

Strange Sounds to Visuals (SSTV)

Janelle Cai

Department of EECS

Massachusetts Institute of Technology

Cambridge, MA, USA

jcai18@mit.edu

Noah Haefner

Department of EECS

Massachusetts Institute of Technology

Cambridge, MA, USA

nhaefner@mit.edu

Amy Shi

Department of EECS

Massachusetts Institute of Technology

Cambridge, MA, USA

amys0930@mit.edu

Abstract—Our system converts strange sounds to visuals. We display images transmitted using Slow Scan Television (SSTV) in the Scottie 2 mode. We decode the image line by line in “real-time” and display the lines on a monitor as they are decoded, taking advantage of the FPGA’s speed and parallelism to perform the signal processing needed for the task. To further increase the technical challenge of this project, the system also features a multichannel mode, where it can decode multiple images transmitted at once.

I. INTRODUCTION

Slow Scan Television (SSTV) is a communications method for transmitting images. It is often used in amateur radio as a fun method of connecting on the air, and historically it was even used for transmitting images from space. Similar to most means of video transmission, SSTV breaks an image into several lines which are transmitted sequentially, separated by a horizontal synchronization signal. To encode colors, multiple “channels” of information are sent sequentially for each display line. The frequency of the baseband signal encodes the value of each channel, and a predetermined amount of time is spent on each pixel, as determined by the chosen mode.

A. Purpose + Requirements

Our project features an SSTV receiver that can decode images transmitted. There are several modes through which images can be transmitted in SSTV; our implementation is for the Scottie 2 mode, which is a commonly used mode in North America. We demonstrate its capabilities by displaying the received image on a monitor using HDMI. Our system is tested on both perfect SSTV signals generated from a Python program, pySSTV [1], as well as a noisy recorded signal captured from an on-the-air transmission.

Besides decoding a normal SSTV transmission, our system also has a multichannel feature that can decode multiple images at the same time. This requires the input signals to be shifted and multiplexed in the frequency domain, so this input is also a program generated signal instead of a radio transmission.

This system will require DSP modules to accurately detect the transmitted frequencies, state machines to decode the pixel values, memory modules to store pixel values, and interfaces with HDMI to produce the output image. The following sections will detail each module and how their designs will meet the deliverable requirements.

II. SYSTEM OVERVIEW + BLOCK DIAGRAM

A high-level block diagram of our system can be found in Fig. 1. The system’s key modules are as follows:

- HF radio with audio output into an ADC
- Band pass filter
- Frequency Detector
 - Frequency Shifter
 - Hilbert transform
 - Derivative
 - IQ frequency extractor
- SSTV Decoder
- Display of decoded data using HDMI

III. BAND PASS FILTER

The purpose of the band pass filter is to reduce noise and attenuate frequencies outside of the expected range, which is between 1100Hz and 2300Hz for an SSTV signal. To incorporate a multichannel feature, the band pass filter can also filter between 3000Hz and 4200Hz for the signal of a second image. It takes in the input of switch 0 as shown in Fig. 1 to know when to filter for one versus two frequency ranges.

The band pass filter is implemented as a Finite Impulse Response (FIR) filter that utilizes 63 taps to achieve an attenuation of over -30dB in the stop bands. See Fig. 2 for the filter bands. First, a fixed value is subtracted from each input because the hardware setup introduces a DC offset that creates artifacts in the decoded image, as shown in Fig. 3. Then, the inputs are multiplied with coefficients designed for the desired range of 1100Hz to 2300Hz or 3000Hz to 4200Hz pass bands [2]. To avoid floating point multiplication, the coefficients are converted to 16 bit integers, and the total sum at the end is right shifted by 16 to account for the gain. Since modules later in the pipeline will also need DSPs, the band pass filter is pipelined to multiply the 63 taps in three stages using only 21 DSPs total.

When in multichannel mode, the module repeats the operations on the inputs a second time but with coefficients designed for the higher frequency range. To reduce the amount of memory needed to pipeline the stages between the multiplication and addition operations, the clock speed was reduced from 100MHz to 50MHz for longer compute time, and with a sampling rate of 12.5kHz, this means each sample has up

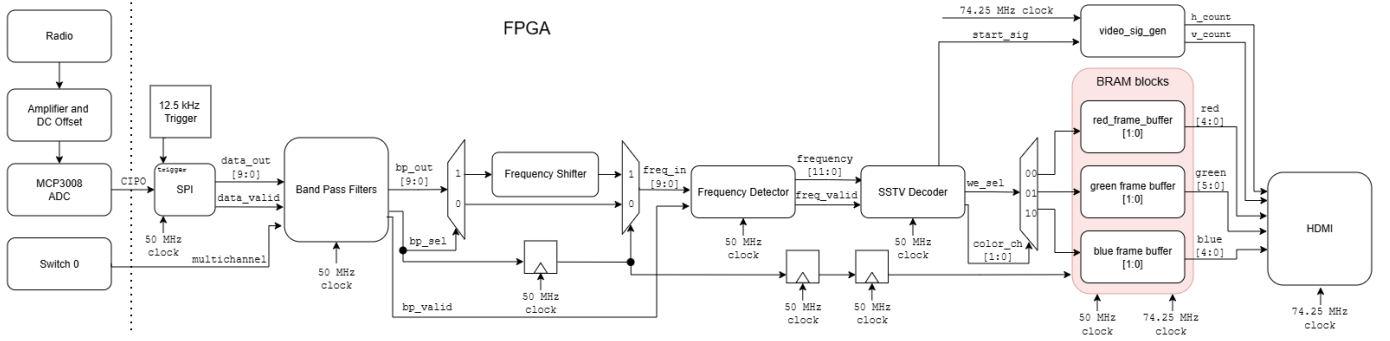


Fig. 1. The block diagram overview of our system. The radio receives the SSTV signal, which is then communicated to the FPGA using the MCP3008 ADC and the SPI protocol. The signal is then passed through a band pass filter and then a Frequency Detector. Then, the SSTV decoder uses the frequencies to reconstruct the pixel values in the image and store them in BRAM frame buffer, which is displayed using HDMI.

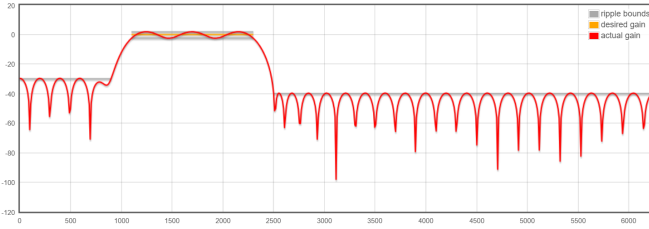


Fig. 2. Band pass filter with pass band from 1100Hz to 2300Hz [2].

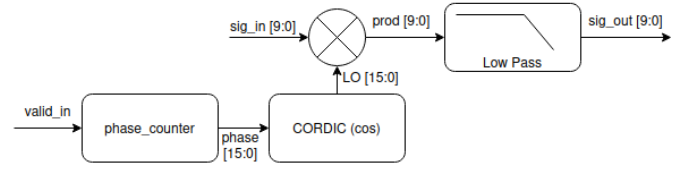


Fig. 4. Implementation of the Frequency Shifter

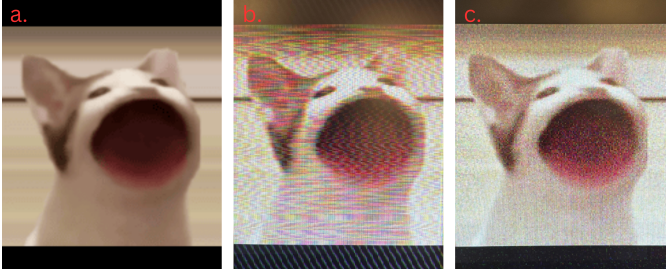


Fig. 3. (a) is the original image. (b) shows artifacts from the DC offset. (c) shows the decoded image after removing the DC offset with the band pass filter. The latter two images were the outputs on a monitor/screen captured by a phone camera.

to 4000 clock cycles for processing. This allows the filter to complete a single range band pass in 6 clock cycles and a multi-range band pass in 11 clock cycles.

IV. FREQUENCY SHIFTER

Because the top of the upper band is at 4200 Hz, about 70% of the Nyquist frequency, the limited bandwidth of the FIR filters implemented in Section V would make it difficult to directly estimate the frequencies of this signal. To solve this, a frequency shifter is implemented to bring the frequencies of the higher band down to 1100 Hz to 2300 Hz. This is done by multiplying the upper band's signal with a local oscillator at a frequency of 1900 Hz, which produces a signal that now spans from $[3000, 4200] \pm 1900$ Hz. This local oscillator is produced using a CORDIC module from the Xilinx IP library. A lowpass

filter isolates the desired band, producing the frequency-shifted signal.

V. FREQUENCY DETECTOR

The Frequency Detector takes the raw time-domain signal and estimates the instantaneous frequency for every sample of the input signal. Because the signal is fairly narrow-band, a typical FM demodulator process can be used. This involves using the in-phase (I) and quadrature (Q) components of the signal to find the phase, then differentiating to obtain an estimate of the frequency. However, because we have only a real signal as the input to the system, we must synthesize the quadrature component, which is done using a Hilbert transform. To accommodate the multichannel feature, two of these modules were instantiated.

A. Hilbert Transform

The Hilbert Transform is implemented as an FIR filter, the coefficients of which follow the pattern of $H_i = \frac{2 \sin(\frac{\pi i}{2})}{\pi i}$ [3]. It is then windowed by a Kaiser window, which reduces “waviness” introduced by the finite number of taps. Experimentally, it was determined that the filter needs 23 taps and a coefficient bit depth of 12 to achieve the desired frequency estimate accuracy.

B. Derivative

While a common approximation for a derivative is $\frac{dI_n}{dn} = I_n - I_{n-1}$, this approach proved insufficient for our needs. This straightforward derivative, often referred to as the first-difference filter, has a frequency response that falls off close to 1/4 of the Nyquist frequency. A sample rate of 12.5 kHz

with signals up to 2.3 kHz means we need a filter that is useful up to around 1/3 of the Nyquist frequency. This was achieved using a 9 tap FIR filter with the coefficients $D_i = \frac{(-1)^i}{i}$ for $i \neq 0$ $D_0 = 0$ [4] windowed with a Blackman window, which produces the frequency response show in Fig. 5.

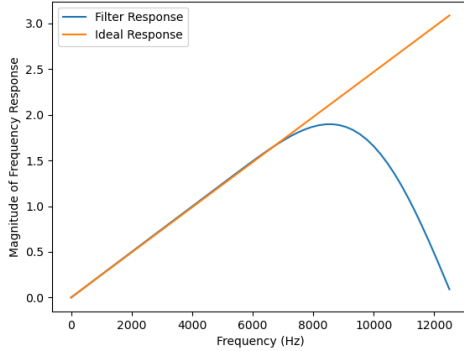


Fig. 5. Frequency Response of the derivative FIR filter

C. Frequency Extraction

The frequency could be calculated as $\frac{d}{dt}(\tan^{-1}(\frac{Q}{I}))$ (where the arctangent is unwrapped to produce a non-periodic phase), but then an arctangent would need to be computed. Instead, we expand the expression into $\text{freq} = \frac{I\dot{Q} - Q\dot{I}}{I^2 + Q^2} (\frac{f_{\text{sample}}}{2\pi})$, leaving only the division, which can be computed with a repeated subtraction divider given the ample time.

VI. DECODER

The decoder module reconstructs the image that is encoded by the instantaneous frequencies. The module can roughly be broken into two components:

- **VIS Decoder:** This checks for the Vertical Interval Signal (VIS), which indicates the start of an SSTV transmission. If the module detects a valid VIS, it sends a single-cycle high signal `begin_pix_decode` to the Pixel Decoder to indicate that the following frequency samples represent pixels and can be decoded.
- **Pixel Decoder:** The Pixel Decoder converts the frequency sample inputs into RGB values that make up the image. The Pixel Decoder also handles syncing between lines.

A. VIS Decoder

Each SSTV transmission begins by transmitting a series of frequencies for a predetermined duration, and the VIS Decoder checks for this pattern with a finite state machine (FSM) as shown in Fig. 7. Real-world signals can be noisy, so an 8 sample running-average is used to increase the resiliency of the detector.

The VIS Decoder breaks the VIS into four sections: WIDE ONE, GAP, WIDE TWO, and SIGNAL, as shown in Fig. 6. The FSM starts at the Detect Wide One state, where it checks if the past 100 samples contain a minimum threshold

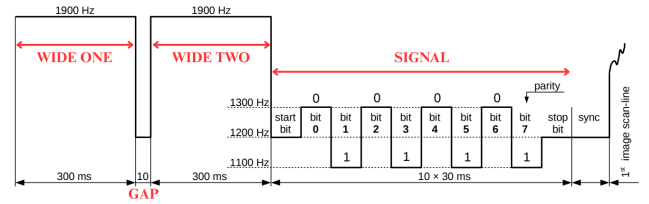


Fig. 6. Example structure of VIS with value 42. The red texts are the names given to the corresponding sections in the signal [5].

of samples within a range of 1900 Hz. 100 samples provide a long enough window to accurately assess if a VIS transmission has started but still short enough that it doesn't miss the real VIS due to a previous false positive. The rest of the FSM states correspond to the three longer sections of the VIS and the transitions between them. To verify the longer sections, the decoder checks for a minimum count that must be met before transitioning to the next state. Since the GAP section is much shorter, the decoder verifies this section by checking for the rising and falling edges instead of only requiring a minimum count.

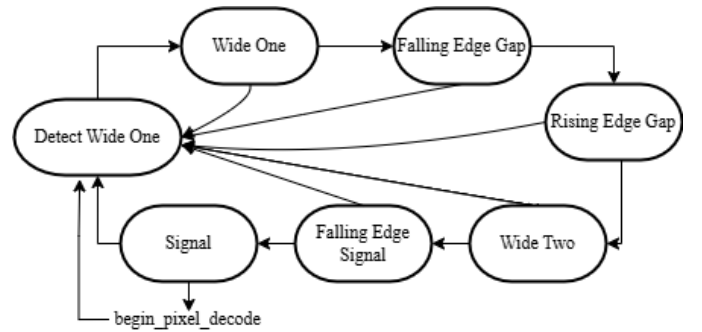


Fig. 7. VIS FSM logic. If a signal fails the requirements for any of the stages, it will restart the detection at the Detect Wide One stage.

The last 300ms of the VIS contains information about the mode of the signal. To reduce complexity, the current implementation only checks to make sure that all of the frequencies are within the acceptable frequency range.

B. Pixel Decoder

Once the VIS Decoder indicates that the VIS was successfully detected, the Pixel Decoder converts the frequencies into pixel values. RGB pixel values are represented with a frequency in the range from 1500 Hz to 2300 Hz and are calculated as

$$\text{pixel} = \left\lfloor 256 \cdot \frac{\text{freq} - 1500}{2300 - 1500} \right\rfloor.$$

For each line of the image, we receive the signals in order of green, blue, horizontal sync, then red. The timing of each region is predetermined and can be found in [5]. An example signal displaying this is shown in Fig. 8.

Initially, we implemented a straightforward timing-based approach. We treat the different regions as different states

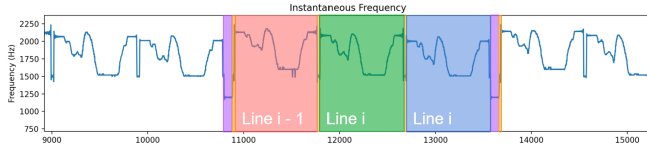


Fig. 8. An example signal. The purple regions represent horizontal sync regions. The orange regions are color sync regions. The red, green, and blue regions correspond to the respective signals for their pixel values. Note that the red signal belongs to the previous line, whereas the green and blue signals belong to current one.

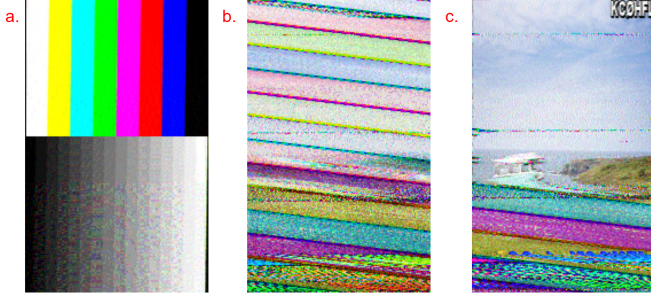


Fig. 9. (a). The decoded image is slightly slanted due to rounding errors in computing the pixel transition zones. Overall, it accumulates an error of 17 pixels. (b). Discrepancies in timing make the decoded image unrecognizable when the timing based approach is used the real recorded signal. (c). Real signals that are noisy can be missing sync regions, and this is a resulting image from a decoding method that doesn't account for this.

(GREEN_SYNC, GREEN_LINE, BLUE_SYNC, BLUE_LINE, HSYNC, RED_SYNC, and RED_LINE), as well as an additional IDLE state. We start in the IDLE state and transition into GREEN_SYNC upon receiving the single-cycle high `begin_pix_decode` from the VIS Decoder. Tracking where we are in the signal with a counter, we can determine when to transition between states with the known timing of the signal and sample rate.

For the most part, this works quite well on clean signals generated from programs. One minor concern is rounding errors accumulating and causing slant, as displayed in Fig. 9a, but this can be resolved with greater precision. However, the timing-based approach is insufficient on noisy real recorded signals, where there may be timing discrepancies and noise. Even minor differences can make the resulting decoded image completely unrecognizable, as shown in Fig. 9b.

A more robust approach uses the 1200 Hz horizontal sync regions to determine the boundaries between states. We implemented a sync detector module that detects the 1200 Hz syncing regions. The sync detector module signals when the syncing region ends, as well as how many samples occurred since between the last syncing region and the current syncing region. The pixel decoder uses the end of the syncing region from the sync detector to determine when to transition from the HSYNC state to the RED_SYNC state. The pixel decoder uses the number of samples (`line_sample_count`) that occurred between the last syncing region and current one to update how much time and how many samples will be spent

transmitting each region, which informs the timing for all other state transitions.

The sync detector uses the timing provided by the SSTV handbook to estimate when it will find the first sync, then uses its own data on `line_sample_count` to estimate where it will find the next syncs. It searches a wide window informed by the expected sync time for a rising edge where $1200 - F_{TOL} \leq \text{freq} \leq 1200 + F_{TOL}$ becomes true. Then, it checks the number of samples that are within F_{TOL} of 1200 for the expected number of samples in the sync, and if there are more samples within the expected range over a threshold of 80%, it updates `line_sample_count`.

One challenge with sync detection is that the real signal can be extremely noisy. This means that that a sync region may be missing, or we might find a rising edge but not find enough samples that are in the correct range, causing a decoded image like Fig. 9c. To address this, we still send a signal when we expect the sync region to be finished (such that the loss of a sync signal, which is quite common, doesn't completely ruin the image), but we don't update our `line_sample_count`. We also take the moving average of the 8 most recent `line_sample_counts` to prevent sudden drastic changes in `line_sample_count`.

VII. FRAME BUFFER AND HDMI DISPLAY

As an image is received, it will be stored in a frame buffer and displayed on an HDMI monitor. The values from the Pixel Decoder are written into BRAM based on their position in the image. Because the pixels are sent in lines based on their color, we split the BRAMs by color, rather than storing their unified RGB565 representation. This is advantageous because in the writing phase, we can just do 3 writes per pixel, as opposed to 3 writes and 2 reads (since we need to update pixels if we store their unified representation). When reading back the colors to display, 3 reads are now required, but these can occur in parallel.

64 out of the 75 available BRAMs are used for the two frame buffers, split by the color channels: 10 red, 12 green, and 10 blue. The red and blue BRAMs will be 5 bits wide and the green BRAM will be 6 bits wide. Each of the BRAMs are deep enough to hold the full image ($256 \times 160 = 40960$ entries).

Additionally, the pixels are being outputted from the Pixel Decoder on a 50 MHz clock, whereas the HDMI reads the pixels on a 74.25 MHz clock. All BRAM reads and writes are on the 74.25 MHz HDMI clock. We use a FIFO for the pixel values to handle the clock domain crossing right before writing to the BRAM. The FIFO uses two 18 KBit BRAMs, bringing the total number of BRAMs used to 65.

We upsample the image by $2 \times$ in each dimension for displaying. We keep track of the last pixel written for each image in order to determine which pixels to display and which pixels should remain black to show the pixels as soon as they are computed. Some basic logic is used to display the different color channels for debugging or display two images side by side when in the multichannel setting.

VIII. SYSTEM EVALUATION

A. Throughput, Latency, and Timing Requirements

At a clock speed of 50MHz and sampling rate of 12.5kHz, the SSTV receiver has 4000 clock cycles between each sample, so throughput and latency were not large constraints on the project. The latency, which is of no practical concern since it will be imperceptible to the viewer, is close to 2000-2500 cycles (40-50 μs) in the worst case, which primarily comes from the divider, which is variable length and takes a number of clock cycles equal to the frequency estimate. The throughput, which is also limited by the divider, is around 20 kSPS, though it will only ever see the 12.5 kSPS set by the input sampling frequency. However, since the system contains multiple stages of signal processing, meeting timing requirements was one of the top priorities. The largest bottlenecks in terms of timing occurred with regards to the 63 summations in the band pass filter and the sequential fixed point multiplications when the number of samples per line was updated.

To overcome timing issues with attempting to sum 63 signed values sequentially in the band pass filter, they were parallelized into 7 different partial sums of 9 values. Furthermore, the summation stage was pipelined to two stages to finish the computation without negative slack.

When we update the number of samples in a line `line_sample_count`, we need to update our moving average then update how often we sample the frequencies for pixel values. The moving average requires summing 8 values. The formula to compute the number of samples per pixel is $\text{samples_per_pixel} = \frac{1}{160} \cdot (\frac{1}{3} \cdot (\text{avg_line_sample_count} - \text{h_sync_samples}) - \text{color_sync_samples})$, and needs 8 bits of precision. The divisions are implemented using fixed point multiplication, with the minimum necessary precision. This did not meet timing, and we fixed it by pipelining and splitting the computation of the moving average and the `samples_per_pixel` into different clock cycles.

We were able to meet timing with a WNS of 0.305.

B. Memory and Resources

The band pass filter, Frequency Shifter, and Frequency Detector all require a non-trivial amount of DSPs. Especially with instantiating multiple of these modules for the multichannel feature, conserving the use of DSPs was crucial. By pipelining and reusing DSPs, we were able to achieve a maximum of 92 DSPs for our system out of the 120 available. The usage is detailed in Table I.

In terms of memory usage, the only modules that required significant resource utilization were the frame buffers to store the pixel values. A total of 64 BRAMs out of the 75 available were used for this purpose. The rest of the modules did not use a significant number of LUTs since they mostly contained smaller shift buffers and registers. The overall resource utilization of the rest of the system included 20.03% of logic LUTs and 0.16% of LUTs used for memory. Of the available flip-flops, 12.08 % were utilized, primarily by various FIR filters,

TABLE I
RESOURCE USAGE

	DSPs	BRAMs	LUTs
Band Pass Filter	21	0	1099
Frequency Shifter	5	0	2343
Frequency Detector1	31	0	256
Frequency Detector2	23	0	259
Pixel Decoder1	2	0	804
Pixel Decoder2	2	0	804
Frame Buffer1	0	32	39
Frame Buffer2	0	32	56
Others*	8	1	1129
TOTAL	92	65	6789

*The Others category includes the VIS Decoder, Clock Wizards, and additional modules whose resource usage were too small to warrant separate rows.

which have large shift buffers to store inputs and intermediate products. These values were calculated by the Vivado build outputs.

C. Checklist

We were able to meet our minimum goals, ideal goals, and one of the stretch goals. We met our minimum goal of decoding an image received through a single-channel audio signal generated by a computer. The physical system can reproduce a program generated signal with little to no artifacts/noise in the image. See Fig. 3.

We also reached our ideal goals of decoding a radio-transmitted SSTV signal as well as receiving and displaying multiple frequency-multiplexed SSTV images at once. For the radio transmitted signal, we recorded a live signal including noise at the beginning and end, in order to use it repeatedly. The decoded image of the radio transmitted signal was compared to the output from an existing SSTV decoding software, as shown in Fig. 10, and the results visually matched.



Fig. 10. The left image is the real signal decoded by MMSSTV, while the right is decoded with our receiver. The right image was the output on a monitor captured by a phone.

The multichannel output is slightly noisier, but this is due to interference from multiplexing two signals that have a relatively small gap between their respective frequency ranges. See Fig. 11 for the result.

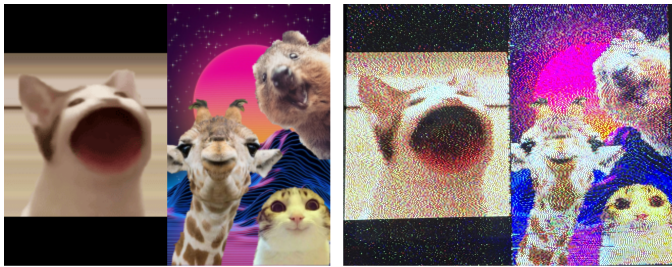


Fig. 11. The left two images are the original, while the two on the right are decoded with our receiver. The right images are the outputs on a screen captured by a phone camera.

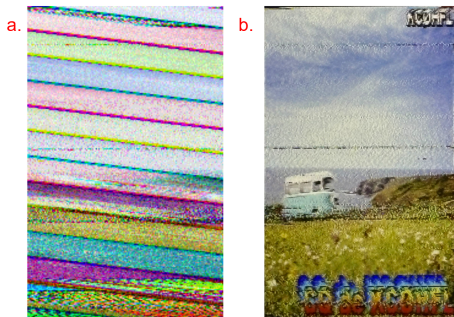


Fig. 12. (a) The real signal prior to skew correction. The image is the output from simulation. (b) The real signal after skew correction. The image is the output on a screen captured by a phone camera.

One of our original stretch goals was to correct skew in the decoded image using the Hough Transform. While the skew correction algorithm was not implemented using the Hough Transform, we corrected skews using our sync detector, as shown in Fig. 12.

D. Implementation Insights

The initial implementation of the band pass filter did not take into account the multiple pipeline stages that would be required to reduce the number of DSPs and meet timing, so the final implementation incorporated a lot of one-bit logics to determine the current state. An improvement for future implementations would be to start with a state machine and use case statements to reduce confusion in the different stages. Additionally, filters with better attenuation in the stop-bands may have helped to reduce interference between images when receiving a multichannel signal.

When designing the frequency detector, we decided to expand the derivative of the arctangent to avoid the necessity of calculating that arctangent. However, in hindsight, this was actually much more difficult for two reasons: the frequency was too high to use a first-difference filter to compute the derivative, and the divider was extremely slow. If we had used a CORDIC module to compute the arctangent, it would have removed the need to divide (since the CORDIC algorithm can easily compute $\tan^{-1}(Y/X)$) and because the result of the arctangent itself has a low frequency (since it is the phase of

a narrowband signal) we could have used just a first-difference filter.

Detection of the VIS proved challenging due to high levels of noise in the frequency estimate, as seen in Fig. 13. Keeping a minimum count of the inputs that fall between an expected range proved to be very unreliable when trying to detect the VIS. Instead, account for noise by looking for rising and falling edges as well as keeping a running average of the current frequency for a more reliable implementation.

When prototyping the pixel decoder in Python, we didn't sufficiently account for the streaming nature of the SystemVerilog implementation, which made the translation harder and required us to eventually change the implementation. Spending more time writing and evaluating more Verilog-like implementations would have been useful. Debugging was also something really important for the decoder. Finding images that were easy to debug and verify was helpful, such as the radio color signal. Finally, we learned that we needed to isolate the pixel decoder from the other parts to test it, since it's at the very end of the pipeline and relies on a lot of other pieces to work.

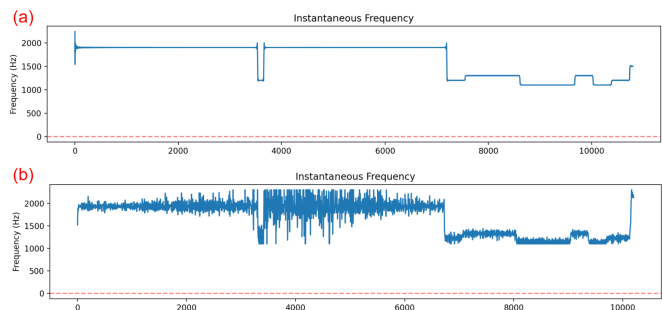


Fig. 13. (a). Clean VIS. (b). Noisy VIS.

E. Code Repository

<https://github.mit.edu/6205F25/fa25-6205-team29.git>

IX. CONTRIBUTIONS

The workload was divided between the team members as follows: Amy worked on the Band Pass Filter and VIS Decoder, Noah worked on the Frequency Detector, Frequency Shifter, and hardware, and Janelle worked on the Pixel Decoder and HDMI pipeline. Each member has contributed to writing up the sections they worked on in the report.

X. ACKNOWLEDGEMENTS

We would like to acknowledge our mentor, Monica Chan, and the rest of the 6.205 staff who helped us plan and debug various aspects of the design and integration process. We would also like to express gratitude to various members of MIT Radio Society for their help in understanding the signal processing aspects of our project.

REFERENCES

- [1] A. Veres-Szentkirályi, "pysstv," Available at <https://github.com/dnet/pySSTV> (2025/12/10).
- [2] P. Isza, "Tfilter," Available at <http://www.t-filter.engineerjs.com> (2025/12/10).
- [3] D. E. T. Romero and G. J. Dolecek, "Digital fir hilbert transformers: Fundamentals and efficient design methods," in *MATLAB - A Fundamental Tool for Scientific Computing and Engineering Applications - Volume 1*, V. N. Katsikis, Ed. London: IntechOpen, 2012, ch. 19. [Online]. Available: <https://doi.org/10.5772/46451>
- [4] Q. Chaudhari, "Design of a discrete-time differentiator." [Online]. Available: <https://wirelesspi.com/design-of-a-discrete-time-differentiator>
- [5] M. Bruchanov, "Image communication on short waves," 2019. [Online]. Available: <https://www.sstv-handbook.com>