CMPE-660 Laboratory Exercise 4

Image Processing

By submitting this report, I attest that its contents are wholly my individual writing about this exercise and that they reflect the submitted code. I further acknowledge that permitted collaboration for this exercise consists only of discussions of concepts with course staff and fellow students; however, other than code provided by the instructor for this exercise, all code was developed by me.

Chris Abajian
Performed 10/26/2020
Submitted 11/11/2020

Section 01S2
Professor: Marcin Lukowiak
TA: Connor Henley
Joshua Willson

Design Methodology

This laboratory exercise investigated image processing through median filters on an FPGA. An image stored in the portable graymap (PGM) format consists of three header lines specifying the image type (P2 for this example), the number of columns and rows, respectively, and the maximum pixel size. The pixel values can range from [0, 255] for this example. Using modules from previous exercises, PGM images can be sent to an FPGA over UART, stored, and transmitted back over UART. The existing top-level UART module needs modifications to include this functionality.

The first addition to the top-level module was the inclusion of two state machines: one for reading input image data and storing it in memory, and another for reading an image from memory and transmitting it back. Figure 1 shows the state diagram for the former process.

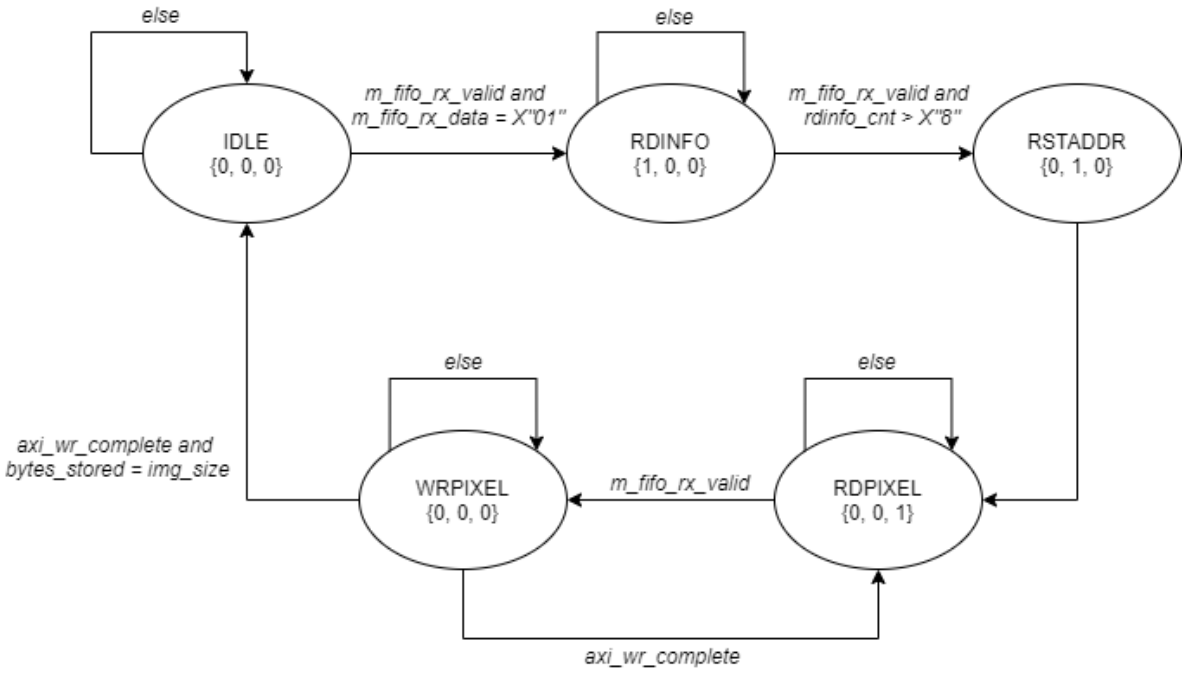

Figure 1: Data Mover write state diagram. Output key: {`rdinfo_ready, axi_rst, m_axis_valid`}.

As seen in Figure 1, specific commands are first sent over UART identifying the operation to be performed. `X"01"` indicates image data should be written to memory whereas `X"02"` indicates image data should be read from memory and transmitted back. Figure 2 shows the state diagram for the read process.

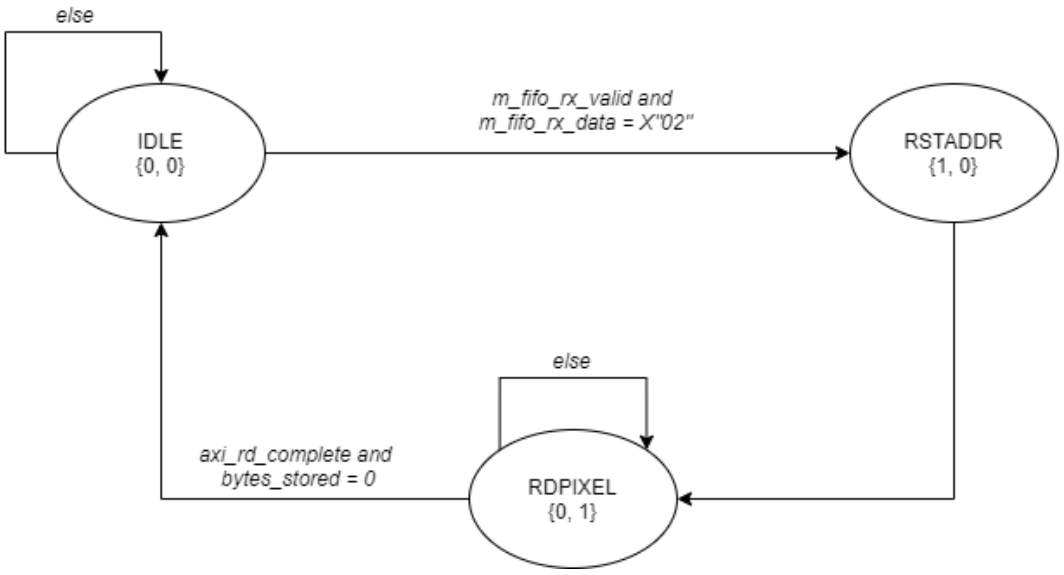

Figure 2: Data Mover read state diagram. Output key: {`axi_rst, rd_en`}.

To decouple these image moving operations from the already existing AXI Stream to AXI Lite conversion, a separate module was created to interface between the top-level FIFO's and the memory. This module operates on a write-enable signal, `m_axis_valid`, and a read-enable signal, `rd_en`. This greatly simplified the image moving state machines.

Another requirement of this exercise was to implement an image filter operation designated by the command `X"04"`. This operation would read the image stored in memory and apply a median filter. This filtered image is then stored back in memory at a separate memory bank. This filtered image can then be requested back by the control system using the command `X"03"`. The state diagram for this process is shown in Figure 3.

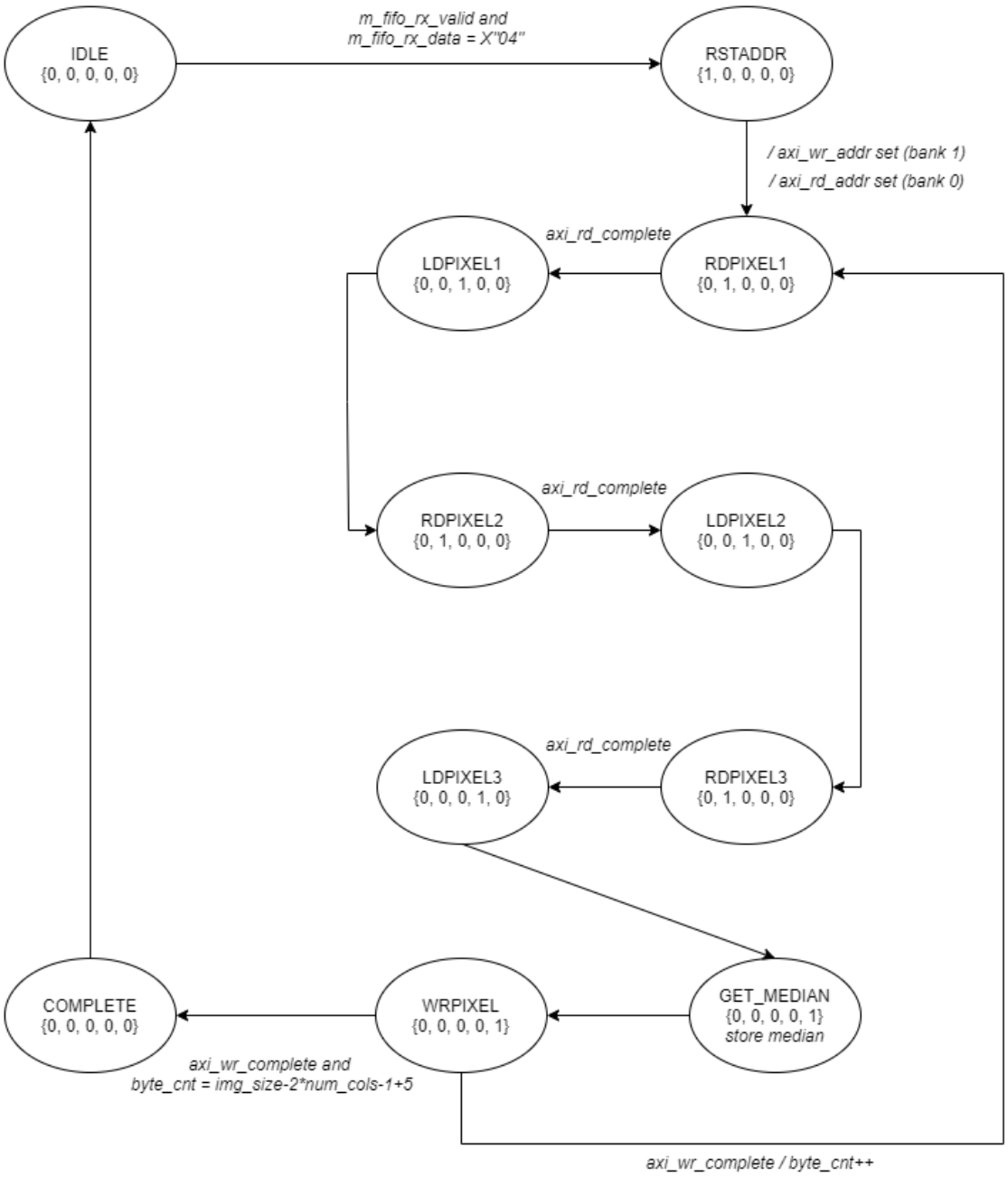

Figure 3: Data Mover filter state diagram. Output key: {`axi_rst, rd_en, en1, en2, m_axis_valid`}.

The median filter operates by individually loading two pixel values using a pre-load enable signal, `en1`. Once two values are loaded, the final pixel value and the two stored pixels are loaded into the filter for processing using the `en2` enable signal. Due to the combinational logic required for the median filter to sort the pixels, a pipeline was implemented to continuously process a three-by-three square of pixels. Figure 4 shows the median filter pipeline.

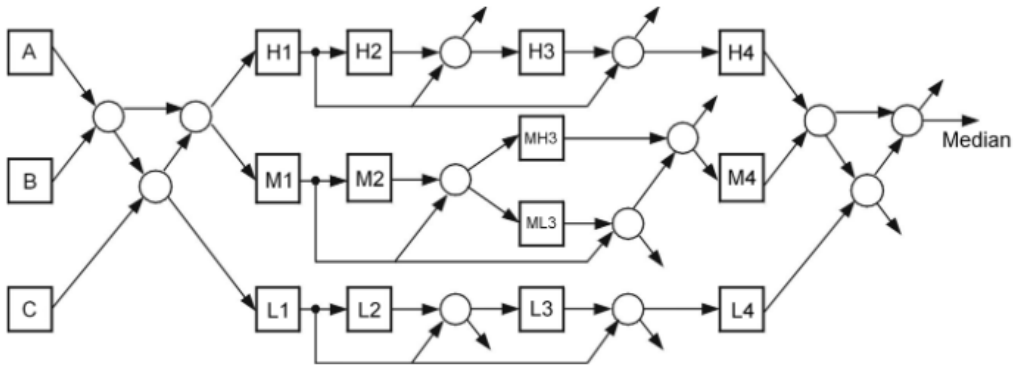

Figure 4: Median filter pipeline. Image source: “Ex4.pdf”, pg. 3 Fig. 4.4, Marcin Lukowiak.

Results and Analysis

The image moving functionality was implemented into the pre-existing top-level UART module. A separate control state machine was created to read incoming commands and dictate which operation is performed. This is shown in Figure 5 where input data is read from UART.

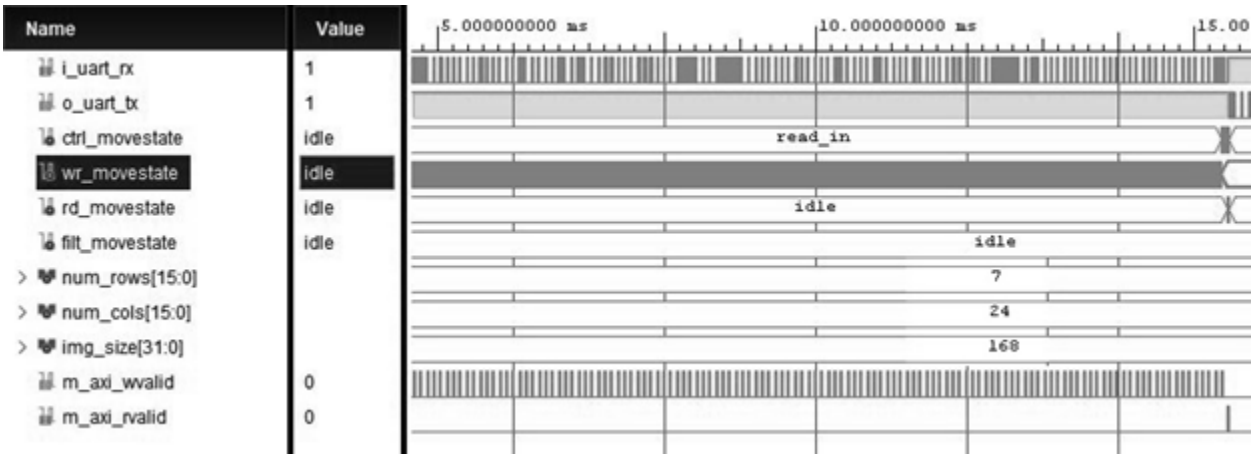

Figure 5: Part 1 simulation of reading feep.pgm in.

From Figure 5, the `ctrl_movestate` signal is in the `read_in` state signifying data is being written to memory. This is further verified by the `wr_movestate` signal constantly changing as pixels are read from `i_uart_rx` and stored to in memory. After this, the testbench requested the raw image data back. This operation is shown in Figure 6.

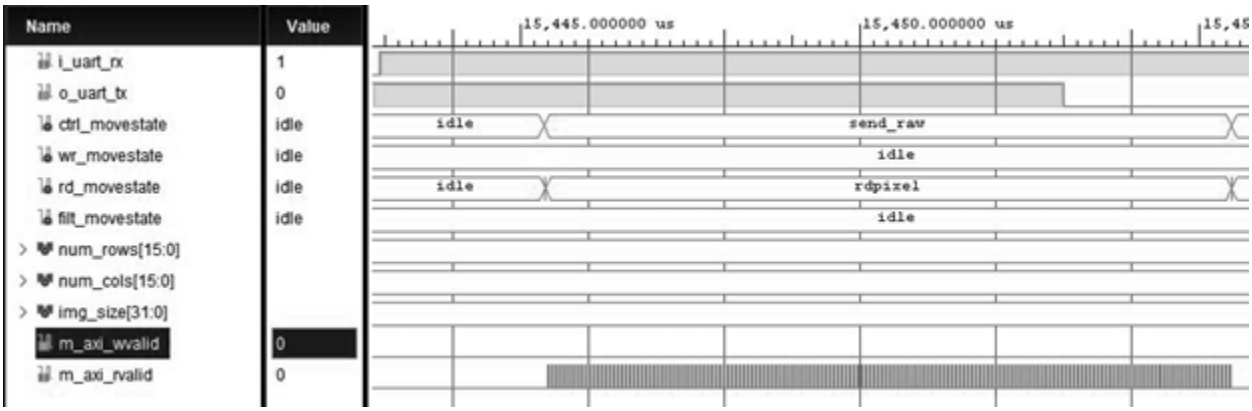

Figure 6: Part 1 simulation of sending feep.pgm back.

The image raw image data was successfully read from memory and sent to the transmit queue. Because the transmit FIFO depth is greater than the number of bytes in the image, all image data is read from memory and stored into the FIFO before a single value is completely transmitted back over UART. The received data was written back to an output file specified in the testbench. The output file correctly matched the input file indicating there was no data loss.

A separate testbench was created to test the median filter on its own before being integrated with the top-level module. A predefined data matrix was sent through the filter. This matrix is shown in Equation 1.

$$A = \begin{bmatrix} 1 & 2 & 3 & 4 & 5 & 6 \\ 7 & 8 & 9 & 10 & 11 & 12 \\ 13 & 14 & 15 & 16 & 17 & 18 \end{bmatrix} \quad 1$$

Due to the ordering of the matrix A shown in Equation 1, the expected medians are 8, 9, 10, and 11 in a perfect scenario. The matrix was sent through the filter as shown in Figure 7 and Figure 8.

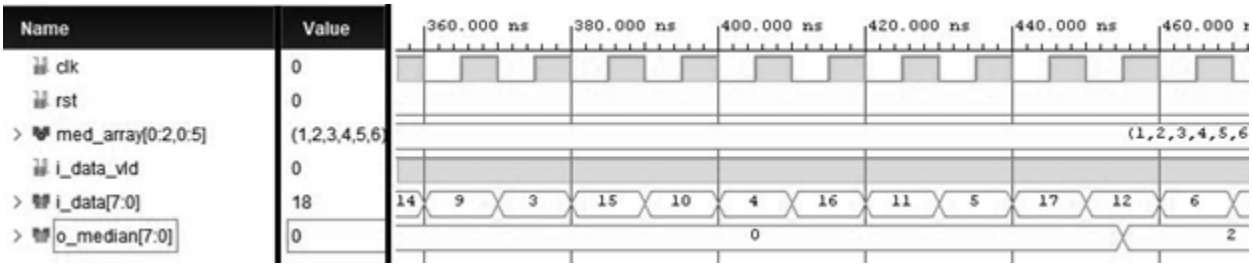

Figure 7: Part 2 simulation of sending a sample data matrix.

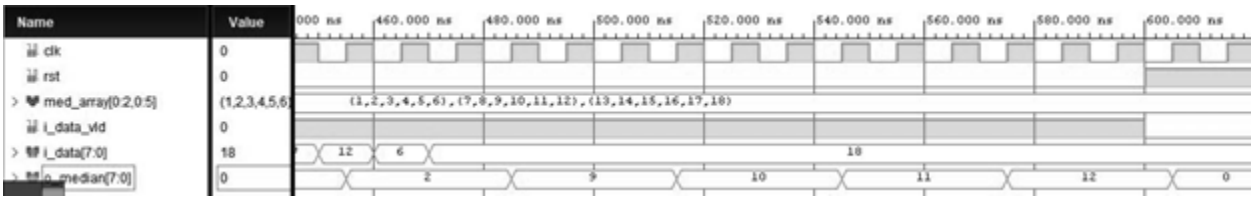

Figure 8: Part 2 simulation of obtaining medians from a sample data matrix. The first valid median is returned at 495 ns.

The received medians were 9, 10, 11, and 12. These values are shifted to the right from the expected due to the pipelined nature of the median filter. The final filtered image edges are therefore not precise, however that is to be expected and unavoidable using this processing algorithm.

The median filter was implemented in the top-level module using the state machine described in Figure 3. The testbench was modified to request the image be filtered. Figure 9 shows the waveforms of this operation in full.

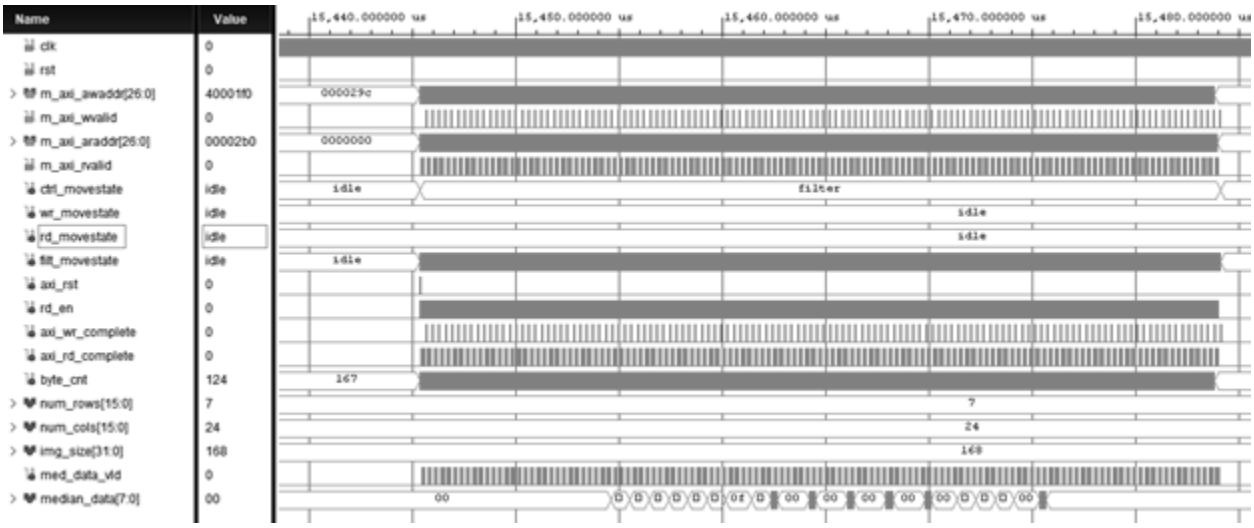

Figure 9: Part 3 simulation of filtering feep.pgm.

An individual cycle of the filtering state machine can be seen in Figure 10.

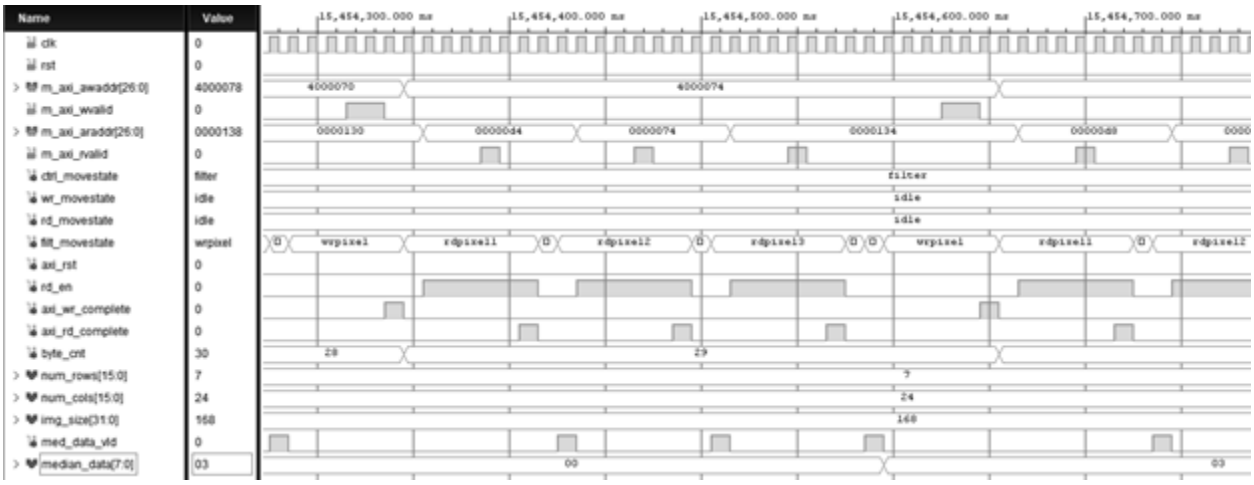

Figure 10: Single filter state simulation.

The waveforms shown in Figure 10 show the filtering state machine correctly operating. The data was then sent back over UART and successfully stored in an output file.

The final design was implemented and loaded onto the FPGA. Matlab was used to send the Lena.pgm image over UART, request this raw image back, request the image be filtered, and finally request the filtered image back. The final output is shown in Figure 11.

Figure 11: Matlab filtering output (left: original image; center: raw image received from FPGA; right: filtered image received from FPGA).

The image was successfully filtered and interpreted by Matlab. The size of the image was reduced by two rows as the top and bottom rows were never the center point of the filter. The resulting image shows a dramatic reduction in the salt-and-pepper effect shown in the original image.

Conclusion

An image moving system was implemented on an FPGA. Through commands sent over UART, a portable graymap image was sent over UART to the FPGA, stored in memory, and received without data loss. A separate command requested the image be processed using a median filter. The median filter correctly functioned in simulations. The design was implemented in hardware and tested using Matlab to send Lena image with salt-and-peppering distortion to the FPGA. The image was successfully filtered and returned to Matlab. The filtered image showed a dramatic decrease in distortion. This exercise was successful.