

Audio Steganography Using Direct Sequence Spread Spectrum (DSSS)

6.2050 Final Project - Team 31

1st Alice Fang

Department of EECS
Massachusetts Institute of Technology
Cambridge, MA, USA
acfang@mit.edu

2nd Nerissa Wong

Department of EECS
Massachusetts Institute of Technology
Cambridge, MA, USA
nerwong@mit.edu

Abstract—This project investigates the implementation of a Direct Sequence Spread Spectrum (DSSS) algorithm for audio steganography on FPGAs. The system captures a longer cover audio signal and a shorter stego (secret) audio signal, applying DSSS techniques to spread and embed the stego audio as low-amplitude noise within the cover audio, rendering it undecipherable to human listeners. The encoded audio can then be transmitted and decoded to accurately recover the original stego content. The complete setup involves recording both audios and performing the encoding/decoding on an FPGA. This approach demonstrates the feasibility of real-time, hardware-based audio steganography using DSSS techniques.

Index Terms—Audio steganography, Digital systems, Direct Sequence Spread Spectrum, Field programmable gate arrays,

- A SPH0645LM4H I2S MEMS Microphone Breakout Board. This will be used to record audio through the I2S communication protocol.
- A MAX98357A I2S Class D Amplifier. This will be used to playback audio through the I2S communication protocol.
- A TRRS audio jack for interfacing the speakers with the I2S amplifier.
- A Logitech Z130 Speaker for interfacing with the TRRS audio jack and playing back our audio.

I. INTRODUCTION

Audio steganography is the process of hiding information within an audio file (cover audio) without noticeable degradation so that only the intended recipient can extract the concealed information. In this case, the hidden information is another audio file (stego audio). Direct Sequence Spread Spectrum (DSSS) will be the technique used to embed the hidden auditory content. This method involves spreading the stego audio throughout the frequency spectrum of the cover audio by artificially lengthening the stego audio using a pre-determined low auto-correlated pseudorandom number (PN) code [1].

This code sequence is used to modulate the Least Significant Bits (LSBs) of the audio while keeping the Most Significant Bits (MSBs) as the cover audio so that the encoded message is imperceptible. To recover the stego audio, the modified audio can be convolved with the PN code to determine the bits of original data and reconstruct the concealed audio [2].

II. SYSTEM OVERVIEW

A. Hardware Overview

This project is made up of several hardware components, including:

- A Real Digital Urbana FPGA Development Board. This is where all of the audio capture, encoding, decoding, and playback will occur.

B. System State Diagrams

Our system seeks to perform audio steganography by embedding a stego audio inside a cover audio. To achieve this, we need to record, store, encode, decode, and output different audios. As a result we have a record state that can be triggered using btn[2] and automatically returns to idle once the allotted record memory is full. This can be used to record both stego and cover audio. We have an encode state which can be triggered using sw[3] and automatically transitions back to idle once its max address in memory has been reached, and a decode and read decode state which perform the decoding algorithm and playback the output. Additionally, there is an output state which can be used to output any audio (cover, stego, encoded, or decoded). These states transition to and from IDLE when computations/recordings are not being interfaced with.

C. Top-Level Block Diagram and Modules

The core modules for this system are:

- Audio Interface
 - I2S Receiver module to read the output from the 24-bit MEMS microphone
 - I2S Transmitter module to take DRAM output and stream audio information into an I2S Amplifier
 - Recording module to take in user input and set recording duration
- DRAM Memory Access

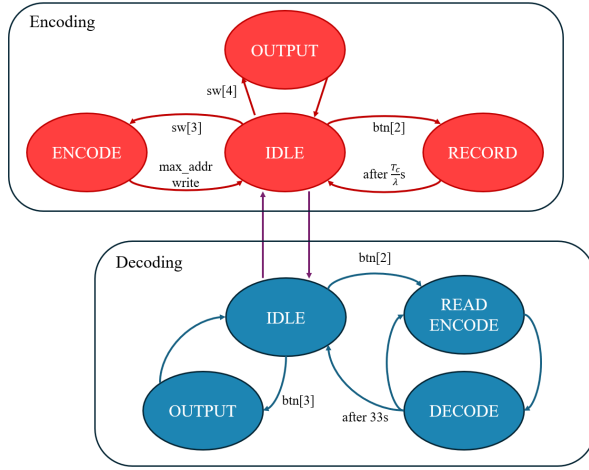


Fig. 1. FPGA State Transition Diagram

- FIFO to queue data to be written and read into the stacker or read out into the unstacker to account for variable DRAM read and write times
- Stacker Module to stack sets of 7 18-bit audio samples into a 128-bit words (Top 2 MSBs are used to denote if the data is stego or combined)
- Unstacker Module to take 128-bit word reads and deconstruct them into 7 18-bit words
- Traffic Generator module to serve as an arbiter to generate read/write commands to the DRAM to read from specific blocks of addresses depending on what state the system is at
- BRAM FIFOs to store the stego and cover audio to be retrieved without limitations from DRAM variable latency in the encoding state
- Encoding and Decoding Scheme
 - Encoding module to spread the stego audio using the PN and overlay it on the cover audio
 - Decoding module to convolve the PN with the modified cover audio to extract the secret audio message
 - Synchronization Module to align the start bit of the encoded message and the decoded message
 - Filtering Module to minimize the impact of noise witemize

III. AUDIO INTERFACE

We are using the I2S protocol in conjunction with a MEMS microphone and I2S Amplifier to capture and playback audio. The microphone outputs 24-bit audio samples, but only the 18 MSBs are relevant data. The I2S Microphone peripheral has the following inputs that need to be provided by the main:

- **BCLK**: The serial CLK of the mic
- **WS**: The word select CLK, indicating whether the right or left channel is being transmitted

To capture the full range of audible human frequencies, we are sampling at 44.1 KHz. With 24-bits per audio sample streaming out of the mic, this requires a BCLK operating at:

$$f_{BCLK} = (44.1 \text{ kHz}) \left(24 \frac{\text{bits}}{\text{samp.}} \right) \left(2 \frac{\text{samp.}}{\text{frame}} \right) = 2.1168 \text{ MHz}$$

The word select clock needs to operate at the sampling rate, and needs to be synchronized to change on the falling edge of the BCLK. In this setup, both WS and BCLK are derived from the Vivado Clock Wizard IP 105.84 MHz generated master clock, which the whole system is clocked off of.

IV. MEMORY ACCESS

The memory access interface is similar to that in Lab 6, but the traffic generator, stacker, and unstacker have been modified. Additional BRAM FIFOs generated by Vivado IP has also been utilized to safely interface the memory access pipeline with the I2S modules, and allow for the patterned access of reads in memory in the ENCODING state.

The stacker module has been modified from Lab 6 to take in 7 18-bit samples to ensure that no bits of the I2S data are thrown away. This totals to 126-bits of relevant data, and the top bit of the stacked data is used to indicate whether it is currently stego or cover audio, that is being recorded. Similarly, the unstacker module also has the same modifications, reading out 7 18-bit samples for each 128-bit chunk and making note of the MSB to determine if the current sample is stego audio or cover audio.

BRAM FIFOs have also been added to ensure safe interfacing with the other modules in the system. Specifically, to pad either side of the memory access pipeline, 128 by 128 AXI BRAM FIFOs have been added for safe data readout. Moreover, 2 4096 by 128 AXI BRAM FIFO follow the 128 by 128 BRAM FIFO after the unstacker which stores the cover and stego data based on the unstacker's MSB. This implementation ensures that variable DRAM latency is not a limitation and that the next stego and cover data are always nearly instantaneously available for the patterned access of memory needs needed in the encoding state.

To implement encoding and decoding on the FPGA, the system needs to be able to read and write from four different sections in memory. To accommodate all of these sections, twelve total states were used: RST, IDLE, WRITE_AUDIO_COVER, READ_AUDIO_COVER, WRITE_AUDIO_STEGO, READ_AUDIO_STEGO, ENCODING, WRITE_COMBINED, READ_COMBINED, DECODING, WRITE_DECODE, and READ_DECODE. To enter RST, btn[0] and btn[1] can be used (btn[1] is tied to DRAM reset). WRITE_AUDIO_COVER is the state used to record and write to memory the cover audio while READ_AUDIO_COVER is the state used to read out/playback the cover audio. WRITE_AUDIO_STEGO and READ_AUDIO_STEGO are the corresponding states for the stego audio. To read out/playback either audio, sw[2] must be toggled high. To write/record either audio, btn[2] must be pressed, while sw[1] is used to denote whether the cover (low) or stego (high) is being accessed. To encode two audios, sw[3] can be used to enter the ENCODING state

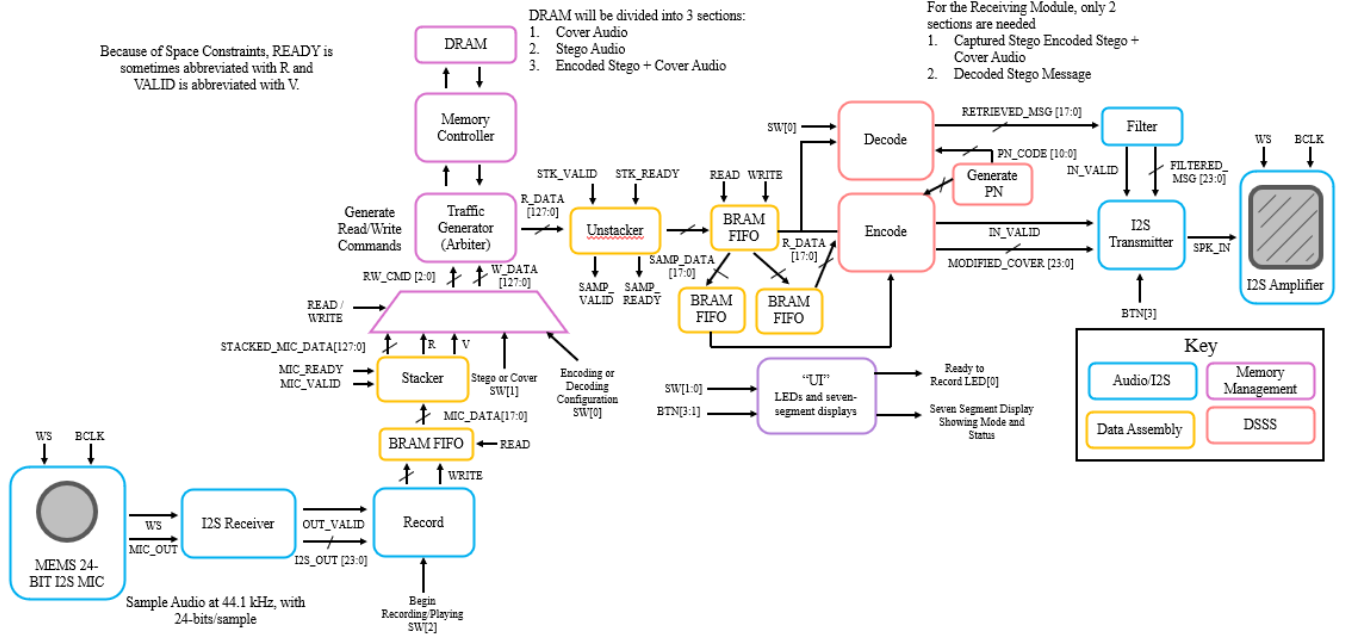


Fig. 2. Top Level Block Diagram

which accesses both the cover and stego sections of memory. ENCODING toggles with the WRITE_COMBINED state which allows for writing back the output of the encoding module, while the READ_COMBINED (sw[4]) state allows for playback of the data in the COMBINED section of memory. DECODING and WRITE_DECODE are entered using sw[5] and similarly toggle to allow for reading and writing for the decoding module, while READ_DECODE (sw[6]) is used to read/playback the output of the decoding module.

All of the states transition back into the IDLE state once they have finished recording, encoding, decoding, or playback. This transition is given by reaching the MAX_xxxx_ADDR from the START_xxxx_ADDR where the address is given by the duration of the audio being recorded or produced. However, when in the playback state for any of the 4 audios, the audio will continue looping until a switch is flipped to leave that specific state. The number of addresses required for each of the sections of memory is given by:

$$\text{NUM_ADDRS} = (44.1 \text{ kHz}) \left(\frac{\text{number of seconds}}{7} \right)$$

To maintain the ratio of 11 cover samples to 1 stego sample feeding into the encoding scheme, we will need to use:

- 207900 addresses (33 seconds) for cover
- 18900 addresses (3 seconds) for stego
- 207900 addresses (33 seconds) for encoded(combined)
- 18900 addresses (3 seconds) for decoded

V. ENCODING AND DECODING SCHEME

DSSS is a digital modulation technique that mixes a signal with a fast pseudorandom code. The output is a signal that appears as noise but with high autocorrelation peaks, allowing a receiver to decode the original signal. While there are a variety of PN codes, the 11-bit Barker PN was selected because it has good autocorrelation, small sidelobes, and it is relatively short. This allows us to encode and decode while maintaining a reasonable stego to cover audio length ratio.

A. Encoding Algorithm

To encode, each sample of the stego audio is multiplied with each bit of the PN. The since the PN (+1, +1, +1, -1, -1, -1, +1, -1, -1, +1, -1) is 11 'bits' long, each stego sample is replicated 11 times. The output of the encode module is a scaled result of the multiplication, added to the cover audio. This means that the cover audio is 11 times the length of the stego audio.

$$\lambda_{\text{ratio}} = \left(11 \frac{\text{bit PN}}{\text{stego}} \right) \left(1 \frac{\text{cover}}{\text{PN}} \right) = 11 \frac{\text{cover samples}}{\text{stego sample}}$$

For our system, we chose a stego length of 3 seconds, meaning our cover audio was 33 seconds long. We opted to scale down our stego multiplication output by a factor of 4, for simple implementation and to limit the presence of the pseudo-noise in the cover audio.

In order to implement this on the FPGA, we used a patterned access of reads and writes from the different sections of memory. The input pattern to our encoding module was a single stego sample, followed by 11 cover samples. Since our computation took a single clock cycle, the clock cycle after a

cover sample was received, a corresponding combined sample was written back to memory.

B. Decoding Algorithm

To decode using DSSS, we break our combined audio into 11 sample blocks. Each sample within the block is multiplied by the corresponding PN bit in the PN code. The 11 samples are then added back together to form the decoded output. Due to the strong autocorrelation of the barker PN, this allows for reconstruction of the original stego audio. The summation without thresholding introduces artifacts from the cover audio which appear in the high frequency range. To reduce the appearance of these in our output, we applied a low pass filter as described in the filter section.

To implement the decoding algorithm a pattern of reads from the COMBINED section in memory and writes into the DECODED section of memory was needed to ensure that decoded outputs were not lost. By toggling sw[7] we were able to either write the decoded output to memory, or pass the decoded output into a low-pass filter.

C. Synchronization

Since each sample of cover audio is repeated across different samples, it is important to ensure that the decoder splits the samples correctly, otherwise the bits may be correctly transmitted but incorrectly decoded. To synchronize the modules, a synchronization preamble was created. This preamble takes each bit of the 11-bit Barker PN and maps the ± 1 value to the minimum or maximum possible value (-2^{17} or $2^{17} - 1$) of the speaker. Before decoding, the sync_preamble module computes a sliding window correlation between the 11 most recent samples received and the barker PN. If the samples and PN are aligned, the correlation results in a large positive number, otherwise the correlation results in a large negative number. The sync_preamble module counts the number of times the correlation is above the threshold before transitioning into the decoding module. While the relevant modules were created and testbenched, due to timing constraints the system was demoed on a single FPGA, so the synchronization modules were not used in the final implementation.

D. Filtering

To implement our low-pass filter we used the Matlab Filter Generator tool as well as its generate HDL feature to create and apply a low-pass filter with a cutoff of 4khz, since human speech frequencies are much lower. The filter implemented was a 10-tap hanning low-pass filter which was connected to the output of the decode module and could be read into memory using sw[7].

Since the filter also attenuates human speech frequencies to some extent and the stego audio was already scaled down by a factor of four, the output of the filter was scaled up by a factor of eight to compensate.

VI. EVALUATION

A. Audio

In our design, we were able to demonstrate successful record and playback of audio by interfacing with I2S through the MEMS Mic and Speaker. To maximize audio quality, we utilized a sampling rate of 44.1 KHz and kept all 18-bits of audio data as we wrote data to DRAM live piped our data into the speaker. The I2S receiver was built and tested first, integrated with the UART Transmit module to write data to the laptop in a format similar to Lab 3, modified on both the UART Transmission side and the Python Script side which takes the data out of the UART Transmitter and then assembles them together into 18-bit data words by reading the reconstruction bits.

a) *UART Transmitter*: Each transmission will have 6-bits of the audio sample in the LSBs and while the top 2 bits encode which window of the audio sample is being encoded - top, middle, or bottom.

TABLE I
UART AUDIO SAMPLE ENCODING

Reconstruction Bits	Audio Data
2'b00	mic_data_out[11:6]
2'b01	mic_data_out[17:12]
2'b10	mic_data_out[23:18]

The UART needs to run at a baud rate high enough to read data from the I2S Microphone.

$$f_{\text{baud}} > (44.1 \text{ kHz}) \left(10 \frac{\text{bits}}{\text{send}} \right) \left(3 \frac{\text{send}}{\text{sample}} \right) = 1.323 \text{ MBaud}$$

To ensure that no data is lost, we utilize 1.512 MBaud, which gives 70 clock cycles per transmission with a 105.84 MHz master clock.

Originally, our system used an 8 ohm speaker, which had several hardware limitations since the encoded and decoded audios all contain high-frequency artifacts which are still audible due to the limitations of the DSSS algorithm described in the next section.

B. DSSS Algorithm

Using the DSSS algorithm for audio encoding presented a number of challenges. DSSS is originally a digital encoding algorithm, so full implementation of DSSS for audio encoding would require each bit of each stego sample to be encoding using the PN. This would allow for more noise/cover resilience as the summation in the decode stage would be compared against a threshold and outputted as a binary 0 or 1 instead of an "analog" value from -2^{17} to $2^{17} - 1$. However, since our stego audio was recorded at the sample sampling rate with the same resolution, this type of encoding would require a cover audio that was thousands of times longer than the stego audio to be noise resilient, which was unfeasible as a demonstration, so we opted to use a modified version of DSSS where each sample was treated as a bit and no threshold was used to determine the decode result.

While the stego message was audible upon output of the decode and filter modules, there is a lot of noise present in the result. Which is due to both the modified use of the DSSS algorithm, but also the choice of low-pass filter. The Hann window was chosen because it is computationally resource light, however, the filter has a gradual roll off as opposed to a sharp transition, allowing more noise/artifacts to be present in the resulting. If this project were to be redone, a different, sharper filter would have been used to compensate for the deficiencies of the algorithm/implementation.

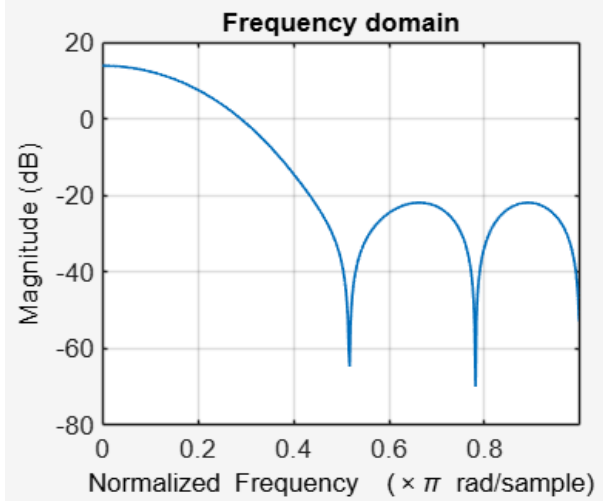


Fig. 3. Filter Magnitude Response Plot

C. Memory

Our project's DRAM memory usage is primarily based on the duration of audio that needs to be recorded, which is fixed based on the length of the stego message that we want to encode. However, in the end, our team utilized 14 BRAMs, some of which were redundant, such as the ones prior to and following the stacker and unstacker respectively. These were put in place to ensure safe access from the top level data to the memory pipeline, though these could be removed with more careful and stringent utilization of the stacker and unstacker's ready signals.

D. Timing

The system has a WNS = 0.571 ns, which meets timing requirements. Because our overall system is clocked using a 105.84 MHz clock, and we are sampling audio at 44.1 kHz, we have 2400 cycles for any set of consecutive accesses to DRAM, so DRAM timing constraints and latency do not pose a huge limitation.

E. Checklist

We were able to complete our commitment of implementing audio recording and playback for both the cover and stego audios. We were also able to demonstrate an FPGA implementation of DSSS by encoding and decoding on a single FPGA. We were also able to implement a state machine using

our traffic generator to automate recording and playback by using address counters and buffer full flags to automatically leave recording/playback states, thus completing our goals as well.

VII. RETROSPECTIVE

The time it took to debug each step of the DRAM was difficult and significantly delayed the integration of the full system, since our inability to access the DRAM consistently led to nondeterministic outputs and accessing DRAM was a critical step in our system. Additionally, our inability to testbench DRAM also contributed to delays in debugging that led to a late integration of our system. Initially we underestimated the number of sections in memory needed as well as the difficulty in interfacing with DRAM, however, we learned about the benefits of using BRAM/FIFOs to minimize the impacts of variable latency reads/writes on the performance of our system. Additionally, had we realized the limitations of the DSSS algorithm earlier we would have used a different encoding algorithm (phase encoding/echo encoding) or started with encoding binary/ascii instead of audio to allow for more resilient encoding/decoding as well as minimize the amount of interfacing with DRAM that was needed.

VIII. CONTRIBUTIONS

We split the work in our team in such a way that allowed each member to separately design, build, and debug their modules before integrating them together. Nerissa worked on the I2S transceivers to interface with the MEMS microphone and the amplifier (speaker) as well as the memory management DRAM pipeline, recording cover and stego audio data, and playing them back accordingly as it transitions through the state machine. Nerissa also worked on packaging the 18-bits of data across three UART transmission samples to verify the practice that the I2S data receiver protocol was working. Alice worked on developing a Python implementation of the DSSS encoding and decoding algorithms, and then translating this implementation into Verilog, developing the DSSS algorithm pipeline, which also included an FIR Filter. Both of us worked on integrating the DRAM access patterns with the DSSS encoding and decoding algorithms, since Nerissa had more familiarity debugging traffic-generator and DRAM-based issues, while Alice knew how the encode and decode modules should behave based on different inputs.

IX. ACKNOWLEDGMENTS

We would like to thank Joe Steinmeyer and Yoland Gao for their endless mentorship, guidance, and patience debugging throughout this final project period. Furthermore, we would also like to thank Yeabsira Hawaz for providing us with an RF24 audio jack last minute. Lastly we'd like to thank the rest of the 6.205 teach staff for being so helpful throughout the term!

X. CODE REPOSITORY

<https://github.mit.edu/6205F25/fa25-6205-team31>

REFERENCES

- [1] D. Hong, G. Huang, and N. Kumar, "Message encryption in audio steganography," May 15, 2024. [Online]. Available: <https://65610.csail.mit.edu/2024/reports/hong-kumar-huang.pdf>
- [2] A. F. Mahmood, N. A. Kanai, and S. S. Mohmmad, "An FPGA implementation of secured steganography communication system," *Tikrit Journal of Engineering Sciences*, vol. 19, no. 4, pp. 14–23, Dec. 2012, doi: 10.25130/tjes.19.4.02.