ECE532: Digital Systems Design
Final Project Group Report
April 7, 2026

Hardware Image Filter

Group 6:
Cheng Su
Aochen Fan
Deyu Cao
Ben Barcados

1. Overview

This project creates a hardware image filter processor using a Nexys 4 DDR board. The image filter can accept input images over UART from desktops with Matlab script or using the ov7670 camera to capture an image as input. The output of this processor can either be a filtered still image sent back to the desktop with UART or a filtered live video through VGA display. Two filters were integrated into the system: the Sobel filter for edge detection and the median filter for removing salt and pepper noise.

1.1. Motivation

We chose to work on a hardware image filter because we have a strong interest in image processing and its increasing relevance in today's world. Another reason we were motivated to do this project was the growing trend of using hardware acceleration with FPGA. We were expecting to build something that could outperform the software approach. Finally, we would like to see how different filters react to each other by cascading them. We assumed that some filters could yield better results when combined with another filter.

1.2. Goal

This project aimed to build an image processor that contained 1 filter first. The system should be able to get input from the desktop using UART, then filter the image and send it back to the desktop. Then, add more filters to the system, let the input image run through each filter, and observe the cascading effect of the different filters. The user can also choose to capture an input image using the ov7670 camera and display the live processed result using a VGA monitor. Users should be able to operate the system using onboard switches to toggle different filters and input/output modes. When using a desktop for input and output, the buttons can be used to send and receive images.

1.3. Block Diagram

The original design of the system involves using UART to transmit and receive image data. Then the UART modules were wrapped in AXI stream(AXIS) protocol, which allowed us to have access to other Vivado IP, like AXIS data fifo, then using a custom AXIS and AXI-Lite bridge to convert the point to point protocol in to memory mapped protocol which allow us to have access to the memory interface generator provided by Vivado to communicate with DDR memory for data storage. The block diagram of the system is shown in Figure.1

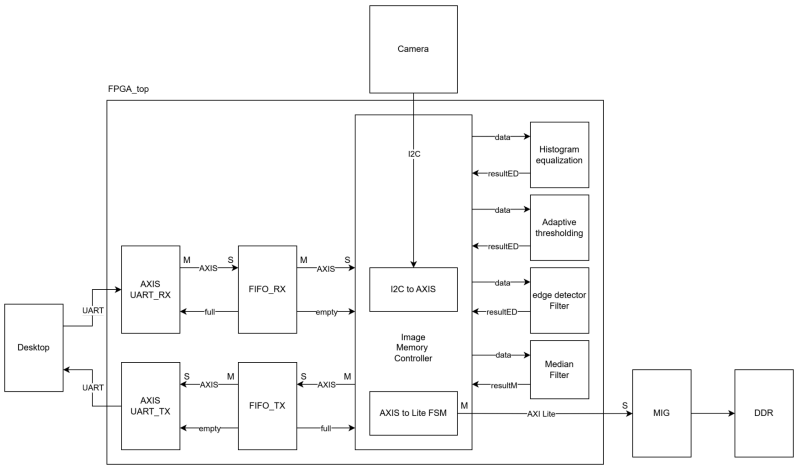

Figure.1 Original Block diagram

As shown in Figure 1, all filters were connected to the image memory controller. This is where the processor receives commands like send, receive, and filter image from the PC, and execute corresponding operations. The camera module should also be wrapped in AXIS protocol so that input from the camera can be fed into the AXIS and AXI-Lite bridge and to the reset of the system.

1.4. System Changes

The block diagram shown in Figure 2 depicts the final implemented version of our image processor.

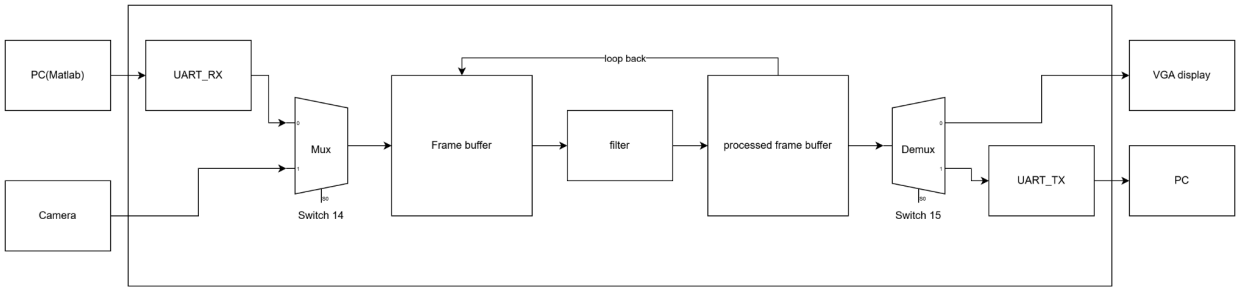

Figure.2: final implemented block diagram

As we progressed using the original block diagram, we were able to complete all modules except the image memory controller due to the complexity of the control signals. Therefore, we had to settle on the design shown in Figure 2 last minute.

In Figure 2, a mux was used to switch between PC input or camera input on the left side. On the right side, there is another switch used to switch between UART output to PC or direct output to VGA display. When a still image was used as input, the UART output must be selected to return the filter image. The camera input can be output onto either UART to PC or VGA display.

1.5. Description of the blocks

Our system contains 2 filter processing paths. The first one is live video processing using a camera and output onto the VGA display. The second path was to process a still image from the PC and send back the filtered image to the PC.

1.5.1. Video live processing path

The live video processing path begins with camera input**.** When the mux, switch14, toggles to the camera input. The camera module video captures each frame and sends to the frame buffer a byte at a time. The camera module would output three signals so that the input frame buffer knows when to store data, which address or position the data is going to be placed, and what data is stored. When a full frame is stored in the input frame buffer, the filter module begins to read data from the image buffer. The filter has a 3x3 sliding window for filter calculation. Each processed pixel is stored in the processed frame buffer. The filter module would also output three signals, which tell the processed frame buffer when, where, and what data is to be stored. The output mux, switch 15, is used to select different ways to output the processed frame. When a live video processing path is selected, the output must go to the VGA display. The VGA display module can read pixels from the processed frame buffer with col and row position on the display.

1.5.2. Image processing path

The Image processing path begins from the MATLAB script on the PC. The Matlab script would read a grayscale PNG file and get the image information like height and width. Then, open a serial port to send a PNG file. The UART_RX file would read a bit at a time with a baud rate of 115200. The UART_RX module would read bits until a full byte is received, then send the full byte to the frame buffer. A helper module, image_loader, was placed in between UART_RX and input frame buffer to generate an address and tell image buffer when to receive the byte data. The input image buffer reads byte data until a full image is received, then starts to send data to the filter module. The filter module works the same way as the video processing path, takes in the full frame, builds a 3x3 compute window, computes the result, and sends it to the processed frame buffer. When the processed frame buffer receives the full frame, each byte data is then sent to another helper module, image_sender, where an enable signal was generated telling the UART_TX module when to read from the processed frame buffer. Once the UART_TX receives a byte of data, it begins to output the byte one bit at a time to the PC. Finally, the Matlab script on the PC would read the bit data and generate a filtered PNG file.

2. Outcome

The project was successfully implemented in a minimally viable state, and it covers all the features that we proposed in the original design with a different approach.

2.1. Result

In the original design, we expected to use external memory like DDR RAM as storage for the input and output buffer. Therefore, we built a lot of infrastructure to enable us to use the Vivado IP like AXIS data fifo, AXIS and AXI-Lite bridge, AXI protocol converter, MIG and DDR, and finally, the integration with image memory controller. This approach did not make it to the end due to time limitations. However, we built a different system, as shown in Figure 2. This system essentially has the same functionality, without AXI protocol, DDR, and memory controller. The still image filter result is shown below in Figure 3.

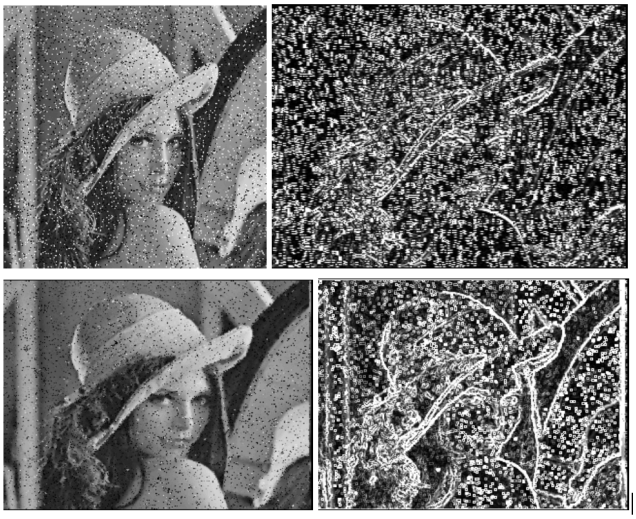

Figure.3: Result of the image processor. Top left: the original input image with salt and pepper noise. Top right: Sobel filter applied directly to the original input image. Bottom left: median filtered original image. Bottom right: result image for Sobel filter and median filter.

As shown in Figure 3, the edge detection filter returns poor results on noise images. This is because edge detection filters also process the salt and pepper noise as tiny circles. Due to the large number of tiny circles, it is very difficult to observe the edge of the image. The median filter can eliminate some noise on the image, so when the median filter image is passed into the edge detection filter, the result has much better quality, where edges can be easily observed. Although the median filter can remove noise on the original input, but cannot completely. A possible improvement would be to find a better method for the median filter so that noise can be removed completely. Another possible improvement is to speed up the performance of the median filter. The median filter uses a sorting algorithm to find the middle value, which is a very time-consuming algorithm in software. Currently, we implemented our median filter in a software approach where we sort then select value, which ruins the purpose of using hardware.

If we could start over again, we would first choose a proper version control tool so that all of us can work in a more organized way. During the developing phase, we created so many Vivado projects that were used to try out different methods and debug different components, which created a bit of hassle at the end when we tried to integrate. We often grabbed the wrong projects that were missing features, and we did

not realize it until we implemented the design on board and then found out that a certain switch or button did not work.

2.2. Future work

The first thing that should be worked on is to design a better median filter, which can remove all noise on the input image. Secondly, improve the performance of the median filter by removing the sorting algorithm and employing pure hardware logic. For example, using registers to store 3 input pixels and build comparators to find the median value between the 3 pixel values. This process is repeated throughout all the pixels in the input image to generate the filtered image. This median filter can also be pipelined to further increase the performance, for example, by dividing the filter into two stages. The first stage would load input pixels, and the second stage would find the middle value. Finally, the median filter can be duplicated multiple times so that there could be multiple median filter engines running at the same time.

3. Project Schedule

3.1. Milestones

Original Milestone	**Completed Milestone**
Milestone1	
● Research project components such as UART, flash memory, and camera PMOD.	● Research was completed.
Milestone2	
● Implement Verilog modules for UART RX and TX working with testbenches.	● Implemented UART RX and TX modules and testbenches. ● Both modules worked in simulation. ● Only RX worked on FPGA. ● Designed working median filter module and testbenches
Milestone3	
● Write a median filter module. ● Begin working on the read and write to memory.	● Debugged UART TX and AXI-Stream wrapper. ● Implemented camera PMOD and VGA output. ● Worked with Microblaze to control a hardware timer interrupt to receive useful analytics from the board.

	● Implemented Lacplacian filter.
Milestone4	
● Integrate UART, flash memory, and filters into one system. ● Test integrated components ● Use Microblaze to control a hardware timer interrupt.	● AXI-Lite to AXI Stream conversion module in development. ● Flash memory implementation ran into errors. ● Camera input can be modified and stored in the BRAM memory of the board. ● Implemented adaptive thresholding filter and histogram equalization modules.
Milestone5	
● Getting VGA output working. ● Finish integration and test for errors.	● AXi-Lite wrappers for filters implemented and tested. ● Finished implementing modules for both directions of AXI-Lite to AXI-Stream conversion. ● Updated camera to capture 256x256 frame so that the input is compatible with the rest of the system. ● Flash memory implementation experienced more errors. ● UART baud rate error fixed.
Milestone6	
● Project integrated and in presentable state	● Filters without integration working on FPGA together. ● UART and singular filter working on FPGA. ● Asynchronous flip-flop for camera, VGA, and AXI-Stream added. ● DDR memory implementation unsuccessful due to issues with Vivado IP core. ● UART RX to AXI-Stream/Lite converter to memory works in simulation.

3.2. Discussion

While a working minimum viable project was completed for the demo, the team missed multiple planned milestones and did not complete the project according to their original vision. The team did complete a lot of work individually but due to the issues at the time of integration much of it was unable to be showcased. When comparing the milestones outlined in the project proposal to the progress made in the

milestone reports a few issues are clear. First the team underestimated the difficulty of the integration stages of this project. Individual components working on the Nexys DDR4 board did not mean that when put together that combination would be successful. This led to more issues where time ran out for finished components of the project that could not be implemented. In the future, when possible, starting integration earlier in the project's life cycle would be beneficial. First, the team would have their minimum viable product soon. Second, it would give a better understanding of how adding more components would affect the project's scope.

Another issue with integration in this project was that the image memory controller was a major component to the project that would really only work with the other parts of the project completed. With this in mind the team decided that saving the memory controller for last made sense but in reality it meant that a quick smooth integration was impossible without this component. When looking at how important this component was to the project what would have been better would be to have an individual work on the memory controller in parallel to the rest of the project and have them integrate each component immediately as soon as it is ready. This would have broken integration into more manageable jobs instead of the singular great one the team forced on itself at the end of the project. As well it would allow for iterations of the project to exist so if issues were encountered there would always be a previous working state of the project. It is believed that this approach to integration would have allowed the team to present a more complete form of project closer to the original design.

4. Description of the Blocks

4.1. Top_ov7670_nexys

This is the top module, where all the other modules are instantiated. The following two switches are used to control the input and output of the system.

- Switch 14 is used to switch between camera input mode and UART input mode.
- Switch 15 is used to switch between VGA mode, where the processed image is live displayed via VGA, and Photo mode, where a static processed image is sent via UART to the desktop.

4.2. UART_TX/UART_RX

The UART RX module is used to read data bit by bit from the PC. The RX module is operated using a baud rate of 115200 and 16-time oversample to ensure the accuracy of the bit transmitted. When each bit is sent from the PC, the bit is sampled 16 times, and the 8th sample is used as the valid input bit. This requires a counter that increments on every rising edge of the clock until 16 is reached and the transmission of the current bit is completed. This process is repeated 8 times for a single pixel input. To

use the 115200 baud rate, another counter is created to generate a clock enable signal, which allows a valid input bit to be read every time the counter reaches a certain value. This value is calculated using the formula as shown in the equation.1

$$Clock\ frequency/(baud\ rate\ *\ oversample) \qquad \text{Equation.1}$$

The clock frequency used in the system is 100MHz, which means the clock enable signal would be generated every time when the counter reaches 54, and a valid input bit can be read from the PC. Once a bit is read, it is stored in an 8-bit width register. When the register is full, a read data complete flag would turn on high to indicate a pixel is read from the PC and is ready for downstream logic components.

The TX module works in the same way as the RX but in reverse. The upstream logic component would signal the TX module to start sending data. Then the TX module would grab the input byte and send it out bit by bit using a baud rate of 115200 and 16 time oversample. When the byte is sent out completely, a send_data_complete flag will turn on high to signal upstream logic ready to send the next byte data.

4.3. UART_sender/UART_loader

The UART_sender is a custom module that reads pixel data from the frame buffer and sends it over UART. It sends one pixel at a time and then increments the read address to move on to the next pixel.

The UART_loader does the opposite of the UART_sender. Whenever a pixel is received via UART, this module writes it to the correct address of the frame buffer, and increments the write address. This process is repeated until the end of the whole image.

4.4. Mode_select

This module handles the following two mode selections.

- Cycles through the different filters (in a predefined order) when the button is pressed shortly.
- Toggles test mode, which switches between camera input and the test image (a color bar).

4.5. VGA_sync

The VGA_sync module is responsible for generating the standard 640×480 VGA display timing at 60 Hz. Base on our project, the active image resolution is 320×240, the synchronization signals, hsync and vsync, and scanning coordinates follow the 640×480 VGA specification to ensure compatibility with standard monitors. The visible image is rendered only within the upper-left 320×240 region of the screen. The module outputs additional control signals, such as visible, which indicates when the current pixel is within the active display area, and new_pxl, which triggers pixel updates at the correct rate. These signals are used by the VGA display module to read image data from the framebuffer in sync with the screen refresh.

This IP block was reused from an open-source implementation without changes to its timing logic. However, in our design, the visible and new_pxl signals were utilized to conditionally enable memory reads and synchronize pixel output. The module is clocked at 25 MHz, the required pixel clock for 640×480 resolution at 60 Hz. We verified the functionality through onscreen image display and confirmed that the 320×240 grayscale images were correctly aligned and scaled in the display area.

4.6. VGA_display

The VGA_display module is responsible for rendering grayscale image data from the framebuffer to the VGA output in synchronization with the display timing provided by the VGA_sync module. It accepts row and column coordinates along with synchronization signals to determine when to read and display valid pixel data. The module supports an enable signal vga_en, to control whether VGA output is active, allowing the system to disable display during certain modes.

During normal operation, when a new pixel is triggered and the coordinates are within the 320×240 display area, the module increments the frame_addr pointer to read the next pixel from memory. This address is later used to retrieve grayscale values from the processed framebuffer. Since the image is monochrome, the module replicates the 8-bit grayscale value across the red, green, and blue VGA channels, creating a uniform grayscale tone. In addition, for debugging or aesthetic visualization, the module includes logic to colorize pixels outside the main image area using the row and column indices.

4.7. Frame_buffer_in

This is a simple two-port RAM module, used for storing a frame of image temporarily before being processed. At each clock edge, a byte of data (corresponding a brightness value of a pixel) will be written to the specified address when the write_enable is on, and another byte of data will be read from the specified address.

4.8. Edge_proc

Ths is the wrapper module that has different filters instantiated in it and also takes care of the data buffering. This module reads one byte at a time from the data buffer and stores the correct 9 pixels in its internal registers. This module has 2 row buffers to store the values in the two preceding rows, and the correct values are retrieved based on the current position in the row and stored in the 9 internal registers in each clock cycle.

4.8.1. Median_3x3

This is the median filter module that is instantiated inside the 4.9 Edge_proc module. This module takes 9 pixels (each 1 byte) as input and calculates the output. No data buffering is used inside the median filter module and the 9 inputs are assumed to be corresponding to pixels in the right position.

4.9. Frame_buffer_out

This uses the same module as 4.8 Frame_buffer_in, but this is used instead to store the processed image before sending to VGA_display or UART_sender.

4.10. Ov7670_capture

The ov7670_capture module is designed to interface directly with the OV7670 camera sensor, receiving pixel data in real-time and writing it into the system's framebuffer. It monitors synchronization signals from the camera, such as vsync and href to determine frame boundaries and valid pixel windows. Specifically, a new frame begins on the falling edge of vsync, and valid pixels are transmitted during periods when href is high, aligned with the pixel clock pclk.
The camera outputs 8-bit grayscale values, which the module directly samples and stores. Internally, the module uses row and column counters to compute a linear address that maps the 2D pixel grid into 1D framebuffer space. The signal write enable is asserted only when valid data is received, ensuring the framebuffer is updated only with meaningful pixel values. This mechanism ensures robust real-time image acquisition.

4.11. Ov7670_top_ctrl

The ov7670_top_ctrl module serves as the top-level configuration controller for the OV7670 camera module. Its primary function is to perform initialization and configuration of the camera via the SCCB interface. Upon reset, the module begins sending a predefined sequence of register-value pairs to the OV7670 through the sclk and sdat lines, setting up essential parameters such as resolution, color format, clock polarity, and output range.

4.11.1. Sccb_master

The sccb_master module is responsible for handling low-level communication between the FPGA and the OV7670 camera via the SCCB. This module generates the appropriate clock and data signals to sequentially transmit configuration data to the camera. It takes as input the 8-bit device address, 8-bit register address, and 8-bit data to write, along with a send control signal that initiates the transaction.

Internally, sccb_master implements a finite state machine that transitions through the standard I²C start condition, address phase, data write phase, and stop condition. It also includes timing control to meet the SCCB protocol's requirements. To interface with the OV7670, the module outputs bidirectional control signals.

Upon completing a transfer, the module asserts the ready flag, which signals to the higher-level ov7670_top_ctrl module that the next transaction can proceed.

4.11.2. Ov7670_ctrl_reg

The ov7670_ctrl_reg module provides the configuration data necessary to initialize the OV7670 camera sensor. It serves as a lookup table that stores a predefined sequence of register address-value pairs, which are written to the camera via the SCCB protocol during the initialization phase. Each register setting in the sequence corresponds to a specific camera feature, such as color format, resolution, gain, or clock control.

5. Design Tree

This project implements a real-time image capture, processing, and display pipeline on the Nexys4 DDR board using the OV7670 camera. The design features UART communication for both image upload and download, supports grayscale and edge filtering (Sobel/median), and displays images via VGA at 320×240 resolution.

All source code is hosted on GitHub at: **https://github.com/Veclex/ECE532_Hardware_Image_Filter**
Please see the directory structure and explanation below.

Hardware Image Filter

Overview
This project implements a real-time image capture, processing, and display pipeline on the Nexys4 DDR board using the OV7670 camera. The design features UART communication for both image upload and download, supports grayscale and edge filtering (Sobel/median), and displays images via VGA at 320×240 resolution.

Features
- **OV7670 Camera Support**: Captures real-time grayscale video at 320×240 resolution using the OV7670 CMOS camera module.
- **VGA Display Output**: Displays processed images at 640×480 VGA timing with active resolution of 320×240.
- **Real-Time Image Processing**: Includes 4 modes of processing: Raw grayscale, Sobel edge detection (horizontal / vertical / combined), and 3×3 median filtering (noise reduction).

- **Bi-directional UART Interface**: User-friendly script for image upload/download

Switch & Button Control
- sw14: UART image input mode
- sw15: UART image output (transmit)
- btnr: capture/send trigger
- btnl: switch image processing mode
- btnc: reset

Structure
```
├── /final_3.srcs
│   └── constrs_1/                    # constraints file
│       └── new/
│           └── nexys4ddr_filter.xdc
│   └── sources_1/                    # source file
│       └── imports/
│           └── nexys/
│               ├── top_ov7670_nexys.v      # Top-level module
│               └── ov7670_ctrl_reg.v      # register  configuration
│           └── ov7670_yuv_80x60_sobel/
│               ├── mode_sel.v             # model selection
│               ├── edge_proc.v            # edge_processing filter
│               ├── sccb_master.v          # initialization
│               ├── ov7670_top_ctrl.v      # camera configuration top
│               ├── ov7670_capture.v       # capture logic
│               ├── frame_buffer.v         # Frame_buffer
│               ├── vga_display.v          # rendering data
│               └── vga_sync.v             # synchronization
│       └── new/
│           └── median_3x3.v              # median filter
│           └── uart_img_sender.v
│           └── uart_rx.v                 # RX
│           └── uart_img_loader.v
│           └── uart_tx.v                 # TX
│
├── /matlab
│   ├── uart_to_board.m                   # MATLAB script for uploading image to board
│   ├── uart_out_board.m                  # MATLAB script for recive image from board
│   ├── uart_loop.m                       # Main interactive loop: upload + receive + resend
│
├── /doc
│   └── report.pdf                        # Group report (NOT individual reports)
│
```

```
├── /test_img
│   ├── Lena.png                    # Test image used for upload
│   └── output/                     # Folder to save received images
│       └── median_only.png
│       └── median+sobel.png
│       └── sobel_only.png
│
├── README.md                       # This description file
```

Notes
- The /final_3.srcs directory contains all synthesizable and behavioral modules for FPGA synthesis.
- The system is driven by a 100 MHz input clock. VGA display resolution is 640x480 (timing), with active image region of 320×240.
- MATLAB handles image upload/download via UART

6. Tips and Tricks

- Design filters with hardware in mind
    - When designing image processing filters, always consider what can be done in a single clock cycle. Avoid trying to port a software algorithm directly to Verilog without accounting for pipelining or parallelism. For example, the median filter is computationally intensive in software due to sorting, but in hardware, a fixed comparator-based approach is more efficient and can be pipelined.
- Pipelining is essential for performance
    - Especially in filters like Sobel or Median, splitting the logic into pipeline stages allows you to achieve better clock speeds and throughput. For instance, consider a 2-stage median filter pipeline: one stage loads the 3x3 pixel window, and the next stage computes the median.
- Integrate early and often
    - Don’t wait until all modules are “finished” before integration. As we learned, many bugs and system-level issues only appear during integration. Start connecting pieces as soon as any one of them is working to ensure compatibility and smooth debugging.
- Use proper version control from Day 1
    - Use Git (or similar) and commit often with meaningful messages. This prevents the confusion of multiple Vivado project folders and avoids the issue of integrating outdated or incomplete versions of modules.
- Modularize and reuse your components
    - Write small, well-defined modules that can be reused in different configurations (e.g., UART_TX/RX, frame buffers). This makes debugging easier and improves maintainability.
- Simulate with realistic testbenches before FPGA deployment

    - Use test images, realistic UART streams, and timing constraints in simulation. Simulations helped us verify UART RX functionality before moving to hardware and avoided unnecessary time spent debugging on the board.
- Know when to pivot your architecture
    - Our decision to move away from AXI and DDR memory late in the project was difficult but necessary. Simplifying the design helped us reach a presentable minimum viable product. Be ready to adjust your system architecture when integration proves too complex under time constraints.
- Document signal conventions carefully
    - Keeping consistent naming and directionality (e.g., ready/valid, enable, addr/data) across modules made it easier to plug components together. This is especially critical when working with synchronous signals across multiple domains (camera, VGA, UART).

7. Appendix

Links to demo videos

Image Processing Project (ECE532 of University of Toronto) : still image processing

Image Processing Project (ECE532 of University of Toronto) : real time processing