Lab 4 Project Report: Video Player

Aneesh Bonthala, Jacob Cunningham, Vincent Lin

Introduction and Requirement

For our project, we implemented a video player on the spartan-6 FPGA board. The board displays the video to a monitor connected through VGA. The video file is preloaded onto the device by the user; the file is originally a .mp4 file converted to an array of 8-bit VGA color-encoded pixels per frame, which plays in 40x30 px resolution at 2 frames per second.

The video player runs the video at this default number of frames per second unless the user wishes to change the speed. Using the switches of the FPGA board, the video can also be played at either 0.5x speed (1 frame per second) or 2x speed (4 frames per second). Additionally, a button on the FPGA will be used to pause/play the video. This button simply causes the video playback to toggle between playing and stopped.

Design Description

top Project Status (03/16/2023 - 15:17:08)			
Project File:	lab4-2.xise	**Parser Errors:**	No Errors
Module Name:	top	**Implementation State:**	Programming File Generated
Target Device:	xc6slx16-2csg324	**•Errors:**	No Errors
Product Version:	ISE 14.7	**•Warnings:**	48 Warnings (40 new)
Design Goal:	Balanced	**•Routing Results:**	All Signals Completely Routed
Design Strategy:	Xilinx Default (unlocked)	**•Timing Constraints:**	All Constraints Met
Environment:	System Settings	**•Final Timing Score:**	0 (Timing Report)

Device Utilization Summary	[+]

Performance Summary			[-]
Final Timing Score:	0 (Setup: 0, Hold: 0)	**Pinout Data:**	Pinout Report
Routing Results:	All Signals Completely Routed	**Clock Data:**	Clock Report
Timing Constraints:	All Constraints Met		

Detailed Reports					[-]
Report Name	**Status**	**Generated**	**Errors**	**Warnings**	**Infos**
Synthesis Report	Current	Thu Mar 16 15:14:30 2023	0	44 Warnings (40 new)	1 Info (1 new)
Translation Report	Current	Thu Mar 16 15:14:35 2023	0	0	0
Map Report	Current	Thu Mar 16 15:15:22 2023	0	1 Warning (0 new)	7 Infos (0 new)
Place and Route Report	Current	Thu Mar 16 15:16:36 2023	0	3 Warnings (0 new)	3 Infos (0 new)
Power Report					
Post-PAR Static Timing Report	Current	Thu Mar 16 15:16:42 2023	0	0	4 Infos (0 new)
Bitgen Report	Current	Thu Mar 16 15:17:06 2023	0	0	0

Secondary Reports			[-]
Report Name	**Status**	**Generated**	
WebTalk Report	Current	Thu Mar 16 15:17:07 2023	
WebTalk Log File	Current	Thu Mar 16 15:17:08 2023	

Date Generated: 03/16/2023 - 15:22:08

Figure 1. Design report as generated by the Xilinx ISE

Our project was split into three submodules to handle the main tasks. We connected these via a top level file. The modules were split into a clock divider, vga controller, and main logic for drawing frames. The top module was used to also route the correct hardware components like buttons, switches, and main 50MHz clock into the correct modules as well. The clock divider was mainly used to create a vga sync pulse and frame counting pulses. The vga controller was used to correctly time which pixel to display color to. The draw_new module contained the logic to handle drawing frames at different speeds. The frame information was encoded into the file as well.

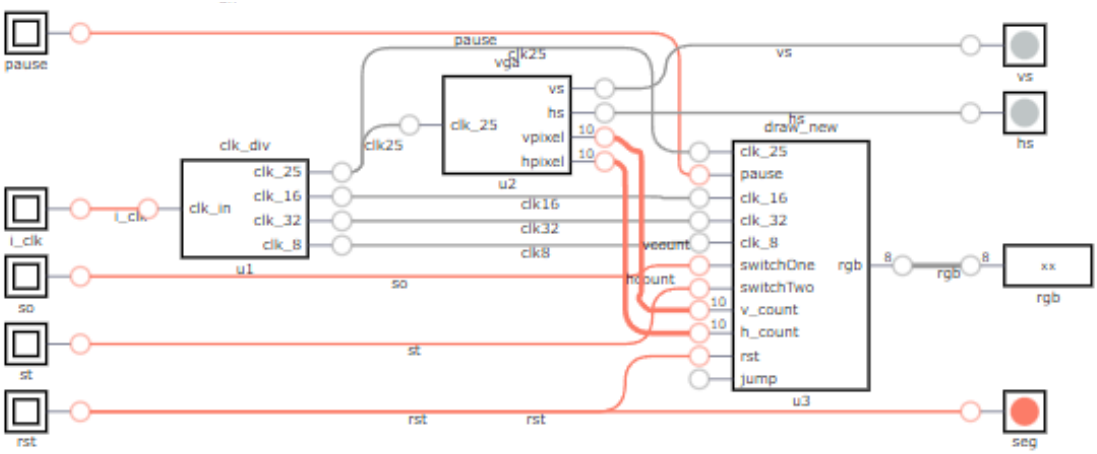

Figure 2. Top module schematic

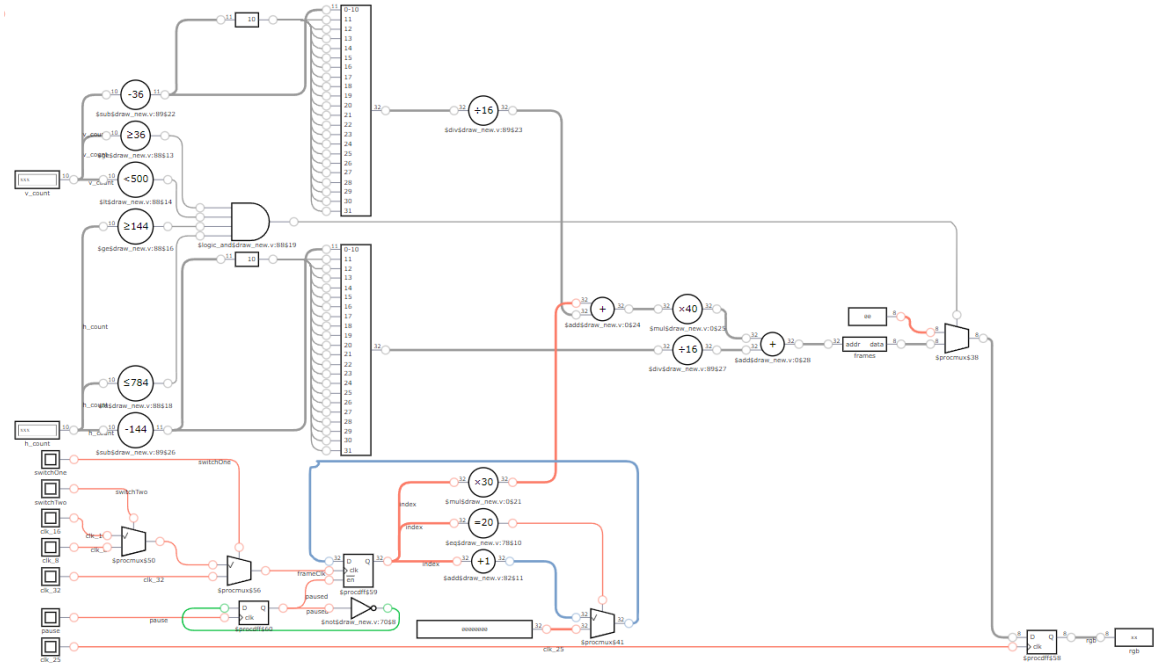

Figure 3. Draw module schematic

The draw_new module was the core logic to the video player program. This module contained the code to figure out which frame to display and send the correct rgb color for the correct pixel. For this we needed the x,y pixel coordinate from the vga module which was piped in via a port v_count and h_count. These were used to display the correct pixel. Each frame was held in a 3-d array, so we used these values to access the correct pixel color. We also piped in the user control like switches and buttons to this module. The speed control was selected using the switches. We had a switch for slow mode and a switch for fast mode, .5x and 2x respectively. The base display rate we ended with was 2fps or 2Hz. We would pick the current clock to a register value that would then act as the clock for how fast frames incremented. We simply looped the video so the frame counter just incremented the frame index and overflowed to 0. For displaying to the vga port, we assign to a 8 bit register rgb, with 3 red, 3 green, and 2 blue bits. The vga screen size was 640x480, but our video was encoded to a smaller 40x30 and then expanded to save space. When displaying the pixel count was used to select the pixel, however we also needed to factor in the vertical and horizontal front and back porch in order to display correctly. This was mainly to add a buffer between the vsync and hsync pulses so that data won't be cut off. To display we also updated the rgb register on the 25MHz pixel clock.

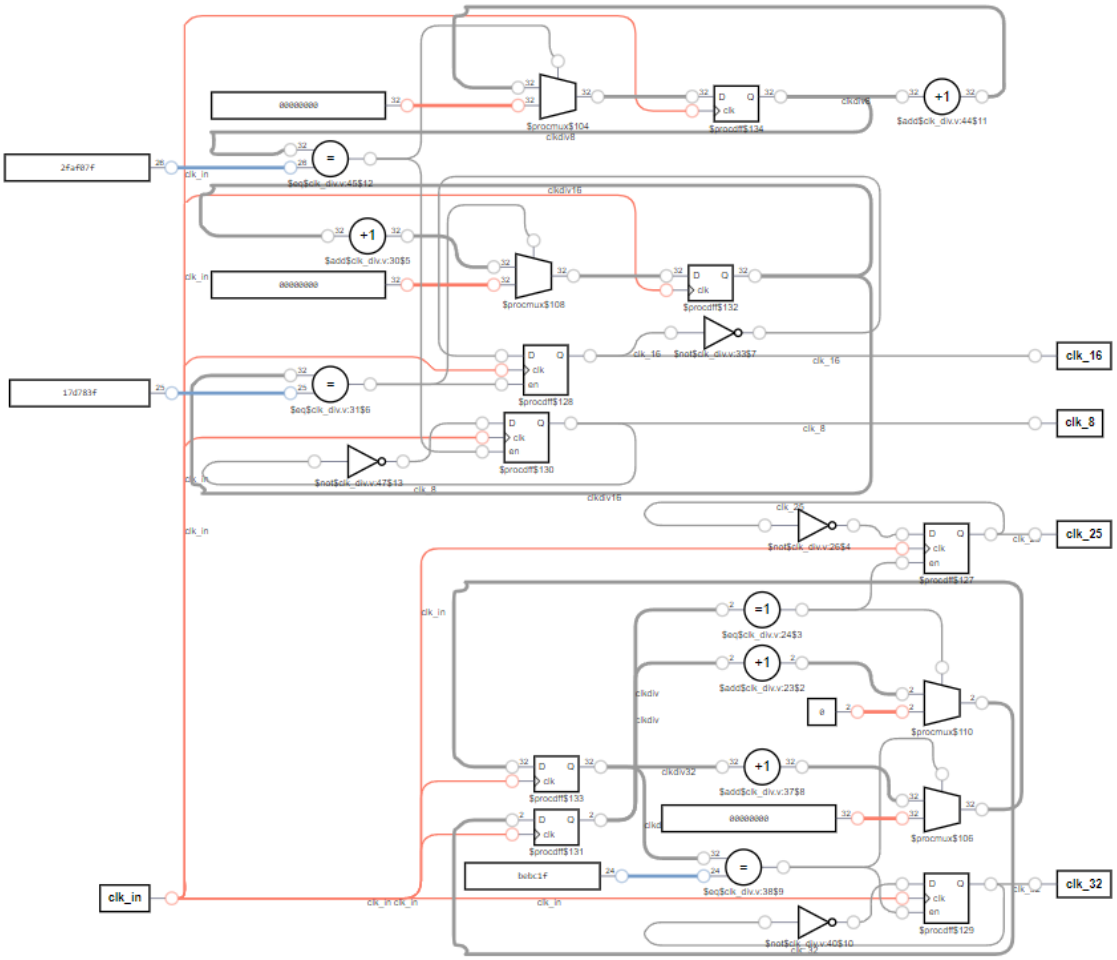

Figure 4. Clock divider schematic

The clock divider was used to split the main 50MHz fpga clock into the 4 used clocks. The main clock ran at 50 MHz whereas we need clocks in the Hz range. Therefore, we designed a simple counter that incremented on clk posedge. For the base frame clock, we needed a 2 Hz clock such that the stopwatch would keep time at the rate we would expect. To achieve this effect, we counted 24999999 master clock cycles and overflowed to 0 on the last count, inverting the state of our custom 2 Hz clock. Thus, we got a clock with duty cycle 50% with a period of 2s. We repeated this technique for the other clocks to get the 25 Mhz pixel clock and faster and slower mode clocks at 1Hz and 4Hz.

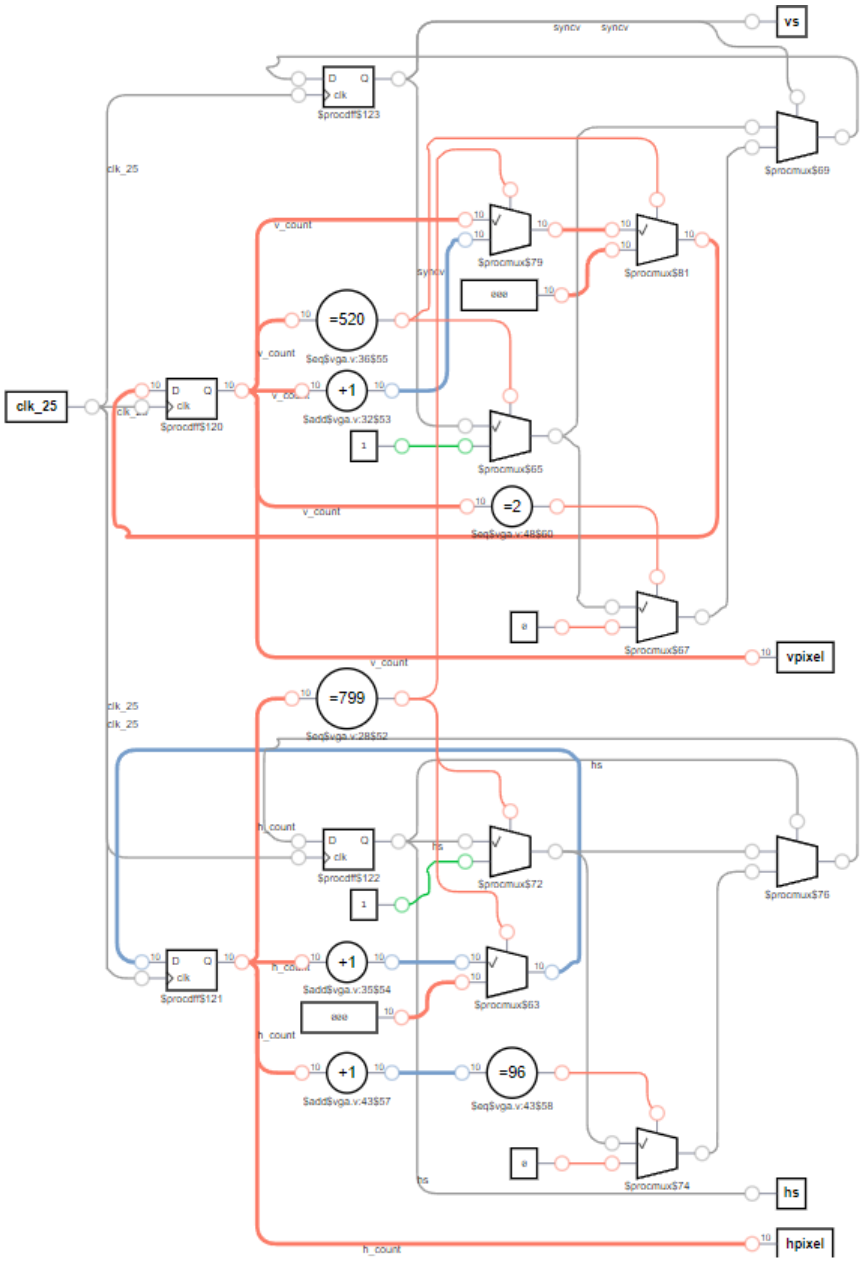

Figure 5. VGA controller schematic

The vga controller was used to create the needed pixel location and hsync and vsync clocks. We wanted to display at 640x480, which is what the monitor supported. To do this we created counters for 800 horizontal and 521 vertical. This is because we needed to factor in the porch sizes for the display as well. The incrementer would increment on the 25MHz clock pulse, and be reset at 799 and 520 back to zero. For the hsync and vsync pulses, these were used to tell the monitor when to display the row and to move to the next. For these we used a horizontal pulse width of 96 and vertical pulse width of 2 as per the spec on the diligent website.

Simulation Documentation

Figure 6. Working display during project demo. This is a processed frame from an arbitrary video about quesadillas we ended up using: https://www.youtube.com/watch?v=Vj4Y1c-DSM0

Because this project worked with a physical FPGA and its peripherals, there was no need for the simulation or waveform features of the Xilinx ISE.

To prepare the video data to upload to the FPGA, we used Python scripts to automate the conversion from arbitrary MP4 files to a binary format with the video properties we wanted. The following graphic illustrates a high-level overview of the file formats and video properties used as an input video is processed into a form appropriate for our use case:

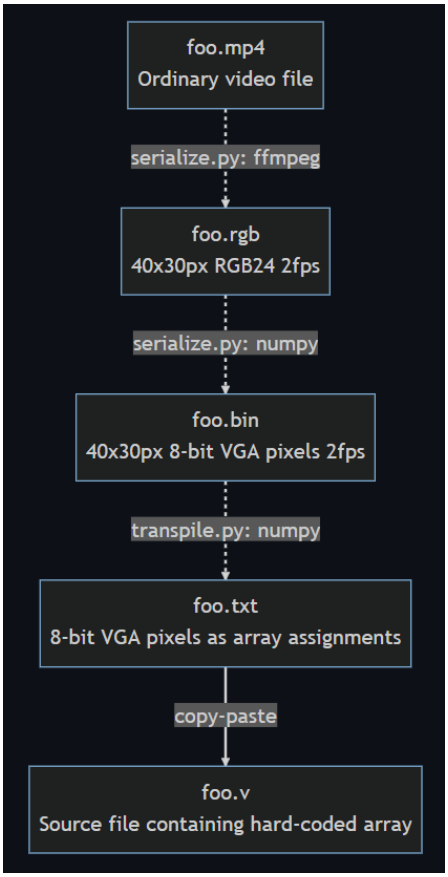

Figure 7. The different formats into which an input MP4 file is transformed as it goes through our custom converter suite

Firstly, the serialize.py script takes in an input MP4 file and uses the FFmpeg backend to convert it into the raw RGB24 format - that is, where every 3 bytes represent the red, green, and blue color values for that specific pixel. We needed this raw format because we ultimately need to transmit the value for every individual pixel through the VGA as each frame is rendered, and it makes more sense to perform the decompression on more familiar software beforehand instead of on the FPGA hardware where memory and complexity constraints become a concern. However, this intermediate is not yet compatible with VGA because VGA encodes each pixel as a single 8-bit value (rrrgggbb), not RGB24. Thus, serialize.py then makes use of the numpy library to perform vectorized bit manipulation to convert RGB24 into 8-bit color for all pixels of all frames. The resulting *.bin file was a binary file encoding the 8-bit color pixel values for all 40x30 compressed pixel frames at 2 frames per second.

```python
def compress_rgb_frames(frames: np.ndarray) -> np.ndarray:
    # Vectorized operations go brr.
    r = frames[..., 0]
    g = frames[..., 1]
    b = frames[..., 2]
    # Compress 24-bit RGB to 8-bit RGB by taking the most significant
    # bits of each channel.
    r_slice = (r & 0b11100000) >> 0
    g_slice = (g & 0b11100000) >> 3
    b_slice = (b & 0b11000000) >> 6
    # Concatenate the components.  This also flattens the frames array
    # from 4D to 3D by converting the pixel axis into uint8 scalars.
    compressed_frames = r_slice | g_slice | b_slice
    print("Compressed 24-bit RGB channels into 8-bit scalars.")
    return compressed_frames
```

Figure 8. Python function for converting from RGB24 to the 8-bit format compatible with VGA.

The *.bin file was initially the final product for this preprocessing stage. We were under the impression that we could upload this file into some kind of ROM segment on the FPGA, and the main process could then load from it one frame at a time to render pixels to the screen. However, this proved more complex than what could be completed in the timeframe of this project as it involved not only figuring out how to interface with the ROM but also implementing our own memory manager for it. We ended up using a crude workaround. Because we knew how to upload Verilog source code, we would simply hard-code the entire video as an array in the draw_new.v source file. Obviously, it would be impractical to manually write out all 24000 pixel values by hand. Thus, we tacked on another Python script to our converter suite called transpile.py, which took a *.bin file from prior steps and converted it into Verilog array assignments. We would then copy-paste the output into draw_new.v.

```
reg [7:0] frames [0:19][0:29][0:39];
initial begin
frames[0][0][0] = 8'hb1;
frames[0][0][1] = 8'hb1;
frames[0][0][2] = 8'hb1;
frames[0][0][3] = 8'hd1;
frames[0][0][4] = 8'hd1;
frames[0][0][5] = 8'hd1;
frames[0][0][6] = 8'hd1;
frames[0][0][7] = 8'hd5;
frames[0][0][8] = 8'hd5;
frames[0][0][9] = 8'hd1;
frames[0][0][10] = 8'hd1;
frames[0][0][11] = 8'hd5;
```

Figure 9. Snippet of draw_new.v at the time of demo. A video ended up being implemented as a 3D array of 8-bit values. This example shows 20 total frames of 40x30 resolution. The assignments generated by transpile.py would continue for all 20*40*30=24000 pixel values.

Because we were working blindly for the most part at this stage, we also wrote a visualize.py script to make sure we were on the right track. It made use of the cv2 and matplotlib.pyplot libraries to render our processed binary representations into images. This way,

we could check if our intermediate files were encoded with the right bytes in the right places and that our color compression algorithm produced something that was still recognizably similar to the source video. This was also useful in isolating one of the major challenges we faced regarding color. We encountered a bug where we were able to display pixels to the screen via VGA, but the color and offset were all warped. We were confident that our color encoding and UCF mappings were compliant with the VGA interface, so after referencing the provided VGA demo in the assignment spec, we found that our issue was elsewhere. In summary, we were incorrectly handling the porches in our signal processing, causing the main loop to read from the middle of each 8-bit value, resulting in the corrupted display we observed.

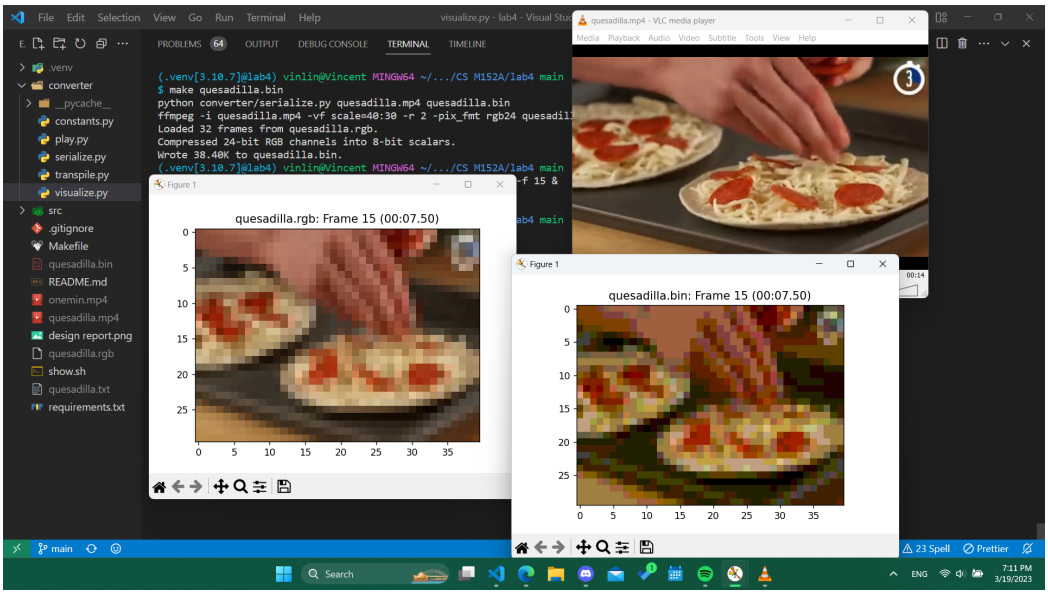

Figure 10. Top-right: the corresponding frame in the original video. Left: the compressed RGB24 version. Bottom-right: the *.bin version with 8-bit compressed color.

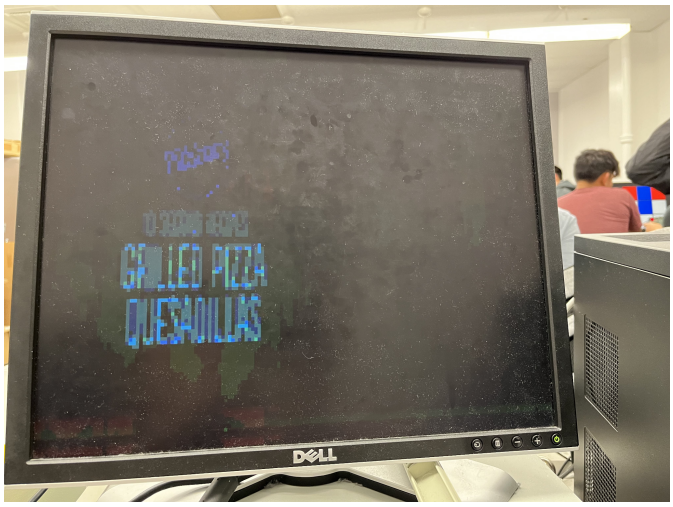

Figure 11. Warped display caused by improper handling of VGA porches.

The advantage of our converter suite became clear when it came time to put the parts together with the Xilinx ISE and physical FPGA during lab sections. Expectedly, our hard-coded array assignments massively bloated the code size. This made compilation time impractically long, and we even encountered an out-of-memory error once. In the effort to just get it working, we incrementally sacrificed video properties such as resolution and framerate. We expected memory to be a challenge from the start, so we originally strived for 160x120 px and 16 FPS, a marked deterioration from the 640x480 viewport of the monitor, but this was already too much memory for the compiler to handle. We experimented with lower and lower quality until we got a compromisable compilation time, at 40x30 px and 2 FPS. Because these updates were just a matter of changing a few constants defined in the converter suite, we were able to change our video quality many times on demand by simply changing a few variables and rerunning our Makefile frontends. This assisted us tremendously in completing the lab on time.

Conclusion

Our project taught us how to use a Nexys 3 Spartan 6 FPGA and VGA to not only render pixels to a monitor but also update them on a clock to produce the effect of a video player. We also utilized switches and a button on the board to implement some basic features of a video player, such as pause/resume and different playback speeds. Our implementation can be divided into two major parts. Firstly, in the Python preprocessing part, we wrote extensive project-specific tooling in the form of our custom video converter suite. This provided a toolchain that abstracted the on-demand conversion from an arbitrary MP4 file to a format that can be used in our Verilog code. The second part is the Verilog code consisting of the actual video player logic, such as the clock divider for controlling playback speed and a main loop that interfaces with user input and the VGA.

We encountered two major challenges during this project. Firstly, we had a bug where we were able to display pixels to the screen via VGA, but the color and offset were all warped. Using a combination of our converter suite and the reference demo, we were able to isolate the cause as improper handling of the VGA porches. Secondly, we struggled with memory and time constraints of the Xilinx compiler. Because we were not able to utilize the FPGA's ROM, we settled for a "brute force" approach of hard-coding pixel values as Verilog arrays. This was very memory-intensive, so we ended up having to massively cut down on video quality such that our program could compile in time for testing and demo. Thus, the resulting display was not as crisp as we were hoping, but it still fulfilled our specifications of a video player.

The attached VGA demo was a helpful resource for groups utilizing VGA like ours. However, I think it would also be helpful to link resources for specific FPGA features such as the different memory segments or in general how to use the Nexys 3 documentation. We also noticed there were often not enough working FPGAs to go around, especially during busy office hours where many groups had to line up in hopes of completing their project on time. We hope that future offerings of this course can remedy this by updating inventory and maybe increase lab availability through more office hours, etc.