkoff, you will demonstrate to your TA that your design successfully computes the full histogram of a 128x128 image. To do this you must use the unmodified `histogram.m`

MATLAB script. Your TA will choose a few random images to test. For full points, the histogram of your module must match the histogram computed by MATLAB exactly.

4.5 Workflow Tips

1. As always, parametrize as much of your module as you can. At the very least, you should parametrize the depth (number of addresses) for the RAM. This will allow you to run testbenches using small data sets to verify the logic of your design before increasing the RAM size to the full amount needed to buffer a 128x128 image.

2. Modify the provided test bench and write your own to verify your designs at each stage. Definitely do this.

3. Consider how to efficiently combine the distributed histogram data at the end of the computation. With only four computational lanes, this simply amounts to adding four numbers together. However, if you decide to increase the number of parallel computational units, this has the potential to become a bottleneck in your design.

5 Lab Requirements

1. **Prelab** [30 points]

   (a) Assume you wanted to implement each filter on a simple microprocessor. Estimate how many simple assembly instructions (and thus roughly the number of clock cycles) each of the filters described in Table 2 would take per pixel. Using this, estimate the total number of clock cycles needed to process a 512x512 gray-scale image. Assuming a 50 MHz clock, how many seconds would each filter take.

   Record your answers in a table and describe how you got the number of instructions needed to execute each of the filtering operations.

   (b) Now, assume that each of the four filters in Table 2 can be done in two clock cycles using an FPGA. Assuming that we did not have the transmission limitation imposed by the serial port and again assuming a 50 MHz clock, how many seconds will each filter take? What is the speed up relative to the microprocessor?

2. **Task 1** [20 points]

   (a) Implement the simple echo circuit described above. The circuit should instantiate an instance of **uart_handler** and connect the *ready* output to the *start* input.

   (b) Demonstrate to your TA that your circuit can receive and transmit an image to and from the MATLAB script using the serial cable. Have your TA sign a verification sheet.

3. **Task 2** [50 points]

   (a) Design and implement a circuit to perform the four filtering operations described in Table 2. The filter to be applied should be selected using switches 0 and 1.

   (b) Interface your circuit with the MATLAB script `serialtest.m` and verify that your filters are working as expected.

   (c) Demonstrate your design to your TA by successfully filtering one of the large 512x512 images using a random filter of the TA's choice. Have them sign a verification sheet.

   (d) Record the resources used for your design (logic elements used, flip-flops, memory blocks etc.) as well as the maximum operating frequency reported by Quartus.

4. **Task 3** [200 points]

   (a) Design and implement the histogram computational unit.

   (b) Interface the design with the MATLAB script `histogramtest.m` and verify that the histogram calculated by your design matches that produced by MATLAB.

   (c) Demonstrate your design to your TA by computing the histogram of several 128x128 images. Have them sign a verification sheet.

   (d) Record the resources used and maximum operating frequency of your design.

5. **Lab Report:** Be sure to include all the Verilog source files and test benches you have written for this lab, including any testbenches you wrote to help verify your design. Be sure to include resource utilization numbers for Tasks 2 and 3.

   Also, include a couple of sample images from the MATLAB script that show the results of your filtering. The final result should be similar to Figure 2.

   As always, include a short summary of the problem, your approach, any ideas you have for increasing the efficiency of the design, and a brief conclusion.

6 Extra Credit

There are many opportunities for extra-credit on this lab. More points will be awarded for more complex options. Several ideas in order of increasing complexity are:

1. Research and implement a new image filter.

2. Restructure your filter architecture to allow the user to run multiple filters on an image.

3. Incorporate the four filters from Task 2 into your design from Task 3 to see how applying a filter to an image changes its histogram. For this step, you may choose to either filter each pixel after receiving it from the UART and before storing it in memory. More points, however, will be given for designs that operate on the image data in memory, especially if these designs incorporate the same SIMD style parallelism as the histogram computation.

4. Parametrize the number of bins in the histogram and the number of parallel computational units. Try to exploit as much parallelism as possible. The author of this lab was able to perform a 32-bin histogram using 32 parallel computational units, processing an entire 128x128 image in 512 clock cycles. See if you can beat this!

Feel free to propose your own extension!