

SpectroRun: Final Report

Dougy Ouyang

Department of EECS

Massachusetts Institute of Technology

Cambridge, MA, USA

dougyo@mit.edu

Kyle Fu

Department of EECS

Massachusetts Institute of Technology

Cambridge, MA, USA

kylefu@mit.edu

Brayden Reece Leggett

Department of EECS

Massachusetts Institute of Technology

Cambridge, MA, USA

bleggett@mit.edu

Abstract—SpectroRun is an interactive spectrogram game rendered via HDMI, where the player controls a character that dodges obstacles in a gameplay environment that is procedurally generated and evolves in real time based on both prerecorded audio data and live microphone input. This project serves as both a source of artistic inspiration and a fun way for students to de-stress after long projects.

I. HIGH-LEVEL OVERVIEW

This project implements real-time audio processing and video output on an FPGA, requiring data-flow management, communication protocols, signal processing, and video generation. It starts with audio processing which retrieves prerecorded song data from an external SD card, where it is mixed with live microphone inputs and passed to the DFT module. This then computes the frequency-domain representation of the mixed signal which is used to generate a spectrogram. The spectrogram is both displayed on screen and treated as a (non-normalized) probability distribution to influence enemy spawns in a game. A visualization of the game is shown in Fig. 1, with the spectrogram at the top in red (and its derivative over time in yellow), falling square enemies, and the player at the bottom. The player is controlled via a PS/2 keyboard to move back and forth across the screen and dodge the enemies.

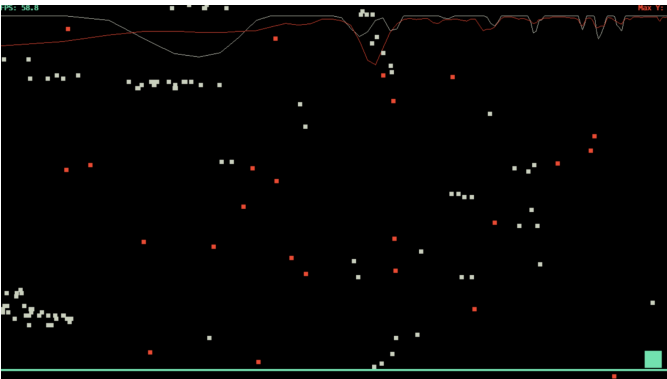


Fig. 1. Screenshot of a Python SpectroRun prototype.

II. PERIPHERALS

A. PS/2 Keyboard

The keyboard used for player movement and song select uses the PS/2 communication protocol. There are two signals

for PS/2 communication: Data, and Clock. Both signals are meant to be open-collector, where either the host or peripheral device can pull the signals low and pull-up resistors drive the voltage up otherwise. However, since we only need one way communication from the keyboard to the FPGA, the signals were simply taken as inputs from the keyboard.

The PS/2 communication protocol is similar to UART in that it only needs clock and data signals. Unlike UART, there is no agreed-on BAUD rate. Instead, the clock is generated by the keyboard. Because this clock is external to the FPGA and asynchronous, a dual flip-flop buffer is required for edge detection to prevent metastability issues. The communication happens in 11-bit messages from peripheral to host and 12-bit messages from host to peripheral. Each message consists of 1 start bit, with the data line being pulled low, 8 data bits, 1 odd-parity bit, and one end bit, with the data line returning high. As previously mentioned, if the host wants to send a message to the device, it would pull down the clock line to inhibit communication and then run the same protocol with the addition of an acknowledgment bit from the peripheral to the host at the end of every message. As this functionality was not necessary, we omitted the capability of the FPGA to output to the two lines.

PS/2 communication devices normally run on 5V and we were advised by Professor Joe Steinmeyer to follow this. Even though the keyboard says that it can run on 3.3V-5V power our project uses the FPGA 5V servo power supply instead of using the 3.3V from the FPGA's logic drivers. The two signals pass through a Logic Level Converter to change from the 5V out of the keyboard to the 3.3V input into the FPGA.

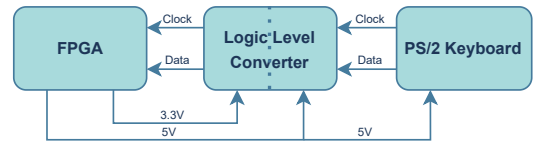


Fig. 2. PS/2 Keyboard Wiring Diagram

B. I2S Microphone and Speaker

The I2S protocol uses WS (word select), SCK (serial clock), and 1 data pin (DIN or DOUT). The protocol is similar to SPI, with SCK being DCLK (data clock), WS being similar to

CS (chip select) but switching continuously instead of waiting for a trigger, and DIN and DOUT corresponding to CIP0 and COPI respectively, but outputting data on the left and right channels based on WS. The I2S ICS-43434 microphone outputs 24-bit signed PCM audio and requires WS to have a period of 64 SCK cycles. We set the I2S MAX98357A speaker amp up to accept 24-bit signed PCM audio output, which required WS to have a period of 48 SCK cycles. Both devices were configured to use only the left audio channel by leaving SEL on the microphone unconnected and wiring a 110 k Ω resistor from the 5V Vin port to SD (to get around 2.4 V on the pin). The speaker was set to 6 dB gain by connecting GAIN to Vin. As this was a late addition to our project, we ran the I2S transmitter and receiver directly off the 100 MHz clock, configuring the period of SCK to get an approximate 44.1 kHz sampling rate for each device (± 1.2 kHz), and we further quantized the audio input to send unsigned 8-bit PCM samples through the main pipeline. Our pipeline always uses the latest values available when communicating with the microphone and speaker. The I2S modules were testbenched with manual waveform inspections, and in an audio-only top level, we did not notice any artifacts resulting from the slightly different sampling rates.

III. AUDIO PROCESSING

The audio-processing section retrieves song data from external storage and mixes it with a microphone source before sending it to the rest of the pipeline. The song data is stored on a 2 GB microSD card which we interact with through SPI using an SD controller module written by Jonathan Matthews and provided to us by MIT 6.205 staff. A music controller handles the addressing and timing of requests which send the music data into a FIFO. The data is sent out from here at a 44.1 kHz frequency matching the song sample rate. A PWM driver delivers the sound to the FPGA audio jack and the audio samples continue on to the next stage. We have one song stored on the microSD card and were able to get it to play out of the FPGA audio jack.

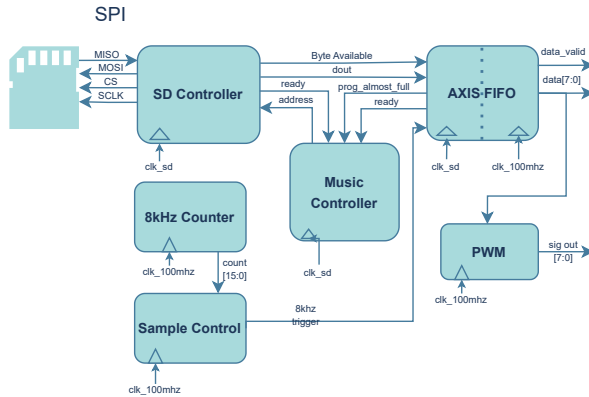


Fig. 3. Audio Processing Block Diagram

A. SD Card

The SD card is used as external storage for the song data to allow for the storage of multiple songs. The SD controller module gives direct access to the SD memory in 512-byte sectors where songs are stored sequentially to simplify accesses. Near the end of the project, we ran into an issue where the audio became infrequently garbled and static was playing during the silence. This was determined to be an issue with the SD card and, after acquiring a new one and re-copying the songs over the issue was fixed.

B. Music Files

The songs are WAV files that were originally preprocessed before storage to be mono-audio 8-bit 8 kHz frequency in order to match the MAX9814 microphone whose audio it was mixed with. In the final iteration, once the I2S microphone capable of playing 44.1 kHz was set up, the songs were changed to be mono-audio 8-bit 44.1 kHz frequency.

C. Music Controller

The music controller handles the logic for requesting song data from the SD card. The start and stop addresses of each song are recorded and the music controller runs until it hits the end of the song, in which case it restarts the selected song. This module controls the flow of reads to avoid overfilling the FIFO by using *prog_almost_full* that signals when there is less than 512 spaces free in the FIFO. To switch songs, start and end addresses of the song are provided and the song begins once a start signal ceases. These addresses correspond to the saved values from uploading to the SD card from a laptop. They are stored in our *top_level* file. Switching between songs relies on keyboard inputs; W cycles between the songs and S starts playing the selected song.

D. FIFO Buffer

The SD Card runs on a 25 MHz clock and has variable latency in its accesses so we use an AXI-Stream FIFO both to convert to the 100 MHz system clock and give consistent access to future modules. It has a width of 8-bits and a depth of 2048 so that it can store four SD sectors' worth of data. Instead of a ready signal from the next module to signal to the FIFO, we have a trigger module that goes off with an approximately 44.1 kHz frequency to request the song data (44.092 kHz).

IV. DISCRETE FOURIER TRANSFORM

A. Overview of DFT Logic

The DFT component sits in the middle between the Audio Processing and Game Logic stages. The main function of DFT, as its name suggests, is to take the time-domain data produced by the audio pipeline and microphone, transform it into the frequency domain, and pass the resulting values to game logic for display on HDMI.

The overall block diagram (shown in Figure 3) for the DFT stage consists of four modules: DFT Computation, Magnitude Squared, Square Root, and a Frequency Buffer module. The

DFT Computation module takes in 8 bit data samples (from the microphone and wav files), stores them in a BRAM, and runs a Discrete Fourier Transform on each 8192 sample-window as soon as the previous computation finishes. The Magnitude Squared module takes the outputs (complex numbers) from the DFT Module, and calculates their magnitudes squared. We separated out this module because squaring 32 bit numbers is computationally expensive and requires pipelining to meet timing constraints. Next, we pass those values into a square root IP module (CORDIC IP) to get the actual magnitude of each frequency bin. Finally, we pass the resulting frequency-domain data to a frequency buffer module, which stores the magnitudes and allows the game logic to look up specific frequency bins whenever needed. All components follow the standard AXI handshake mechanism and support a blocking flow control.

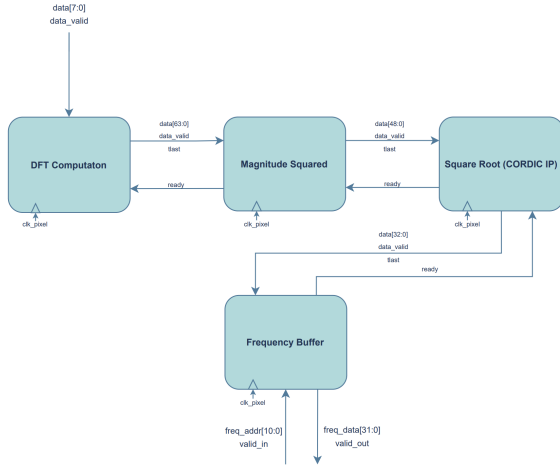


Fig. 4. Block diagram for the DFT stage.

B. DFT Computation

It is worth examining the DFT Computation module in greater detail. Although this module is primarily responsible for performing Discrete Fourier Transformations, it also has to take care of managing data collection. The microphone samples at a rate of 40 kHz, meaning that collecting a full 8192 sample-window would require 1/5th of a second. However, our goal is to have the DFT module produce a new result every time the monitor refreshes, or 1/60th of a second. Since our audio samples do not arrive quickly enough to populate a 8192 sample-window within that time frame, we must reuse previous collected values.

Our solution is to have a circular BRAM buffer that is twice the size of the sample window (16384 entries). Incoming samples are written sequentially in this buffer and wrap around once the end is reached. Whenever we start a new DFT computation, we simply take the most recent 8192 values from the BRAM; this would guarantee that we could get a 8192 sample-window from the circular buffer whenever we needed it, without having to wait for 8192 samples every time we do a

new computation. Furthermore, we do not need to worry about the buffer overwriting data needed for the current computation because the sample collecting rate is a lot slower than the time it requires to finish a DFT computation.

With this setup, the DFT module is organized into a state machine with the following four states: IDLE, CAPTURE, COMPUTE, and OUTPUT. After reset, the module transitions from the IDLE state to the CAPTURE state, where it waits for 8192 samples to be collected. Once this is ready, it moves on to the COMPUTE stage to perform the actual transformation. After computation, it moves to the OUTPUT state to pass the results downstream. Once OUTPUT is complete, it returns to the CAPTURE stage to get the next 8192 sample-window, and the process repeats itself.

Lastly, for our DFT algorithm, we initially implemented a quadratic $O(N^2)$ algorithm to test and validate our overall design. The algorithm itself is very straightforward: for each frequency bin, we're simply summing up the products of every input sample with it's corresponding root-of-unity. We kept track of the real and imaginary parts separately, and used fixed-point representation to represent decimal numbers. We precomputed the 8192 roots-of-unity and stored them in a .mem file, which then gets loaded into a BRAM lookup table when the program starts. The figure below shows the DFT formula written in Euler's notation.

$$A_k = \sum_{m=0}^{n-1} a_m \exp\left\{-2\pi i \frac{mk}{n}\right\} \quad k = 0, \dots, n-1.$$

Fig. 5. DFT Computation

The DFT module was testbenched against NumPy's FFT implementation and matches the expected result within an error margin of 0.1% (due to fixed-point representation).

C. Fast Fourier Transform

Since an $O(n^2)$ implementation is relatively straightforward, we chose to pursue a more efficient $O(n \log n)$ FFT algorithm for the goal version of our project. To achieve this, we decided to implement the commonly used Cooley-Tukey FFT algorithm on our FPGA. Normally, the Cooley-Tukey algorithm is understood and implemented via recursion; however, lucky for us, there exists a relative simple iterative formulation of this algorithm which is better suited for hardware implementation. Below is the pseudocode (from the Cooley-Tukey wikipedia page) we referenced when implementing FFT.

One important detail in the pseudocode is that each iteration of the inner loop (a butterfly operation) requires two reads and two writes, but our BRAM only has two ports! We could wait an extra cycle for the BRAM to process all four memory operations, but we would be wasting "valuable" cycles and introduce a lot of unnecessary complexities, especially once pipelining is involved. Our solution is to use a "ping-pong" ram design - using two BRAMS that alternates roles between

```

algorithm iterative-fft is
  input: Array  $a$  of  $n$  complex values where  $n$  is a power of 2.
  output: Array  $A$  the DFT of  $a$ .

  bit-reverse-copy( $a$ ,  $A$ )
   $n \leftarrow a.length$ 
  for  $s = 1$  to  $\log(n)$  do
     $m \leftarrow 2^s$ 
     $\omega_m \leftarrow \exp(-2\pi i/m)$ 
    for  $k = 0$  to  $n-1$  by  $m$  do
       $\omega \leftarrow 1$ 
      for  $j = 0$  to  $m/2 - 1$  do
         $t \leftarrow \omega A[k + j + m/2]$ 
         $u \leftarrow A[k + j]$ 
         $A[k + j] \leftarrow u + t$ 
         $A[k + j + m/2] \leftarrow u - t$ 
         $\omega \leftarrow \omega \omega_m$ 
      end for
    end for
  return  $A$ 

```

Fig. 6. FFT Pseudocode

reading and writing in each stage. This would ensure that both reads and writes happen within one clock cycle without stalling, and significantly simplifies our pipelining logic.

With this setup, we created another state machine to support the actual computation: FFT_IDLE, BIT_REVERSE, FFT_COMPUTE. The state machine sits idle whenever our main DFT logic is not in its COMPUTE state. Once triggered, the state machine would first move to the BIT_REVERSE phase, where it performs the bit-reversal permutation required to reorder the samples into the correct layout. In our code, this is achieved by storing the samples in their bit-reversed address location in the ping-pong ram when we first load it from our circular buffer (reversing bit address can be done entirely with wiring). After this stage, the state machine would move on into the FFT_COMPUTE phase where it iteratively processes each FFT stage, performing the butterfly operations and swapping the roles of the BRAMS after each stage. We pipelined each butterfly operation into six stages, three for bram accesses, one for the multiplications, and two for the addition and subtractions. We also precomputed the twiddle factors in python and stored them in a .mem file using 16-bit fixed-point precision. We have variables that keeps track of the correct twiddle factor address to read from and how the twiddle factor “moves” within each stage. Once the FFT computation is finished, we would signal the main state machine to move into the OUTPUT stage where it passes the results to the next stage at once.

The FFT module was testbenched against NumPy’s FFT implementation and matches the expected result within an error margin of 0.1% (due to fixed-point representation).

D. Hanning Window

To make our spectrogram look extra nice, we applied a Hanning window prior to our FFT to help reduce spectral leakage and noise components. This is accomplished by multiplying the samples with the hann function $w(n) = 0.5[1 - \cos(\frac{2\pi n}{N})]$ at the moment we store the samples into the ping-pong ram. As with the twiddle factors, we precomputed the hann function

coefficients with 16-bit fix-point precision and used a BRAM lookup table during loading.

NumPy provides a .hanning() function which applies the hann window to a set of samples. We used this to verify our windowed FFT, and our result matches the expected result within an error margin of 0.1% (due to fixed-point representation).

E. Frequency Buffer

Once the results are computed, we store them in a frequency buffer, allowing downstream modules to access the frequency-domain magnitudes as needed. We initially planned on implementing a double frame buffer approach to ensure that data does not get overwritten before the display could fully capture all the frequency bins. However, after we finished implementing FFT, we decided on using just one buffer since the FFT is much more performant than our previous implementation. We decided to just wait until the FFT finishes its computation before reading from the frame buffer, stopping FFT updates until the display has fully captured all the frequency bins. This approach is a lot less bug-prone and simplified our design significantly.

V. GAME CONTROLLER

A. BRAM Frame Buffer

We decided to use a double frame buffer approach for rendering to the HDMI display, as attempting to render a single pixel at a time would be infeasible with many sprites, and rendering would take too long to fit into the HDMI blanking period, ruling out a single frame buffer. We started with a simpler, 320 x 180 16-bit RGB565 BRAM frame buffer manager, as shown in figure Fig. 7.

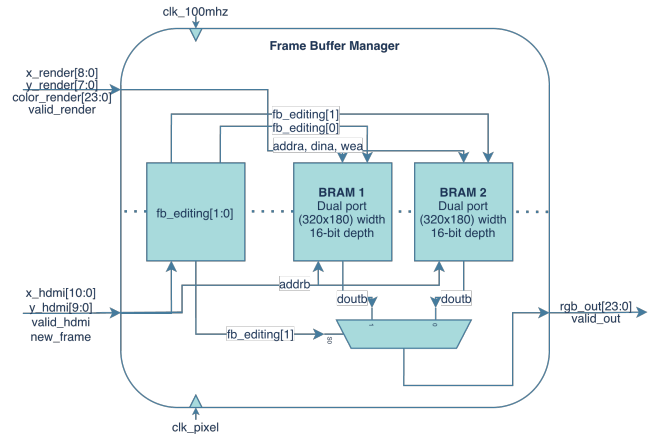


Fig. 7. Block diagram for the BRAM frame buffer manager.

For a given frame, 1 BRAM is modified by the game controller, while the other BRAM is read out to HDMI. On a new frame, the BRAMs are effectively swapped using a one-hot register with clock domain crossing. This repeats for each frame. This module passed a thorough testbench.

B. DRAM Frame Buffer

In order to render the game at 720p resolution, we used a 1280 x 720 16-bit RGB565 DRAM double frame buffer, as shown in Fig. 8. We opted to use RGB565 instead of a compressed color palette, as we thought we might use more than 256 colors, and the DRAM is easier to interface with when the frame buffer is byte-aligned, motivating the use of 2 bytes per pixel.

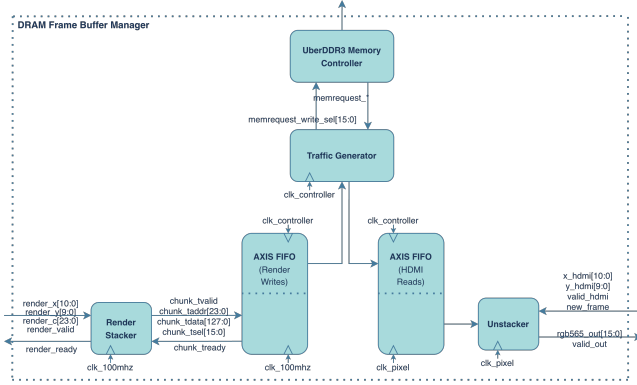


Fig. 8. Block diagram for the DRAM frame buffer manager.

The DRAM interface design mirrors the one used in Lab 6, but with a few important changes. The two frames are stored contiguously in DRAM memory, so the traffic generator simply reads up to twice of the original max address to alternate HDMI reads between frame buffers. There is an additional `WR_CLEAR_SCREEN` state in the traffic generator's FSM, which clears the pixels in a frame buffer right after the HDMI read for those pixels is returned. Single pixel, 2 byte render requests are combined into 16 byte requests when possible using the render stacker module, which outputs all the info needed for a DRAM write request, in order to reduce the load on the UberDDR3 controller. The write byte strobe is used to write only the correct bytes if they cannot be combined into a full, 16 byte write. These render write requests are sent to the memory controller in the `WR_RENDER` state of the traffic generator. The render stacker and traffic generator modules were testbenched with manual waveform inspections, and the correctly pipelined DRAM interface passed an on-board testbench that renders a test square and a 1 pixel wide border pattern to the HDMI display.

C. Sprite Spawning and Rendering

We took a major-minor FSM approach to rendering sprites, where the game controller stores a BRAM array of sprite data, iterates through the list of sprites, and calls a different handler module for each sprite type. The handler module renders up to 1 pixel per cycle, returning the potentially updated sprite data at the end. We opted for this approach because although it leads to inefficient resource utilization (as only 1 of the many handler modules are active at a time), it made the SystemVerilog implementation easier to modularize and

develop. A separate module spawns enemies, communicating with the game controller using AXI Stream. Fig. 10 shows the block diagram for the game controller, and Fig. 11 shows the FSMs for the involved modules.

To render and update a frame, the Ground Sprite draws the HP bar at the bottom, then a Player Sprite draws the player on the ground. Next, the Spectrum Sprite renders the audio's frequency spectrum in the top portion of the screen. Finally, the Enemy Sprite is repeatedly used for each enemy, which renders the enemy and moves it according to its velocity. We initially used a Clear Screen Sprite to clear the screen, but had to move this into the DRAM traffic generator to render at 720p. At the start of this process, the Enemy Spawner uses a linear-feedback shift register (LFSR) for pseudo-randomness, querying the audio's frequency spectrum to spawn enemies with random velocities where the frequency spectrum has higher magnitudes. Due to the gameplay being better than expected without the need for spectrogram smoothing or derivative tracking, we decided to simply use/render the lower portion of the frequency buffer instead. The game controller passed manual waveform inspections and visual checks of the rendered sprites with mock data.

VI. INTEGRATION AND EVALUATION

A. Integration

We connected our audio processing, FFT, and 720p game controller sections in sequence to create a working SpectroRun game on the FPGA. The game is shown in Fig. 12, with the wiring setup in Fig. 9. The speaker output is signed 24-bit 44.1 kHz, while the 8192-sample FFT uses unsigned 8-bit 44.1 kHz. The game is updated and rendered at 720p 60 FPS.

B. Timing, Latency, and Throughput

Our design meets timing with a WNS of 0.278. The latency of the audio processing portion is negligible compared to the latency of later modules. The latency of the 8192-sample FFT pipeline is approximately $8192 \log_2(8192) + 8192 \approx 111500$ cycles, since the FFT takes $n \log n$ cycles with no constant factor and the post-processing output phase takes approximately n cycles. The FFT only operates on 1 batch of samples at a time, so its throughput is approximately $1/111500$. We continuously restart the FFT calculation when it finishes, ensuring that the data pulled from the frequency buffer is never more than 1 frame old. Through outputting cycle counts to the seven segment display, we empirically determined that each frame's rendering latency is around 221 000 cycles with 1000 enemies, well under the theoretical $10^8/60 \approx 1\,600\,000$ cycle limit before the HDMI display needs the next frame. Most of these cycles are spent rendering pixels, but the render stacker is able to combine many of the pixel writes in a horizontal line into 16-byte chunk writes. Empirically, the DRAM memory controller was never close to being the bottleneck, as the FIFO for render writes was nearly empty most of the time, and it passed a testbench that rendered far more chunks. However, since frame rendering only takes place once per frame, the

worst case end-to-end latency, from audio input to HDMI output, is a bit less than 2 frames (around 33 milliseconds).

C. Resource Utilization

Our design uses 6586 LUTs (20%), 6495 flip-flops (10%), 67.5 BRAM blocks (90%), and 11 DSP blocks (9%). The high BRAM usage is due to the 8192-sample FFT, which uses 52 of the BRAM blocks: 36 for the ping-pong BRAMs, 8 for the precomputed twiddle factors, 4 for the precomputed Hanning window factors, and 4 to store the frequency buffer output. It also uses 9 of the DSP blocks. This resource usage could be reduced by lowering the FFT sample size, lowering the fixed-point decimal precision from 14 decimal places to a lower number, and by computing the twiddle and Hanning window factors on the FPGA instead. If an even larger FFT is needed, the intermediate computations could be stored in DRAM instead of BRAM at the cost of higher latency, since DRAM has a larger memory capacity.

D. Project Checklist

Overall, we were able to meet all of the goal requirements we set up in our presentation. We have a fully working audio processing unit that selects songs from the sd card, and records microphone inputs and passes the result data samples to the DFT unit. Our DFT unit is also fully functional and achieves the theoretically optimal $O(n \log n)$ time complexity with no constant factors in front (constant factor=1). The DFT unit also incorporated a Hanning window to help reduce spectral leakage and noise on the spectrogram due to inherent sample alignment problems, as discussed in our presentation. The FFT we implemented fully met timing and was test benched against python's FFT implementation, confirming correctness. The game controller unit is fully polished with 720p DRAM resolution and presents the user with a nice looking gaming experience. We integrated the ps/2 keyboard as well to make the playing experience feel more natural and enjoyable. In fact, we went beyond the goal to meet part of our stretch portion as well. For instance, we integrated an I2S microphone which samples at the higher 40kHz frequency, ensuring both smooth and responsive looking spectrogram than the naive lab provided 8-bit microphone. We did not have time to implement multiplayer game play because doing so would require us to handle communication between multiple keyboards/FPGA and perhaps even coming up with our own communication protocol. Thinking back, this goal was perhaps overly ambitious to squeeze in a 7 week project, but nonetheless is an interesting concept that future projects could attempt to explore.

E. Additional Insights

After having made the music controller, it really would have been better to come up with a FSM to handle state changes and accept or block requests when necessary before designing the module. The PS/2 interpreter could have also been two modules in order to keep it more general in case we wanted to add mouse support for another game in the future.

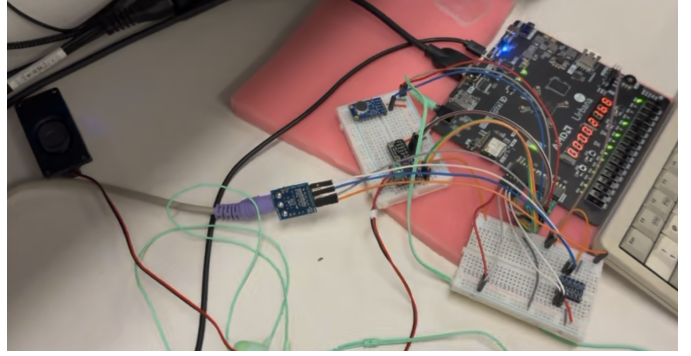


Fig. 9. Photo of the wired hardware.

For the DFT portion, we would like to point out how easy it is to blow timing requirements. When we first wrote the DFT module, we did not consider too much about timing, and we incorporated a 32-bit multiplication, a 32-bit addition, and a muxes within a single pipeline. This obviously blew timing, and led to a lengthy process to re-pipeline our DFT stages. In the future, we want to be more careful about timing restrictions so we don't run into the issue of having to redo our pipeline over and over again.

For the game controller portion, combining the sprite handlers would likely have made more sense, as it currently wastes resources (only 1 handler is ever active at a time).

VII. CONTRIBUTIONS

Dougy researched, implemented, and testbenched the entire DFT pipeline, from audio sample input to the frequency buffer output, using both an $O(n^2)$ DFT and an $O(n \log n)$ FFT. He also implemented a Hanning window prior to the DFT, helped debug the DFT during integration, and wrote the DFT section of the report.

Kyle was responsible for handling the entire game controller portion, the I2S microphone/speaker, DRAM double frame buffering, project integration, and writing the I2S, game controller, and the first 3 Integration and Evaluation sections of the report.

Reece handled communication with the SD card and information flow as well as communication and hardware for the PS/2 keyboard. Reece also worked with project integration and wrote the PS/2 and Audio Processing sections of the report.

VIII. CODE REPOSITORY

The SpectroRun Github repository and source code can be found at <https://github.mit.edu/6205F25/fa25-6205-team39>.

IX. ACKNOWLEDGMENTS

We would like to thank Professor Joe Steinmeyer, Yoland Gao, and the rest of the 6.205 teaching/lab assistants for providing invaluable feedback and assistance throughout the project period.

