ECE 385

Spring 2025

Final Project

FPGA-Based Camera Capture and Real-Time Display System with Image Processing

Rohan Shah and Guy Robbins
Section AL1
Elijah Ye

Introduction:

For our final project in ECE 385, we interfaced an OV7670 camera with the AMD Urbana FPGA to generate a real-time video display on a monitor, enabling basic image processing effects such as color inversion, grayscale, and a “light tunnel” filter. The project was structured around two main objectives: first, to achieve a functional live video output, and second, to implement image processing for photobooth-style effects.

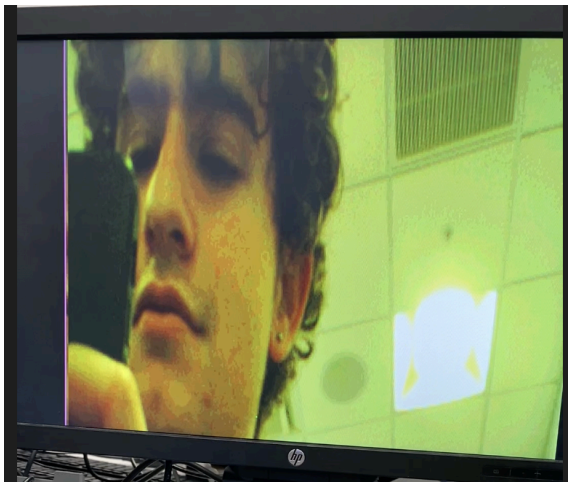
Achieving the live video output proved to be the most time-consuming and technically challenging aspect. Due to memory constraints on the AMD Urbana board, we chose to use QVGA resolution (320×240 pixels) and RGB444 mode, which required us to later scale the video up to VGA for display. Our system design included a custom I2C driver for camera configuration, a ROM containing the initialization sequence, a pixel capture finite state machine, a dual-port BRAM for frame buffering, and a top-level module to connect all components and handle video output and processing.

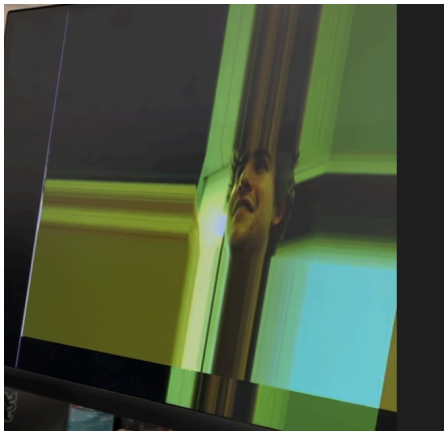
We also developed a constraint file to map the OV7670’s signals to the FPGA’s GPIO pins, using female-to-male jumper wires for physical connections. Some of our biggest challenges involved debugging the I2C interface, tuning the camera’s configuration registers, and ensuring correct timing across modules. Despite the complexity, we had a great time building this system and are proud of what we accomplished.

The general flow of the project was, we press the start button on the fpga to initialize the camera. After this is done, the camera starts sending out pixel data in parallel, with 2 bytes corresponding to each pixel. Then our pixel capture module writes this pixel data into our framebuffer, in the form of a dual port BRAM. Then, our top level module reads from the framebuffer, does some image processing, and feeds the pixel data into a vga to hdmi IP. The top level also upscales our QVGA resolution to VGA resolution by repeating each pixel four times in a 2x2 box.

Outputs:

As you can see below, we have a real time video feed display from the camera. In the first image you can see the raw, unedited output of the camera under the configuration we set. In the second image you can see greyscaling. The third is color inversion, and the fourth is the photobooth like effect of “light tunnel”.





OV7670 Camera:

The OV7670 camera is a compact, low power VGA image sensor commonly used for real time video applications. It features a CMOS active-pixel array that can output up to 640 by 480 resolution images, with configurable options for scaling and windowing to lower resolutions such as QVGA (320 by 240). The camera outputs data over an 8-bit parallel bus (D[7:0]) along with synchronization signals VSYNC, HREF, and PCLK to organize frame and line timing. The OV7670 must be configured at startup through its internal register set by I2C communication. Key configuration parameters include selecting the color format (in our case RGB444 mode), setting scaling factors, enabling or disabling auto gain and auto exposure, and adjusting image orientation or timing offsets (such as HSTART, HSTOP, VSTART, and VSTOP). In our project, we implemented an I2C controller and camera rom to automatically write a specific sequence of values to these registers, ensuring the camera operated in the desired mode and resolution compatible with our FPGA processing pipeline. Without proper configuration, the camera outputs invalid or misaligned data, so careful tuning of these settings is essential for stable and correct video capture.

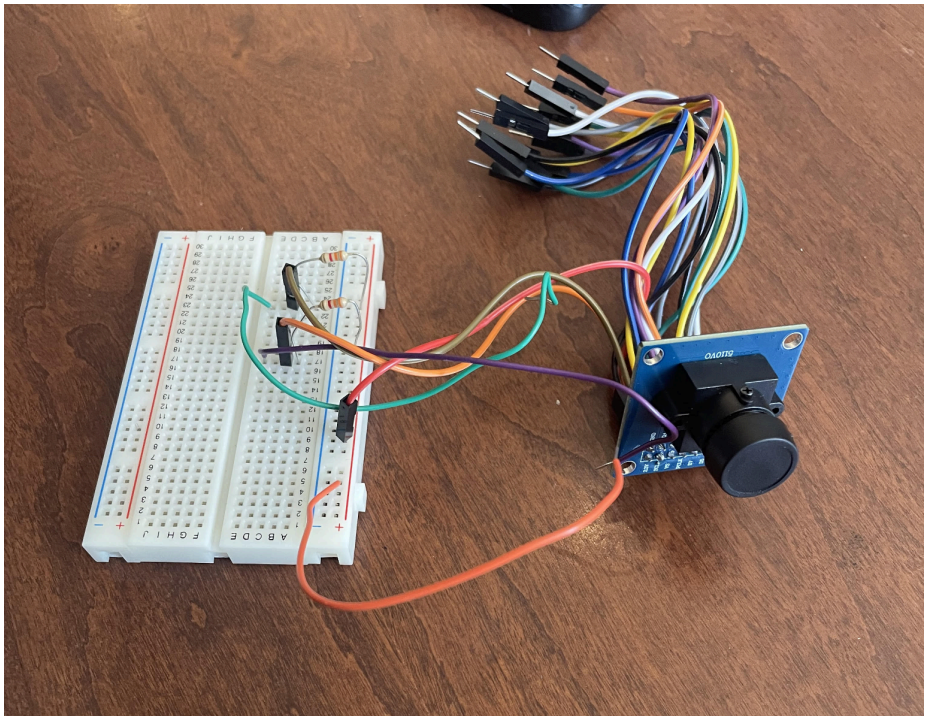


Figure 1: OV7670 Camera Powered with Breadboard

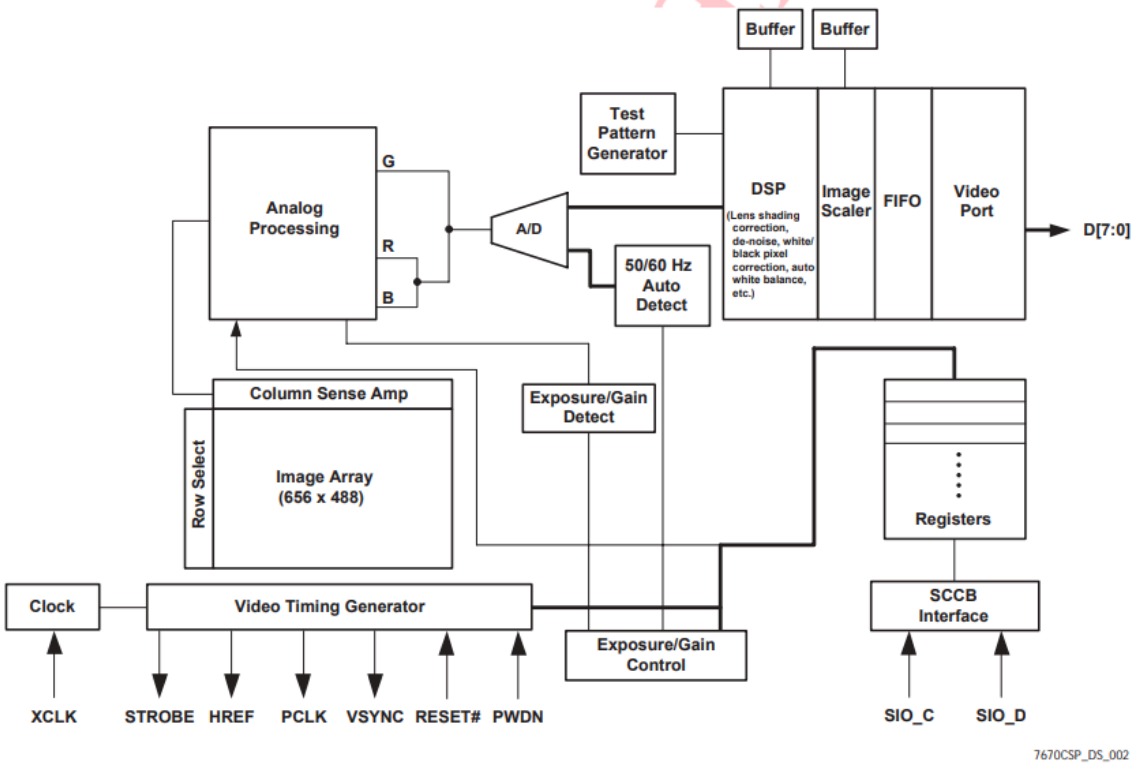


Figure 2: OV7670 Camera Internal Block Diagram

(Hex)	Name	(Hex)	R/W	Description
0A	PID	76	R	Product ID Number MSB (Read only)
0B	VER	73	R	Product ID Number LSB (Read only)
0C	COM3	00	RW	Common Control 3 Bit[7]: Reserved Bit[6]: Output data MSB and LSB swap Bit[5]: Tri-state option for output clock at power-down period 0: Tri-state at this period 1: No tri-state at this period Bit[4]: Tri-state option for output data at power-down period 0: Tri-state at this period 1: No tri-state at this period Bit[3]: Scale enable 0: Disable 1: Enable - if set to a pre-defined format (see COM7[5:3]), then COM14[3] must be set to 1 for manual adjustment. Bit[2]: DCW enable 0: Disable 1: Enable - if set to a pre-defined format (see COM7[5:3]), then COM14[3] must be set to 1 for manual adjustment. Bit[1:0]: Reserved
0D	COM4	00	RW	Common Control 4 Bit[7:6]: Reserved Bit[5:4]: Average option (must be same value as COM17[7:6]) 00: Full window 01: 1/2 window 10: 1/4 window 11: 1/4 window Bit[3:0]: Reserved
0E	COM5	01	RW	Common Control 5 Bit[7:0]: Reserved
0F	COM6	43	RW	Common Control 6 Bit[7]: Output of optical black line option 0: Disable HREF at optical black 1: Enable HREF at optical black Bit[6:2]: Reserved Bit[1]: Reset all timing when format changes 0: No reset 1: Resets timing Bit[0]: Reserved

Figure 3: Camera Register list and functions

I2C Driver

One of the most important and technically challenging modules we developed was our I2C driver, which configured the OV7670 camera. This need arose from the camera's design: the OV7670 contains over 400 internal registers that control its output behavior. To set the correct color mode, resolution, pixel format, and other settings, we had to write to around 75 of these registers.

The communication protocol used for configuring these registers is called SCCB (Serial Camera Control Bus), which is a variant of I2C. While SCCB omits some features like the acknowledge bit, it is functionally identical to standard I2C. This allowed us to design a finite state machine (FSM) that transmits register values from a ROM to the camera by following the I2C protocol. Further, since we only needed to write to the camera, we only had to design the write component of the protocol, and did not implement a read part to our FSM.

Before diving into the implementation, it’s helpful to understand the basics of I2C in a single-master, single-slave setup, with the FPGA acting as the master and the camera as the slave. I2C is a synchronous serial communication protocol where data is transferred one bit at a time over a shared data line (SDA), which is synchronized by a clock line (SCL). Both lines are

pulled up to a supply voltage (3v3 in our case), meaning they idle high and are actively pulled low by devices on the bus. This connection is shown below in figure four. In our implementation, only the master (the FPGA) pulled the lines low, and the timing for each transition was carefully controlled by our FSM.

Our I2C FSM consisted of nine states: IDLE, LOAD_NEXT, START, SEND_DEVICE_ADDR, SEND_REG_ADDR, SEND_DATA, STOP, WAIT1, and DONE. These states walk through the SCCB initialization process, allowing the FPGA to send register configuration data from a ROM to the OV7670 camera. Explaining the purpose of each state naturally illustrates how the I2C protocol operates in our implementation.

We begin in the IDLE state, where internal counters are reset and both SDA and SCL are set high, which is the idle condition for I2C. If the user presses the start button (start signal), we latch it and transition to LOAD_NEXT.

In LOAD_NEXT, we first check whether all 75 entries in our ROM have been sent. If so, we jump to DONE. Otherwise, we set byte_to_send to 0x42, the write address of the OV7670, and prepare for the start condition.

The START state initiates communication by pulling SDA low while SCL remains high, then pulling SCL low after one I2C clock cycle. We manage this timing with a counter, as we do in several other states where precise sequencing is required.

Next is SEND_DEVICE_ADDR, where we transmit the 8-bit write address one bit at a time over SDA, synchronized with the toggling of SCL. On the ninth cycle, we release SDA to simulate an acknowledge slot, even though SCCB does not send an ACK. This step turned out to be necessary, as omitting it caused register writes to fail, likely due to internal timing expectations in the camera.

After this, we load the register address from the ROM and move to SEND_REG_ADDR, where the address is sent in the same bit-wise fashion. Then we set byte_to_send to the associated data value and move to SEND_DATA, which follows the exact same transmission pattern.

Once the data byte is transmitted, we move into the STOP state. Here, we raise SCL while SDA is low, then raise SDA after one more clock cycle to generate the I2C stop condition.

To allow the camera enough time to complete the write internally, we enter the WAIT1 state, where we insert a short delay of 7 I2C clock cycles. After this, we increment the ROM address and loop back to LOAD_NEXT.

If all ROM entries have been sent, we enter DONE, where the done signal is asserted, the busy flag is cleared, and the FSM returns to IDLE.

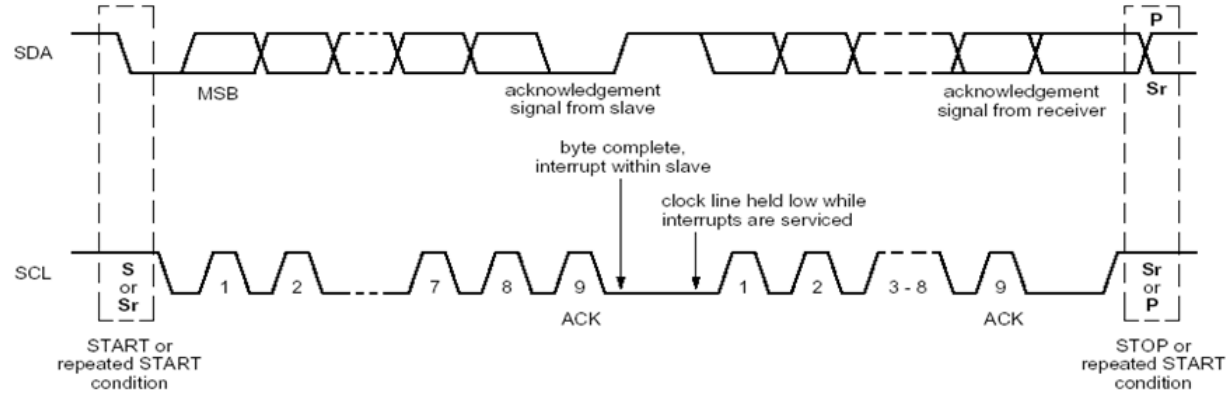


Figure 4: Timing diagram/handshake of SDA and SCL signals

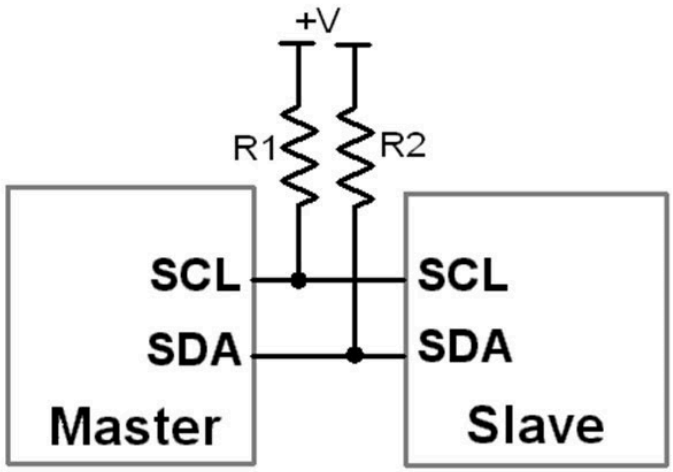


Figure 5: SDA and SCL signals pulled high with pull-up resistors

Pixel Capture

A critical component of our camera-to-display pipeline is the pixel_capture module, which handles real time data acquisition from the OV7670 camera and formats it for storage into the FPGA's block memory (BRAM). The OV7670 outputs pixel data using the RGB 444 format, which transmits each pixel using two consecutive bytes, the first containing only red and the second containing green and blue.

Figure 13 RGB 444 Output Timing Diagram

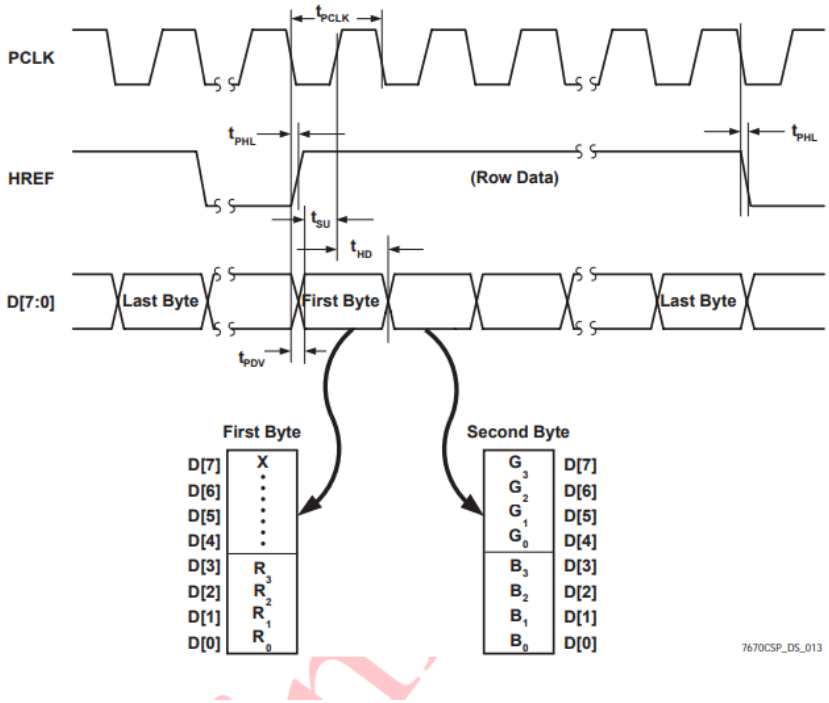


Figure 6: RGB 444 Timing Diagram

Camera Data Protocol

The camera generates a PCLK signal (24 MHz) that indicates when data is valid on the D[7:0] bus. The HREF signal marks the active period of each row of image data, and VSYNC signals the start of a new frame. As shown in the diagram, during each row, two bytes represent a single pixel:

First Byte: Contains the 4-bit red value (R3:R0) and padding bits.

Second Byte: Contains the 4-bit green value (G3:G0) and 4-bit blue value (B3:B0).

This data structure requires our design to latch two consecutive bytes and combine them to form a single 12-bit RGB value for each pixel.

Functionality

The pixel_capture module relies on an FSM to synchronize the incoming pixel data from the camera with the internal memory system. It ensures that only valid pixel data, corresponding to active frame data, is captured and written to the BRAM for later display. The FSM effectively manages the handshake between the OV7670 camera's streaming data and the FPGA's storage, avoiding corrupted or misaligned pixel values.

All of the FSM states and their explanations (in order) are listed below:

IDLE:

- Waits for HREF (horizontal reference) to go high, indicating valid pixel data for the start of a new row.
- Prepares the module to start capturing the first byte of a new pixel.

BYTE_1

- Captures the first byte of the pixel from the D[7:0] bus.
- This byte contains the 4-bit red component (R3:R0) and unused padding bits.
- Transitions to BYTE_2 when the next byte becomes available.

BYTE_2

- Captures the second byte from the camera.
- This byte contains the 4-bit green (G3:G0) and 4-bit blue (B3:B0) components.
- Once both bytes have been latched, the FSM moves to WRITE.

WRITE

- Combines the two bytes into a single 12-bit RGB value (4 bits per color).
- Calculates the BRAM address using current row_index and col_index values.
- Asserts wr_en to initiate the write into BRAM.
- Transitions to COL_ADD.

COL_ADD

- Increments the col_index to prepare for the next pixel in the row.
- If the end of the row (col_index = image width − 1) has not been reached, it returns to BYTE_1 to continue capturing the next pixel.
- If the end of the row is reached, resets col_index, increments row_index, and either starts the next row or moves to DONE if the entire frame is complete.

DONE

- Occurs when both col_index and row_index indicate the full frame has been captured (all 320×240 pixels).
- Waits for HREF to go low, signaling the end of the current row and frame.
- Once HREF de-asserts, the FSM returns to IDLE to prepare for the next frame.

The vsync input resets the row and column counters to prepare for a new frame. During normal operation, when HREF is high, the module captures and formats each pixel before writing it to memory.

Addressing & BRAM Interface

The module flattens the 2D image array into a 1D address using the row major formula we learned in ECE 220:

wr_addr = row_index × image_width (# of columns) +col_index

In our QVGA implementation, image_width = 320. This ensures that pixels are written sequentially in memory to allow for smooth display readout.

However, due to the time intensive nature of the multiplication operator on an FPGA, by just having the above algorithm our output was very flawed. Due to this, we had to implement separate pipelines, utilizing only addition or shifts to improve the timing to calculate addresses for both reading and writing. On the writing side, where timing was more important, we opted for a running counter, bounded by the col_index and row_index, resetting once the bounds are reached. On the read side, however, we opted for a pipelined multiplication. Specifically, instead of computing read_addr = y * 320 + x with a direct multiplier, we split the multiplication into two shift-based operations: y << 8 (which gives y * 256) and y << 6 (which gives y * 64). These

two values are added to compute y * 320, and then x is added to get the final address. All intermediate results are stored in registers and updated on the clock edge, allowing the computation to complete over a few pipeline stages using only additions and shifts, which are much more timing-friendly on FPGA.

Timing and Synchronization

The FSM aligns with the PCLK to avoid race conditions and ensures that only valid pixels are stored. The pixel capture logic accounts for data setup and hold times to reliably latch incoming values. By following the protocol defined by the OV7670's timing diagram, the pixel capture module effectively bridges the external camera hardware with the internal digital memory of the FPGA, forming the first essential stage of the camera pipeline.

Overall, pixel_capture transforms raw camera data into structured frame data. It works as the pipeline's front-end, enabling subsequent modules like BRAM and HDMI output to function correctly. Without reliable pixel capture, image corruption, tearing, or flickering would occur during display.

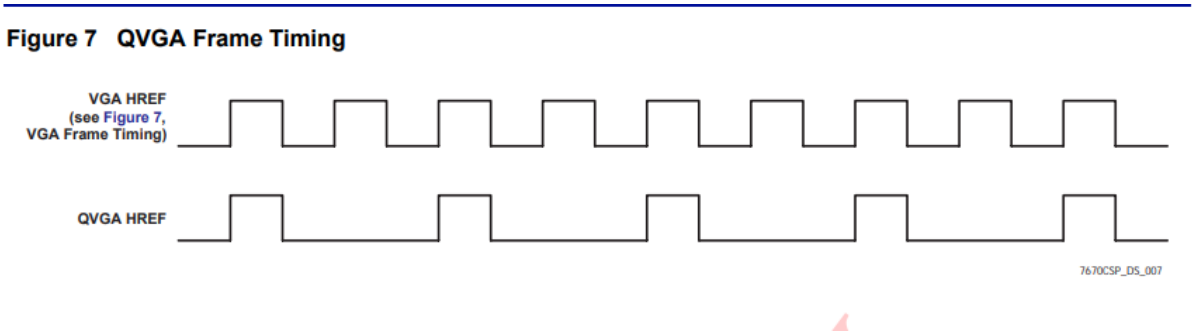


Figure 7: QVGA Frame Timing Diagram

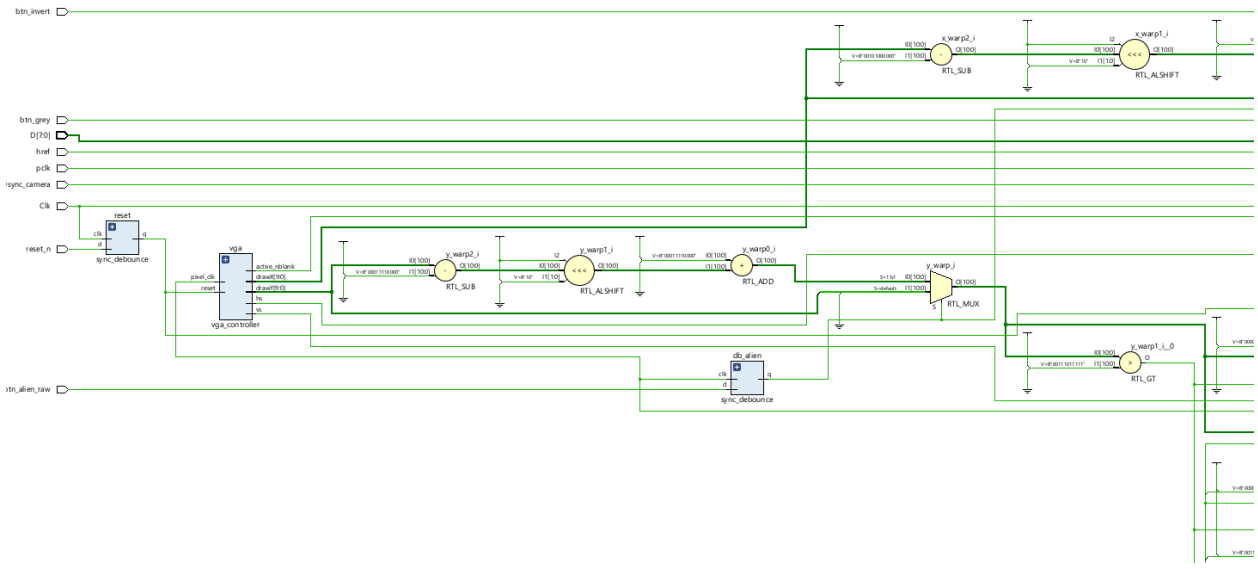
Filter Implementation:

We had time to implement three main filters: color inversion, grayscale, and a light tunnel filter. Each filter was applied one pixel at a time during the read stage from BRAM. For color inversion, we simply used the bitwise NOT (~) operator on each 12-bit RGB444 pixel value, flipping every bit to produce the inverted color. For grayscale, we extracted the red, green, and blue components, summed them, and divided the result by three to approximate luminance. The output grayscale value was then assigned to all three color channels to maintain the RGB format.

The most complex effect was the light tunnel filter, which we designed to simulate a lens distortion effect centered in the middle of the screen. For this filter, instead of reading from the pixel located at (x, y), we offset the coordinates to point closer to the image center, effectively "pulling" pixels inward. This created a radial distortion where the image appeared to tunnel toward the center. We controlled this transformation using a simple scaling factor based on the pixel's distance from the center. The farther a pixel was from the center, the more it was pulled inward.

We implemented filter selection using two physical buttons on the FPGA, and a switch, each mapped to a specific filter. When no button was pressed, the system displayed the original color output from the camera. This allowed for real-time switching between filters during live video display. When the flip was switched, we outputted the distorted light tunnel effect. When one of the buttons was pressed, we implemented the color changes. This allowed us to both have the filter on and change colors at the same time.

Block Diagrams



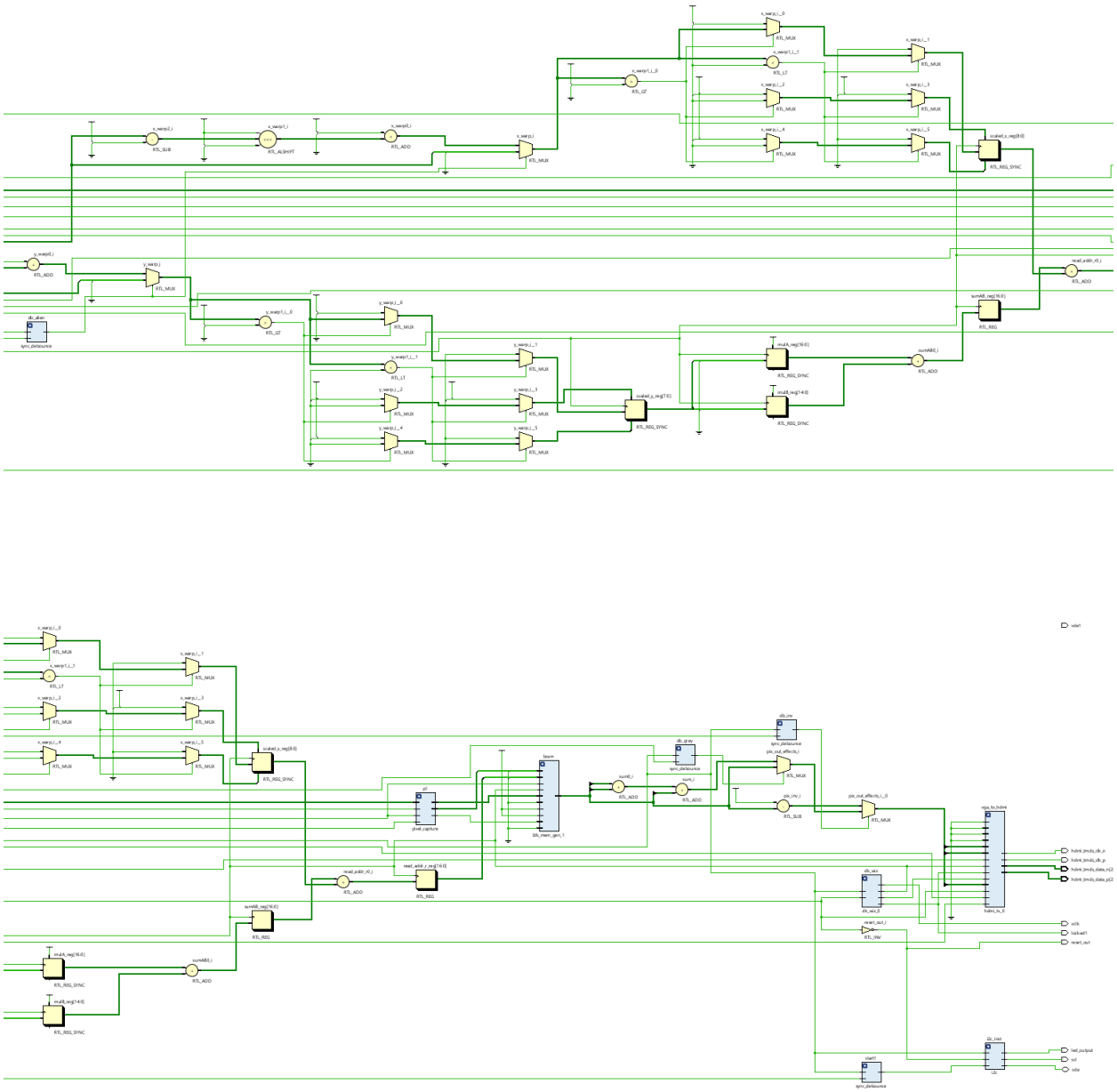


Figure 8: Top Level Block Diagram split into thirds (RTL view)

SV Module Descriptions

To program our FPGA with the adder system architectures, we implemented multiple .sv modules, with many instantiated within each other.

Module: camera_top.sv

Inputs: Clk, start, reset_n, [7:0] D, pclk, vsync_camera, href, sda, btn_invert, btn_grey, btn_alien_raw

Outputs: scl, led_output, xclk, reset_out, hdmi_tmds_clk_n, hdmi_tdms_clk_p, [2:0]hdmi_tmds_data_n, [2:0]hdmi_tmds_data_p

Description: This module is the top level integration of the camera to HDMI pipeline. It instantiates and connects the clock generator, I2C camera configuration module, pixel capture, BRAM frame buffer, HDMI encoder, VGA timing generator, and some image filters. It also debounces and routes input buttons.

Purpose: This module routes signals from inputs to the FPGA to the necessary instantiations of other modules so that camera data can be captured, processed, stored, and sent to an HDMI display.

Module: i2c.sv

Inputs: clk, rst_n, start, sda

Outputs: sda, scl, done, busy

Description: This module controls the I2C communication with the OV7670 camera sensor. It reads a camera configuration ROM and sequentially sends the required register settings to the camera upon start signal.

Purpose: To configure the OV7670 camera registers over I2C at boot to enable proper pixel format (RGB 444), resolution, and scaling modes for capturing frames into the FPGA.

Module: camera_config_ROM.sv

Inputs: i_clk, i_rstn, [7:0] i_addr

Outputs: [15:0] o_dout

Description: This is a simple synchronous ROM containing the OV7670 camera initialization sequence as a list of register addresses and their respective value pairs.

Purpose: To supply the camera configuration data to the i2c.sv module so that the FPGA can automatically initialize the camera.

Module: pixel_capture.sv

Inputs: [7:0] D, pclk, vsync, href

Outputs: [11:0] RGB, [16:0] wr_addr, wr_en

Description: This module samples data from the camera over the pclk domain, decodes the RGB444 data from two camera bytes, and outputs a write enable signal with a BRAM write address.

Purpose: To continuously capture pixel data from the camera in real time and prepare it to be written into the BRAM frame buffer for later reading by the display controller.

Module: Block Memory Generator (BRAM)

Inputs: [17:0] addra, clka, [11:0] dina, ena, [0:0] wea, [17:0] addrb, clkb, [11:0] dinb, enb, [0:0] web

Outputs: [11:0] douta, [11:0] doutb

Description: Dual-port block RAM IP used as a frame buffer. Port A writes incoming pixel data from the camera at pclk. Port B reads out pixel data to the HDMI display pipeline at 25 MHz pixel clock.

Purpose: To act as a buffer between asynchronous camera input and video output. It stores full video frames so that the display controller can read stable image data independent of capture timing.

Module: VGA_controller.sv

Inputs: pixel_clk, reset

Outputs: hs, vs, active_nblank, sync, [9:0] drawX, [9:0] drawY

Description: Generates the necessary VGA timing signals, including horizontal and vertical sync pulses, and pixel coordinates to drive a VGA display.

Purpose: Coordinates the display scanning process, ensuring that video signals are synchronized and the image is shown properly on the screen.

Module: Clocking Wizard

Inputs: reset, clk_in1

Outputs: clk_out1, clk_out2, clk_out3, locked

Description: An IP-generated module that derives multiple clock signals with a 25 MHz, 24 MHz, and 125 MHz clock from a single input 100 MHz clock.

Purpose: Provides stable, synchronized clock outputs to different parts of the design, ensuring proper timing and operation of all subsystems.

Module: VGA to HDMI

Inputs: pix_clk, pix_clkx5, pix_clk_locked, rst

Outputs: red, green, blue, hsync, vsync, vde, TMDS_CLK_P, TMDS_CLK_N, TMDS_DATA_P, TMDS_DATA_N

Description: Converts VGA video signals into HDMI compliant TMDS signals.

Purpose: Bridges the analog VGA domain with the digital HDMI interface, allowing VGA generated content to be displayed on HDMI monitors.

Testbench Waveforms

The waveforms below show the timing diagrams for both the I2C driver and Pixel Capture module, which both rely on FSMs. The longer hexagons with words in them represent the current state the FSM is on, and other signals can be visualized as well based on the description and instructions described in the earlier sections for each module.

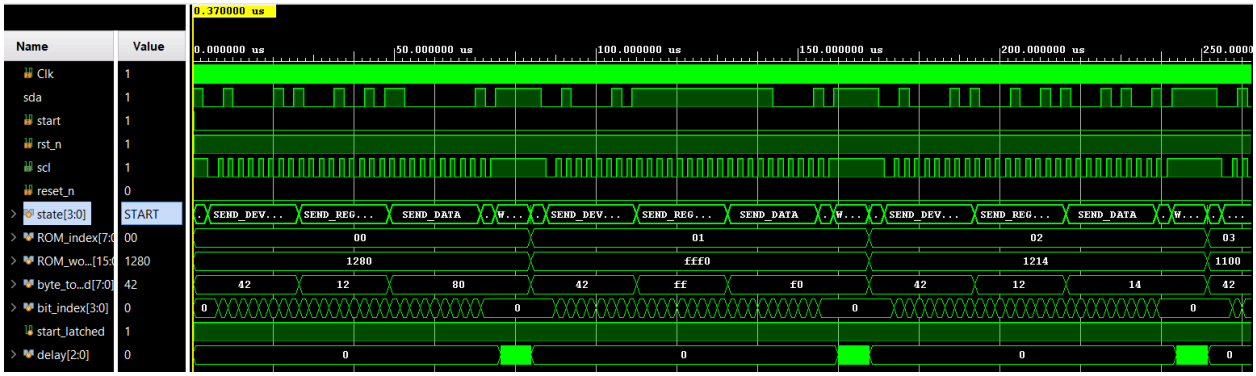


Figure 9: Simulation Waveform of I2C Driver

Figure 10: Simulation Waveform of Pixel Capture

Design Resources and Statistics:

LUT	**539**
DSP	**0**
Memory (BRAM)	**44**
Flip-Flop	**434**
Latches	**0**
Frequency	**126.36 MHz**
Static Power	**0.074 W**
Dynamic Power	**0.275 W**
Total Power	**0.349 W**

Conclusion:

In the end, our camera functioned almost perfectly, and we had an output on the monitor that we were very satisfied with, where the frame the camera was seeing was clearly visible on the monitor and not that pixelated. However, there was one bug that we couldn't solve, no matter how much we debugged. Our resolution and QVGA format of the camera translated well to the monitor screen, but the frame would always display some sort of on and off flickering on the top portion of the screen. After analyzing closer, it looked like the camera was rewriting over that part of the screen constantly, which resulted in the flickering. We thought it might be something with the code we wrote in our files, so we double checked that first. Then, we thought it might be something with the way the camera was configured, so we double checked those with the documentation sheet on the OV7670. We then thought it might be a problem with the write and read pointers being too close to each other, and so we added a buffering method called ping pong buffering, where we read and wrote alternatively from two brams essentially. However, none of these debugging methods fixed the flickering, and then we realized it was actually due to the frame rate, which was out of our control. The camera runs at 30 Hz, while the VGA to HDMI IP runs at 60 Hz, so there was a mismatch there that we couldn't fix without having a 30 Hz IP, which was impossible to do in the scope of our class's resources.

If we had more time, we were planning to extend our camera project to include edge detection. Making sure the camera displayed the correct output on the monitor was our first priority, and then we did some simpler image processing techniques like the inverter, greyscale, and alien filter effect. If we were able to have time for edge detection, we were planning to use a Sobel filter, which is essentially a convolution-based image processing algorithm that detects sharp intensity changes (edges) by computing gradients in the horizontal and vertical directions. This would have allowed us to increase our difficulty score as well.

Overall, the lab took us a good amount of time, around 25-30 hours total, to initially get the camera working and displaying the proper output on the monitor. However, there wasn't much help from the 385 resources, like TA office hours as many have not worked with interfacing with cameras, and did projects like video games with sprites. We had to spend a lot of time looking through documentation online and videos for the configuring of the camera, as well as the i2c driver which was different from AXI. A lot of our initial bugs came from not configuring the camera properly, because there were so many different types based on the type of resolution a user wants (QVGA, QQVGA, VGA). As a result, we had to keep messing around with the settings and use test outputs in a sophisticated and clever way to obtain the desired output.