RFSoC For RF Environment Monitoring

First Semester Report

Yana Galina, Jaime Mohedano Aragon, Kakit Wong, and Isaac Yamnitsky

Executive Summary (Yana) —The radio frequency (RF) spectrum is becoming increasingly congested, making high-fidelity measurements in the geospace and radio astronomy communities difficult. This congestion results in the need for RF interference mitigation techniques, and mitigation techniques require the use of RF spectrum monitoring tools. We plan to create an interactive Web application which will use a variety of RF sources to display the RF spectrum, efficiently parse and store data. Our main focus is on the application of the Xilinx Radio Frequency System-on-Chip device to RF spectrum monitoring, as well as other techniques of interest to current interference mitigation research. We will also handle previously recorded data, as well as live data from GNURadio.

Index Terms—software-defined radio, signal processing, radio astronomy

- *Yana Galina is with the Department of Electrical and Computer Engineering, Boston University, Boston, MA 02215, Email: yanag@bu.edu*
- *Jaime Mohedano Aragon is with the Department of Electrical and Computer Engineering, Boston University, Boston, MA 02215, Email: jainem@bu.edu*
- *Kakit Wong is with the Department of Electrical and Computer Engineering, Boston University, Boston, MA 02215, Email: kakit@bu.edu*
- *Isaac Yamnitsky is with the Department of Electrical and Computer Engineering, Boston University, Boston, MA 02215, Email: isaacy@bu.edu*

1 Introduction (Yana)

The radio frequency (RF) spectrum is becoming increasingly congested due to usage from both the public and private sectors. This congestion creates challenges for researchers in the geospace and radio astronomy communities, as congestion makes high-fidelity measurements difficult to achieve. Radio Frequency Interference (RFI) mitigation techniques are essential to carry out work in these fields [1]. Our clients at MIT Haystack Observatory are interested in a wide variety of RFI mitigation techniques relating to cancelation over space, time, and frequency. These techniques require the ability to monitor the wideband RF spectrum.

There is a specific need for monitoring tools that are both inexpensive and high bandwidth, and the recent progression of Software-Defined Radio (SDR) has the potential to fill this gap. SDR is becoming increasingly popular for RF applications due to its potential for rapid design cycles and also because SDR allows for reusing hardware over multiple applications [2]. SDR systems are characterized by having components implemented in software which traditionally have been implemented in hardware. SDR allows for more flexibility in changing radio parameters, such as bandwidth or center frequency, on the fly.

One particular SDR device which shows promise for RF monitoring is the Xilinx Radio Frequency System-on-Chip (RFSoC). Our project will focus on exploring the full capabilities of this device. This device has multiple inputs, and can be phase-synced with other boards allowing for spatial filtering applications, which are a current major topic of research [3]. The device has a large potential bandwidth of up to 2.5 GHz. It was designed to minimize energy cost per RF channel, which is critical since current techniques for wideband spectrum monitoring tend to be power intensive. It is relatively cheap compared to other boards with similar capabilities, with a price of $2,149 per board as of October 2021 [4]. All of these factors will help facilitate deployment of this technology in many different spaces, furthering its potential use cases for radio astronomy research.

The Xilinx RFSoCs also come with the Xilinx PYNQ framework, which simplifies development by allowing it to be carried out using mostly Python, an extremely popular high level programming language. Traditionally, development of this kind is done in a hardware description language such as Verilog or VHDL, which requires a specialized skill set. The PYNQ framework allows for users who do not have this niche skill set to develop the boards themselves without having to farm out the work to a digital hardware engineer (and having to pay for that engineer's time). The addition of the PYNQ framework makes the RFSoC device a promising candidate for developing complex RF applications that can be prototyped easily and cost effectively.

Along with the RFSoC, our project will provide monitoring tools for other RF data sources. These additional data sources will include pre-recorded data stored in the DigitalRF format, as well as data from other

SDR devices processed through GNURadio. DigitalRF is a standardized format for reading and writing RF data and was developed by our clients at MIT Haystack Observatory [5]. GNURadio is a widely-used signal-processing software toolkit for SDR devices [6].

This project is ultimately exploratory in nature. The complete depth and breadth of the potential of the Xilinx RFSoC with regards to geospace and radio astronomy applications is not yet known. We will continue to work closely with our clients at MIT Haystack Observatory as the project progresses to adjust and refine our goals based on their feedback and the functions we are able to achieve. We aim to create a practical base that can be extended as needed to serve future research needs.

2 CONCEPT DEVELOPMENT (YANA, JAIME)

Originally, we planned to create an application which processed and visualized data only from the Xilinx RFSoC board. However, we had to modify this concept due to not having access to an RFSoC board for the entirety of this semester. By mid-October, when it became clear that it would take a while to receive a board, we decided to shift to a broader goal of displaying and processing a variety of data sources. This shift widened the scope of our project, and set up our application to serve as a more flexible RF research tool.

For our front end, rather than starting from scratch, we began by piecing together related existing components. We chose to modify plots from StrathSDR for our visualization components. StrathSDR is a library that comes installed on the Xilinx RFSoC board, and contains a variety of graphs and visualizations for the board, including spectrum and spectrogram plots [7]. We decided to use plot classes from StrathSDR as a base to work from because the library is open source, contains code that pertains directly to the board we will be working with, and produces outputs similar to our desired outputs.

StrathSDR uses Plotly graphs, so we will continue using Plotly for additional visualizations. We considered options such as creating graphs from scratch with Plotly or Matplotlib. However, compared to Matplotlib, Plotly offers higher levels of customization and options for user interaction [8].

We decided to make our front end a Dash-based app. Dash was specifically developed for complex dashboard visualizations using Plotly graphs. Also, Dash applications are written in Python, which has many data processing libraries and tools relevant to our project (including the DigitalRF library), and is the programming language our team is most comfortable with. We considered building a dashboard in ReactJS, but chose Dash due to the tools that Python has to offer.

Initially the data processing and visualization was handled by the same Python script. Later in the semester, we decided to separate the front end (visualization) from the back end (processing). Decoupling the different parts of our application is good software practice, as it makes the code base more flexible and scalable. In our app, it will facilitate having multiple different data sources and will make it easier to add more features in the future.

We decided to use a Redis Stream database for communication between the front and back ends. This Redis Stream database will serve as an interface between the front end and the variety of data sources in the back end. Redis Stream is well suited for time series and message queues, which fits well with graphs that are updated over time.

In order for there to be communication between multiple Redis servers, the data must first be serialized. Then it can be transmitted, de-serialized and displayed on the front end web application. We decided to use PyArrow, the Python implementation of Apache Arrow, for this serialization. We chose this tool because it is more elaborated than serializing in JSON. It also has a greater efficiency for streaming applications.

We decided to use Python as the back-end programming language. Our reasons for using Python in the back end are similar to the reasons we are using Python in the front end. Python is easy to develop for, and has many data processing libraries and tools like Numpy and SciPy. Additionally, and perhaps most critically, the RFSoC board has tools specifically designed for integration with Python as part of the PYNQ Framework. We have not ruled out the possibility of adding C/C++ modules as needed for extra efficiency.

Our focus for this project is on the functionality of the different components, rather than any particular deployment scheme. This focus is due to the intended use case of our work. This project is meant to serve as a base which can be easily extended to suit future research needs. Our target audience is software-savvy researchers who will likely tweak the codebase to suit whatever particular goals they may have. As such, these researchers will presumably use their own deployment methods, whether that be running parts of the code on the cloud, or on machines in a lab.

3 SYSTEM DESCRIPTION (YANA)

3.1 System Block Diagram

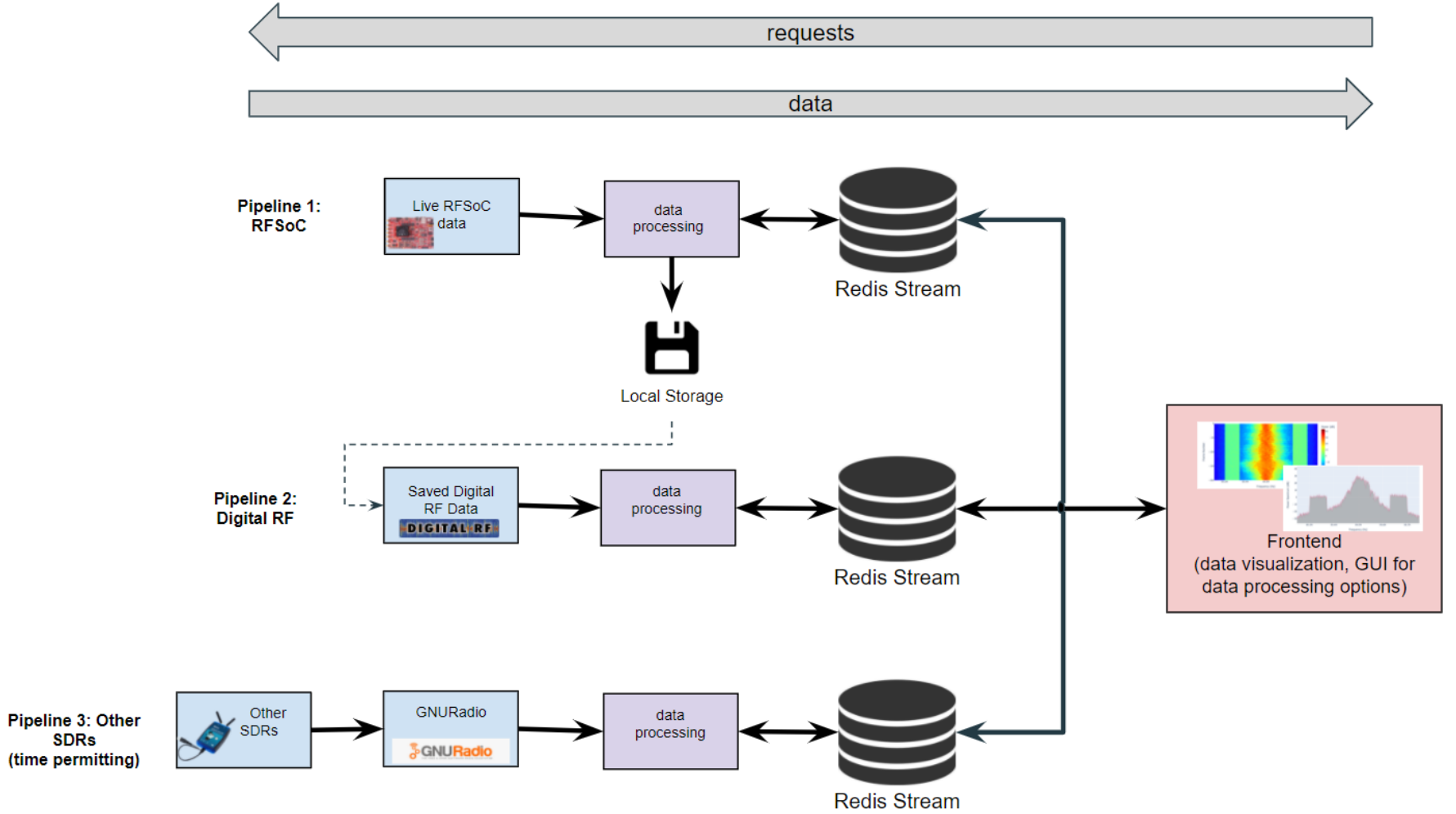

Fig. 1. Block Diagram of our system. The diagram shows the flow of data between the front end and the different back-end data pipelines.

3.2 System Overview

Our project consists of a Dash-based front end which interfaces with a variety of data-processing back-end applications. The front end makes requests to the back end on the user's behalf, processes the incoming data from the back end, and visualizes the data according to the user's preferences using Plotly graphs. The front end also serves as a GUI for the various data-processing options available to the user. The front end interacts with the back end through a Redis Stream server, which provides a single clean interface for the variety of data processing pipelines. The front end server will run on a host PC.

The core data processing options we aim to provide include spectrum visualization, digital down-conversion, and storing live incoming data. Additional data processing options will be added if time permits, and may include demodulating certain RF signals, implementing a polyphase filter bank, or any other kind of processing our client specifically requests.

The main data source is the Xilinx RFSoC board. The entire pipeline of taking in data, processing it, and sending it to the front end will be done on board. The core of the signal processing functions, such as the Fast Fourier Transforms (FFTs) necessary for spectrum analysis, will be done on the board's field-programmable gate array (FPGA) for efficiency. Programming the FPGA will be done using the PYNQ framework, which allows logical programming to be carried out using mostly Python.

The board's ARM-based processor will carry out further data processing such as appending timestamps and serializing the data, as well hosting the Redis server itself. The Redis server running on the team's board will be accessible via an IP address on the Boston University network.

An additional functionality of this pipeline will be storing the live data coming into the board in the DigitalRF format. Visualizing and storing incoming RFSoC data are the most critical parts of this project. As such, these two functions need to be implemented as efficiently as possible. The front end needs to be able to display the full spectrum of incoming live data (up to 2.5 GHz) no more than 5 seconds after the data enters the RFSoC device.

Another data source we will be using is previously recorded DigitalRF data. In this pipeline, data is processed by a Python back end using the DigitalRF Python library, and then fed into the Redis server, to be streamed to the front end. Both the Python back end and the Redis server will reside on a host PC.

The last data source we will include is data coming in from other SDRs through GNURadio. This data processing pipeline is not a critical component of our project, and we will only implement this portion of the project if time permits. For this component of the project, we will use an ADALM-PLUTO board which we have worked with this semester to learn SDR fundamentals. Our plan for this pipeline is to have the PLUTO board connected to a PC running GNURadio. GNURadio will process the incoming data (for example, run the raw data through a FFT filter), and output the result for additional back-end data processing and then into the Redis Server. All of these components will be running on a host PC connected to the board. Similar to the other data pipelines, the Redis server will stream data to the front end.

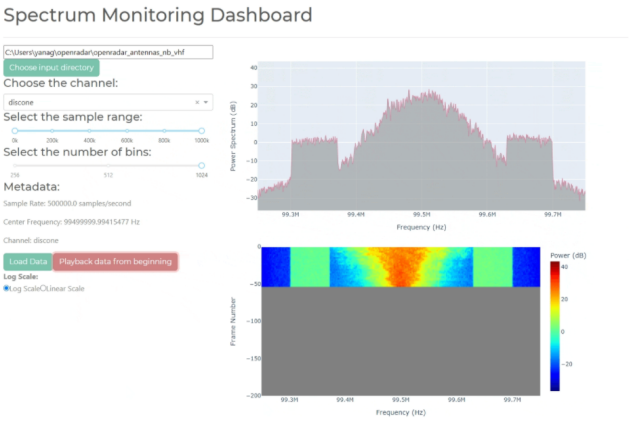

Fig. 2. Our preliminary GUI for the DigitalRF pipeline. Processing options are displayed on the left side of the dashboard, while the right side shows the spectrum (top) and spectrogram (bottom) graphs.

4 First Semester Progress (Yana, Jaime)

In September, after our time was first formed, we began by trying to understand the complete scope of our project. Then, our team focused on learning RF and SDR fundamentals. Our client provided us with the materials from an SDR workshop that he had previously run. This workshop served to familiarize us with SDR concepts and technologies that would be directly relevant to our project, including GNURadio and the DigitalRF library.

After gaining a baseline understanding of the tools we would be working with, we shifted to developing our first prototype. Since we did not have access to an RFSoC board, we decided to focus on creating a web application which can display previously-recorded data in the DigitalRF format. This corresponds to "Pipeline 2" in the system block diagram (Figure 1).

The first thing we got working was displaying static spectrum graphs with the data averaged over the entirety of a DigitalRF file. Then, achieved dynamic playback of a file over time. This functionality was achieved by splitting the file into many chunks, and using a Dash component which fires every 100 milliseconds to update the graph with the next chunk of data. The spectrum plot is the top graph shown in Figure 2.

After we were able to get playback of spectrum graphs, we added in a spectrogram waterfall plot, which displays the same data as the spectrum plot in a different format.

The spectrogram is the bottom graph shown in Figure 2.

Then, we added user interaction. We added a button for toggling the axis scales of the plots between linear and logarithmic. We added in an input field so that users could select which DigitalRF field to play back. We added a "metadata" section to display additional information from the DigitalRF file.

Next, we presented our prototype. During the prototype testing, we were able to confirm that our application produced accurate representations of DigitalRF data by comparing the graphs our application produced to the static graphs produced by the graphing tool found in the DigitalRF library.

After our prototype demo, we added additional UI components, including a drop down which lets the user select the antenna channel, a slider which selects which data points of the saved file to play back, and a slider which selects the number of Fast Fourier Transform bins for the spectrum graphs. We also rearranged the UI layout so that the options are on the left side of the page, while the visualizations are on the right side. We also began incorporating the Redis Server into our data pipeline.

5 Technical Plan (Jaime, Yana)

- *Task 1 - Adding additional graphs to front end*
  This task will involve adding histogram, voltage, and other graphs to the front end dashboard. We will also add select options, so that the user can choose which graph to see with the receiving data. We will be implementing Plotly graphs with DigitalRF data for this task.
  Lead: Yana, Assisting: Isaac
- *Task 2 - Arranging a better looking UI*
  This task will involve arranging the UI to be more visually pleasing and scalable, complete with tabs and dropdowns.
  Lead: Isaac, Assisting: Yana
- *Task 3 - Setting up and familiarizing ourselves with the RFSoC*
  This task will involve the basic setup of our RFSoC board in Professor Semeter's lab space and getting the demo project (StrathSDR) successfully running on the board.
  Lead: Kakit, Assisting: Jaime
- *Task 4 - Testing rate of possible data storage of RFSoC*
  This task will involve testing the rate of data we will be able to pull and store from the RFSoC while the board is receiving live data. This task is non-trivial because the board has a very large bandwidth, and the rate of raw data coming in will almost certainly be greater than the rate of data that can be pulled off the board via its ports.
  Lead: Jaime, Assisting: Kakit
- *Task 5 - Incorporating a Redis Server to decouple front and back ends for DigitalRF pipeline*
  As of writing this report, there is no logical separation between the front and the back end. This task will involve incorporating a Redis Stream server to serve as an interface between the two, and also designing the message structure necessary for this functionality. This task will focus on serialization of DigitalRF data so that it can be processed in the back end, transmitted to the front end server via the Redis server, and displayed in the front end.
  Lead: Yana, Assisting: Jaime
- *Task 6 - Testing workable rate of streaming server*
  This task will involve testing the rate of data we can stream through the Redis server between our front and back ends.
  Lead: Jaime, Assisting: Yana
- *Task 7 - Connecting RFSoC to streaming server*
  This task involves adding the RFSoC data pipeline to the rest of the codebase. This will involve setting up a Redis server on the board, and serializing incoming data into the server, to be received and displayed by the front end.
  Lead: Yana, Assisting: Jaime
- *Task 8 - Arranging front end to handle multiple data sources*
  This task is the corresponding front-end work for Task 7. It involves arranging the UI so that it can handle both live RFSoC and pre-recorded DigitalRF data.
  Lead: Isaac, Assisting: Yana
- *Task 9 - Processing RFSoC data*
  This task will involve adding additional data processing options for RFSoC. This will include demodulating certain RF signals, implementing a polyphase filter bank, or any other kind of processing our client specifically requests.
  Lead: Kakit, Assisting: Jaime
- *Task 10 - Adding Other GNURadio data*
  This is a lower-priority task involving incorporating a GNURadio data pipeline into our application. We will be using data from the ADALM-PLUTO board for this.
  Lead: Kakit, Assisting: Jaime
- *Task 11 - Testing, Optimization, Cleanup*
  This task will involve testing our code, coming up with any optimizations as needed, and cleaning up the codebase.
  Lead: All members

6 BUDGET ESTIMATE (YANA)

Item	Description	Quantity	Cost
1	Xilinx RFSoC 2x2 kit	2	$2,149
2	ADALM-PLUTO Evaluation Board	1	$208
3	Antenna for RFSoC board	1	$50 (estimated)
4	SD Card Reader	1	$17 (estimated)
	Total Cost		$4573

Two Xilinx RFSoC 2x2 kits are listed because one board is the property of our customer, and one was purchased for our team by the BU ECE department. Our customer is allowing us to remotely use their board while we wait for our board to arrive. Our RFSoC kit was purchased by the ECE Department with the coordination of Professor Josh Semeter.

The ADALM-PLUTO Evaluation Board was lent to us by Professor Josh Semeter.

The antenna and the SD card reader are the only components that we are currently planning on purchasing using the standard allotted budget for the course.

7 ATTACHMENTS (JAIME, YANA)

7.1 Appendix 1 – Engineering Requirements

Requirement	Value, range, tolerance, units
Displaying the *full range* of live RFSoC data	Display a bandwidth of up to 2.5GHz (depending on antenna) Spectrogram should display at least 30 seconds of data at a time
Minimize delay between live data entering RFSoC board and being displayed on front end	Delay should be less than 5 seconds
Storing (digitally down-converted) live data	Must be able to store a bandwidth of *at least* 30 MHz
Easy set up	A new user should be able to download and run the code on their board in < 2 hours

7.2 Appendix 2 – Gantt Chart (Yana)

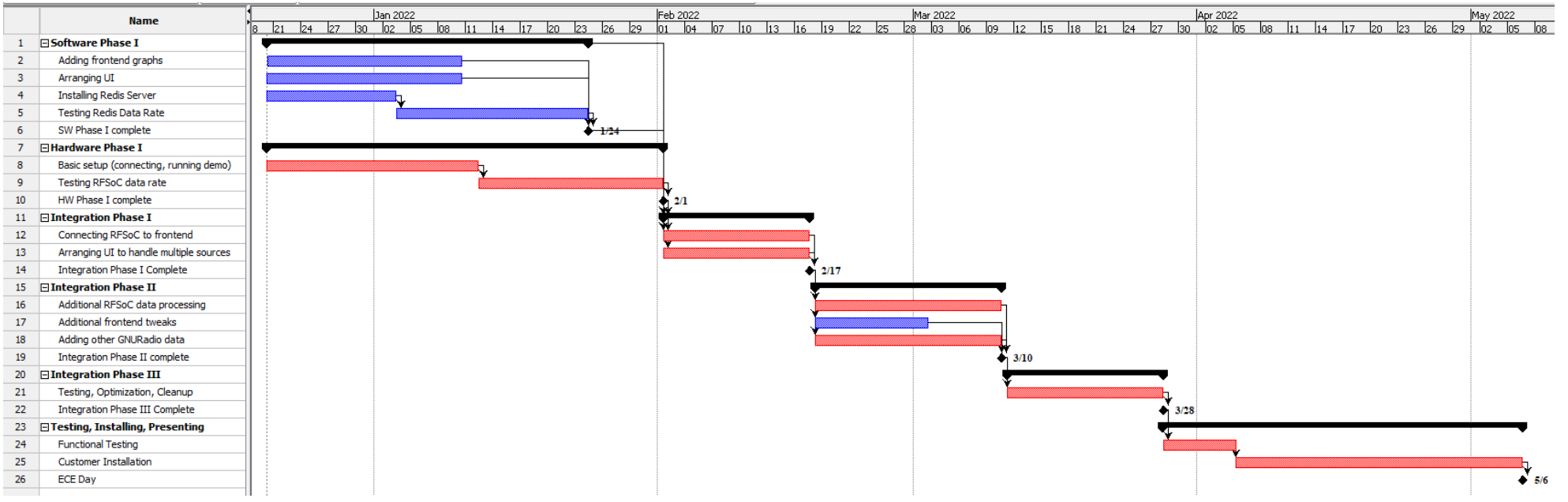

Fig. 3. Our Gantt Chart, showing the tasks we will be working on from the end of this semester to ECE Day on May 6th, 2022.

7.3 Appendix 3 – Other Appendices

7.3.1 Acknowledgment

We wish to thank our clients John Swoboda and Sharanya Srinivas for their continued guidance on this project. We also wish to thank Professors Joshua Semeter, Alan Pisano, and Osama Alshaykh for their continuing advice and support.

7.3.2 References

[1] A. Leshem and A. -. van der Veen, "Radio-astronomical imaging in the presence of strong radio interference," in *IEEE Transactions on Information Theory*, vol. 46, no. 5, pp. 1730-1747, Aug. 2000, doi: 10.1109/18.857787.

[2] A. M. Wyglinski, D. P. Orofino, M. N. Ettus and T. W. Rondeau, "Revolutionizing software defined radio: case studies in hardware, software, and education," in IEEE Communications Magazine, vol. 54, no. 1, pp. 68-75, January 2016, doi: 10.1109/MCOM.2016.7378428.

[3] Veen, Alle-Jan & Wijnholds, Stefan. (2013). Signal Processing Tools for Radio Astronomy. 10.1007/978-1-4614-6859-2_14.

[4] https://www.xilinx.com/support/university/boards-portfolio/xup-boards/RFSoC2x2.html

[5] https://github.com/MITHaystack/digital_rf

[6] https://www.gnuradio.org/about/

[7] https://github.com/strath-sdr/rfsoc_studio

[8] A Guide to Data Visualization with Plotly. https://towardsdatascience.com/python-for-data-science-a-guide-to-data-visualization-with-plotly-969a59997d0c