Justin Sapun
Arun Guruswamy

ENGS 128 Final Project Report

Introduction

The goal of this project was to process audio signals from DDS or I2S passthrough using FFT and identify the peak frequencies of the audio in order to manipulate the HDMI output accordingly. The peak frequency dictated what color the moving object generated on the screen would be. The audio system was implemented the same way it was for the previous labs and the video processing followed the template provided by Digilent's HDMI project. Video pixels were generated by using the pixel clock and control signals from the Video Timing Controller. The inputs for this project include a mux select that chooses between DDS or I2S Passthrough for audio signals, and a mute enable that toggles the audio's mute state. There is also AXI Lite input for the DDS from the previous lab.

Github Repo Details:

Github Link: https://github.com/arun-guruswamy/128-final-prj/tree/main

Hardware Design:

New Sources (found in hw/sources_1/new):

- audio_passthrough.vhd: Primary wrapper responsible for audio generation from DDS or I2S passthrough.
- video_gen.vhd: Responsible for generating the pixels that will be transformed by video_transform before being displayed on the screen. Uses the video timing controller for timing and the dynamic clock generator for the pixel clock.
- axi_fifo.vhd: A custom fifo compliant with axis protocol, that buffers 512 samples for the FFT, and handles multiplying coefficients as part of our Hanning Window implementation.
- video_transform.vhd: A wrapper for rgb_transform and fft_axi_rx and FFT IP core that helps connect each of them
- fft_axi_rx.vhd: A modified version of the standard AXI receiver from previous labs. It receives frequency data from the FFT IP Core and identifies the peak frequency bin which it outputs to rgb_transform.
- rgb_transform.vhd: Receives video data from video_gen and frequency data from fft_axi_rx. Uses both of these inputs to make sure it changes the color of the pixels for moving blocks only.

Old Sources (found in hw/sources_1/imports)**:** Existing sources imported from previous labs.

Hardware Simulation (found in hw/sim_1/new)**:** Relevant testbenches created.

- tb_vivado_fft.vhd
- tb_video_transform.vhd
- tb_video_transform.vhd
- tb_video_gen.vhd
- tb_rgb_transform.vhd
- tb_general.txt
- tb_fft_axi_rx.vhd
- tb_axis_fifo.vhd

IP Cores (found in hw/sources_1/ip)**:** A folder containing .xci files of Vivado IP core blocks.

Software (found in sdk/)**:** Integration of the lab 3 AXI LITE DDS serial control and the given HDMI simple demo display sdk package.

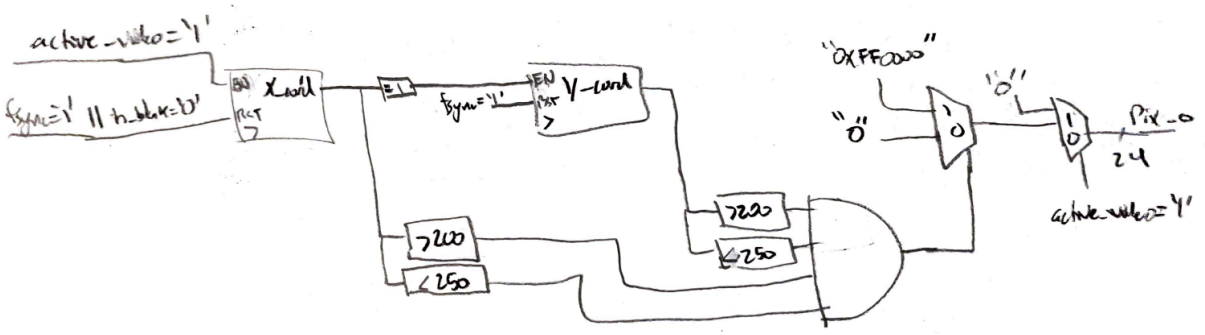
Matlab (found in matlab/)**:** Coefficient and coefficient generating files for 3 memory blocks for dds, rgb colormap, and the han window.

Task 1: Video generation

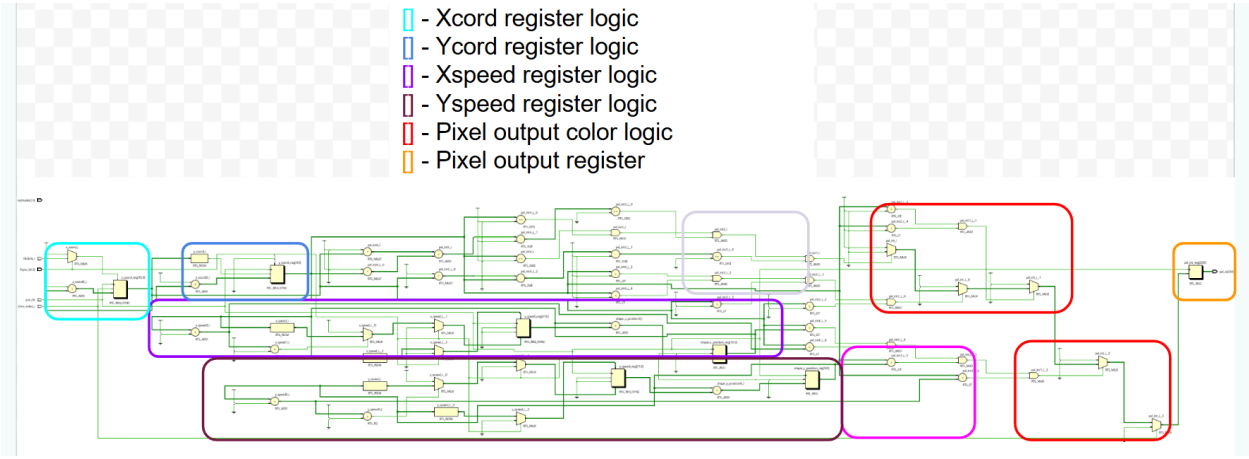
The first past of the project involved being able to generate pixels dynamically using the pixel clock and video timing controller. These pixels were generated with the goal of creating a simple scenery in the monitor and having a moving square similar to a screensaver logo. This moving square is colored a specific color that allows the subsequent video processing to identify it uniquely and color it depending on the peak frequency of the audio data coming in.

Paper Design

Here is our elementary paper design for the video generator. It consists of two pixel counters for the x and y cord, and a condition in another process which constantly checks the current pixel and if it falls without a set boundary. If it does, we change the color of it, otherwise, it stays black. We later improved the design to include a moving square, which required the addition of more registers to continuously update the position in each axis. We also decided to include a simple visual of grass and a tree, for fun.



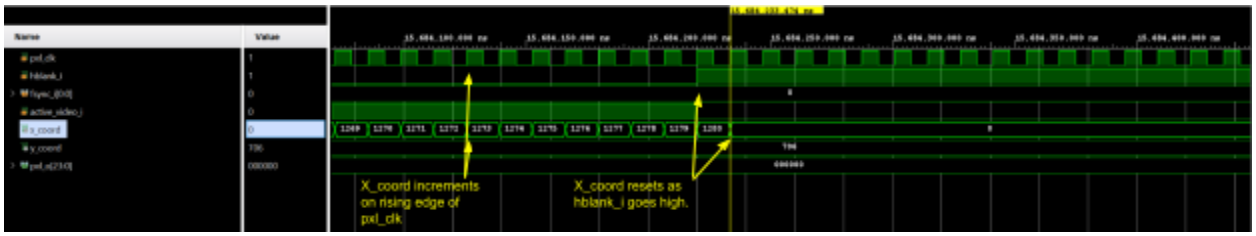
Elaborated Design



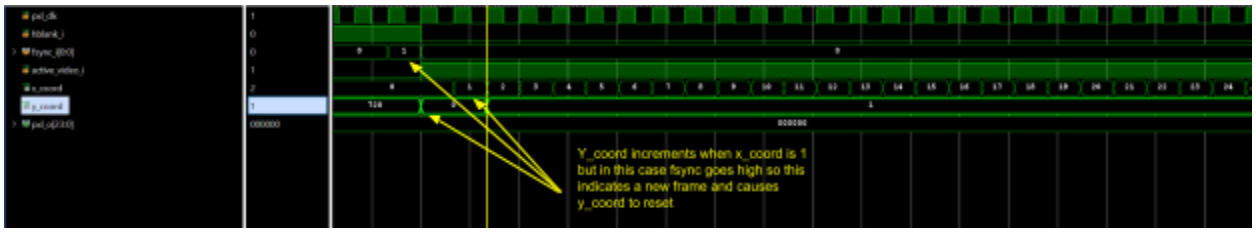
Annotated Screenshots for Simulation of Video Generation

The first screenshot is just a high level zoomed out screenshot that shows how the coordinates are updating in response to the main control signals like active_video_i and hblank_i. It can be seen that x_coord changes quickly when active_video_i is high as it is updated for every pixel clock cycle. It can also be seen that the y coord only updates every time hblank_i goes high. Pixel output only changes if the x and y coords are within the range of the shape.

The second screenshot shows a zoomed in version where X is resetting and also shows it updating every pixel clock cycle.



Similar to the previous screenshot this one shows Y resetting.

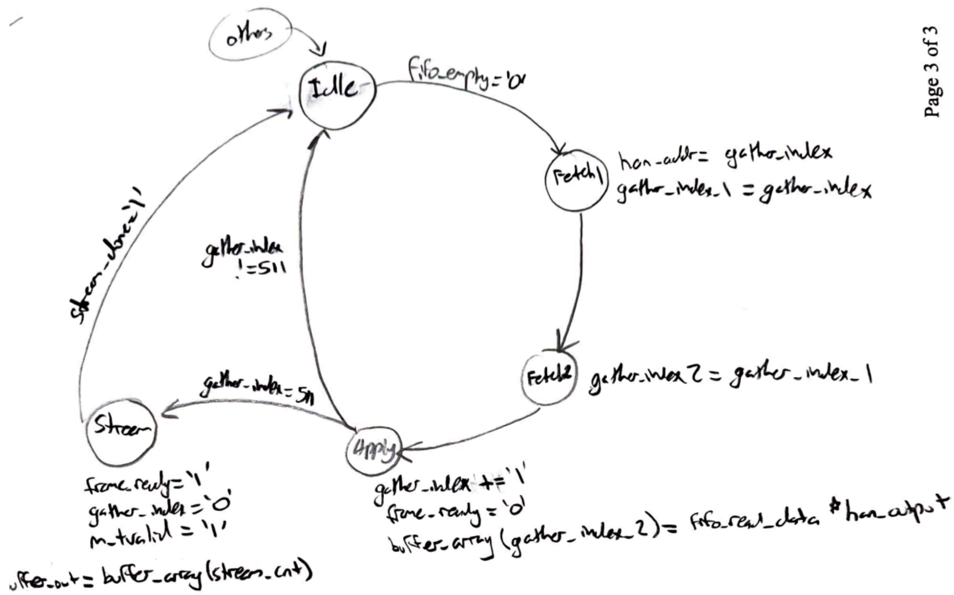


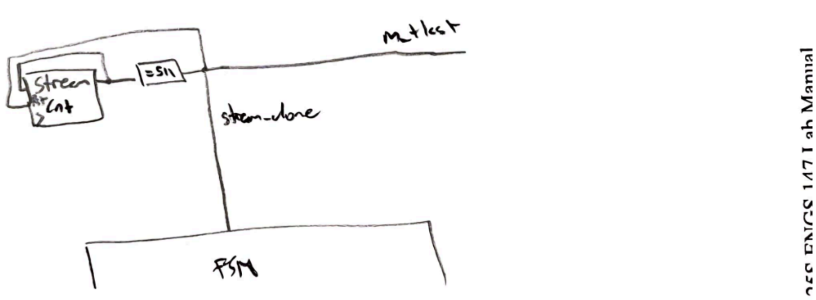
Task 2: Audio Input to the FFT (axis_fifo)

In this task, we set out to transform real time audio data into the frequency domain to gain meaningful information using an FFT IP core. From debugging, we discovered that the FFT required a burst input of sample data, meaning you could not send data to the FFT using the 48khz audio sampling rate. We found the easiest approach was to modify our FIFO to buffer 512 samples at the 48khz sampling rate of our audio data, and send that continuously to the FFT on the axis clock, running at 100Mhz. We also had to set up a configuration protocol for the FFT, which we will discuss later. The conversion from the FFT enabled us to identify dominant frequency bins, which we later used for video color changes.

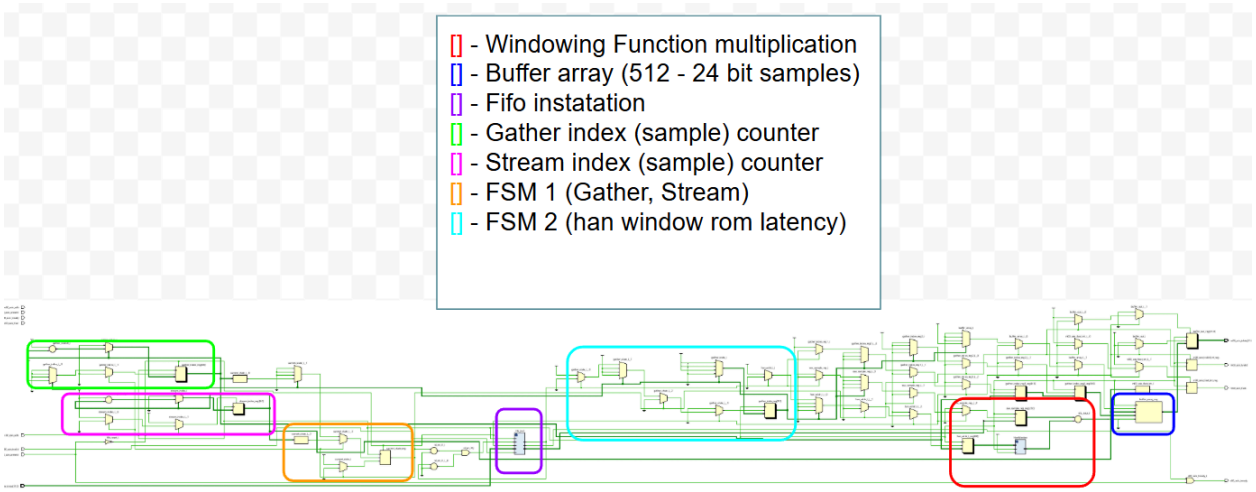
Paper design

Below you will find the paper design for the AXI FIFO. In the design, you will also see our attempt in trying to reduce spectral leakage by implementing what's known as a windowing function. We found this [resource](#) to be especially helpful in our understanding, and we believed a hanning window would suit our needs.



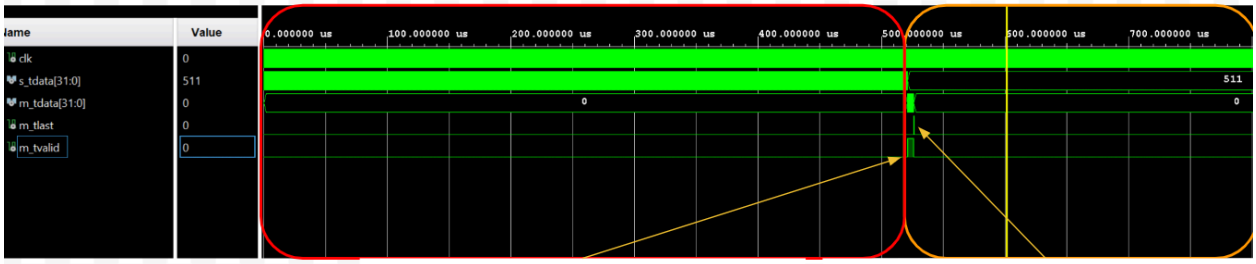


Elaborated Design



Annotated Simulation

[] - AXI input sample (48khz)
[] - AXI output burst output of 512 samples (100Mhz)



Tvalid goes high for all 512 samples

Tlast goes high for last sample

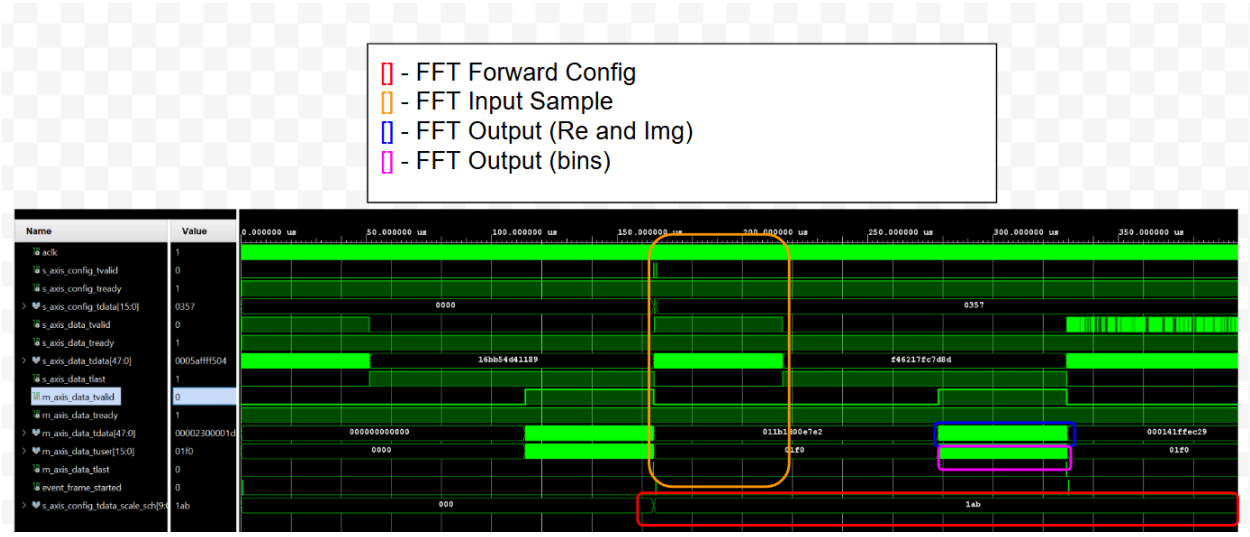
Task 3: Audio Processing with the FFT IP Core

In this task, we are creating a general wrapper for all the logic associated with the FFT IP Core.

FFT Instantiation

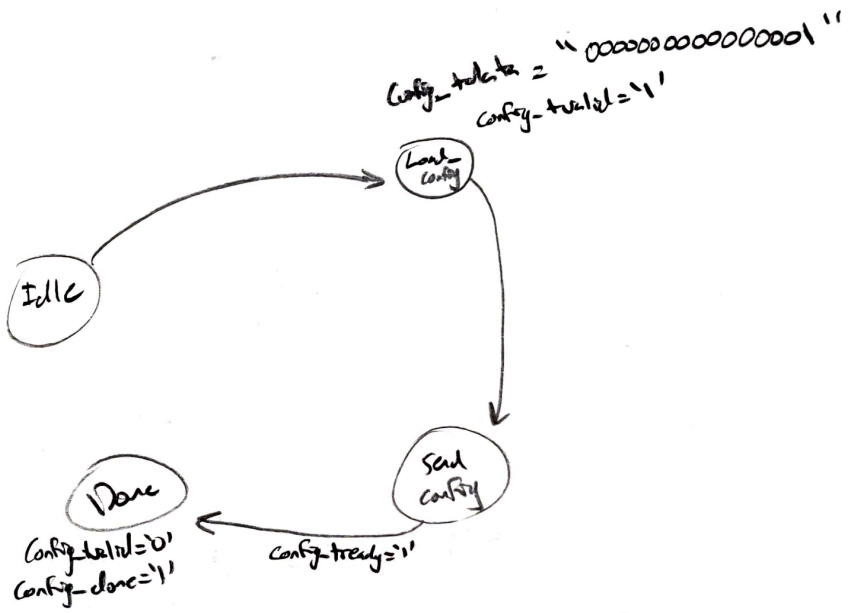
From the default settings, we set up our FFT to process an input of 512 samples with a target clock frequency of 100Mhz. We choose a pipelined/stream architecture with a data width and phase factor of 24 to match our existing audio data stream. We are outputting the ‘xk_index’ to receive the respective bin address for each output. We stuck with non real time throttle as we wanted to make sure the FFT was receiving 512 samples before outputting data. Lastly, we changed the output ordering to natural order, to make the real and imaginary bins be symmetrical about half the bins, meaning we can ignore the second half of the output.

FFT IP Core Annotated Screenshot

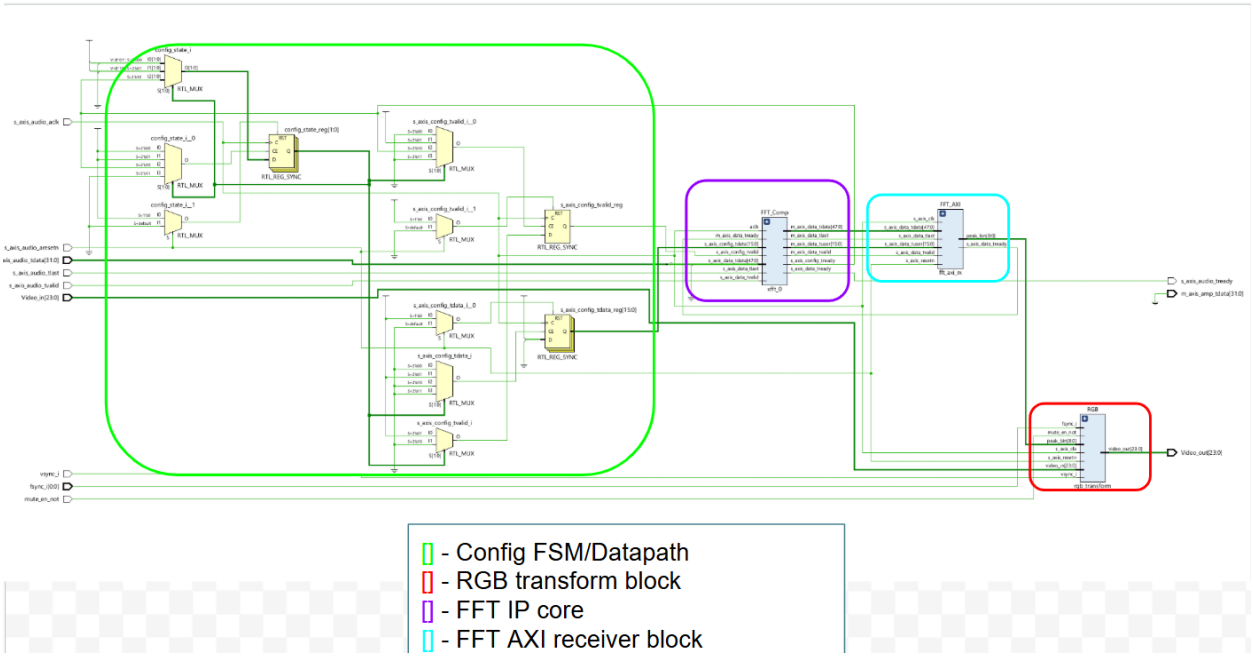


FFT Configuration

In order to use the FFT IP core, we need to configure it properly. This is handled in our video_transform wrapper which comprises the FFT IP, our fft_axi_rx, and our rgb_transform designs. Our wrapper is relatively simple, as it just connects the other files, but it also serves to config the FFT in hardware. The configuration process is not complicated at all, we are only sending a logic high bit in the LSB in the ‘s_axis_config_tdata’ line in accordance with normal axis communication protocol. This is handled via a FSM to make sure it only occurs at start up or reset. See the FSM design below:



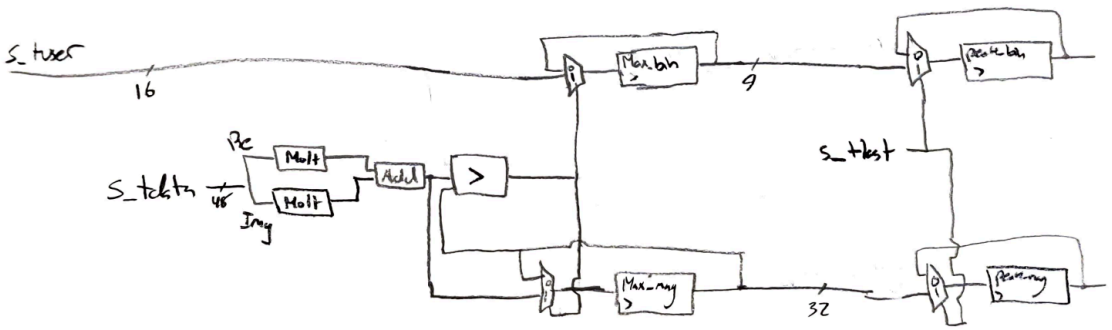
Video_transform Wrapper Elaborated Design



Task 4: Extracting the Peak Frequency Bin (fft_axi_rx)

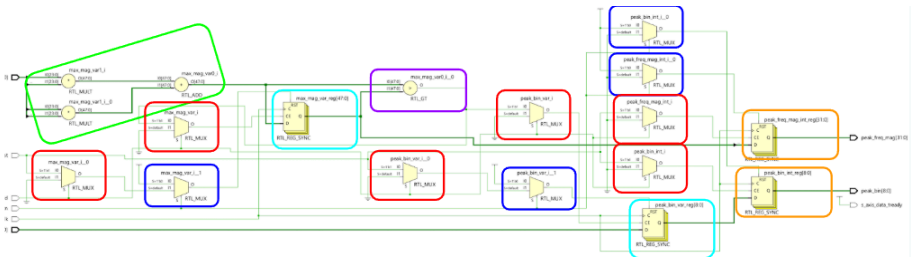
After getting valid FFT output, the next step was to extract the peak frequency bin from the stream of frequency data. We created a custom AXI receiver module called fft_axi_rx that waits for the FFT output and tracks the bin with the highest magnitude. Since the FFT outputs both real and imaginary parts, we calculated the magnitude squared for each sample using a simple sum of squares (to avoid using square roots). At the end of each frame, the module outputs the index of the bin with the highest energy, which we planned to use to control the video color.

Paper Design



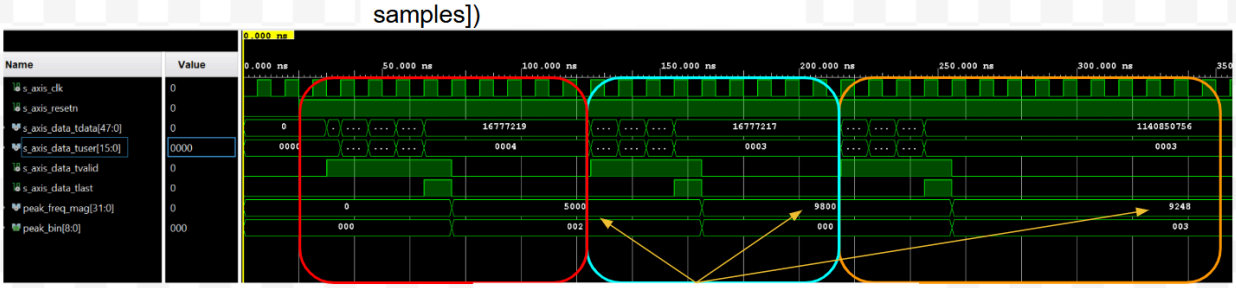
Elaborated Design

[] - Tvalid mux logic
[] - Reset mux logic
[] - Max magnitude squared check
[] - Magnitude squared computation
[] - Input audio register
[] - Output registers
[] - Max magnitude and bin registers



Annotated Simulation

[] - Test1 (peak freq in middle [5 samples])
[] - Test3 (peak freq in last [4 samples]
[] - Test2 (peak freq in front [4 samples])



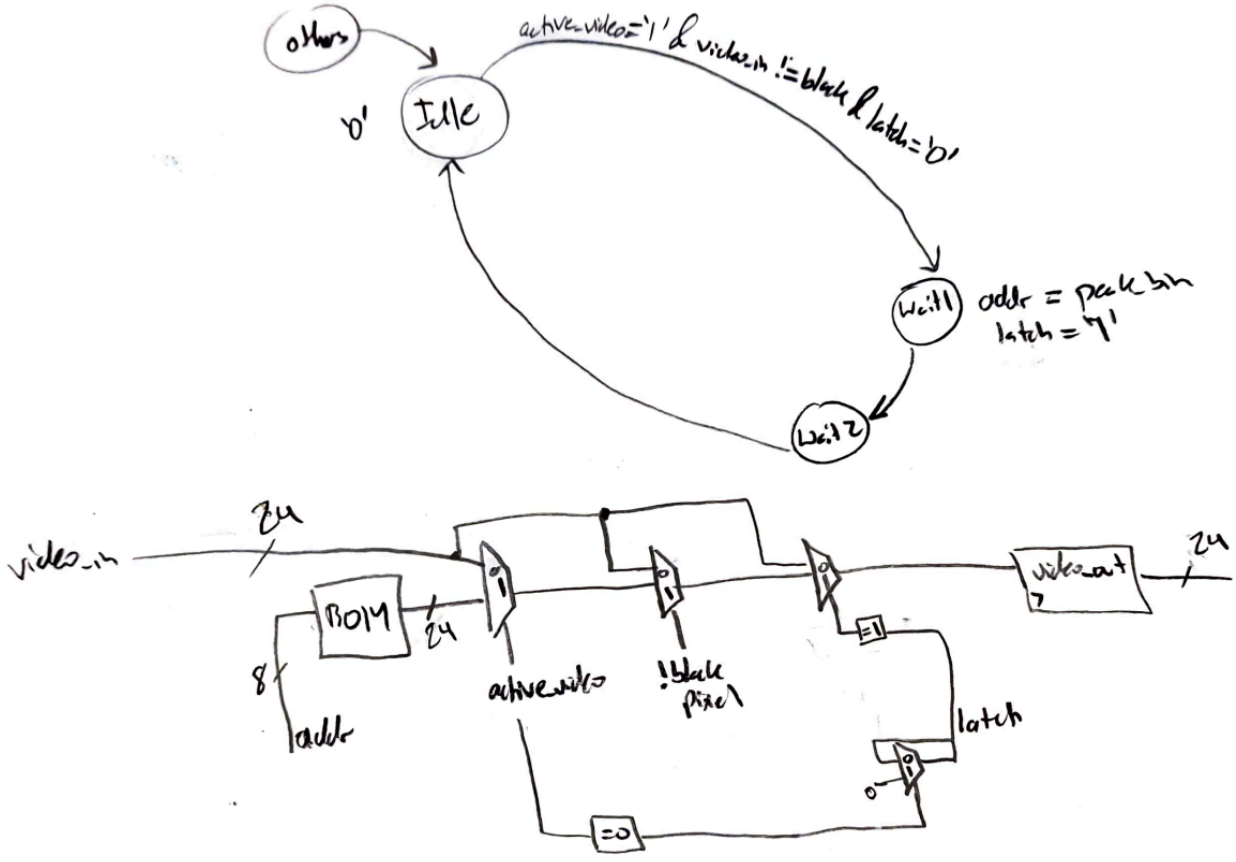
Magnitude is correctly computed

Task 5: Video Transformation From Audio (rgb_transform)

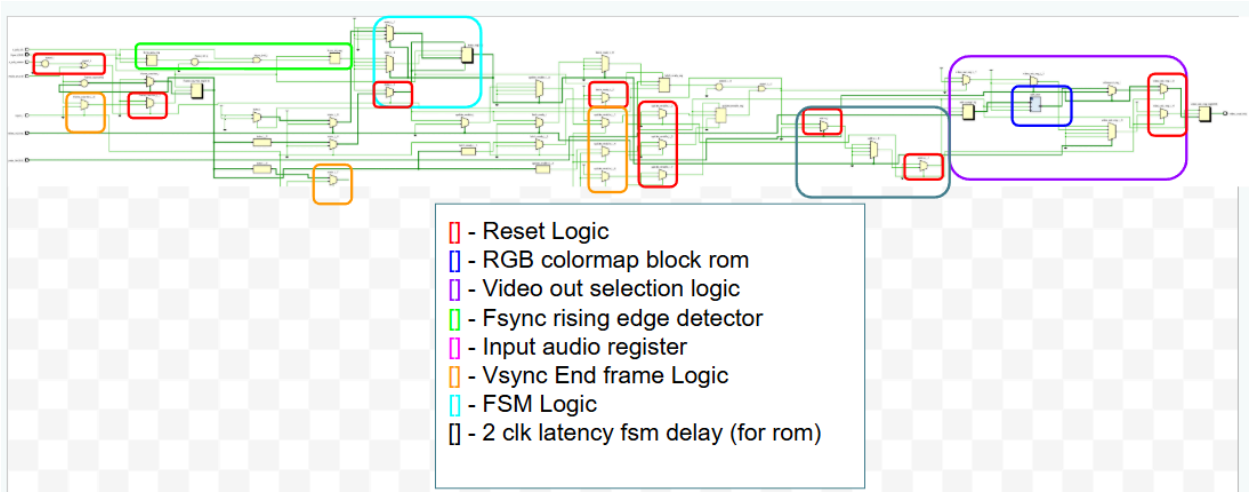
With the peak frequency bin extracted, we used it to influence the visual output on the HDMI display. This was done through our rgb_transform module, which receives both the generated video animation and the peak bin value. The goal was to color a moving object on the screen based on the dominant audio frequency. The module checks if a given pixel matches a certain color, predefined, and if so, changes its RGB values according to the bin index. This allowed us to create a direct, real time visual representation of the incoming audio signal.

Paper Design

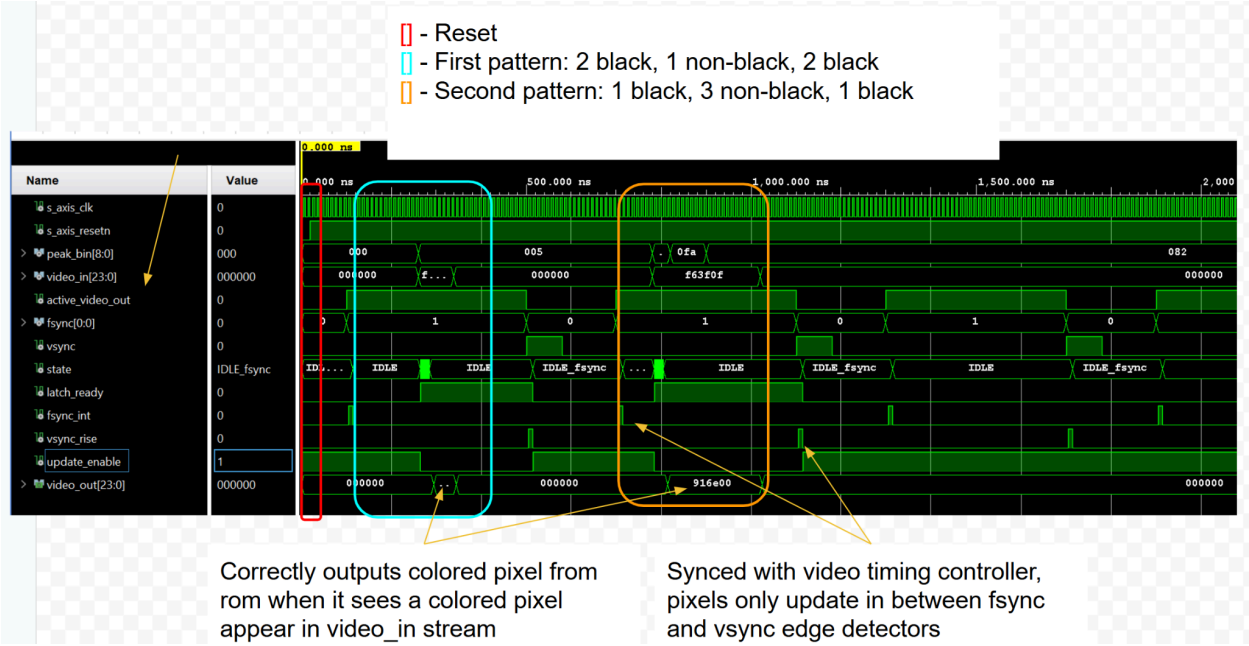
In the paper design below, you will see the FSM and datapath we designed for the video transformation block. The main principle is that it will detect whether the pixel is not black (or a predefined color) and change it quickly using block memory for a specific peak bin (depending on the FFT output). Please note that this initial design was made under the assumption that active video would be high during the entirety of the frame display. This was wrong, and we corrected this in code by using fsync and vsync to synchronize the pixel changes to the video timing controller. We felt it was an easy fix and did not need to draw a new paper design for it.



Elaborated Design



Annotated Simulation

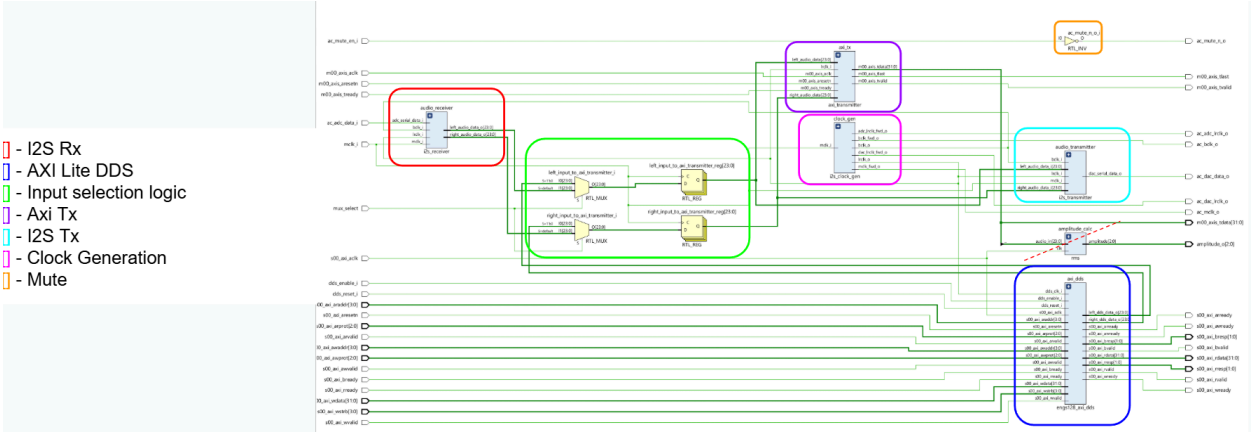


Task 6: Audio Passthrough

For audio streaming, we reused our existing I2S wrapper from previous labs with some changes. The module now serves as an audio passthrough, taking in data from either the I2S audio codec or the DDS (controlled via AXI Lite) and sending it directly to the DAC for playback. At the same time, the audio samples are forwarded to the AXI FIFO for FFT processing. This allowed us to maintain continuous audio output while also enabling frequency analysis for video transformation. The passthrough design kept the audio path simple and reliable throughout the project.

Elaborated Design

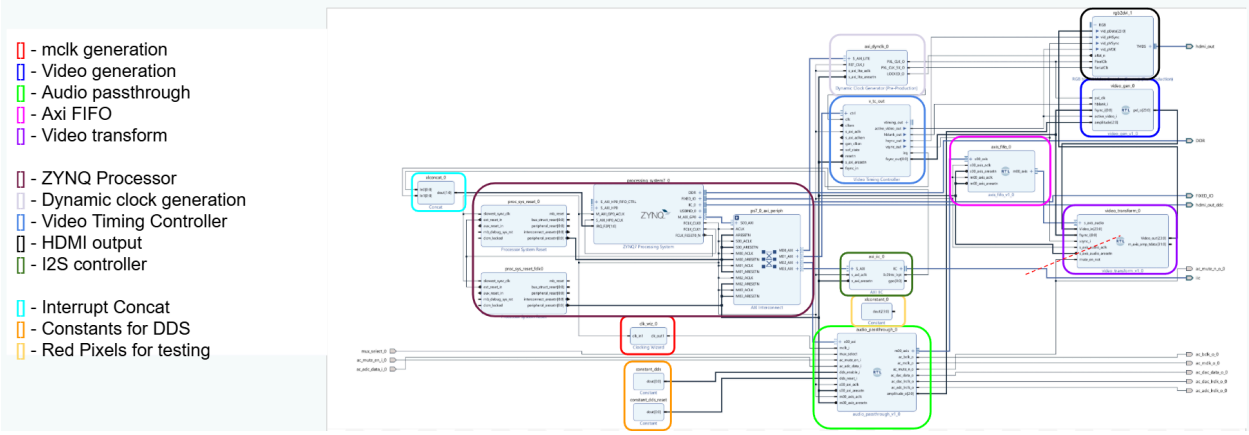
As we are confident in our implementation of our I2S wrapper from earlier labs, we did not write a testbench for the audio passthrough component. Instead, we relied on testing everything together in one final simulation.



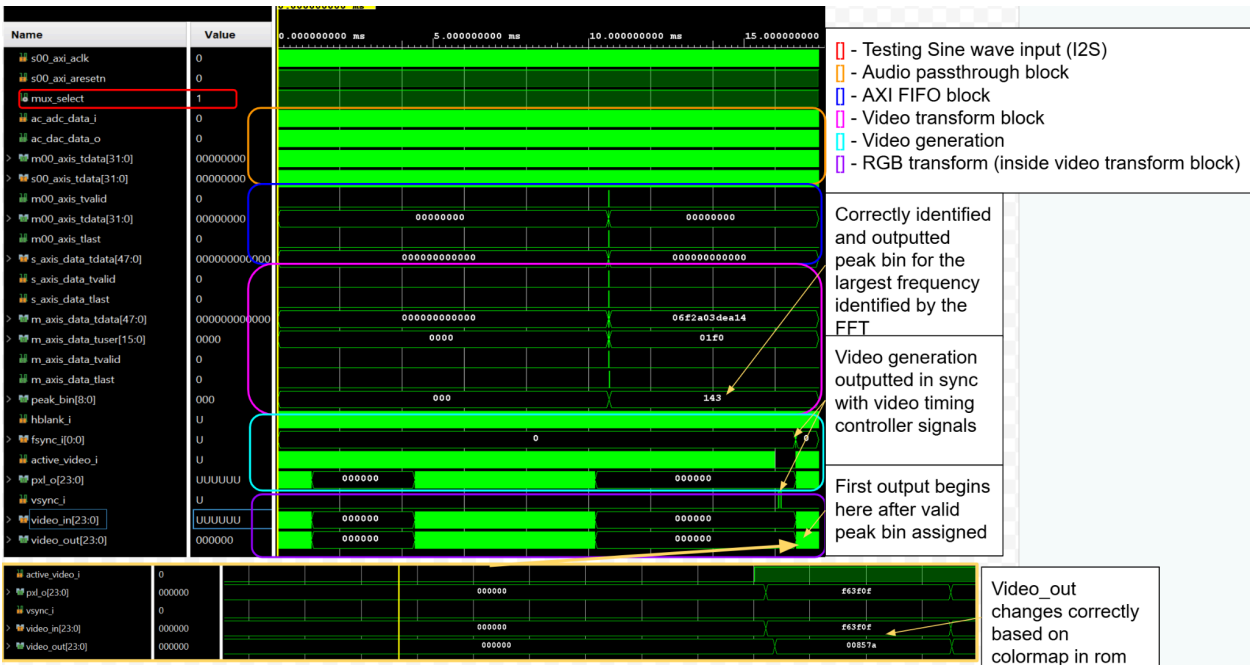
Task 7: System Integration

In the final stage of the project, we brought together all individual components: video generation, audio passthrough, FIFO buffering, windowing, FFT processing, peak frequency detection, and video transformation into a single top level design. This was the final step before we could test our design in hardware.

Final Block Design



Final Simulation



Post-Project Discussion

Methods

What steps did you take to solve the problem, and how did you leverage the incremental build and test process?

We broke the project down into as many modules as we could so that this would allow us to test each module thoroughly with its own testbench and ensure functionality before integrating it with another module. This made the overall integration of the system much easier, especially between the overall video generation and video processing modules.

What challenges did you encounter? How did you address these challenges?

With video generation a difficult challenge was figuring out the polarity of the main control signals for timing. Initially based on the simulation and configuration of the video timing controller we assumed that the polarity of important timing signals like hsync and vsync are high but in reality through the use of the ILA we discovered that the polarity of the control signals was low. Overcoming this discrepancy in the system made the video generation process much easier.

On the FFT side, the biggest challenge we had to deal with was spectral leakage. For context this means the energy of a signal's frequency content appears to spread or "leak" into neighboring frequency bins, especially when the input signal is not periodic or doesn't perfectly fit the sampling window. This caused some frequencies to work perfectly but others to produce a multitude of colors on the screen at once. To resolve this problem we tried to use the Hanning Windowing algorithm however our success was limited and it would’ve required more time to optimize the algorithm.

What code/designs were you able to reuse, and what did you need to change or add?

We were able to use all of our audio code from the previous labs to generate the audio and did change anything in it. We also used the code for the AXI FIFO however we did make modifications to it. We adjusted it to have a FSM that consisted of two mains states - GATHER and STREAM. In GATHER it accumulates 512 audio samples from the audio wrapper and then in the STREAM state it sends all 512 audio samples in a single burst to the FFT IP core.

If you were to tackle this design challenge again, what would you do differently?

We would use the ILA sooner for debugging since it pointed out an issue that the simulation failed to. This is in reference to the discrepancy in polarity for the video timing controller's output signals between the simulations/software and the hardware. Realizing sooner that these signals were actually active low in the hardware would've saved a lot of time (especially generating bitstreams).

Results

How well did the system as a whole work? Did you observe any unexpected behavior or timing discrepancies upon implementing your design in hardware? How did you resolve them?

The system worked well but not perfectly. We were able to change the color of the drawings on the screen using the FFT but the only issue was that for certain frequencies there was spectral leakage that caused the screen to flip through multiple colors instead of sticking to one. We tried to resolve it using windowing algorithms like Hann's but they didn't fully fix the issue.

On the video side the only unexpected timing discrepancy was with the polarity of the control signals like hsync and vsync but once these were recognized it was fixed quickly.

What is the critical path in your implemented design? Did your design fail timing at first? If so, how did you ultimately meet timing? If not, why do you think that is?

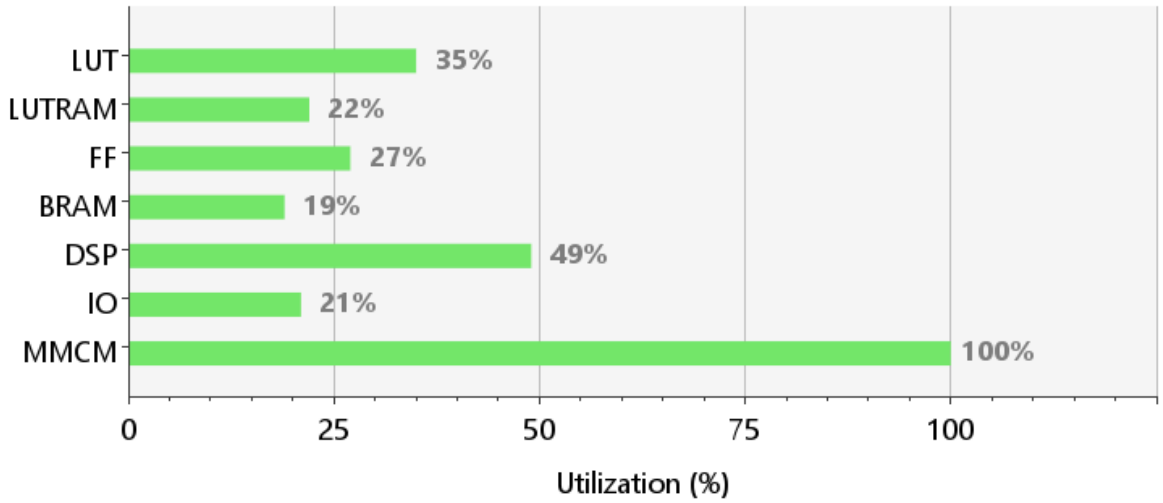
Design Timing Summary

Setup		Hold		Pulse Width	
Worst Negative Slack (WNS):	0.658 ns	Worst Hold Slack (WHS):	0.049 ns	Worst Pulse Width Slack (WPWS):	0.333 ns
Total Negative Slack (TNS):	0.000 ns	Total Hold Slack (THS):	0.000 ns	Total Pulse Width Negative Slack (TPWS):	0.000 ns
Number of Failing Endpoints:	0	Number of Failing Endpoints:	0	Number of Failing Endpoints:	0
Total Number of Endpoints:	27874	Total Number of Endpoints:	27798	Total Number of Endpoints:	11117

All user specified timing constraints are met.

Yes our design did fail timing in setup at first, so we set a false path in our constraints file and our design met timing constraints.

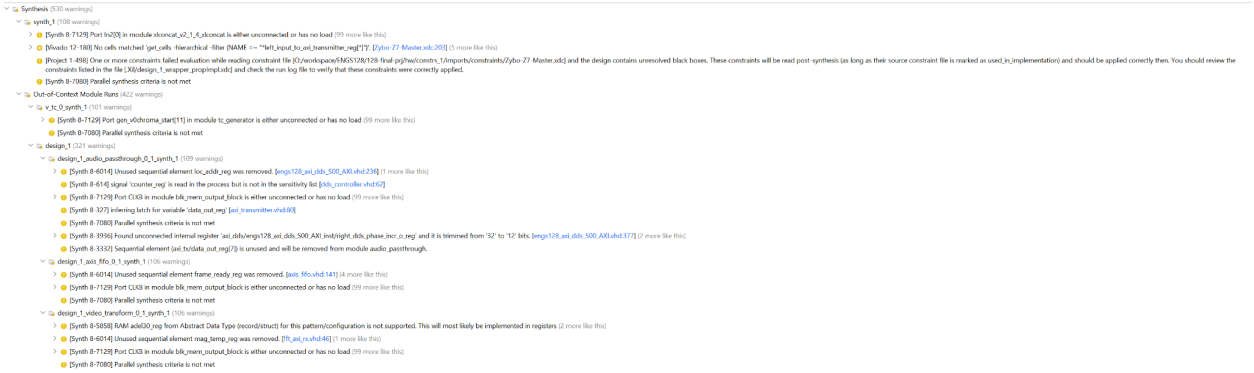
What is the resource utilization of your design? Which component(s) are the most resource-intensive?



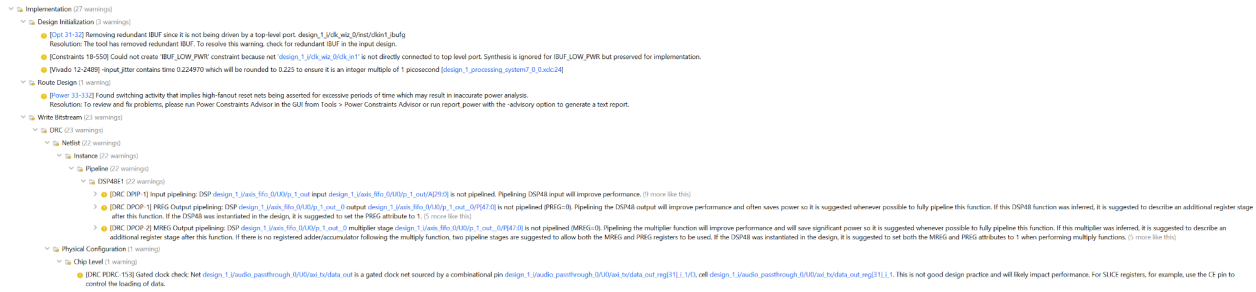
We can see the clocking network is the most resource intensive followed by digital system processing.

Resource	Utilization	Available	Utilization %
LUT	6126	17600	34.81
LUTRAM	1312	6000	21.87
FF	9499	35200	26.99
BRAM	11.50	60	19.17
DSP	39	80	48.75
IO	21	100	21.00
MMCM	2	2	100.00

Include a print-out of your Vivado synthesis and implementation Messages panel (deselect Info, and select Warnings, Critical Warnings, and Errors). If any warnings remain, explain why they are benign.



In synthesis, most of the warnings are unused signals. We do have a latch warning from our AXI transmitter but the protocol stick works and we can hear audio. We had this warning from a previous lab and decided to ignore it since it works.



For implementation, it appears we have an interesting warning about a gated clock in our AXI transmitter, but again, it works so we left it alone.

Reflection

What was the coolest thing you learned in this course?

We listen to audio everyday everywhere yet we never stopped to think about how exactly this audio was being processed and transmitted to us. Getting to understand how exactly the I2S protocol works especially in conjunction with AXI Stream was very useful and interesting.

After taking this course, what do you wish you understood better?

We wish we understood how the DMA worked and how to use it because it seemed like a very useful feature available on the FPGAs. Of course the class was already packed with material so it makes sense why it fell out of scope from lab 2 but certainly something that would be cool to know if time allowed it.

What about your system is the most exciting to you? This project was a massive undertaking--be proud of your accomplishment!

The most exciting part for us was just watching all the major components work together successfully. We worked on the audio processing the entire term so watching it work well and then be able to use it to manipulate the video data successfully with the FFT was very satisfying and rewarding. The class was very cumulative in the sense that each lab built on the previous so it was great to see the final product work after adding all the layers of complexity over time.