Trumpet Analyzer

ECE532 FINAL REPORT

Professor Jason Helge Anderson
TA: Daniel Rozhko
Team: Jason Hayhoe, Krishanth (Kris) Suthaharan, Tristan Robitaille & Yudi Wang
Date: April 12th, 2024
Group #2

1.0 Overview

1.1 Motivation and Goals

The motivation of this project is to create a hardware design similar to the functionality of a tuner and display the results on a screen. While tuners nowadays are widely available over software applications, it would be a great learning experience to implement the same functionality in hardware. Therefore, the goal of the project is to replicate the basic functionality of a tuner, while also achieving other functionalities like recording and playing audio data through a speaker.

Figure 1. An example of a tuner app in the App Store[1]

The system would continuously take audio input from a microphone, push the data through a Fast-Fourier Transform (FFT) block, and display the resulting frequency onto a display through HDMI. In addition, the system implements more advanced features, like audio playback and controlling playback through capacitive buttons.

1.2 Project Specifications

Inputs

The music analyzer's main source of input is an audio signal that gets processed in real time. We primarily focused on having input from a Trumpet instrument.

Outputs

[1] Image adapted from: https://apps.apple.com/us/app/tuner-t1/id453056916

Our project aims to output the note being played from the audio data. This is achieved by utilizing a graphical interface that displays the trumpet valve combinations being correctly actuated on an HDMI monitor. Additionally, the correct note and frequency processed by the FFT are shown as text on the GUI. Lastly, the music analyzer will be able to play back music after recording the incoming audio signal. For debugging purposes, the output of the SPI MEMS microphone and FFT are displayed as PWM signals on the leds of the Nexys Video board as well as the I2C buttons being displayed on the leds.

Constraints

There are many restrictions we have set in place to help improve the accuracy of our project and achieve more meaningful output. The project must be able to:

1. Display results for only 1 note and distinguish between notes of about a semitone, which means the smallest frequency difference between the two lowest notes is 10 Hz
2. Identify pitches over the trumpet's whole range of notes which is about 10 Hz to 1 kHz
3. Display results on screen before the next note has changed such that the screen and note must update in less than 500 ms.

Requirements

In order to succeed in processing the audio input data correctly, the Trumpet Analyser will follow a set of functional requirements:

1. The audio processing algorithms should work in real-time to recognize notes from the audio stream of data for every note change at 500 ms or less.
2. The project should be able correctly map frequencies to their respective note pitches at least 85% of the time.
3. During audio playback, the output should match the input audio signal, which requires accurate processing of the audio data and storage into memory.
4. The project should work from anywhere in a noise controlled environment.
5. There should be sufficient memory in the BRAM to sample audio at a frequency of 5 kHz storing 12-bit samples. The DRAM on the Xilinx Nexys Video board should have enough capacity to store all the possible GUI images and text overlays for display (i.e. 12 note combinations and 10 numbers as well as the empty case). The resolution of the image frames should be sufficient to display the notes and frequency information clearly.
6. The correct valve combinations on the animated trumpet should be displayed for the current note and show the current note name and frequency in text format on an HDMI monitor. The transitions should be synchronized.

1.3 Block Diagram of Design

Compared to the original proposed design, some changes were made to accommodate the limited FPGA resources and time. In the final design, over-reaching functionalities like stepper motors and modulation were removed, and more software was added for HDMI and GUI images. The following figure shows a block diagram of the final implemented system. Functionalities related to each block will be described in later sections.

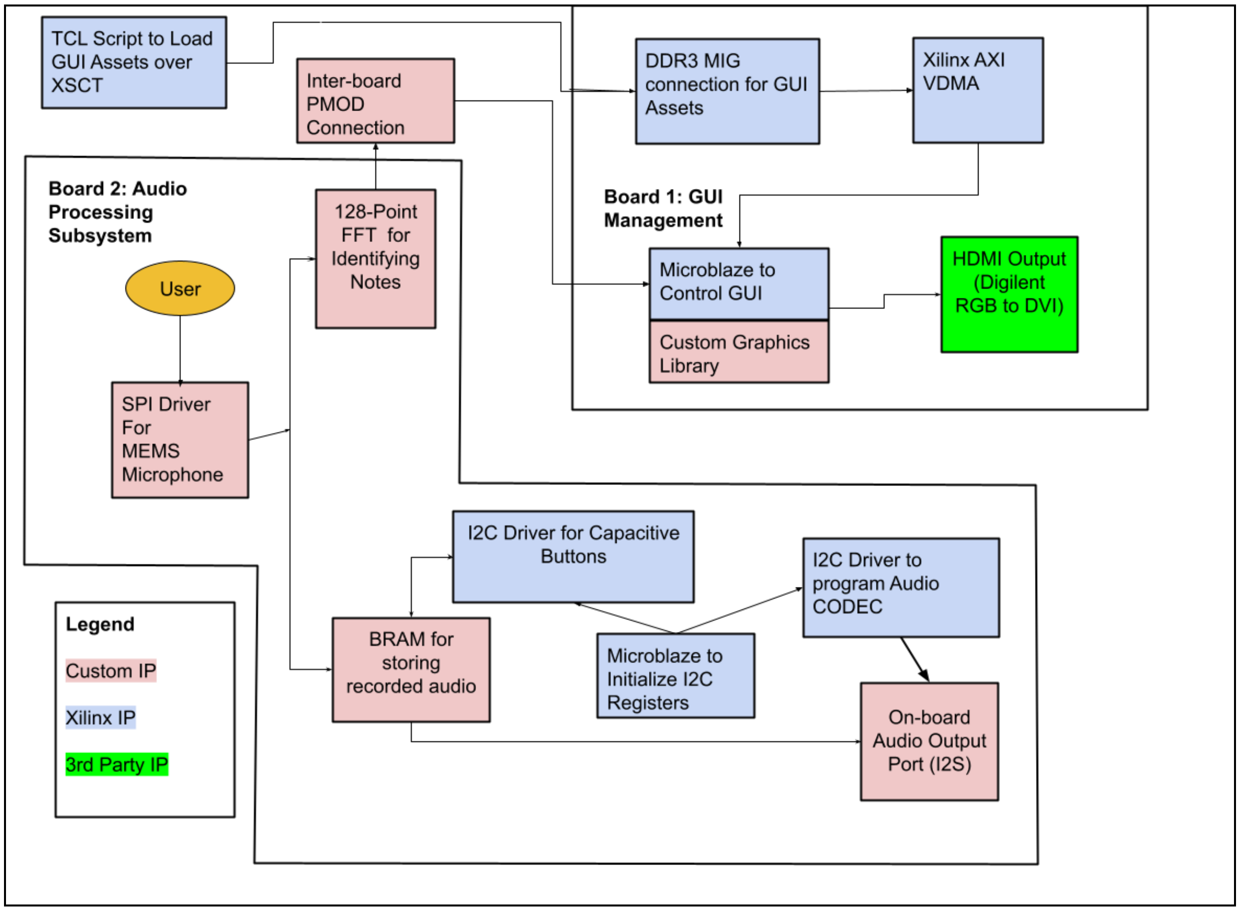

TCL Script to Load GUI Assets over XSCT
Inter-board PMOD Connection
DDR3 MIG connection for GUI Assets
Xilinx AXI VDMA
Board 2: Audio Processing Subsystem
128-Point FFT for Identifying Notes
Board 1: GUI Management
User
Microblaze to Control GUI
HDMI Output (Digilent RGB to DVI)
Custom Graphics Library
SPI Driver For MEMS Microphone
I2C Driver for Capacitive Buttons
I2C Driver to program Audio CODEC
BRAM for storing recorded audio
Microblaze to Initialize I2C Registers
On-board Audio Output Port (I2S)
Legend
Custom IP
Xilinx IP
3rd Party IP

Figure 2. Block diagram of the final implementation

1.4 System Modules Descriptions

Table 1: Modules in the final implementation

Modules	Type	Description
SPI Controller	Custom HW IP	SPI MEMS microphone controller to read audio data at a rate of 5 kHz.
128-Point FFT	Custom HW IP	Transform time-domain signal into frequency domain and identify corresponding pitch.

I2S Interface for Speakers	Custom HW IP	Uses I2S communication protocol to send audio data to the audio codec.
I2C Interface for Speakers	Xilinx IP	Necessary for writing to configuration registers.
BRAM for Audio	Custom HW IP	Store real-time audio data in memory.
MicroBlaze	Xilinx IP	Processor for controlling the entire system.
DRAM for GUI Assets	Hardened on Board	Used to load GUI assets once converted to one-dimensional binary arrays using MATLAB
DDR3 MIG	Xilinx IP	For HDMI display, the GUI is stored at specific memory addresses in the output frame buffer.
Xilinx AXI VDMA	Xilinx IP	For HDMI display of GUI, located output framebuffer and transferred video stream from framebuffer to RB2DVI without CPU involvement.
HDMI Output (RGB to DVI)	3rd Party IP	From the VDMA framebuffer, the RGB2DVI IP converts the digital video signals from RGB format to digital video interface standard to display on to the monitor.
UartLite	Xilinx IP	Helped facilitate serial communication between the MicroBlaze and UART port.
AXI Interconnect	Xilinx IP	Allowed to standardize communication between many hardware modules like MicroBlaze, custom IP blocks and controllers.
I2C Interface for Capacitive Buttons	Xilinx IP	To control capacitive buttons PMOD for start/stop audio recording/playback.
AXI-wrapped Wire PMOD	Xilinx IP	To receive frequency values from FFT board and map them into MicroBlaze memory for the HDMI.

2.0 Outcome

2.1 Results

Our original design of the Trumpet Analyser was modified to allocate for hardware resources as well as prioritize on achieving success in the main requirements of the project. The comparison is outlined in the table below:

Table 2: Feature differences between start and end of the project

Original Feature	Final Feature	Rationale
Modulation of recorded audio for playback.	No modulation is implemented.	We chose to drop this feature in order to improve the core functionality of the project since it wasn't necessary to identify the pitches from incoming audio. We were also limited on time due to pressure from other course work.
Store GUI assets at a resolution of 1280 x 720 pixels.	Store GUI assets at a resolution of 800 x 600 pixels.	Lowering the resolution helped take up less memory and storage capacity as well as helped boost loading times of the images to meet system latency requirements.
Stepper Motors to show valve combinations moving in real life.	No stepper motors were implemented.	This was a stretch goal feature that we realized was not attainable given the amount of time we had left.
Add noise reduction module to output of SPI controller.	Noise reduction is not integrated.	Since the output of the processed audio data from the SPI controller had relatively good resolution, we wanted to prioritize getting other modules working such as the FFT.
HDMI frame generation.	Added XSCT flow for loading GUI images.	Command line interface helped automate the task of loading the necessary GUI images which help make our project more efficient.

Based on our final demo, our project worked well in performing its core goals including displaying the correct GUI of trumpet valve combinations being correctly actuated on an HDMI monitor. The correct note and frequency processed by the FFT are shown as text on the GUI and we were able to play back music after recording. Despite these changes, our project was able to successfully meet its goals initially set.

2.2 Future Improvements

We believe there are a couple of features that can be added in the future to help improve our Music Analyser project. These include:

1. Integrating the noise reduction module and pitch detector dial
    a. In terms of the audio processing, the noise reduction can help improve the signal-to-noise ratio which would result in more accurate data representation of the audio data.

    b. By reducing noise, we hope to minimize the unwanted signals to achieve better pitch detection accuracy that is reliable and does not glitch.

    c. We would like to add a display on our GUI to show a pitch detector dial that can help users tune their instruments for example. This would be displayed to the HDMI monitor and would require significantly more accuracy.

2. Hardware acceleration of GUI assets
    a. Currently, the loading of the GUI images at the start takes a bit of time. So by accelerating this, we hope to make our design more efficient in the overall process of analyzing incoming audio data.

3. Improve music analytics
    a. With the audio signal processed, we could display more information to the user about the piece of music. For example, we could try to detect the tempo in beats per minute, perform harmonic analysis to identify relationships between notes.

    b. In addition, we can have a GUI for spectral analysis to analyse the frequency spectrum of the audio signal and visualize the distribution of frequencies in the audio.

4. Hardware Note Transcriber
    a. Listening to the audio input, the trumpet analyser can process the audio input to be exporting sheet music for other people to play the music

5. Increase efficiency of project
    a. From our demo, we used about three Nexys Video Xilinx FPGA boards due to the 128-point FFT consuming a significant amount of the FPGA resources. We would like to make our design compact and fit on one FPGA board.

    b. We can also incorporate more pipelining wherever possible in our system to help accelerate the processing time of the audio processing and be more efficient at detecting the pitches for the notes.

2.3 Recommendations for Next Steps

Starting over, we would recommend anyone taking over our project to use the Xilinx FFT instead. This is because the IP core would already be verified and pre-optimized for resource efficiency utilizing the least number of FPGA resources such as logic cells, DSP slices and BRAM blocks, which means we wouldn’t have to be worried about consuming too many FPGA resources. In addition, it would probably already result in higher performance since the FFT computations would have low latency. Lastly, we believe that it would help improve the accuracy of the pitch detection as well as help make the integration into the FPGA designs to be a lot easier.

It would also be a cool, nice step to have stepper motors to actuate the valves of a real trumpet since an audio input from a real trumpet player would be replicated by the FPGA project which can help another amateur player learn to play the Trumpet.

Lastly, future developers of the project can incorporate modulation into audio recording and playback to help make the use of bandwidth more efficient as well as improve the audio signal integrity. There can be error correction coding to improve the reliability of the audio transmission.

3.0 Project Schedule

Please see appendix for original milestones from proposal (Appendix A) and weekly accomplishments (Appendix B). Comparing and contrasting some of the difference between the original and final weekly accomplishments, I believe our team has made some changes in order to help improve our team's performance. For instance, in our proposal, we wanted to start our HDMI setup in week four whereas in our final schedule we ended up starting it in week two since that was going to be a challenging IP block for our group to complete. Some tasks seem to have taken longer to implement than initially projected such as the FFT, HDMI and SPI controller modules. For example, initially we wanted to get a working SPI MEMS microphone by the second week, but that was delayed due to working on DSP blocks for filtering as well as discovering a reliable means to test the module at the time. Reading week in mid-February showed an uptick in progress catching up on deliverables that were a bit late. Near early March, it seems we were a bit behind from our proposal schedule which may be due to the workloads of other courses running in parallel. In the end, we were able to integrate all our major components on time and succeed in our trumpet analyser project.

4.0 Implementation and Algorithm Description of Blocks

4.1 SPI MEMS Microphone PMOD and SPI Interface

SPI driver

This SPI PMOD uses the SPA2410LR5H-B microphone and ADCS7476 SPI ADC. The ADC does not have any configuration registers. A negedge of the CS signal launches an ADC input capture and conversion, which takes 4 SCLK cycles. The output is 12-bit, so a read of 16-bit is required, the first 4 bits being "dummy". The SPI module is written from scratch and operates on a few always blocks. One generates the SCLK from the 100MHz system clock, another generates a 5kHz sampling and another performs the SPI communication. We also have an always block to cross domain to 100MHz of a new_data_ready signal, which is generated on a 500kHz timebase (it simply generates a 100MHz pulse on a posedge of new_data_ready). Finally, the SPI module generates a PWM signal from the deserialized audio data to display on an LED on the board. The SPI communication FSM simply sleeps until requested (5kHz signal) and, at which point it desserts CS and shifts in 16 bits of MISO data. It outputs the new_data_ready pulse once all 16 bits have been read.

4.2 Identification of Pitches with FFT

FTT_128 and FFT_32 source code with testbenches

As the DFT can be recursively split into a summation of "even" and "odd" components, it is possible to decompose the DFT into multiple levels of multiplication and addition of "even" and "odd" units[2].

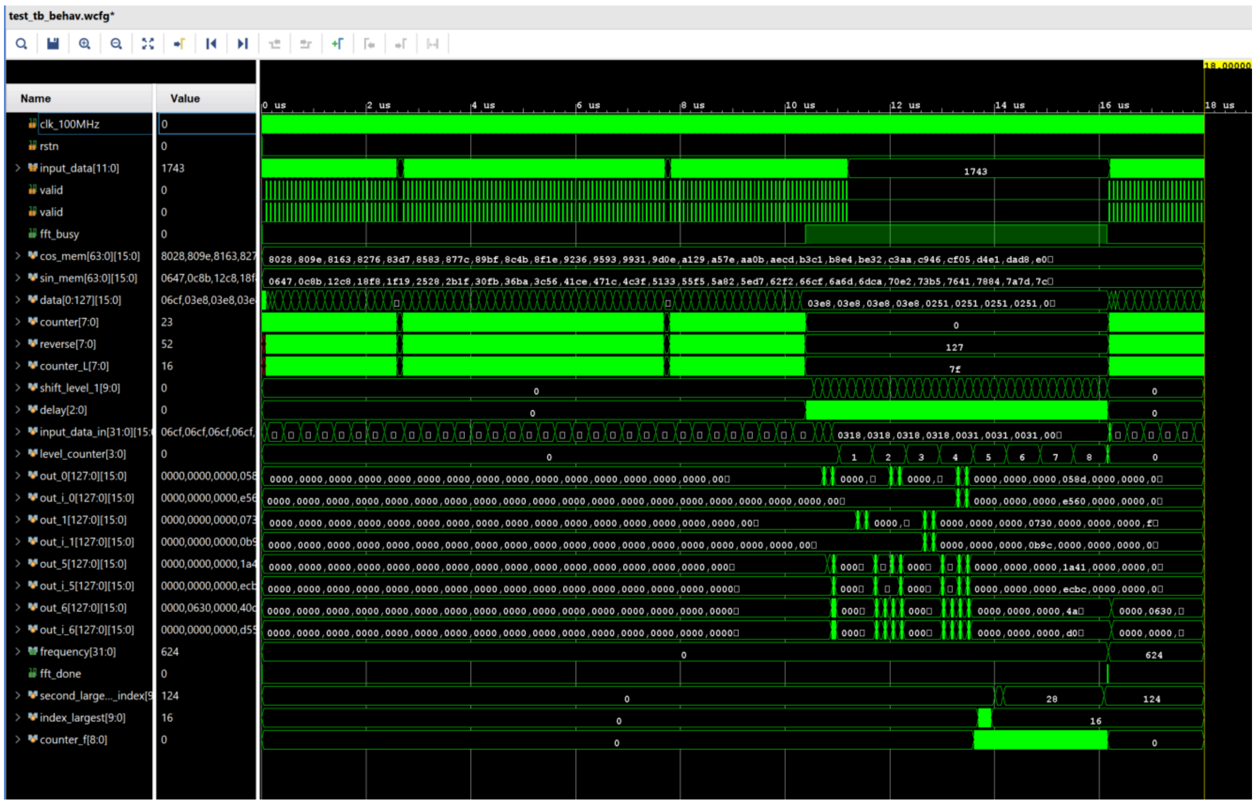
$$X_m = \sum_{n=0}^{N/2-1} x_n w^{nm} + w^{mN/2} \sum_{n=0}^{N/2-1} x_{n+N/2} w^{nm}$$

Figure 3: Recursive Split of the DFT Algorithm[2]

For a 32-point FFT, this would require 5 levels. For a 128-point FFT, this would require 7 levels. The current 128-FFT code originated from a previously made 32-point FFT design. The 32-FFT was hardcoded with regard to the butterfly diagram found in this GitHub link[3]. For the 32-FFT, each level was wired manually, and butterfly units (that contain DSP blocks) were used in each level. In terms of butterfly unit calculation, incoming SPI data would be 12 bits. Within the FFT block, this data is sign-extended to 16 bits. The twiddle factors are stored in Q15 format, thus the multiplication would be 16*16. Fixed point arithmetic was used in the butterfly units.

To expand this to a 128-point FFT implemented in the current project, the first 5 levels of the 32-point FFT can be maintained by re-routing the first layer. When valid mic data enters the FFT from the SPI controller, they are fed into a local memory block which stores the data in bit-reversed order. A counter is used to keep track of the number of data currently in this local memory block. Once the memory block is filled, the first layer will take these data in incrementing order. For example, the first butterfly unit of the first level would take data point 0 and data point 64, located in data[0] and data[1] respectively. The first 5 levels would maintain their structure as in the 32-point, and each layer would take 4 rounds to get through all 128 data points. The intermediate signals are stored in 2 buffers out_0 and out_1. Since the last two layers require data points that are 32 data points and 64 data points apart, the same logic cannot be applied. Thus the last two layers are hardcoded. The following shows the FFT testbench:

[2] https://web.mit.edu/6.111/www/f2017/handouts/FFTtutorial121102.pdf
[3] https://github.com/AhmedAalaaa/32-point-FFT-Verilog-design-based-DIT-butterfly-algorithm



Name	Value
clk_100MHz	0
rstn	0
input_data[11:0]	1743
valid	0
valid	0
fft_busy	0
cos_mem[63:0][15:0]	8028,809e,8163,827
sin_mem[63:0][15:0]	0647,0c8b,12c8,18f
data[0:127][15:0]	06cf,03e8,03e8,03e
counter[7:0]	23
reverse[7:0]	52
counter_L[7:0]	16
shift_level_1[9:0]	0
delay[2:0]	0
input_data_in[31:0][15:	06cf,06cf,06cf,06cf,
level_counter[3:0]	0
out_0[127:0][15:0]	0000,0000,0000,058
out_i_0[127:0][15:0]	0000,0000,0000,e56
out_1[127:0][15:0]	0000,0000,0000,073
out_i_1[127:0][15:0]	0000,0000,0000,0b9
out_5[127:0][15:0]	0000,0000,0000,1a4
out_i_5[127:0][15:0]	0000,0000,0000,ecb
out_6[127:0][15:0]	0000,0630,0000,40c
out_i_6[127:0][15:0]	0000,0000,0000,d55
frequency[31:0]	624
fft_done	0
second_large..._index[9	124
index_largest[9:0]	16
counter_f[8:0]	0

Figure 4: Simulation waveforms of the FFT block

After the final result from level 7 has been computed, the absolute value is then taken. The addition of the absolute values of the real and imaginary numbers is sufficient to estimate the amplitude[4]. The largest amplitude is then taken to calculate the final frequency, and the module then toggles appropriate control signals to indicate that the FFT calculation has finished.

4.3 On-board Audio Output Port and I2S Speakers

I2S driver
I2S driver CocoTB testbench
Audio library configuration code
The audio codec, ADAU1761, onboard the Nexys Video requires two communication interfaces: I2C for writing to configuration registers and I2S for sending/receiving audio data. For these, we instantiated two blocks: one using the Xilinx IIC IP wrapped in an AXI slave and a custom I2S driver. The MicroBlaze can send register read/writes to the ADAU1761 through simple memory-mapped AXI using the `XIic_Send()` Xilinx-provided function. The only C code needed for the audio codec is configuration of the clocks, the I2S interface settings, power domains enable and mixer/gain control in the audio output signal path. This code can be seen in the configuration code linked above.

[4] https://dspguru.com/dsp/tricks/magnitude-estimator/

The I2S driver implements the I2S communication protocol (transmit only). It outputs three clocks derived from the 100MHz system clock: master clock (clock used for system functions in the ADAU1761), bit (data) clock and left/right “clock” (indicates whether the data corresponds to the left channel or the right channel). The module, when enabled, buffers the incoming (parallel) data and sends it out serially according to the I2S protocol. The same data is sent to the left and right channels.

This module was verified manually using a waveform viewer (GTKWave) and very basic CocoTB testbench. I2S being such a simple protocol, this verification and initial debug were very quick and painless (!). Synthesis was left as vanilla Verilog; no directives needed.

4.4 I2C Capacitive Buttons PMOD and I2C Interface

Capacitive button library configuration code
The capacitive buttons PMOD’s capacitive sense IC, the AD7156, also requires an I2C interface, also made from the Xilinx IIC IP. The capacitive buttons are used to start/stop recording, start/stop playback and switch between live playback and recorded audio playback mode. The chip makes available a number of signal processing constants such as fixed or adaptive threshold select, sensitivity adjustment, averaging function, etc. These have to be adjusted as the default values do not give reliable sensing with the PMOD’s board layout and capacitance. These are adjusted in software. The actual button signal isn’t read over I2C; it is a GPIO output that is read by the FPGA, made available to the MicroBlaze through AXI and distributed to the other hardware blocks. The C code’s sole responsibility is configuration at startup and to reflect the state of the buttons on the LEDs. This could be done in hardware, but we thought it was a good idea to do it in software in order to have a good AXI slave reference in the project as an example (this module was the first one to go on the main branch) and as a means to sanity check the MicroBlaze throughout the project. No testbench was needed for this module.

4.5 Audio Playback and Record

Microphone storage RTL
Testbench
This functionality is achieved through two RTL modules, both written from scratch. The `mic_storage` module is responsible for interfacing with the BRAM used to store the audio recording. It instantiates a BRAM (specified with the `ram_style = "block"` directive) for 5s of 5kHz 16-bit recording. It changes into `idle`, `recording` or `playback` mode based on the state of the two capacitive buttons. In the recording mode, it increments a counter on every `new_sample` pulse from the microphone module and writes the microphone data to the BRAM. In the playback mode, it walks the BRAM at a period equal to the microphone module’s period until the index of the end of the recording that was saved during the recording mode. Finally, this module also outputs the correct audio data (either live measurement or playback from BRAM) to the FFT and I2S modules.

The playback_ctrl module is responsible for determining the playback mode based on the states of the capacitive buttons. Its output is sent to the mic_storage module. Button #1 starts/stops the playback mode and button #2 starts/stops recording, when they are not both pressed. If both are pressed, the playback mode toggles between live playback and playback from recording. Being able to deal with this conditional functionality of the two buttons was more complicated than anticipated, so a more thorough testbench was written to exercise all button state transitions.

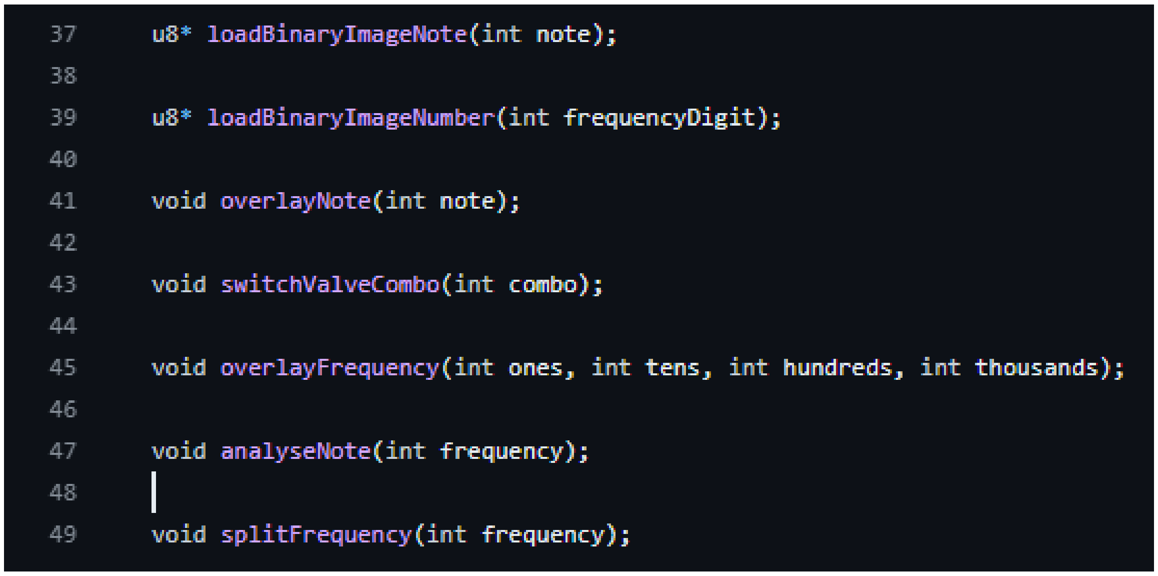
4.6 HDMI Output on Nexys Video

HDMI Output Vivado Project
Digilent Nexys Video HDMI Demo
For HDMI, we started with a passthrough HDMI demo for the Nexs Video, which is linked above. This demo project uses Xilinx IPs such as the AXI VDMA (Video Direct Memory Access), the MIG(Memory Interface Generator) for interfacing with DDR3, and the MicroBlaze processor. It also uses Digilent IP cores to handle conversion between RGB and DVI for the Nexys Video HDMI input and output pins.
To use this demo in our project, we ported the design to 2018.3, which required upgrades to most of the IP blocks but otherwise worked without major modifications. We also needed to switch from using passthrough HDMI to displaying our own GUI. The modifications required for this change were mostly made in the MicroBlaze software and in the sdk.
To get the HDMI to display our GUI, we modified the initialization of the VDMA driver to make it look for the output framebuffer in a memory address that was mapped into the DDR3. We would then copy our GUI assets from elsewhere in the DDR3 memory into the framebuffer based on the output of our GUI library. Once the GUI library output was stored in the VDMA framebuffer, it would be sent to the RGB2DVI IP, which would then output the frame over HDMI.
To load the GUI assets into the DDR memory, we used the XSCT console to run a TCL script. It would load binary encodings of all the GUI assets into the part of the MicroBlaze memory map that corresponded to the DDR, using the XSCT mwr command. These GUI assets would stay in the DDR until the board was powered off, even if the FPGA was reprogrammed. In the next section, we will discuss the details of the GUI assets and the GUI library.

GUI Library and Assets

GUI Library
To make our GUI, we created custom GUI assets and a custom C library to generate framebuffers based on the detected frequency. The first step was preparing the assets. We prepared 8 base images of a cartoon trumpet player, each showing a different valve combination. We then created 22 overlays, one for each of the twelve note names, and one for each of the ten digits, as well as one for a blank digit. We made the background white for these overlays, which corresponds to (255,255,255) for all pixels in RGB. These overlays were generated using a custom MATLAB script.
Once the GUI assets were prepared, they were converted to one-dimensional binary arrays and written into binary files using MATLAB, before being loaded into DRAM using the XSCT flow described above.

The GUI library, running on the MicroBlaze, made use of these GUI assets to prepare the output framebuffer based on the frequency provided by the FFT. The full list of functions is shown in the figure below. Every time the frequency changes, to a value that is within the range for this project (200 Hz - 1000Hz), the main loop calls analyseNote with the frequency value. The GUI library then calls switchValveCombo to load the corresponding image for the valves, overlay Note to overlay the note and splitFrequency on the frequency. splitFrequency then calls overlayFrequency with all the digits of the frequency to overlay the frequency digits in the framebuffer.
Two techniques were used to manage the overlays. For the notes, whose positions never change with respect to the position stored in DDR, we simply ignored the white background, only copying the non-white pixels that made up the note name. For the digits, the need to shift through different positions meant that this technique was not fast enough to keep up with real-time note switches. For the frequency digits, we used hard-coded offsets to shift the overlay pixels into each position based on the input frequency. We used memcpy to copy the base image and the frequency overlays to the framebuffer, while using pointer arithmetic to copy the note overlays.



```
u8* loadBinaryImageNote(int note);

u8* loadBinaryImageNumber(int frequencyDigit);

void overlayNote(int note);

void switchValveCombo(int combo);

void overlayFrequency(int ones, int tens, int hundreds, int thousands);

void analyseNote(int frequency);

void splitFrequency(int frequency);
```

Figure 5: Functions used in the GUI library

4.7 Integration

As described above, many audio modules were needed in this project. Integrating them was relatively straightforward, as most communicated with basic parallel signals. To ease compatibility and debugging, all interface signals were timed based on the 100MHz system clock.

Integrating the Microphone and FFT with the HDMI flow was challenging, since the FFT was large and it wouldn’t fit on the same board as the HDMI setup. To get around this issue, we put the mic and FFT on one board and the HDMI flow on the other board. We transferred the 16-bit frequency value between the two boards using wires connected to 16 PMOD pins on each board. On the receiving board, we wrapped the PMOD pins in an AXI slave IP, which we then memory-mapped into the Microblaze address space. This setup allowed the GUI library, which was running on the Microblaze on one board, to read the output of the FFT from the other board. This integration method met our requirements for speed and kept the GUI up to date based on the FFT output.

5.0 Design Tree Description

Project Github Repository

Our hardware project is structured in a set of folders. The following list shows the relevant directories:

- HDMI_gui_project: contains source code for the HDMI setup and GUI display
- Trumpet_project: contains source code for the audio processing including the SPI controller, and audio playback/record
- doc: contains final demo videos and final demo presentation slides as well as datasheets for various components in our system. This group report will also be in this folder.
- fft_sw: contains code for 32-point FFT version in Python
- fft_32_fft_128_source_files: contains source code for the 32-point and 128-point FFT custom IP blocks
- Mic_FFT_project: contains the Vivado project for the mic and 128-point FFT
- warmup_demo: contains warmup demo project for having a TCP client on MicroBlaze

Setup Instructions for Audio Record/Playback (Trumpet Project):

To use the audio part of this project, we need the SPI microphone PMOD, some standard wired speakers and the capacitive button PMOD. The microphone needs to be plugged into PMOD port JB and the capacitive button PMOD into port JA of the Nexys video used for audio processing. Plug the speakers into the line out (green) jack. The MicroBlaze needs to be flashed and its code must be running for the audio codec to output any audio.

Setup Instructions for the Mic_FFT_project:

The Mic_FFT_project has the same audio setup as the trumpet project, except the PMOD microphone location is moved to PMOD port JXADC. The FFT_128 frequency pwm output is displayed on LED3, while the actual frequency output is output onto PMOD port JB and PMOD port JC. Since the mic doesn't need any configurations and FFT takes control signals from the SPI mic controller, no setup is needed other than loading the bitstream onto the board.

Setup Instructions for the HDMI_gui_project:

Detailed instructions for setting up the HDMI GUI project can be found in the README for that project.

6.0 Tips and Tricks

Synthesis schematics: Before going too far in the implementation process and hardware testing, spend a few minutes reviewing the schematics generated from synthesis to check that pins or signals are not tied to a supply.
Restarting SDK after it Crashes:The Vivado SDK in 2018.2 and 2018.3 has a tendency to crash fairly frequently on windows machines, and many times after it crashes it fails to start up again, crashing without any error message immediately upon launch. When this happens go into your project directory and navigate to *<project_name>.sdk\.metadata\.plugins\org.eclipse.core.resources* and delete any files that have .snap as their extension. Once these files are deleted the sdk should launch again. This solution was given as an answer on the Xilinx forums here: https://support.xilinx.com/s/question/0D52E00006iHvUKSA0/xilinx-sdk-20182-crashes-quietly?language=en_US, but it wasn't easy to find, and we helped multiple other teams with this issue in the leadup to the demo, so we thought it was worth including.

7.0 Video

We have two demo videos, one showing our microphone and FFT integrated with the HDMI and GUI, and another showing our audio recording and playback. These videos can be accessed at the links given below:
FFT and HDMI
Audio Recording and Playback

8.0 Appendix A

Proposed Schedule

- Week 1 (01/22-01/28):
  - Vivado project setup and all IP blocks instantiated (operational settings may change)
  - Setup GitHub repository (appropriate .gitignore, etc.)
  - Collect datasheets and reference documents for HDMI, SPI, I^{2}C, DSP blocks, audio interface

- Week 2 (01/29-02/04):
  - Determine detailed interfaces between all blocks (memory-mapped, simple GPIO, AXI, etc).
  - Working SPI MEMS microphone
  - Note: “Working” implies that the data is accessible (and accurate) through defined interfaces
  - DSP interface integrated into the project (i.e. can receive streams of data and perform desired computation)
  - FFT Python prototype (to validate the data processing algorithms, we will prototype them using software). May need to add filtering or equalizer filters to audio.

- Week 3 (02/05-02/11):
  - Working I^{2}C capacitive buttons
  - Frequency stretching modulation Python prototype
  - Microphone audio can be saved to memory as controlled with I^{2}C buttons

- Week 4 (02/12-02/18):
  - Output HDMI signal recognized by monitor
  - Required audio processing algorithms can be performed by DSP blocks controlled by MicroBlaze
  - Audio port outputs constant-frequency signal (controlled by MicroBlaze)

- Week 5 (02/27-03/03 = Mid-project demo):
  - Output desired graphics over HDMI, with arbitrary animation controlled by MicroBlaze
  - Prepare demo:
    - Audio from the microphone fed through FFT and frequency calculated output on the terminal.
    - I^{2}C buttons showed as reading accurately and accessed by MicroBlaze.
    - Constant frequency output on audio jack
    - “Hello World” on HDMI

- Week 6 (03/04-03/10):
  - Integration: System controlled by buttons, HDMI output fully functional with audio output

- Week 7 (03/11-03/17):
  - Buffer week for delayed milestones

- Week 8 (03/18-03/24):
  - Buffer week for delayed milestones

- Week 9 (project to be completed by April 1st):
  - Animated trumpet responding to live audio and metrics on GUI
  - Record and audio modulation, followed by playback through audio port
  - Film demo

9.0 Appendix B

Project Schedule

Week 1 (Feb 5)

- Found documentation for mapping trumpet notes to frequencies, and for HDMI version 1.3 (newest publicly accessible version).
- Researched DAC and DSP blocks, as well as how to use the Audio interface by referencing documentation. Test Vivado project setup on remote ECF machines
- Looked into version control for Vivado projects, specifically setting up .gitignore and TCL scripts for building Vivado projects and SDK projects.
- Found documentation for the Nexys-Video board, PMOD interface, and IPs (SPI, I2C, I2S, FFT, HDMI).

Week 2 (Feb 12)

- Set up Digilent HDMI example project in Vivado 2018.2 to generate an HDMI example design for Xilinx HDMI 2.1 Transmitter subsystem.
- Implemented audio preprocessing with sampling and low pass filter to remove noise with the help of MATLAB (using Parks-McClellan algorithm) to generate the filter coefficients.
- Implemented FFT (and STFT) with Numpy and plotted both frequency vs amplitude and frequency vs time graphs. Determined an FPGA FFT implementation/algorithm.
- Integrated I2C IP into Microblaze codebase, and configured AD7156. Added GPIO IP for capacitive button module output, accessible with Microblaze.

Week 3 (Feb 19)

- Digilent HDMI demo project running on the Nexys Video using Vivado 2018.2. Displays test pattern, forwards output from computer HDMI port to monitor.
- Testbench for audio preprocessing and generated frequency response for filter coefficients. Implemented SPI controller module.
- Completed the software algo implementation, starting testbench with the FPGA design.
- 90% Developed AXI slave I2S + I2C driver for audio output.

Week 4 (Feb 26)

- Figured out data flow through block diagram of Digilent HDMI demo. Looked into different options for connecting our image logic to the existing HDMI flow.
- Testbench for SPI controller module and started microphone integration.
- Design fsm to “take” audio data and write to bram in bit-reverse order, a “butterfly” calculation unit to do the multiplication and addition (half-working).
- I2S, I2C modules for audio codec and small C library for them. GUI assets painted.

Week 5 (Mar 4)

- Concluded that the best way to get images onto board is to load them into 512 MB DDR3 on board, where the VDMA and the microblaze can read them. Will load images from microSD using SPI.
- Integrated preprocessing module (12-bit) with SPI controller ready to feed FFT. Created test GUI using lightweight graphics library for trumpet and display information like frequency.
- FFT working and standalone project can output a frequency with dummy data as input seeing on the ILA, no output seen while connected to the whole design
- Audio codec works without AXI, completed BRAM recording/playback module

Week 6 (Mar 11)

- HDMI is working with an image loaded into DDR using the XSCT console. Once the image is first loaded into memory, it can be displayed to a monitor from the output HDMI without the UART cable or the input HDMI cable connected.
- Retrieved pixel information from GUI setup from last week to format into efficient colour format into an array to be processed on MicroBlaze. Tested by printing pixels to screen.
- Able to use ILA properly to see frequency and incoming audio signals. Output frequency on a led as pwm.
- Since we use DDR for GUI asset storage, we simplified GUI generation logic to rely on pre-made trumpet valve frames. I made images for the 8 possible notes.

Week 7 (Mar 18)

- Got HDMI Running on 2018.3, with valves switching based on input of UART, either with an unpredictable delay or with only 3 valve combinations available.
- Pixelated all possible combinations of letters and numbers and developed binaries with correct positional offsets to match GUI valve combinations to overlay text.
- Rewrote the FFT to be hardwired so that there are fewer control signals. FFT can measure around the correct frequency when given certain frequency tones at the mic input.

Week 8 (Mar 25)

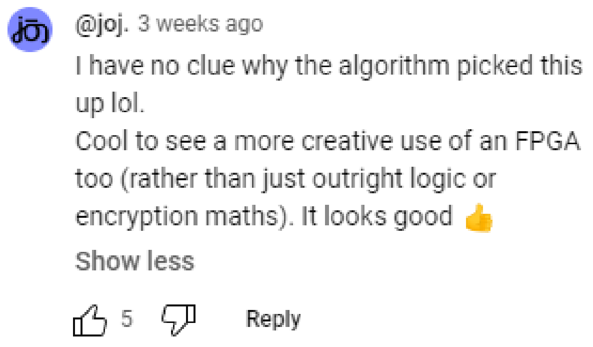
- HDMI displaying different valve combinations either based on uart inputs or FFT, running in 2018.3.
- Integration of text overlay with HDMI display and ensure FFT reading is displayed
- Working FFT with reasonably correct frequency output.
- Final report skeleton.

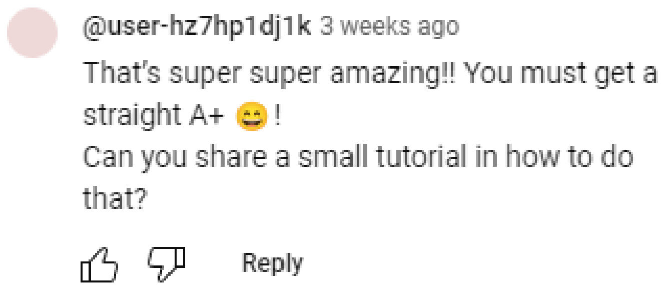
Week 9 (Mar 31)

- Integrate HDMI into the rest of the project in preparation for the final demo.
- Prepare for the final demo by testing the whole project and work on the final report.
- Help with integration. If time allows, try to fix the clipping issue without the workaround.
- AXI slave for FFT output such that it can be read by Microblaze. Full integration.
- Final Demo Week!

10.0 Appendix C

Our midproject demo video somehow got a little bit of traction on YouTube, and some users left some great comments, such as:


@joj. 3 weeks ago
I have no clue why the algorithm picked this up lol.
Cool to see a more creative use of an FPGA too (rather than just outright logic or encryption maths). It looks good 👍
Show less
5 Reply


@user-hz7hp1dj1k 3 weeks ago
That's super super amazing!! You must get a straight A+ 😄 !
Can you share a small tutorial in how to do that?
Reply