FPGA-Based Real-Time ADS-B Scanner

Youran Gao, Hasan Zeki Yıldız, Yichen Gao

***Abstract*—Automatic Dependent Surveillance-Broadcast (ADS-B) is an aircraft surveillance system that is installed on most aircraft. The ADS-B system is used by air traffic control, in-air collision avoidance systems, and other similar applications. The messages are transmitted over UHF radio at 1090 MHz using a standardized scheme published by the International Civil Aviation Organization (ICAO). In our system, the signals are captured using a Software Defined Radio (SDR), digitized using a Raspberry Pi microcomputer, and processed on a AMD Xilinx Spartan 7 FPGA platform in real-time. The signals are then visualized over an HDMI interface to display aircraft locations and information on a monitor. The system offers low-latency and low-power visualization and processing of ADS-B data packets.**

***Index Terms*—Automatic Dependent Surveillance-Broadcast, Field Programmable Gate Array**

I. Introduction

Automatic Dependent Surveillance-Broadcast (ADS-B) signals, emitted periodically by aircraft transponders, provide critical navigational information such as position, speed, and heading. This project aims to achieve real-time decoding of these signals using FPGA. Further, utilizing an HDMI interface, the decoded information is visually presented on a screen, enabling accurate tracking of aircraft locations.

ADS-B is used in critical aviation safety applications such as air traffic control, the Traffic Collision Avoidance System (TCAS), and fleet management. Real-time processing is not just a technical requirement but a crucial necessity, as any interruptions during key events, particularly during traffic avoidance alerts, could have serious implications. Our project aims to provide a proof of concept for a low-latency decoding system using FPGA.

II. The ADS-B Message

A. Radio Transmissions

ADS-B signals are transmitted at 1090MHz by aircraft up to twice a second, and present a unique challenge in processing. To process the data, we determine the noise floor to establish a practical threshold for distinguishing between "high" and "low" signals. Given that each bit is transmitted every microsecond and the entire ADS-B message takes only 120 microseconds, the theoretical bandwidth of 1 Mbit allows for the potential transmission of approximately 8300 messages per second. To ensure that all messages are processed in real time, we must commit to a throughput of no less than 8300 messages per second. The structure of the downlink message is shown in Figure 1.

Manuscript received December 13, 2023; Youran Gao, Hasan Zeki Yıldız, and Yichen Gao are with the Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology (email: youran, hzyildiz, ygao7@mit.edu).

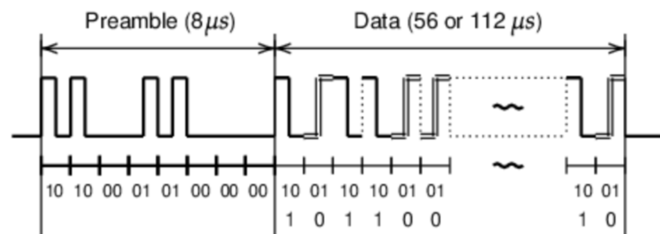


Fig. 1. An example of ADS-B message [1]

B. ADS-B Data Format

An ADS-B packet is 128 bits long and consists of six components:

- 16 bit preamble
- 5 bit downlink format
- 56 bit message (including a 5 bit type code)
- 24 bit ICAO code
- 3 bit CA (transponder capability)
- 24 bit CRC

We will highlight the components that are most important to the decoding process, which are the message, ICAO code, and the CRC.

C. Type Code

The ADS-B system utilizes a 5-bit type code for message identification, of which there are 8 unique message types, and of which we specifically concern ourselves with 5. Despite the existence of additional message types in the standard, not all are actively used by aircraft. Each message is comprised of 51 bits, and the message types of interest encompass crucial information such as aircraft identification, surface position, airborne position, airborne velocities, and aircraft status.

D. ICAO Code

ICAO assigns a unique 24-bit code to every aircraft for their transponder, a designation that remains fixed throughout the entire lifespan of the aircraft. These codes are systematically organized by the country of registration. Of particular note is that every ICAO code on a U.S. registered aircraft starts with 0xA, a fact we will utilize in our design.

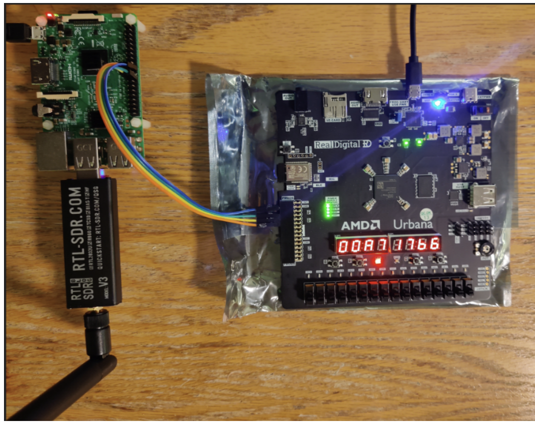
E. CRC Error Correction

The cyclic redundancy check (CRC) error correction bits in ADS-B provides a robust means of error detection, allowing us to identify and disregard messages containing errors. Furthermore, the system is designed to allow for the correction

of 1-bit errors. However, since the radio frequency used for ADS-B transmission is in the Ultra High Frequency range, the error rate is quite high. In addition, there is no synchronization between planes. Despite this, since most messages are re-broadcasted within one second, and there are only a limited number of aircraft present in any given area, this does not present a issue in practice.

III. Hardware Interface

The hardware interface of the device is as follows. The analog signals that are broadcasted from aircraft are received by our SDR, which communicates with the Raspberry Pi over USB. Then, the Raspberry Pi translates the analog signal into a stream of digital signals to be sent over the SPI interface to the FPGA. The FPGA then processes this signal, calculates the aircraft information, and then outputs the result over HDMI to a monitor.



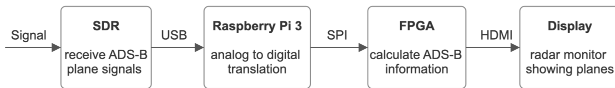


Fig. 2. Physical Hardware Interface of the System

A. SDR to SPI

A simple program was written in C to run on the Raspberry Pi to interface between the SDR and the FPGA. The module tunes the SDR to 1090MHz and sets a sampling rate of 2 Mbits/s. As the data is received, a noise floor is calculated, and the analog signal is converted into a digital signal through the scheme detailed in [1]. This digital signal is then sent to the FPGA over SPI.

IV. Message Decoding and Processing

A. Message Extraction and Error Correction

In this project, the `SPI_rx` module, previously employed in other labs, has been repurposed for the ADS-B system. This module operates with a SPI clock speed of 1MHz and incorporates a combinational CRC error-checking mechanism. Upon successful decoding and validation, the module communicates the ICAO code, ADS-B message, and type code to the message multiplexer.

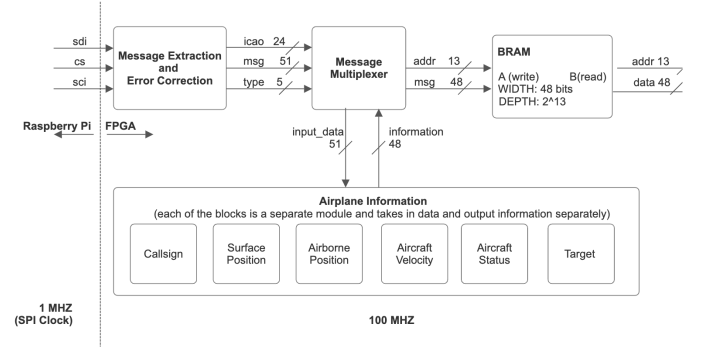


Fig. 3. Block diagram of the ADS-B Decoder

B. Message Multiplexer

After the error correction and message extraction, the message multiplexer coordinates the processing of the ADS-B message. Based on the 5-bit type code, the multiplexer sends the message to the appropriate module to process the information received. This could be the callsign, airborne position, airborne velocity, etc. These modules are implemented according to the ADS-B specification. Once it receives the processed information back from the airplane information decoding modules, it is stored to the BRAM.

To calculate the BRAM address the information is stored to, we take the ICAO address of the aircraft and hash it from 24 bits to 10 bits. It is unlikely for there to be any hash collisions as we expect to pick up signals from 30-50 aircraft at any given time. The last three bits of the BRAM address is based on the type code, as there are 8 types of information that is transmitted over ADS-B. In total, the BRAM address is 13 bits (10 bits from hashed ICAO address, and a 3 bit type code).

C. ADS-B Message Decoding Modules

There are different modules that correspond to the several types of ADS-B messages that can be sent by aircraft. Each of these modules has a different implementation based on the ADS-B specification.

Some modules are relatively straightforward, while others require significant processing. Of particular note is the module that processes aircraft location. Due to the encoding used by ADS-B, there is be non-trivial amounts of math calculations that are required, which necessitates the use of floating-point and CORDIC IP modules.

In total, we have 5 submodules, each of them takes in 52 bits long messages and output up to three 48 bit outputs of decoded information.

- `callsign`: We exclude the 3 bit aircraft category and encode the 8 character callsign using 6 bits. Then, it is

converted into our own encoding for use by the graphics pipeline.

- `airborne_position`: The airborne position of the aircraft can be calculated through a method known as "locally unambiguous position decoding". This method involves knowing a reference position within 180 nautical miles of the airplane. As we know that our receiver is not strong enough to receive signals from more than even 100 nautical miles away from the receiver, it is appropriate to use MIT's coordinates as the reference location. This approach is less complex than the method known as "globally unambiguous position decoding", which does not require a reference location but does requires two different messages (which are received at different times). This necessitates the use of more BRAM storage on the FPGA and complicates the processing required.

  Additionally, we calculate the altitude in this module. The decoding process for altitude is more straightforward, and aircraft sends altitude information in either barometric altitude (based on atmospheric pressure) or GNSS (based on GPS satellites). We decode accordingly and convert the result to feet, following the convention used in this project.
- `surface_position`: Surface position is similar to airborne position but does not contain altitude; instead, it includes the speed of the aircraft on the ground. We use exactly an identical procedure to decode the position in this module as the airborne position module.
- `aircraft_velocity`: Aircraft velocity is transmitted as two separate components. There is a north-south component and an east-west component. This allows us to calculate both the velocity (by calculating the magnitude of this vector), as well as the heading (by calculating using inverse tangent).
- `status`: The aircraft operational status message is designed to provide various ancillary information about an aircraft. Different versions of the ADS-B standard have different format for the status message. In our implementation, we use version 2 of the specification, which is the most widely adopted version. This ancillary information contains properties about the aircraft itself, such as the navigational accuracy, barometer accuracy, operational mode, among others.

D. Aircraft Information BRAM

This BRAM of width 48 bits and depth 2^{13} bits is the primary storage location for all processed aircraft information. The write port is clocked at 100 MHz, and data is written by the message multiplexer. The read port is clocked at the 72.5 MHz (HDMI clock speed), and is read by the graphics pipeline.

Given the constraint of limited BRAM capacity, our project employs a hashing strategy for ICAO codes, since the number of unique planes 2^{24} exceeds available resources. The chosen hash function involves truncating the initial four bits (typically '1010' for U.S. planes) and XORing the top 10 bits with the bottom 10 bits. This approach significantly reduces the required storage capacity, while reducing the chances of hash collisions. To accommodate the different message types associated with each plane, a 3-bit message type code is implemented, allowing for the storage of up to 8 distinct message types per aircraft.

Therefore, our BRAM address has length 13 (10 bit ICAO hash + 3 bit message type code). In Figure 4, we have detailed the bitfields of each of the messages we store. The timestamp is based on a 10ms counter starting from the time the FPGA has been powered on. The callsign is stored using an ASCII-like encoding with 6 bits per alphanumeric character. The heading, horizontal rate, vertical rate, altitude, display X, and display Y are stored using fixed point integers. The latitude and longitude are stored using single-precision floating point numbers.

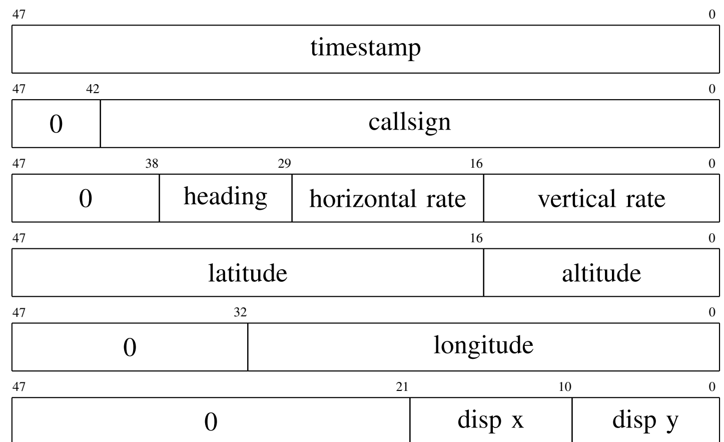


Fig. 4. BRAM format for each type of aircraft information. Each row represents a different message type code, starting with 0 (top) and ending with 5 (bottom)

V. Graphics Pipeline

The graphic pipeline is tasked with visualizing the ADS-B data and producing a video signal over HDMI. The visualization consists of sprites representing each plane over a color map of the state of Massachusetts and its immediate surroundings. The plane sprites show the location of each plane, the 7 character alphanumeric call sign, the heading in degrees and as an arrow, and the airspeed in knots. Figure 5 below shows the general data flow in the graphics pipeline.

A. Plane Lookup

During the active draw period of the frame (including horizontal blanking intervals) this module iterates through the entire Airplane Information BRAM. For each plane entry, it checks the timestamp to make sure the messages enclosed are newer than 60 seconds. Once a plane which has been updated recently is found, it will then read the rest of the entry related to that plane from the BRAM. If any required data field is incomplete, the module advances to the next airplane.

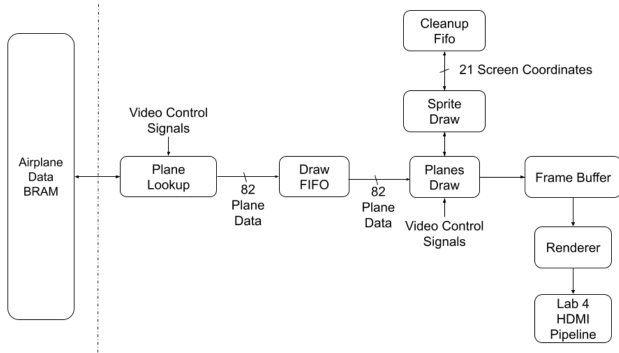


Fig. 5. Block diagram of the Graphics Pipeline

Once an entire plane's worth of data is gathered, the Plane Lookup module puts it into the Draw FIFO to be drawn in the next vertical blanking interval. This continues until the module has gone through all 1024 potential plane entries, after which it halts until the next active draw interval.

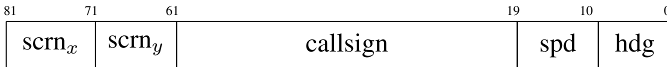


Fig. 6. Bitfield diagram of the data in the Draw FIFO

B. Draw FIFO

The Draw FIFO is a BRAM FIFO generated using the Xilinx FIFO Generator 13.2 IP module. It is 82 bits wide and 64 entries deep. This depth was chosen to limit BRAM usage while not limiting the number of airplanes that could be drawn. It exposes to the modules using it an empty signal and a valid signal.

We must clear all planes on the screen and draw the fresh planes during the vertical blanking time, as this is the time when the signals in the visible portion of the screen are not being drawn.

Since the vertical blanking time is much shorter than the active draw time, we must not waste any cycles during the vertical blanking time. The use of the Draw FIFO allows us to complete the plane fetching process during the lengthy active draw time.

C. Planes Draw

The Planes Draw module is the main module coordinating the cleaning and drawing of the plane sprites to the Frame Buffer. It activates during the vertical blanking interval and first cleans the Frame Buffer. For cleaning, it holds a Cleanup FIFO that holds the screen coordinates of all planes drawn in the previous frame.

After cleaning, it draws the planes located in the Draw FIFO, putting the coordinates of each plane drawn into the Cleanup FIFO to be cleaned during next frame.

The states of the module are detailed in the state transition diagram in Figure 7. When `sprite_draw_ready` is asserted, the Sprite Draw module is signaling that it can accept a new sprite.

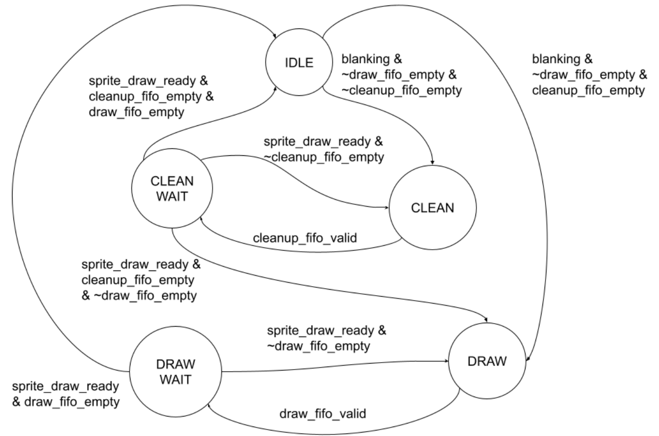


Fig. 7. State Transition Diagram for the Plane Draw Module

D. Cleanup FIFO

The Cleanup FIFO is another BRAM FIFO generated using the Xilinx FIFO Generator 13.2 IP module. It is 21 bits wide and 64 entries deep.

E. Sprite Draw

Sprite Draw module is tasked with turning the plane data coming from the Planes Draw module into actual sprites in the Frame Buffer. It uses sprite based text, numbers, and arrows to draw each sprite into the Frame Buffer. The writing is done 8 pixels at a time, both for simplicity and for speed. It uses a combinational implementation of the Double-Dabbble algorithm (adapted from RealDigital website [2]) to turn the binary heading and speed data into decimal text. It also uses a simple combinational LUT to pick the correct arrow for the heading.

Besides a draw mode, it also has a clean mode used during the clean up phase. In clean mode, it still goes over the same frame locations, except instead of the pixel data from the Sprite Sheet, it simply writes all zeroes.

The Frame Buffer write addresses are protected against wrap-around on the edges of the screen. Each Sprite takes approximately 130 cycles to draw.

Fig. 8. An example sprite for a plane with the call sign "DIGI741" that is moving at 111 degrees at a speed of 205 knots

F. Sprite Sheet

The sprites used by the Sprite Draw module reside in a BRAM module. There are 48 sprites in total: 26 letters, 10 digits, 8 arrows, 1 small square (unused), period, comma, and a space character.

G. Frame Buffer

The Frame Buffer is a large chunk of BRAM containing a 1-bit 1280-by-720 pixel image overlay, containing all of the plane sprites. The data is laid out in 115,200 horizontal lines of 8-pixels each. This buffered drawing method allows us to separate the drawing and fetching of plane data, greatly simplifying the drawing process. It is also by far the biggest chunk of BRAM this project uses, accounting for over half of our usage.

H. Background ROM

This is another large chunk of BRAM, containing a 1-bit 640-by-360 pixel map of the Greater Boston Area. Since the background resolution is less important compared to the sprite text resolution, it is only kept at a quarter resolution to save memory. It is used by the Renderer.

I. Renderer

The Renderer reads the pixels from both the background ROM and the Frame Buffer, and renders them over HDMI. For each pixel it first checks if it is a filled in Frame Buffer pixel. If so this pixel is colored in with the text color.

If not, it colors in the pixel according to the Background ROM. Since both the background ROM and the frame buffer use 1 bit color, this module is where the 24-bit RGB color is added.

J. HDMI Output Pipeline

This last module is replicated from Lab 4, and is tasked with producing the HDMI signals. It produces the frame control signals, such as `hcount`, `vcount`, and `blanking` that is used. The color values created by the renderer is turned into the TMDS signals needed for HDMI in this module.

VI. Analysis of Resource Utilization

The Spartan-7 XC7S50 FPGA we utilized contains 75 dual-port BRAM tiles at 36 kilobits each. It further has 120 DSPs for digital signal processing.

In terms of BRAM tiles, the usage is broken down as follows. For ADS-B decoding, the aircraft information BRAM has a size of $48 \times 2^{13} = 393.2$kb, which requires 11 BRAM tiles. For the graphics pipeline, the frame buffer has a size of $1280 \times 720 = 921.6$kb, which requires 26 BRAM tiles. The background ROM consists of $640 \times 360 = 230.4$kb, requiring 7 BRAM tiles. The sprite sheet, which has a size of 3 kilobits requires 0.5 BRAM tiles. Finally, we use 10 BRAM tiles for FIFO logics. In total, the 54.5 BRAM tiles out of 75 available (72.6%).

In terms of DSPs, we use 19 out of 120 available DSPs (15.8%). They are used for floating-point multipliers, adders, subtractors, and CORDICs that are required to decode the ADS-B information.

It is difficult to optimize the amount of BRAM that is used as the most of the BRAM is used by the frame buffer and background ROM, which are required for the HDMI output. Depending on what specific fields are required by the end user of the ADS-B information, it is possible to reduce number of fields stored, and thereby reduce the amount of BRAM used for storing aircraft information.

VII. Timing Requirements

ADS-B signals have a maximum data rate of 1 Mbit/s, with each message lasting 120 microseconds. In adherence to the real-time processing requirements, our system must be designed to process each message in less than 120 microseconds to ensure reliable processing.

A theoretical critical latency can be calculated by counting the number of cycles required for a worse-case message to get from the SPI input to the Aircraft Information BRAM. As long as the data can make it to the Aircraft Information BRAM in less than 120 microseconds (12,000 cycles at 100 MHz), we can be sure that no information will ever be lost.

The critical latency occurs when an "airborne location" message is received due to the use of many multipliers and dividers along this path. From the SystemVerilog logic and using the Floating-Point Operator v5.0 Datasheet, we determined that the maximum theoretical latency for this message is 230 cycles. This is less than the requirement of 12000 cycles, and represents a theoretical maximum throughput of 55 Mbit/s.

To verify that the FPGA can reliably process data at the maximum ADS-B data rate of 1 Mbit/s, we designed a stress test program to simulate this scenario. The test program written in C contains a pre-created dump of valid ADS-B messages captured over a two hour period, and interfaces with the FPGA over SPI. The dump contains 7,813 ADS-B messages at 128 bits each, which equates 1 megabit of data. During the test, the data was sent over SPI at a target rate of 1 megabit per second. All data was successfully processed on the FPGA at this data rate.

Accounting for delays from the software and the latency of the SPI chip on the Raspberry Pi, the Pi was able to send 1 megabit of data in 1.02 seconds, representing an actual throughput of 0.98 Mbit/s. Therefore, we are confident that we have met the latency requirements.

The worst slack time for the design was 1.835 nanoseconds. This is acceptable within the context of the fastest clock domain (100MHz) of the design.

VIII. Retrospective

The design can accommodate most use cases of ADS-B. Instead of a graphics pipeline, one can use another module more suited to their application's specific needs. For example, one could design a module for collision avoidance for use in aircraft avionics, or a module that takes the ADS-B data and uploads it to flight tracking websites. Since we do not perform any special treatment of the data in the Plane Information

Fig. 9. Example of the system tracking various planes around Boston

BRAM, one can connect the read port of the Plane Information BRAM to any application of their choosing.

In terms of the project checklist, we have reached all of the goals in our goal, as we have successfully implemented all five ADS-B processing modules. We have also successfully rendered a real-time updating map of planes with their callsign, speed, and headings. We also completed one of the stretch goals to display the heading of each plane using a rotated arrow sprite.

For the arrow sprites visually representing the heading, it was suggested to us that we implement it functionally using a module. However, since each sprite is only 8-by-8 pixels big, we found through preliminary testing using image editing software, that any angle that is not a multiple of $45°$ looked unintelligible in a 1-bit color context. This only allowed us to have 8 arrow directions, which could easily fit into our Sprite Sheet. We ended up going with this option.

The biggest bottleneck in the graphics pipeline was the available BRAM. The amount of BRAM limited both the peak resolution (720p), and the time we had available for drawing each frame. With more RAM, a dual frame buffer system could have been implemented. This would increase the time available for the drawing from the current $666\mu s$ blanking time to the whole 16.5 ms frame time. In a future implementation we could move the Frame Buffer from the BRAM to the onboard DDR3 memory to solve this issue.

IX. Contribution Statement

Youran worked on the the C code required to interface between the SDR and the FPGA, the message multiplexer, the SPI receiving and error checking, and test-benches for the decoding modules. Yichen worked on the logic for decoding ADS-B information in each of the five submodules. Hasan worked on the graphics pipeline, including the sprite and plane drawing logic, as well as the plane lookup for interfacing with the BRAM.

X. Source Code

Both the SystemVerilog code as well as the C code used for driving the SDR are available at:
`https://github.mit.edu/youran/fpga-adsb`

XI. Acknowledgements

We express our gratitude for the advice and guidance provided by the 6.2050 teaching assistants, Joseph Feld, and the instructor, Joe Steinmeyer throughout the process of implementing this project.

References

[1] J. Sun, The 1090 Megahertz Riddle, 2nd ed., https://mode-s.org/decode/content/introduction.html
[2] “Binary to BCD and BCD to Binary” www.realdigital.org https://www.realdigital.org/doc/6dae6583570fd816d1d675b93578203d