VL504 Course Project

Veerendra S Devaraddi (IMT2018529)
Shubhayu Das (IMT2018523)
Sai Manish Sasanpuri (IMT218520)

30th December, 2021

Objective

In this project, our primary aim was to implement a video camera, whose output is displayed on a VGA monitor. To implement this, we used a breakout board of the OV7670 camera module and the Digilent Basys3 board, along with a VGA monitor.

Our main task involved:

- Implementing and testing a VGA controller(timing and coloring) on the FPGA
- Implementing the SCCB (I2C) protocol to configure the OV7670 camera module
- Implementing a parallel interface for transferring data from the OV7670 camera module to the FPGA and then displaying it on a VGA monitor.

We have completed all the tasks mentioned. We are not able to configure the OV7670 camera module properly due to poor documentation of the role of its registers. The presets found from a variety of sources fail to produce proper outputs for us.

Additionally, we have implemented and tested the following:

- A high-speed UART interface(1843200 baud rate) for transferring data between a computer and the FPGA
- Utilized a laptop's camera to capture video and display it on the VGA monitor

All the project files and the demo videos can be found here:
https://github.com/Shubhayu-Das/VL504-project

Implementation

VGA

Video Graphics Array is an analog video display controller standard, nowadays being replaced by other digital standards like HDMI and DisplayPort. The working of the protocol is explained in detailed in [reference 1].

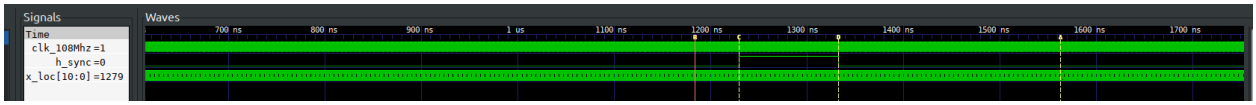
We implemented a VGA controller for a 1280x1024, 60FPS display resolution and update rate. We decided this on the basis of the clock frequency required to generate the timing pulses. The timing requirements can be found in [reference 2].

Here are the signals required in VGA:

Signals	Type	Description
HSYNC	Digital, timing	Synchronisation pulses to mark the end of drawing one **row** of the display
VSYNC	Digital, timing	Synchronisation pulses to mark the end of drawing one **frame**
RED, BLUE, GREEN	Analog, data	Each range from 0 to 0.7V, representing the intensity of the corresponding color in the pixel currently being displayed

The simulation testbench can be found in *simulations/vga/*.. Please note that *iverilog* was used for this simulation and the generated files are very large.

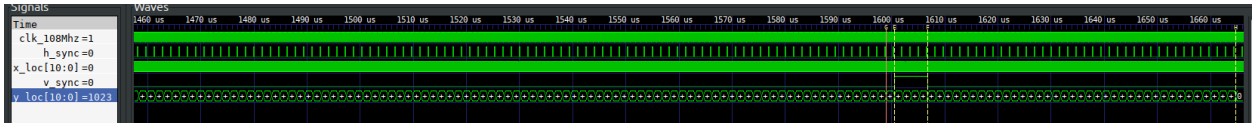
Here is a testbench simulation, showing the timing to display a single **row**:



The 4 vertical markers indicate the positions of critical timing events in the duration of showing a single row. These are:

1. Horizontal Visible area: 1280 clock periods
2. Horizontal Front porch: 48 clock periods
3. H SYNC pulse: 112 clock periods
4. Horizontal Back porch: 248 clock periods

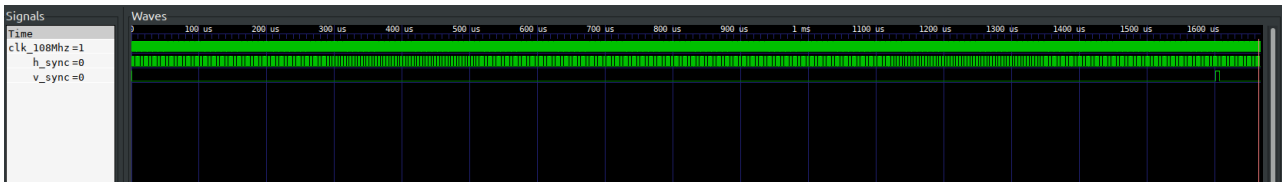
The following image shows the timing to display a single **frame**. Note that this image is zoomed in to show the relevant transition in the *v_sync* signal.



The 4 vertical markers indicate the positions of critical timing events in the duration of showing a single row. These are:

5. Vertical visible area: 1024 clock periods
6. Vertical Front porch: 1 clock periods
7. V SYNC pulse: 3 clock periods

8. Vertical Back porch: 38 clock periods



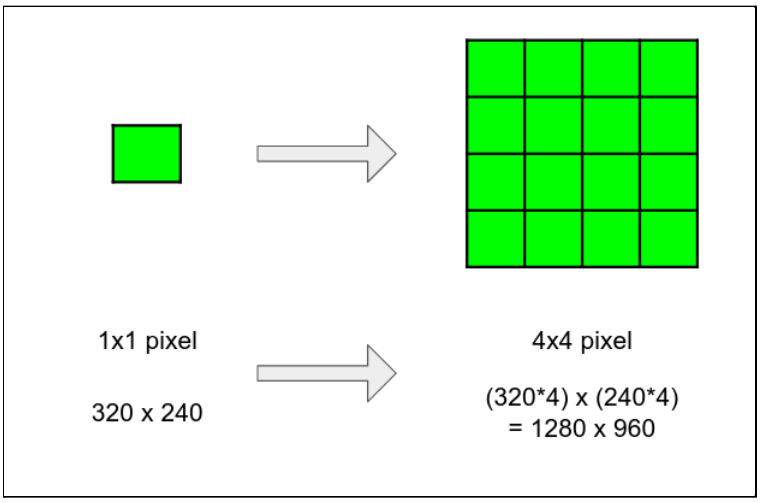
This image show the all the timing signals for showing an **entire 1280x1024 VGA frame**. The image is not clear due to the sheer amount of data contained. There are 1688 H SYNC pulses and a single V SYNC pulse, followed by the vertical back porch.

The color signals are generated using a 4-bit per color “DAC” (using switched resistors), which is described very well on pages 10-11 in the Basys3 Reference Manual.

Here is the reason for using a resolution of 1280x1024: the clock frequency required is exactly 108MHz, which is easily done using the MMCM. At 640x480, we need a clock frequency of 25.175MHz, which can be generated easily too(we just didn’t happen to use it). Our program is parametric, which makes it very simple to swap VGA configurations.

Our test monitor doesn’t support a 320x240 mode, so any image that we can store(see the storing section after this) needs to be scaled up. There are many fancy methods of doing this such as interlacing, interpolation, etc. Interpolation can become difficult because we might have floating point numbers and rounding off might not be good enough. This is something we need to explore in the future. Using HLS might make things easy.

Here is what we are doing now:



This is the simplest way to upscale, by repeating each pixel a certain number of times. Note that the final vertical resolution is only **960**, compared to the screen resolution of **1024**. So there is an *expected* black part at the bottom of the screen. Note that the image transmitted over UART

can easily be scaled to 320x256, but the image from the OV7670 camera cannot be scaled easily(there is some complex combination of registers to do this).

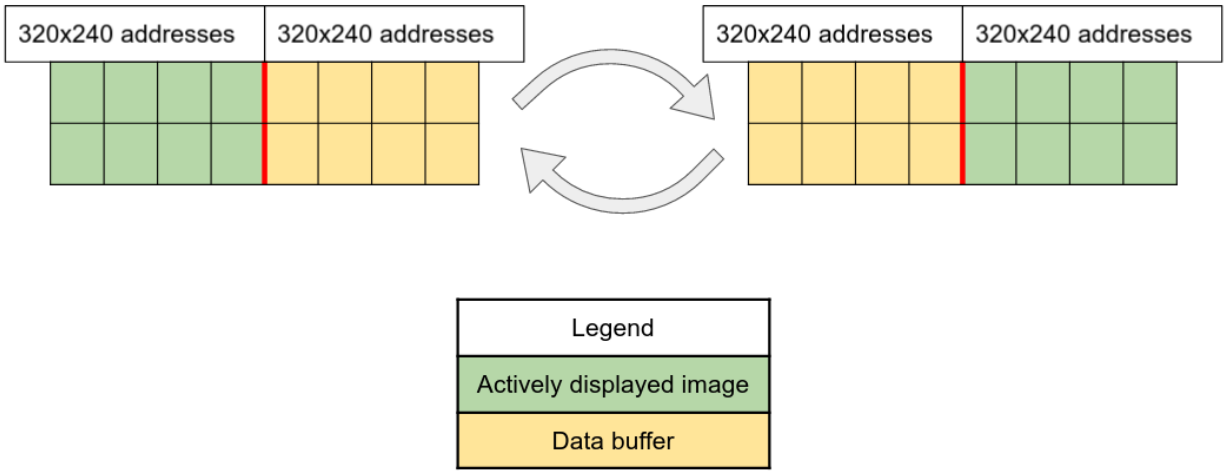
Storing and retrieving images

The Basys3(XC7A35T-ICPG236C) does not have enough space to store a full 1280x1024 image, with 12 bits per pixel color depth. The most that can be stored in block RAMs is

- **One** 320x240 image with 12-bit color depth(12 bits per pixel).
- **Two** 320x240 images with 11-bit color depth

We first tested our VGA module using some hard coded images, in the first block RAM configuration. The coefficient files for the block-RAMs were created using MATLAB to downscale the image and to reduce the number of colors used to form the image. This script can be found as *scripts/downscale.m*. We are not sharing the programs for this, as it is pretty much redundant and is not very relevant to the rest of our project. Needless to say, this worked well enough for us to go ahead.

We utilized the *second* configuration for showing a video on the VGA monitor. One of the images is actively displayed, while the second image space forms a video buffer. We write to this video buffer(over UART or the OV7670 module). Once the image is written, we display this image, and the space for the former first image is used as a video buffer.



This image explains how we have used the block RAM. Each address stores 11-bit pixel data. We store 4 bits of color data for red and blue colors and 3 bits for the green color.

R3	R2	R1	R0	G3	G2	G1	B3	B2	B1	B0

Data stored at each address location, for displaying each pixel. Note that the lowest bit of green(G0) is missing(not stored)

It must be noted that we are crossing clock domains using the block RAM. The data is written to the block RAM at a lower frequency that they are being read. Therefore a simple two ported block RAM must be used. Here are the specifications of the memory:

- **2*320*240 deep address** (corresponding to the maximum number of pixels)
- **11 bit width** (corresponding to the depth of each pixel)
- Simple two ported memory with one read and one write port

A circular buffer or FIFO **cannot** be used, because the previous image must be retained while the other image is being loaded into the video buffer. This is due to the low FPS of the OV7670 camera module/low data transfer rate over UART. Thus, at any given time, we need enough space to store 2 images at once.

When we are displaying the images, we zero-pad the green color in the LSB position. This is to ensure that we are outputting all 12 bits, as per our output mapping constraint.

R3	R2	R1	R0	G3	G2	G1	G0	B3	B2	B1	B0

Data read back from the BRAM to display each pixel. Note that G0 is always zero(1'b0).

VGA over serial

We reused one of our previous UART program(implemented by us, different from the one shared in class) to transmit images from our laptop's camera to the FPGA, and then displayed them on a VGA monitor.

X	R3	R2	R1	R0	G3	G2	X	X	G1	**G0**	B3	B2	B1	B0	X

The 11-bit data payload was split into two 8-bit UART packets, as shown here(separated by the red line). Note that G0 is actually transmitted, but not stored on the FPGA

We were able to achieve stable UART communication upto a baud rate of 1843200. At this rate, we can transfer a 320x240, 11-bit image in 0.8s.

To make this work, we wrote a Python script to read an image or from the laptop camera and send the data over UART. See *scripts/video_over_uart.py* for the program. We used the following Python libraries:

- Numpy - manipulating large matrices efficiently and fast
- Matplotlib - Displaying images(plotting things in general)
- Imageio[ffmpeg] - Accessing the laptop camera in the simplest way. Used ffmpeg in the background

- Pyserial - Transmitting data over UART

We can transmit any image using our Python program:

- Any image opened using numpy
- We take the files generated while make the coefficient files earlier and send those images. This serves as a control experiment to check if data is being sent properly over UART.
- Test patterns to check if the entire 11-bit color spectrum is being transmitted and used properly.
- Most importantly, the data captured from the video camera. We can even stream video files, but the amount of preprocessing required is time-consuming(downscaling in resolution, FPS, and colors).

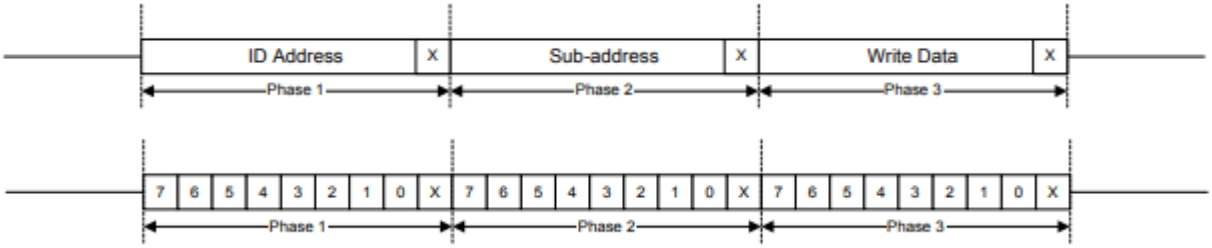
Inefficiencies: It is immediately apparent that we are wasting upto 5 bits - 4 empty bits and an extraneous bit. If we remove these and pack the bits better, we can easily make fewer UART transmissions. There are 2 reasons for us not optimizing this:

- This was just a test that one of us ran to check if we can receive and store the data properly. The structure of the packets was initially similar to the way the OV7670 module sends the data. The structure was changed later to reduce chances of loosing packets at high baud rates.
- We are using Python to process the image. Performing bit manipulations like packing multiple bytes is very difficult, if not outright impractical.

Given these factors, we did not bother optimizing the design. It is very simple to change things in the FPGA program(FSM), because bit manipulations are a breeze in Verilog.

SCCB protocol

Serial Camera Control Bus(SCCB) protocol is similar to the I2C protocol. It uses two or three wire implementations, which are a clock signal, serial data signal and an optional enable signal. It consist of one master and multiple slave devices which communicate over 3 phase write transmission cycle and 2 phase read transmission cycle.

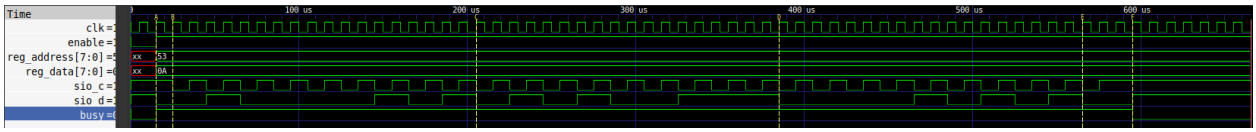


- Phase 1 is Slave device address ID which is used to identify a slave device and the operation(read/write).

- Phase 2 is sub-address which identifies a register of the slave device.
- Phase 3 is data which the master intends to write/read.
- The start and end of the transmission is asserted by a negedge and a posedge of data signal respectively when the clock signal is high.

In the case of OV7670, we use two wire SCCB protocol to configure the camera. Note that the communication has no acknowledgement

We implemented the SCCB protocol and simulated the 3-phase write transmission cycle for a two wire implementation efficiently. Our implementation for SCCB can be found in *src/our-sccb/sccb_configuration.v*. The simulation testbench can be found in *simulation/sccb/*.



Above simulation is an example for 3-phase write transmission cycle for,
Slave address = 0x42 (slave write address of OV7670)
Register address = 0x53
Register data = 0x0A

The 6 markers indicate the following

1. Start of 3 phase write transmission.
2. Start of phase1.
3. End of phase 1 and start of phase 2.
4. End of phase 2 and start of phase 3.
5. End of phase 3.
6. End of the 3 phase write transmission.

Outcome of our implementation and comparison with other implementations:

Even though we had taken known measures to avoid multi-driven net error while implementing the protocol, that is use cases/if blocks such that the corresponding cases in the always blocks do not modify the same variable given that the sensitivity list of always blocks are same, our code could not pass implementation because the always block used different sensitivity lists.

We compared our implementation with other implementations, and it turns out that others have used counters to avoid multi-driven net errors but this is inefficient and unreadable. Therefore we used this code for SCCB communication and modified it accordingly.

Configuring OV7670

Configuring OV7670 as per our requirement turned out to be difficult because of poor documentation. We tried using presets from other sources but it was unsuccessful.

We tried using an Arduino Uno for the initial configuration only, again all Arduino libraries for the camera are undocumented, and when we tried using it by understand the library code, it did not work.

We ended up using the camera with a wrong configuration as a result we are able to recognize only prominent objects like texts and logos on the monitor projection. We used this code as our preset during the configuration.

Reading data from OV7670

The camera needs an input system clock, XCLK. It produces an output clock signal PCLK which is used in latching the output data bus of width eight. That is at every posedge of the PCLK we receive 8 bits of data.

If the camera is configured to use RGB 565 format then it uses 16 bits to represent the color of a pixel therefore it would require two PCLK cycles to transfer the data of a pixel. Similarly for RGB555 and RGB444 we need two PCLK cycles to transfer the data of a pixel where some of the transmitted 16 bits remain don't care.

HREF is an output port and it is high throughout the transmission of data in a row.
VSYNC is an output port and is used to indicate the end of transmission of a frame.

Results

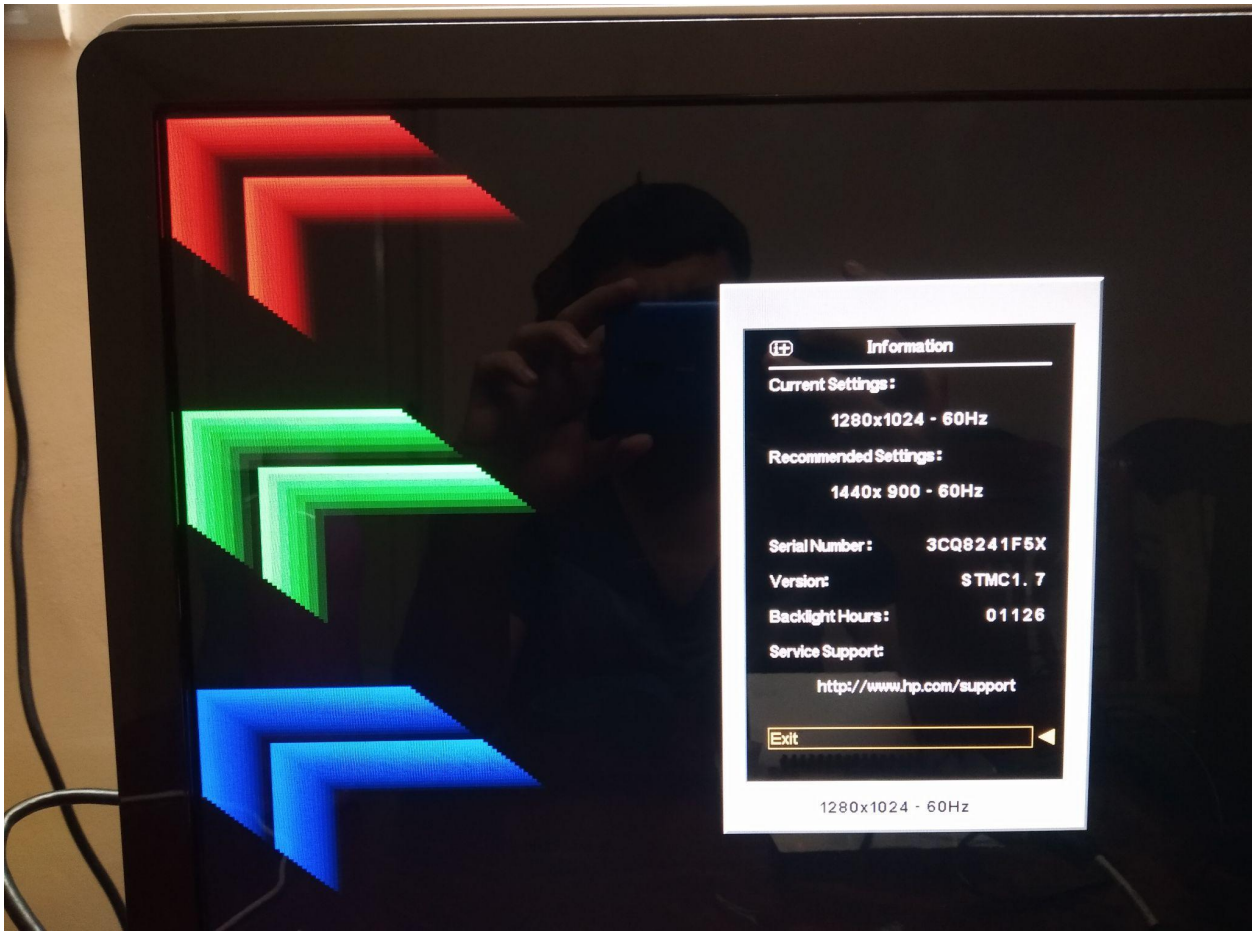
Images from the OV7670

The word Basys can be barely made out. The camera is pointing towards the Basys 3 logo.

The repeating image is an error in the memory address accessing, that is not an error in the OV7670 program.

Images over UART

Test pattern, showing the range of each color



This image shows the range of colors for red, green, and blue, individually. The monitor also shows that it is indeed a VGA monitor operating at 1280x1024 @ 60FPS. The colors are washed out due to the smartphone camera used to capture this image.

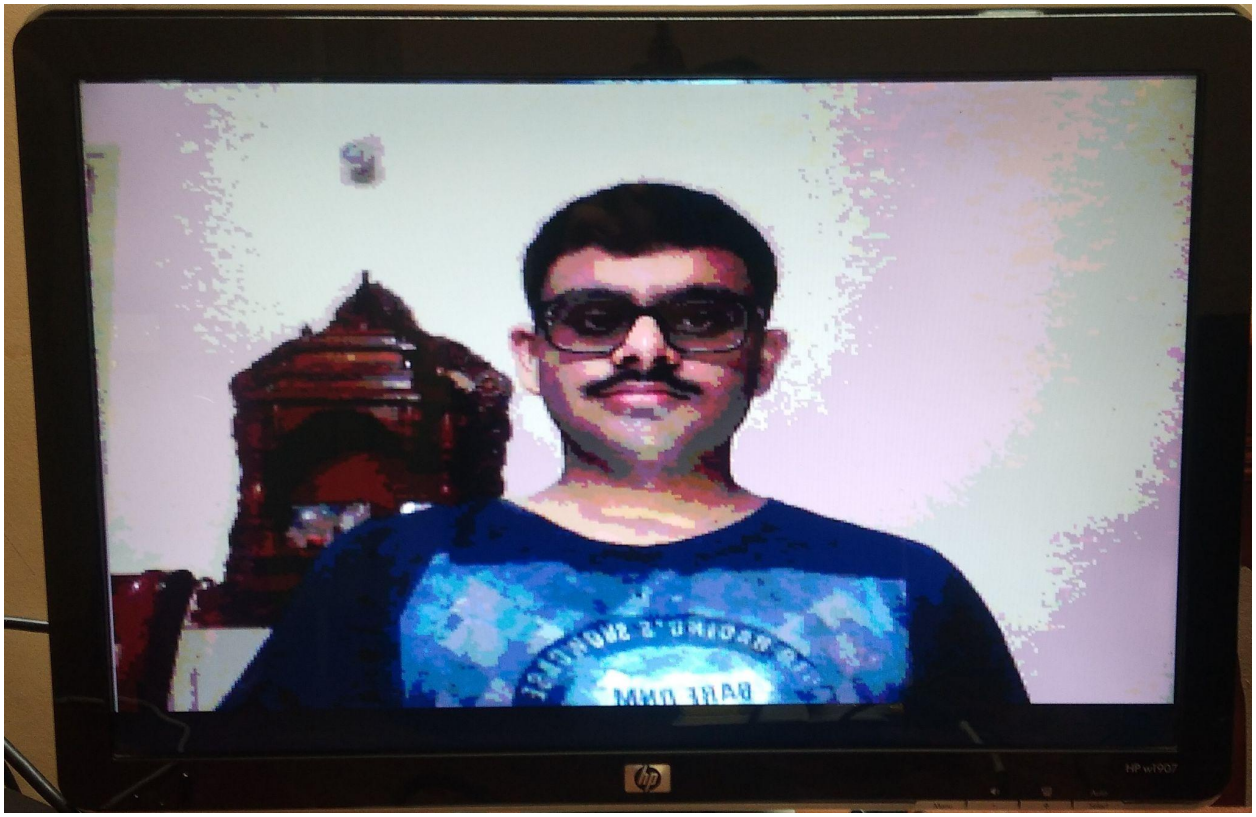
Test image without using a camera

.A comparison between an original test image and the image displayed on the VGA monitor:

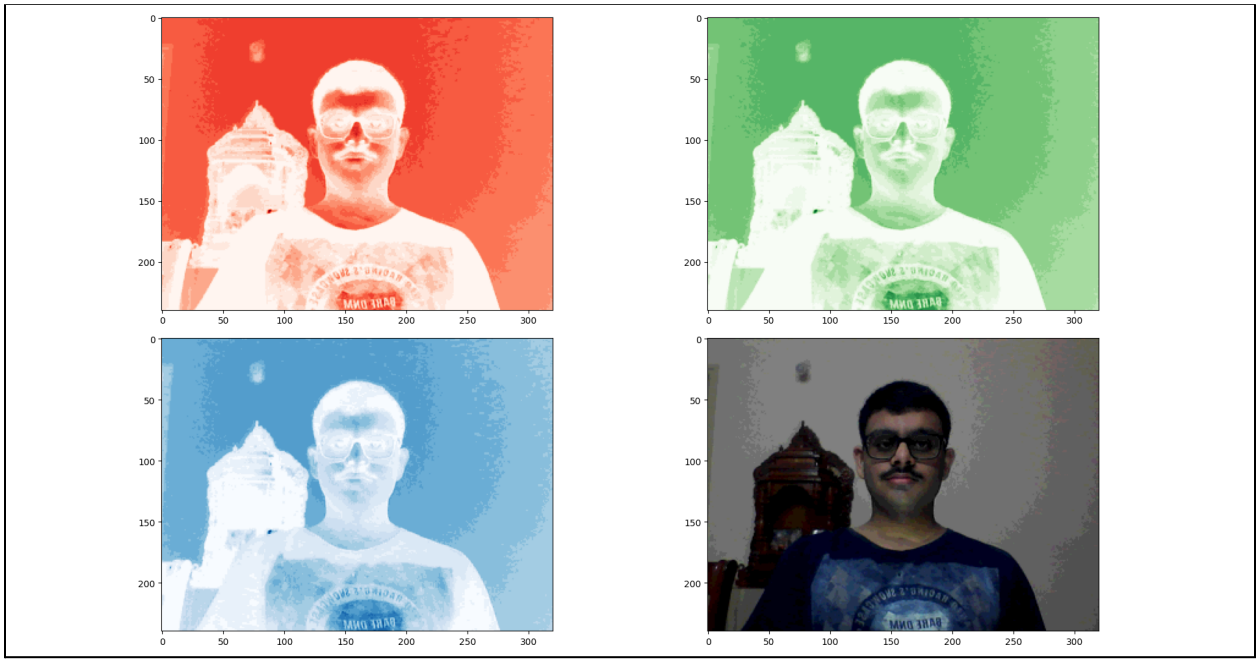
The original test image(see *images/vga-over-uart/test-original.jpg*). This is 1920x1080 image, with full 24 bit color per pixel. This image is taken from the BBC article: Seven Reasons to Love The Baobab Tree. We do *not* own this picture.

Downscaled and downsampled image(320x240, 11 bit color per pixel) on the VGA monitor. All the high level details are captured properly. However finer details are pixellated, because of our upscaling scheme. The color reproductions is reasonably good, but there are bands of colors in some places where the original picture has smooth transitions. An example of this are the tree trunks.

Image using the laptop's camera

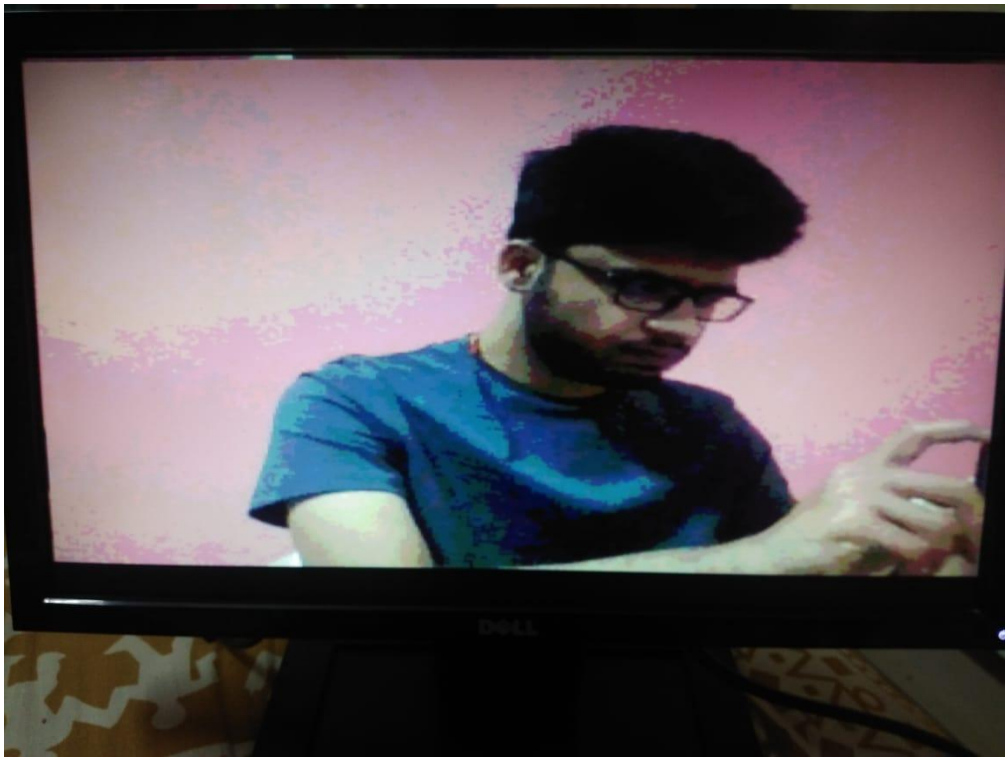


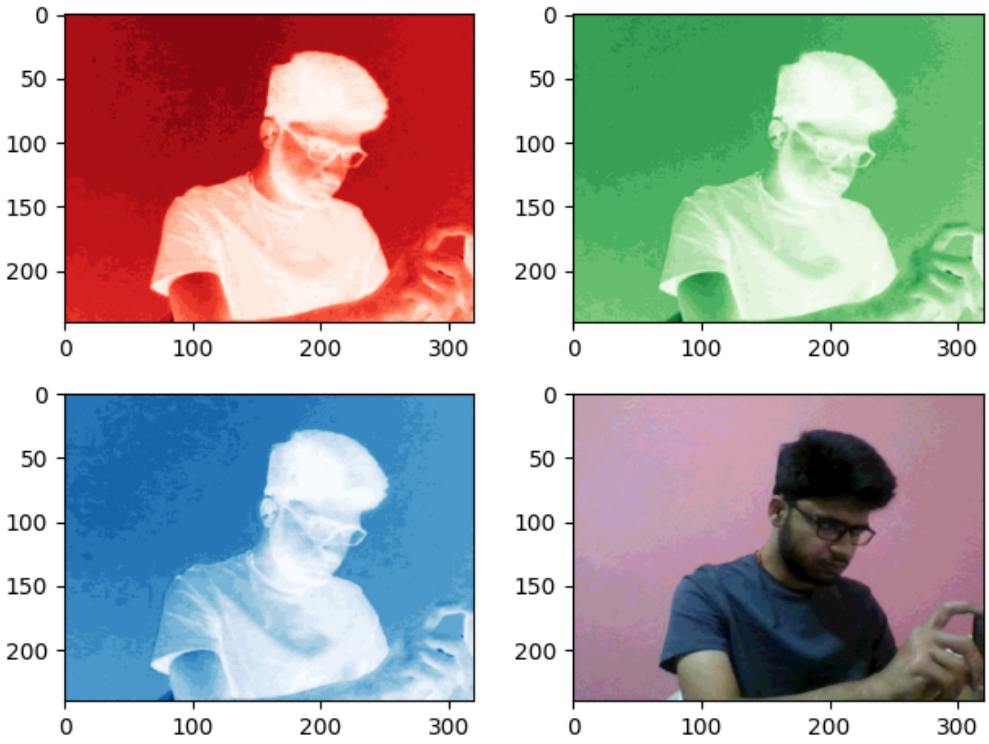
The image is reasonably well reproduced, given that the laptop camera does not have proper lighting(which is why one side of the face is darker).



A plot of the red, green, and blue components along with the final image, as shown on the computer. This image uses 12 bits per pixel, and is of the same resolution.

The banding due to lack of color depth(only 12 bits per pixel or 4 bits per color per pixel) is very apparent in these images.





Future Work

1. Finding better documentation for the OV7670 module or using a better documented camera module.
2. The test image was downscaled in MATLAB, with *dithering.* Dithering is a way to reduce sudden image boundaries when reducing the pixel depth(from 24 bits per pixel to 11 bits per pixel). However, the image from the cameras do not have dithering, which make the color boundaries prominent.

   We can try implementing Floyd Steinberg dithering to help improve the image quality. This can be done on the FPGA to speed things up.

3. The 320x240 image on the FPGA can be upscaled before displaying using interpolation. Bilinear/Bicubic interpolation schemes can be implemented on the FPGA(preferably using HLS).

4. Stream compression algorithms like LZW-4 can be implemented to reduce the total number of transactions/data stored. We successfully implemented the LZW-4 stream compression/decompression scheme in Python, but didn't find the time or motivation to port it over to Verilog(efficiently).(this was a whole project in itself)

5. Further image processing can be added to the image, like rotation, zooming, color variations etc.

Bonus

The creators of this project, in full 1280x1024 VGA glory:
Sai Manish, Veerendra, and Shubhayu*(left to right)*.

References

1. “The world’s worst video card?” by Ben Eater on YouTube
2. VGA signal timing - http://tinyvga.com/vga-timing
3. Basys 3 reference manual and PCB schematic - under “Support”
4. SCCB functional specification
5. OV7670 datasheet
6. OV7670 implementation guide
7. Westonb/OV7670-Verilog on GitHub