OSTBAYERISCHE TECHNISCHE HOCHSCHULE REGENSBURG

WIRELESS SIGNAL TRANSMISSION AND LED CONTROL USING RFSOC-PYNQ IN JUPYTERLAB

By

Maulik Dhorajiya – 3370281

Dhaval Kalathiya – 3447615

A master's Thesis

Submitted to the Faculty of Electrical and Information Engineering

Supervisors

Prof. Dr.-Ing. Florian Aschauer (Asf)

Prof. Dr.-Ing. Susanne Hipp (Hip)

In Fulfilment of the Requirements

for the Degree of

Master of Electrical and Microsystems Engineering

January 2025

Abstract

This project demonstrates the capabilities of the RFSoC-PYNQ platform in achieving wireless signal transmission and real-time control of LEDs using a user-operated push button. Leveraging JupyterLab as the development environment, RF signals were generated, modulated, and transmitted wirelessly from one RFSoC board to another. The received signals were processed to control built-in LEDs and an external RGB LED circuit connected via the PMOD interface. This implementation utilized RFSoC's programmable logic and high-speed data converters to seamlessly integrate RF signal processing and hardware interfacing.

The project successfully achieved wireless signal transmission and reception, enabling efficient real-time control of LEDs based on user inputs. Signal visualization through Jupyter notebooks provided an interactive platform for analysing transmitted and received data. Key challenges included RF interference and circuit design complexities, which were addressed through filtering techniques and iterative design improvements.

This work highlights the versatility of the RFSoC-PYNQ platform for embedded system design and signal processing applications. The project demonstrates its potential in IoT, remote control systems, and RF education by showcasing the seamless integration of signal processing and hardware control within a single platform. It serves as a foundation for future developments in real-time communication and hardware interaction using RFSoC.

Acknowledgment

We would like to extend our sincere gratitude to our supervisors, Prof. Dr.-Ing. Florian Aschauer (Asf) and Prof. Dr.-Ing. Susanne Hipp (Hip), for their invaluable support, guidance, and encouragement throughout this project. Their expertise in the field of electrical and information engineering has been a constant source of inspiration and has significantly contributed to the success of this thesis. We deeply appreciate their patience, constructive feedback, and willingness to share their profound knowledge, which have helped us overcome challenges and refine our work.

Our heartfelt thanks go to the Faculty of Electrical and Information Engineering for providing the necessary resources and an exceptional learning environment. The access to advanced tools, such as the RFSoC-PYNQ platform and related development resources, was crucial for the implementation of our project. We also acknowledge the efforts of the administrative and technical staff, whose assistance ensured the smooth progress of our work.

We would also like to express our gratitude to our peers and colleagues, who provided valuable suggestions, discussions, and camaraderie during this journey. Finally, we are profoundly thankful to our families and friends for their unwavering support, encouragement, and understanding. Their belief in us has been our source of strength, motivating us to persevere and complete this work successfully.

List of Figures

Table of Contents

1 Introduction

1.1. Overview of the Project

The project, "Wireless Signal Transmission and LED Control Using RFSoC-PYNQ in JupyterLab," explores the integration of advanced hardware and software technologies to achieve wireless communication and real-time hardware control. Using the versatile Zynq UltraScale+ RFSoC platform, the project demonstrates wireless signal transmission to control built-in and external LEDs via RF frequencies.

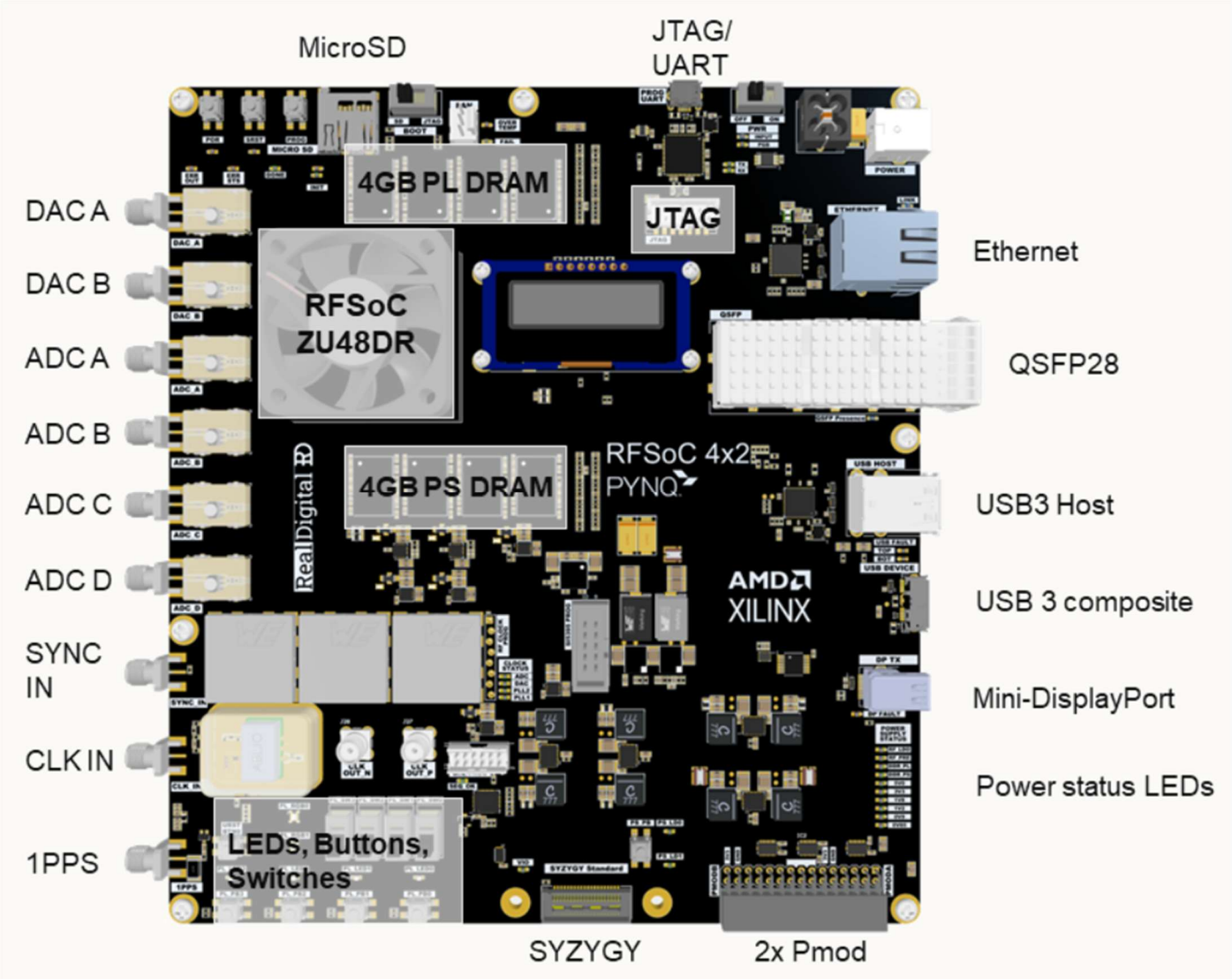


Figure 1.1: RFSoC 4x2 B Board Overview

Python and the PYNQ framework provide a seamless environment for hardware-software integration, enabling dynamic and interactive control. This approach highlights the utility of RFSoC in education, IoT systems, and wireless communications.

1.2. Motivation and Background

The increasing demand for wireless control in IoT and embedded systems underscores the importance of integrating programmable hardware like RFSoC. The RFSoC platform's unique features—such as integrated RF Data Converters (RFDCs) and support for high-speed data processing—make it an ideal choice for real-time wireless applications. By leveraging the flexibility of Python and JupyterLab, this project enables rapid prototyping and debugging, making it a valuable tool for research and practical deployment. The project's application extends beyond academic purposes to industries like automation, smart homes, and entertainment systems.

1.3. Introduction to RFSoC and PYNQ Framework

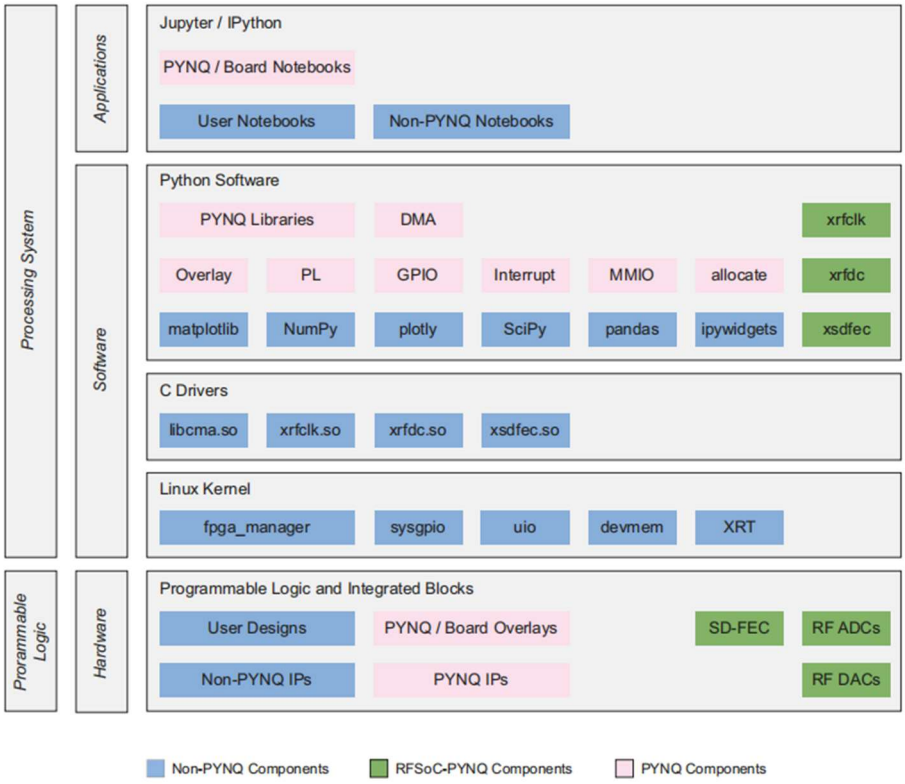


Figure 1.2: Simplified diagram of the PYNQ framework and ecosystem

The Zynq UltraScale+ RFSoC combines high-speed ADCs, DACs, programmable logic, and processing cores into a single chip, making it a compact yet powerful platform for SDR (Software Defined Radio) applications. RFSoC's ability to handle multiple RF channels simultaneously is pivotal for tasks like wireless transmission and control. The PYNQ framework simplifies the development process by abstracting hardware complexities, allowing developers to interact with RFSoC's capabilities through Python in JupyterLab. This integration fosters efficiency and innovation in hardware design and implementation.

1.4. Project Goals

The project aims to:

1. Demonstrate wireless signal transmission using RFSoC.
2. Control built-in LEDs based on transmitted RF signal frequencies.
3. Implement a custom circuit for PMOD RGB LEDs to extend functionality.
4. Showcase real-time control and responsiveness of hardware to transmitted signals.

1.5. Hardware and Software

1.5.1. Hardware

1. **Zynq UltraScale+ RFSoC Boards:**
   - Two RFSoC boards are used: one for transmission and the other for reception.
   - Built-in ADCs and DACs handle high-speed RF signal processing.
2. **PMOD RGB LED Circuit:**
   - Three 2N2222 transistors control Red, Green, and Blue LEDs.
   - Connected to the RFSoC's PMOD interface for signal-based control.
3. **Antennas:**
   - Used for transmitting and receiving RF signals wirelessly.
4. **Miscellaneous Components:**
   - Breadboard, resistors, and connecting wires for circuit assembly.

1.5.2. Software

1. **PYNQ Framework:**
   - Provides Python-based abstraction for interacting with RFSoC hardware.
   - Runs in the JupyterLab environment.
2. **Python Libraries:**
   - NumPy for data processing.
   - **ipywidgets for interactive visualizations.**
3. **Custom Python Code:**
   - Designed for RF signal generation and LED response mapping.
4. **Circuit Design:**
   - EasyEDA Online tool for Circuit and PCB designing

1.6. Technical Approach

1.6.1. Wireless Transmission and Reception

Using two RFSoC boards, the project transmits RF signals in predefined frequency bands to achieve specific hardware responses:

- **Built-in LEDs** are controlled by frequencies:
  - 0.8 GHz: Turns ON LED 0.
  - 1.2 GHz: Turns ON LED 1.
  - 1.8 GHz: Turns ON LED 2.
  - 2.2 GHz: Turns OFF all LEDs.
- **PMOD RGB LEDs** respond to:
  - 0.8 GHz: Turns ON Blue LED.
  - 1.2 GHz: Turns ON Green LED.
  - 1.8 GHz: Turns ON Red LED.
  - 2.2 GHz: Turns OFF all LEDs.

The transmitter generates RF signals based on button presses, while the receiver board monitors the spectrum, detects signals, and maps frequencies to LED actions.

1.7. Hardware Integration

- **Built-in LEDs:**
  - Controlled directly using RFSoC's internal resources.
- **PMOD RGB LEDs:**
  - A custom circuit comprising three 2N2222 transistors controls the Red, Green, and Blue LEDs.
  - Connected to the PMOD interface of the RFSoC board.

1.8. Significance and Applications

1. **IoT Systems:**
   - Remote device management and wireless control.
2. **Educational Tools:**
   - Hands-on demonstrations for teaching RF communication and hardware programming.
3. **Automation:**
   - Smart home lighting and industrial signaling systems.
4. **Debugging and Testing:**
   - Simplifying RF signal-based testing in embedded systems.

2 Transmission and Receiver for LED Control

2.1. Hardware Photo

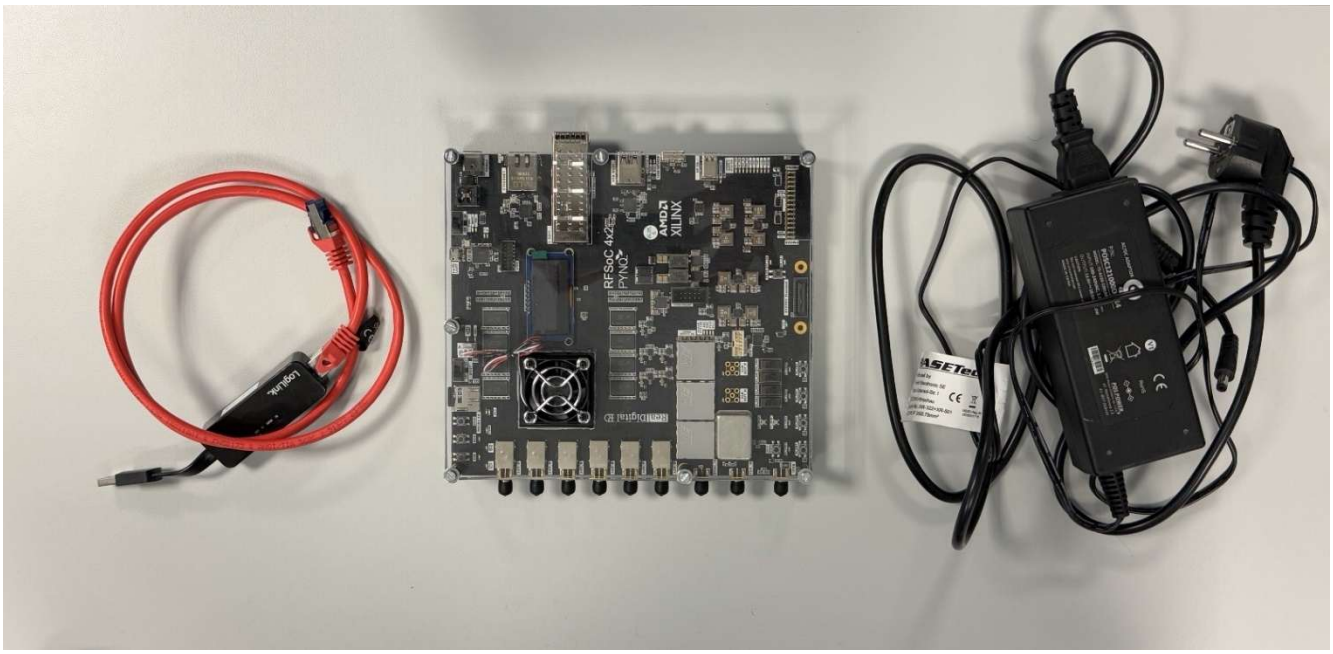


Figure 2.1: The RFSoC boards (transmitter and receiver) with their power supplies and Ethernet connections.

Figure 2.2: two antennas used for transmission and reception.

2.2. Hardware Setup

2.2.1. Transmitter Setup

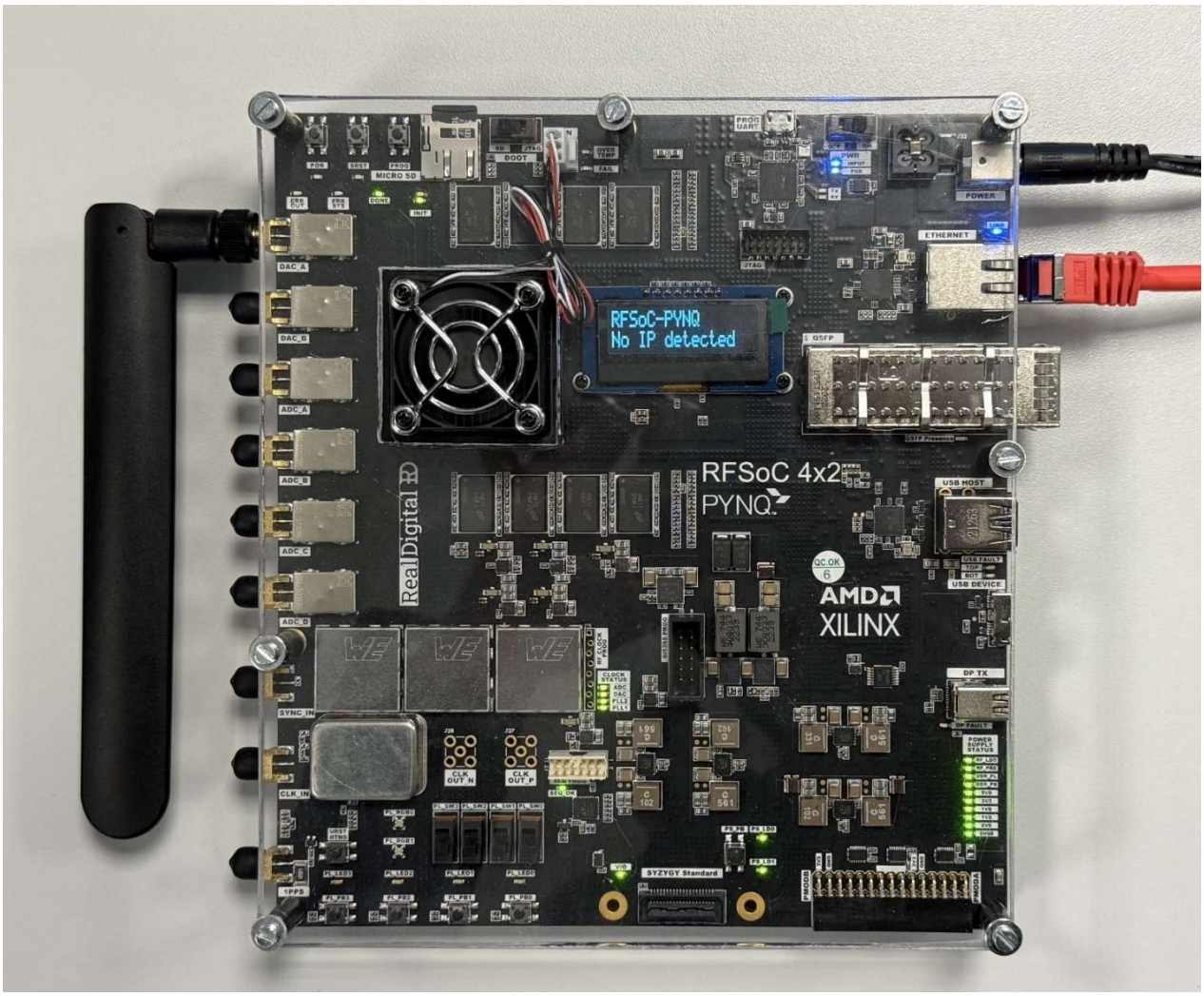


Figure 2.3: The Photo of Transmitter Setup

Explanation:

- **Board Configuration:**
  - The transmitter RFSoC board is connected to a PC using an Ethernet cable for control through JupyterLab (accessed via http://192.168.2.99:9090/lab with the password xilinx).
  - Built-in buttons on the board are allows users to select specific RF signal frequencies.
- **Power Supply:**
  - The board is powered via a 12V adapter, ensuring stable operation.
- **Antenna Connection:**
  - An external antenna is connected to DAC_A for wireless signal transmission

2.2.2. Receiver Setup

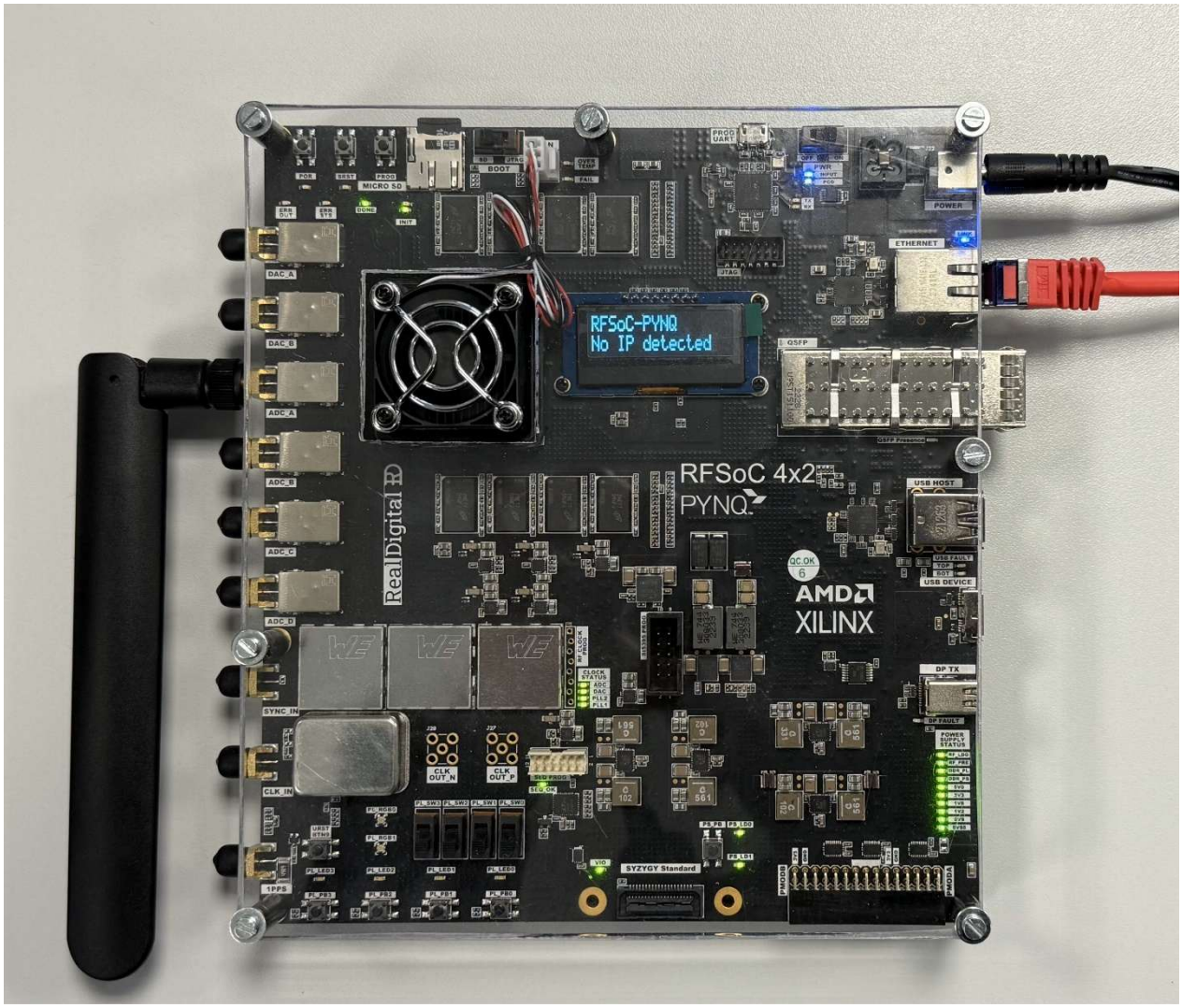


Figure 2.4: The Photo of Receiver Setup

Explanation:

- **Board Configuration:**
  - The receiver RFSoC board is also connected to a PC via Ethernet to monitor spectrum and control LEDs through JupyterLab.
  - Built-in LEDs on the board are configured to respond to specific RF signals.
- **Power Supply:**
  - A 12V adapter provides power to the board.
- **Antenna Connection:**
  - An external antenna is connected to ADC_A for wireless signal reception.

2.3. Transmitter Code for LED Control

2.3.1. Code Overview

[1]:
```
from pynq.overlays.base import BaseOverlay
from time import sleep

print('wait')

base = BaseOverlay('base.bit')
base.init_rf_clks()

def set_transmitter_channel(channel, enable, gain, frequency):
    channel.control.enable = enable
    channel.control.gain = gain
    channel.dac_block.MixerSettings['Freq'] = frequency

print('press now')

while True:
    if base.buttons[0].read():
        set_transmitter_channel(base.radio.transmitter.channel[1], True, 1, 800)
        print("Transmitting 0.8 GHz signal to turn ON LED 0")
        sleep(0.2)

    elif base.buttons[1].read():
        set_transmitter_channel(base.radio.transmitter.channel[1], True, 1, 1200)
        print("Transmitting 1.2 GHz signal to turn ON LED 1")
        sleep(0.2)

    elif base.buttons[2].read():
        set_transmitter_channel(base.radio.transmitter.channel[1], True, 1, 1800)
        print("Transmitting 1.8 GHz signal to turn ON LED 2")
        sleep(0.2)

    elif base.buttons[3].read():
        set_transmitter_channel(base.radio.transmitter.channel[1], True, 1, 2200)
        print("Transmitting 2.2 GHz signal to turn OFF all LED")
        sleep(0.2)
```

Figure 2.5: Transmitter Code for LED Control

The transmitter code configures the RFSoC board to transmit RF signals at specific frequencies based on button inputs. These signals are received by another RFSoC board to control LEDs.

2.3.2. Code Explanation

2.3.2.1. Imports and Initialization

```
from pynq.overlays.base import BaseOverlay
from time import sleep
```

- **pynq.overlays.base:** Provides access to the base overlay for RFSoC, enabling control of buttons, LEDs, and RF modules.
- **time.sleep:** Used to introduce delays between transmissions to ensure smooth operation.

```
base = BaseOverlay('base.bit')
base.init_rf_clks()
```

- **BaseOverlay('base.bit'):** Loads the base bitstream required to configure the RFSoC hardware.
- **base.init_rf_clks():** Initializes RF clocks, preparing the board for RF operations.

2.3.2.2. Function to Configure Transmitter

This function sets the transmitter's parameters:

```
def set_transmitter_channel(channel, enable, gain, frequency):
    channel.control.enable = enable
    channel.control.gain = gain
    channel.dac_block.MixerSettings['Freq'] = frequency
```

- **enable:** Activates or deactivates the channel.
- **gain:** Adjusts the signal strength.
- **frequency:** Sets the transmission frequency in Hz.

2.3.2.3. Main Loop for Button Monitoring

```
while True:
    if base.buttons[0].read():
        set_transmitter_channel(base.radio.transmitter.channel[1], True, 1, 800)
        print("Transmitting 0.8 GHz signal to turn ON LED 0")
        sleep(0.2)

    elif base.buttons[1].read():
        set_transmitter_channel(base.radio.transmitter.channel[1], True, 1, 1200)
        print("Transmitting 1.2 GHz signal to turn ON LED 1")
        sleep(0.2)

    elif base.buttons[2].read():
        set_transmitter_channel(base.radio.transmitter.channel[1], True, 1, 1800)
        print("Transmitting 1.8 GHz signal to turn ON LED 2")
        sleep(0.2)

    elif base.buttons[3].read():
        set_transmitter_channel(base.radio.transmitter.channel[1], True, 1, 2200)
        print("Transmitting 2.2 GHz signal to turn OFF all LED")
        sleep(0.2)
```

- **while True:** Creates an infinite loop to continuously monitor button states.
- **Button 0: When pressed, it transmits 0.8 GHz.**
  - Action: This signal turns ON LED 0 on the receiver.
- **Button 1: Transmits 1.2 GHz.**
  - Action: Turns ON LED 1.
- **Button 2: Transmits 1.8 GHz.**
  - Action: Turns ON LED 2.
- **Button 3: Transmits 2.2 GHz.**
  - Action: Turns OFF all LEDs.

2.4. Output of Transmitter Code of LED Control

When the transmitter code is executed, it generates specific RF signals based on button inputs. Below is a description of the outputs observed during operation:

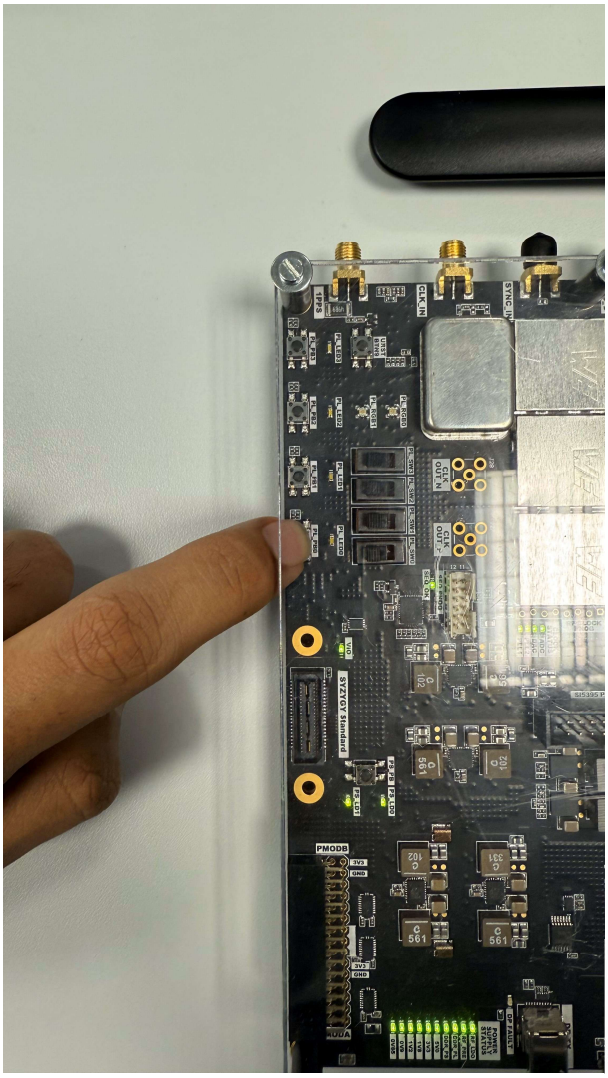
2.4.1. Initialization

```
wait
press now
```

Indicates the board is ready to respond to button presses.

2.4.2.Button Press

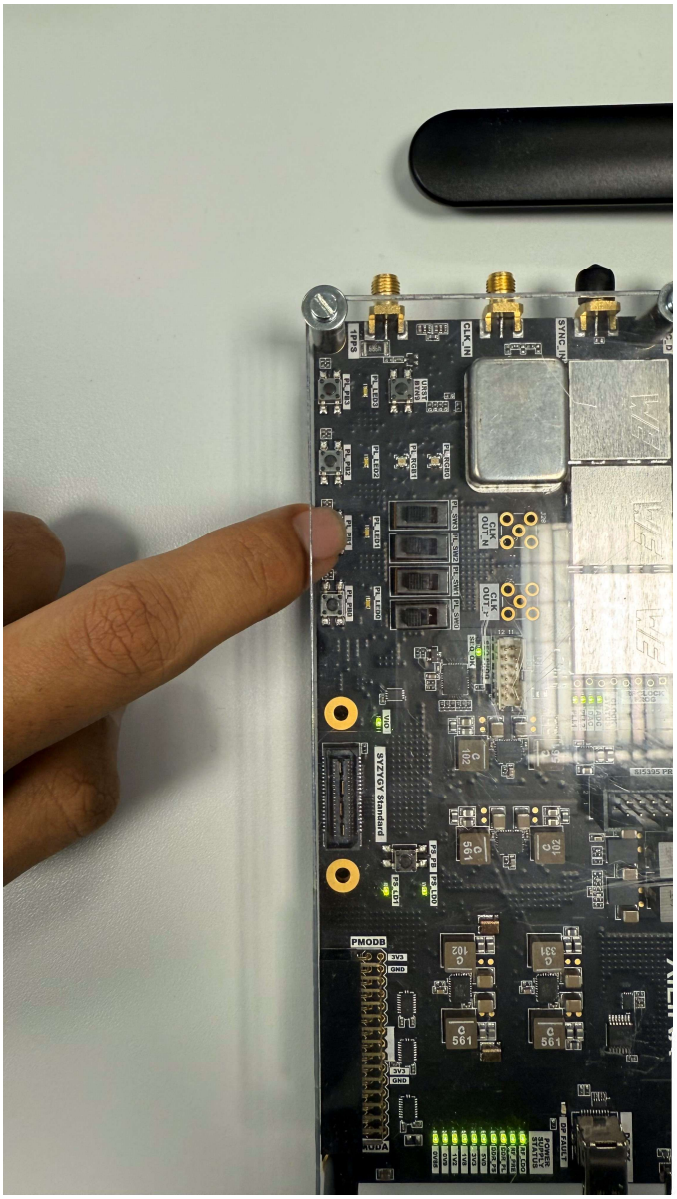
2.4.2.1. When Button 0 is pressed:



```
wait
press now
Transmitting 0.8 GHz signal to turn ON LED 0
```

The transmitter sends an RF signal at 0.8 GHz.

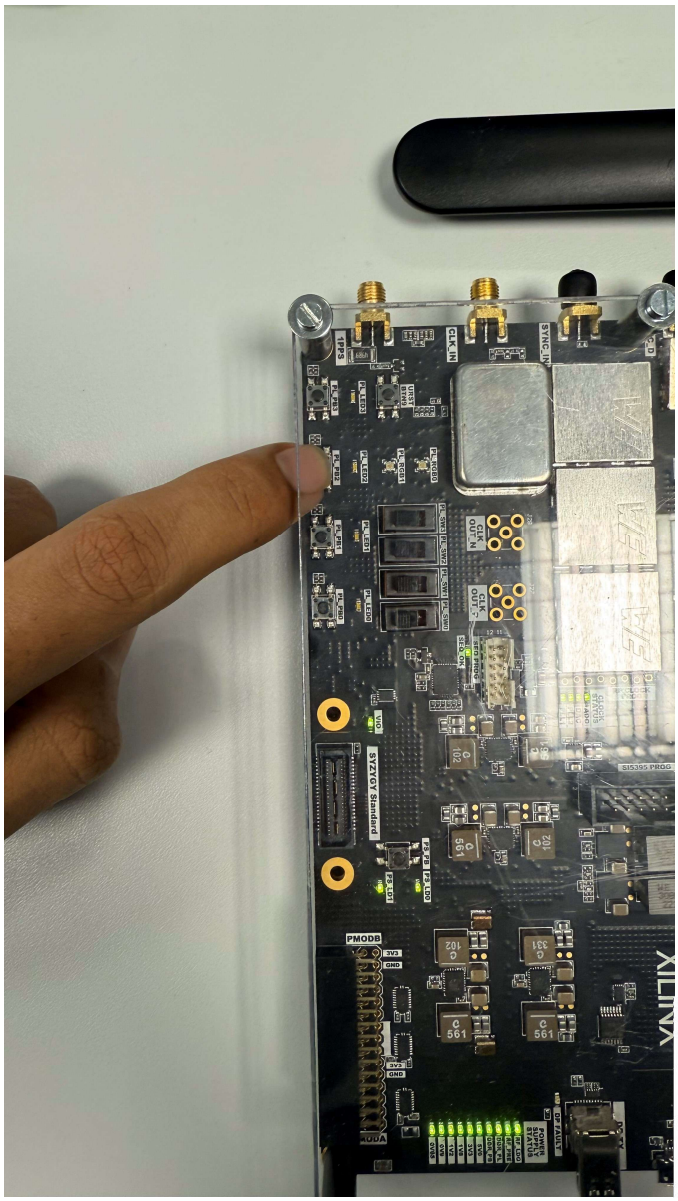
2.4.2.2. When Button 1 is pressed:



```
wait
press now
Transmitting 0.8 GHz signal to turn ON LED 0
Transmitting 1.2 GHz signal to turn ON LED 1
```

The transmitter sends an RF signal at 1.2 GHz.

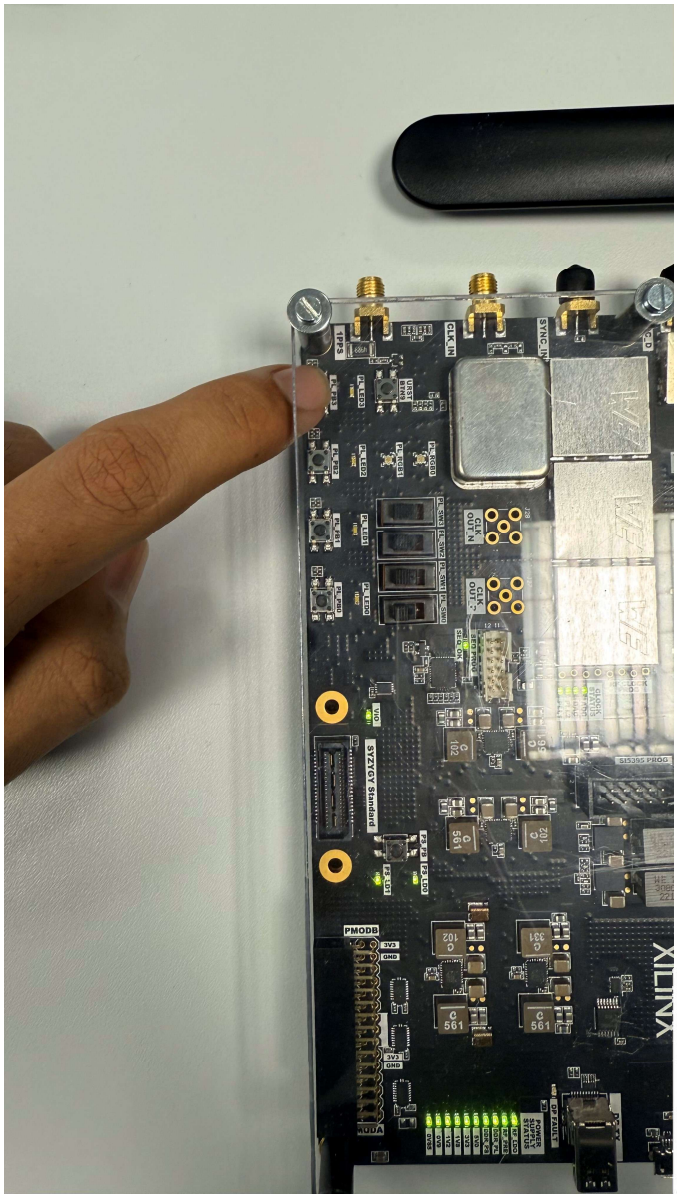
2.4.2.3. When Button 2 is pressed:



```
wait
press now
Transmitting 0.8 GHz signal to turn ON LED 0
Transmitting 1.2 GHz signal to turn ON LED 1
Transmitting 1.8 GHz signal to turn ON LED 2
```

The transmitter sends an RF signal at 1.8 GHz.

2.4.2.4. When Button 3 is pressed:



```
wait
press now
Transmitting 0.8 GHz signal to turn ON LED 0
Transmitting 1.2 GHz signal to turn ON LED 1
Transmitting 1.8 GHz signal to turn ON LED 2
Transmitting 2.2 GHz signal to turn OFF all LED
```

The transmitter sends an RF signal at 2.2 GHz.

2.5. Receiver Code for LED Control

2.5.1. Code Overview

[1]:
```
import time
import numpy as np
import ipywidgets as ipw
from pynq.lib import pmod
from pynq.overlays.base import BaseOverlay
from rfsystem.spectrum_sweep import SpectrumAnalyser

base = BaseOverlay('base.bit')
base.init_rf_clks()

analysers = []
period, duty = 1000, 90
period1, duty1 = 1000, 90
number_samples = 12000
sample_frequency = 2457.6e6

for led in base.leds:
    led.off()

led_0 = 0.8e9
led_0_min_tol = led_0/100 * 99.5
led_0_max_tol = led_0/100 * 100.5

led_1 = 1.2e9
led_1_min_tol = led_1/100 * 99.5
led_1_max_tol = led_1/100 * 100.5

led_2 = 1.8e9
led_2_min_tol = led_2/100 * 99.5
led_2_max_tol = led_2/100 * 100.5

led_off = 2.2e9
led_off_min_tol = led_off/100 * 99.5
led_off_max_tol = led_off/100 * 100.5

analysers.append(SpectrumAnalyser(channel = base.radio.receiver.channel[3],
                                  sample_frequency = sample_frequency,
                                  number_samples = number_samples,
                                  title = ''.join(['Spectrum Analyser Channel A ']), height = None, width = None))

ipw.VBox([analyser.spectrum_plot.get_plot() for analyser in analysers])
```

Figure 2.6: Receiver Code for LED Control part 1

[2]:
```
while True:
    for analyser in analysers:
        analyser.update_spectrum()

    led_0_f_min_index = np.argmin(np.abs(np.array((analyser.spectrum_plot._data.x)) - led_0_min_tol))
    led_0_f_max_index = np.argmin(np.abs(np.array((analyser.spectrum_plot._data.x)) - led_0_max_tol))

    led_1_f_min_index = np.argmin(np.abs(np.array((analyser.spectrum_plot._data.x)) - led_1_min_tol))
    led_1_f_max_index = np.argmin(np.abs(np.array((analyser.spectrum_plot._data.x)) - led_1_max_tol))

    led_2_f_min_index = np.argmin(np.abs(np.array((analyser.spectrum_plot._data.x)) - led_2_min_tol))
    led_2_f_max_index = np.argmin(np.abs(np.array((analyser.spectrum_plot._data.x)) - led_2_max_tol))

    led_off_f_min_index = np.argmin(np.abs(np.array((analyser.spectrum_plot._data.x)) - led_off_min_tol))
    led_off_f_max_index = np.argmin(np.abs(np.array((analyser.spectrum_plot._data.x)) - led_off_max_tol))

    for i in range(led_0_f_min_index, led_0_f_max_index):
        A = np.floor(analyser.get_spectrum()[i]).astype(int)
        if A == -125:
            print('LED_0 is on')
            base.leds[0].on()
            base.leds[1].off()
            base.leds[2].off()

    for i in range(led_1_f_min_index, led_1_f_max_index):
        B = np.floor(analyser.get_spectrum()[i]).astype(int)
        if B == -125:
            print('LED_1 is on')
            base.leds[0].off()
            base.leds[1].on()
            base.leds[2].off()

    for i in range(led_2_f_min_index, led_2_f_max_index):
        C = np.floor(analyser.get_spectrum()[i]).astype(int)
        if C == -125:
            print('LED_2 is on')
            base.leds[0].off()
            base.leds[1].off()
            base.leds[2].on()

    for i in range(led_off_f_min_index, led_off_f_max_index):
        D = np.floor(analyser.get_spectrum()[i]).astype(int)
        if D == -125:
            print('LED_all is off')
            base.leds[0].off()
            base.leds[1].off()
            base.leds[2].off()
```

Figure 2.7: Receiver Code for LED Control part 2

The receiver code monitors the RF spectrum, detects specific frequencies, and activates the corresponding built-in LEDs.

2.5.2. Code Explanation

2.5.2.1. Imports and Initialization

```
import time
import numpy as np
import ipywidgets as ipw
from pynq.lib import pmod
from pynq.overlays.base import BaseOverlay
from rfsystem.spectrum_sweep import SpectrumAnalyser
```

- **time:** Provides time-based utilities for delays and timing operations.
- **numpy (as np):** Used for numerical operations, such as frequency analysis and index calculations.
- **ipywidgets (as ipw):** Supports interactive visualizations in JupyterLab.
- **pynq.lib.pmod:** Provides access to the PMOD interface for hardware interaction.
- **BaseOverlay:** Loads the RFSoC base overlay for hardware control.
- **SpectrumAnalyser:** Offers tools for RF spectrum analysis, crucial for detecting transmitted frequencies.

```
base = BaseOverlay('base.bit')
base.init_rf_clks()
```

- **BaseOverlay('base.bit'):** Loads the default configuration file for RFSoC hardware resources.
- **base.init_rf_clks():** Initializes RF clocks, preparing the system for signal analysis.

2.5.2.2. Spectrum Analyzer and Configuration

```
analysers = []
period, duty = 1000, 90
period1, duty1 = 1000, 90
number_samples = 12000
sample_frequency = 2457.6e6
```

- **analysers:** An empty list to store spectrum analysers for each channel.
- **number_samples:** Specifies the number of samples for RF signal analysis.
- **sample_frequency:** Defines the sampling rate for the RF spectrum (2.4576 GHz in this case).

2.5.2.3. Frequency Definitions and Tolerances

```
led_0 = 0.8e9
led_0_min_tol = led_0/100 * 99.5
led_0_max_tol = led_0/100 * 100.5

led_1 = 1.2e9
led_1_min_tol = led_1/100 * 99.5
led_1_max_tol = led_1/100 * 100.5

led_2 = 1.8e9
led_2_min_tol = led_2/100 * 99.5
led_2_max_tol = led_2/100 * 100.5

led_off = 2.2e9
led_off_min_tol = led_off/100 * 99.5
led_off_max_tol = led_off/100 * 100.5
```

- Each LED frequency (0.8 GHz, 1.2 GHz, 1.8 GHz, and 2.2 GHz) is defined with an associated tolerance of ±0.10%.
- **For example:**
  - led_0_min_tol and led_0_max_tol: Tolerances for LED 0 activation at 0.8 GHz.

2.5.2.4. Spectrum Analyzer Initialization

```
analysers.append(SpectrumAnalyser(channel = base.radio.receiver.channel[3],
                                  sample_frequency = sample_frequency,
                                  number_samples = number_samples,
                                  title = ''.join(['Spectrum Analyser Channel A ']), height = None, width = None))
```

- A Spectrum Analyser instance is created for channel 3 of the RF receiver.
- sample_frequency and number_samples determine the resolution of the analysis.
- title labels the spectrum plot for interactive use in JupyterLab.

2.5.2.5. Interactive Spectrum Plot

```
ipw.VBox([analyser.spectrum_plot.get_plot() for analyser in analysers])
```

- **ipw.VBox:** Creates a vertical box layout in JupyterLab to display interactive spectrum plots for all analyzers.

2.5.2.6. Main Loop for Spectrum Analysis

```
while True:
    for analyser in analysers:
        analyser.update_spectrum()
```

- **while True:** Keeps the receiver code running indefinitely to monitor RF signals.
- **analyser.update_spectrum():** Updates the spectrum analyzer with the latest RF data.

2.5.2.7. Frequency Detection and LED Control

For each LED frequency, the code checks if the signal falls within its tolerance range:

```
led_0_f_min_index = np.argmin(np.abs(np.array((analyser.spectrum_plot._data.x)) - led_0_min_tol))
led_0_f_max_index = np.argmin(np.abs(np.array((analyser.spectrum_plot._data.x)) - led_0_max_tol))

led_1_f_min_index = np.argmin(np.abs(np.array((analyser.spectrum_plot._data.x)) - led_1_min_tol))
led_1_f_max_index = np.argmin(np.abs(np.array((analyser.spectrum_plot._data.x)) - led_1_max_tol))

led_2_f_min_index = np.argmin(np.abs(np.array((analyser.spectrum_plot._data.x)) - led_2_min_tol))
led_2_f_max_index = np.argmin(np.abs(np.array((analyser.spectrum_plot._data.x)) - led_2_max_tol))

led_off_f_min_index = np.argmin(np.abs(np.array((analyser.spectrum_plot._data.x)) - led_off_min_tol))
led_off_f_max_index = np.argmin(np.abs(np.array((analyser.spectrum_plot._data.x)) - led_off_max_tol))
```

- **argmin:** Finds the closest index where the detected frequency matches the tolerance bounds.
- The same logic applies to led_1, led_2, and led_off.

```
for i in range(led_0_f_min_index, led_0_f_max_index):
    A = np.floor(analyser.get_spectrum()[i]).astype(int)
    if A == -125:
        print('LED_0 is on')
        base.leds[0].on()
        base.leds[1].off()
        base.leds[2].off()

for i in range(led_1_f_min_index, led_1_f_max_index):
    B = np.floor(analyser.get_spectrum()[i]).astype(int)
    if B == -125:
        print('LED_1 is on')
        base.leds[0].off()
        base.leds[1].on()
        base.leds[2].off()

for i in range(led_2_f_min_index, led_2_f_max_index):
    C = np.floor(analyser.get_spectrum()[i]).astype(int)
    if C == -125:
        print('LED_2 is on')
        base.leds[0].off()
        base.leds[1].off()
        base.leds[2].on()

for i in range(led_off_f_min_index, led_off_f_max_index):
    D = np.floor(analyser.get_spectrum()[i]).astype(int)
    if D == -125:
        print('LED_all is off')
        base.leds[0].off()
        base.leds[1].off()
        base.leds[2].off()
```

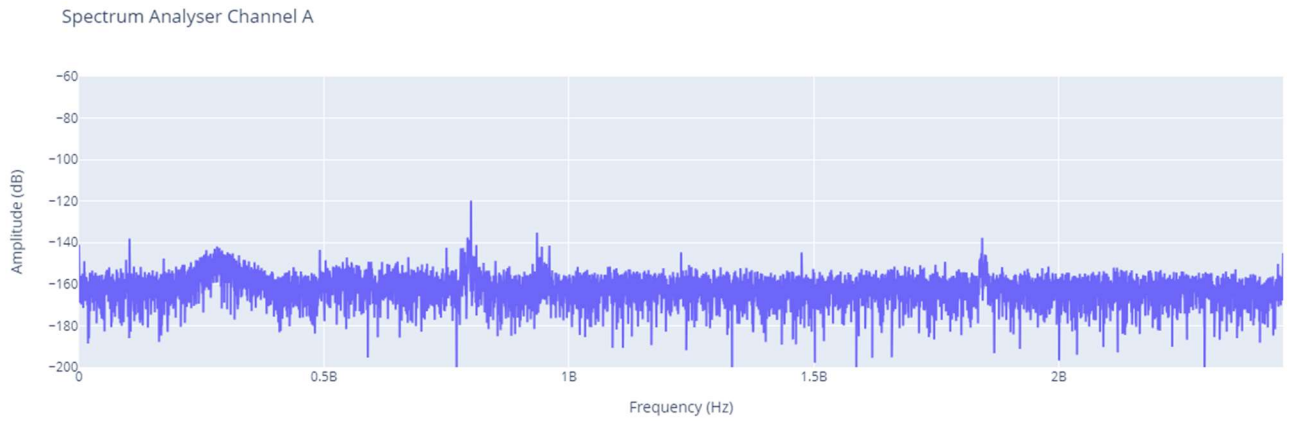
- If the signal strength at the detected frequency is strong (-125 dB), the corresponding LED is activated.

2.6. Output of Receiver Code of LED Control

2.6.1. Interactive Spectrum Plot

2.6.1.1. Before Detection

- The spectrum analyzer continuously updates the RF spectrum plot within the while loop.
- The plot displays a real-time graph of signal strength (dB) versus frequency (GHz).
- The frequency peaks indicate detected RF signals.



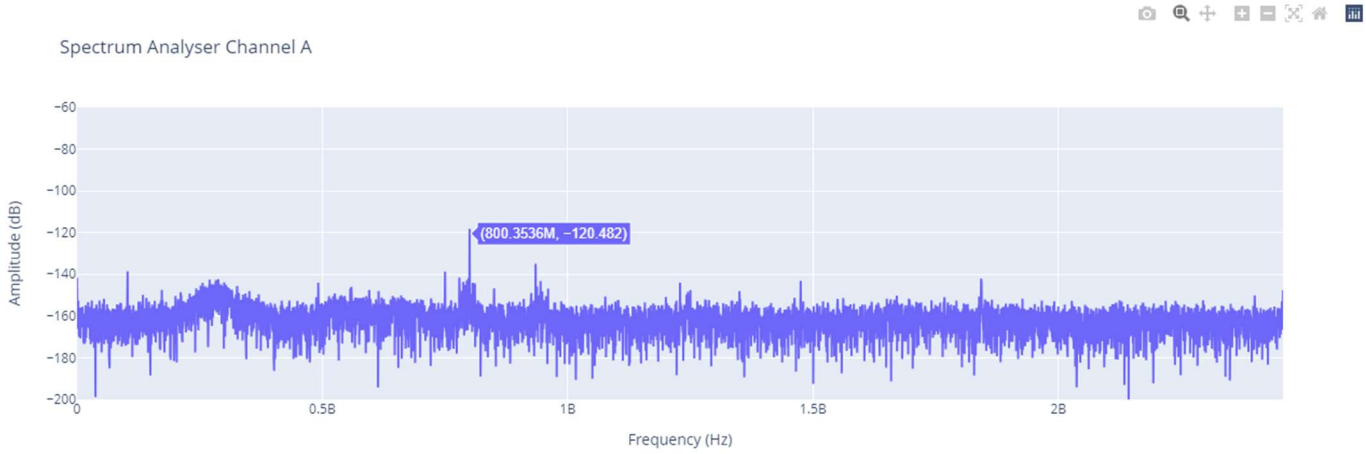
- the spectrum plot shows no significant peaks (background noise).

2.6.1.2. Visualization

- When a transmitted frequency (e.g., 0.8 GHz) is detected, a peak appears on the plot.
- The corresponding LED action is triggered based on this detection.

2.6.2. Frequency: 0.8 GHz

2.6.2.1. Spectrum Plot



- A peak at 0.8 GHz is visible on the plot.

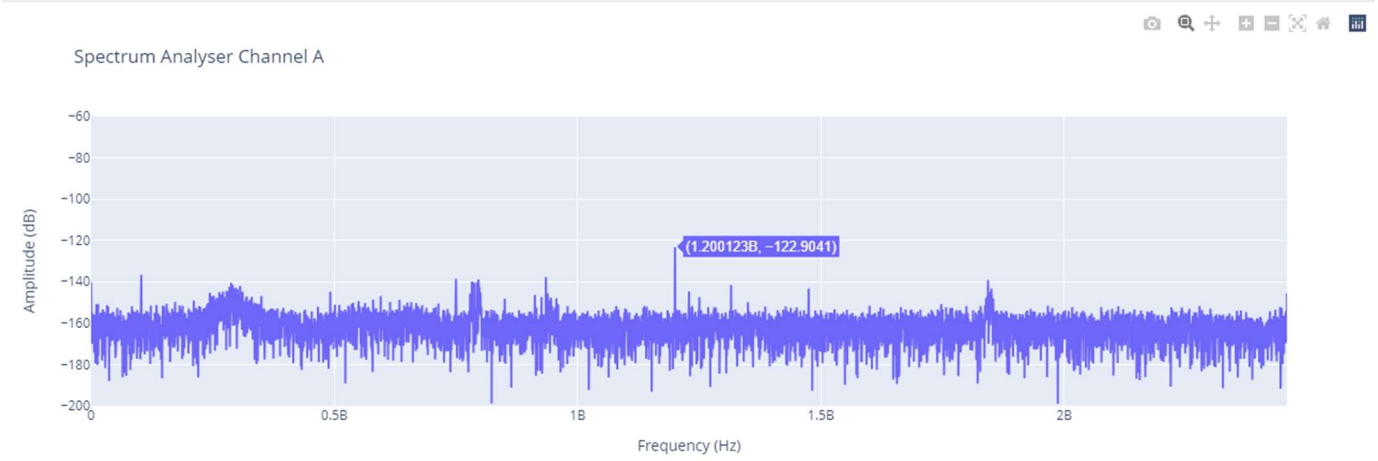
- LED 0 is ON.

2.6.2.2. Output Message

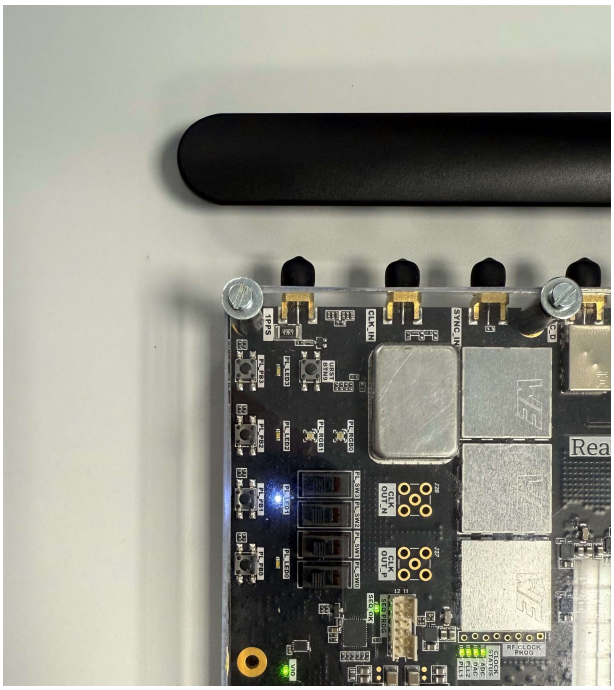
```
LED_0 is on
LED_0 is on
LED_0 is on
LED_0 is on
LED_0 is on
```

2.6.3. Frequency: 1.2 GHz

2.6.3.1. Spectrum Plot



- A peak at 1.2 GHz is visible.



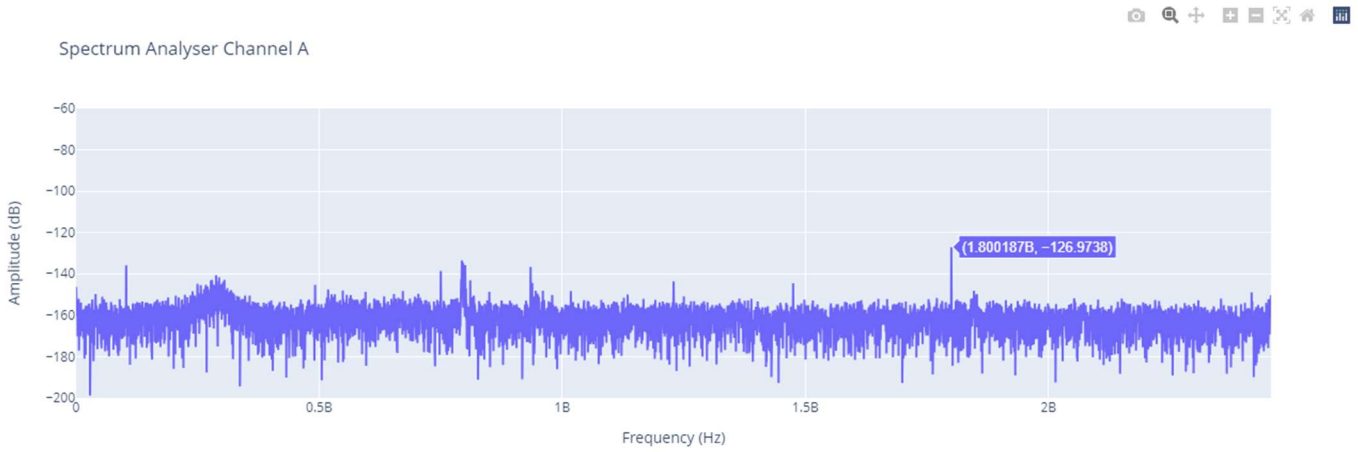
- LED 1 is ON.

2.6.3.2. Output Message

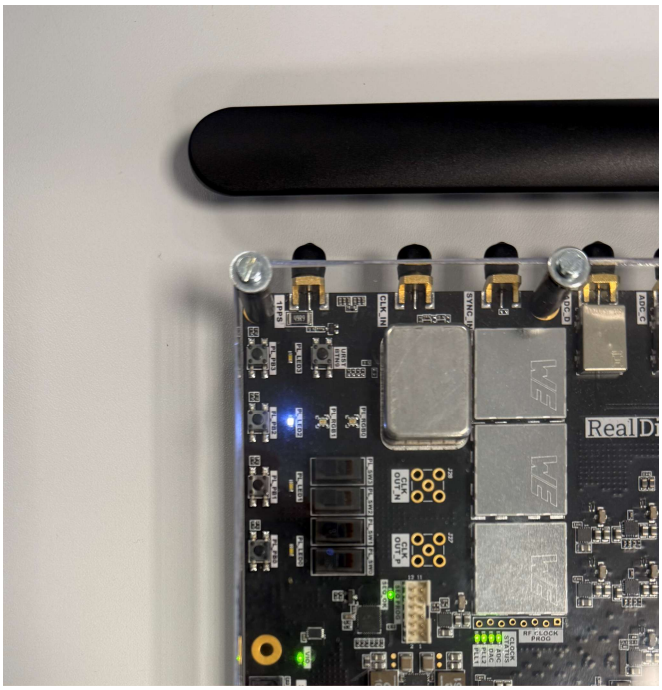
```
LED_1 is on
LED_1 is on
LED_1 is on
LED_1 is on
LED_1 is on
```

2.6.4.Frequency: 1.8 GHz

2.6.4.1. Spectrum Plot



- A peak at 1.8 GHz is visible.



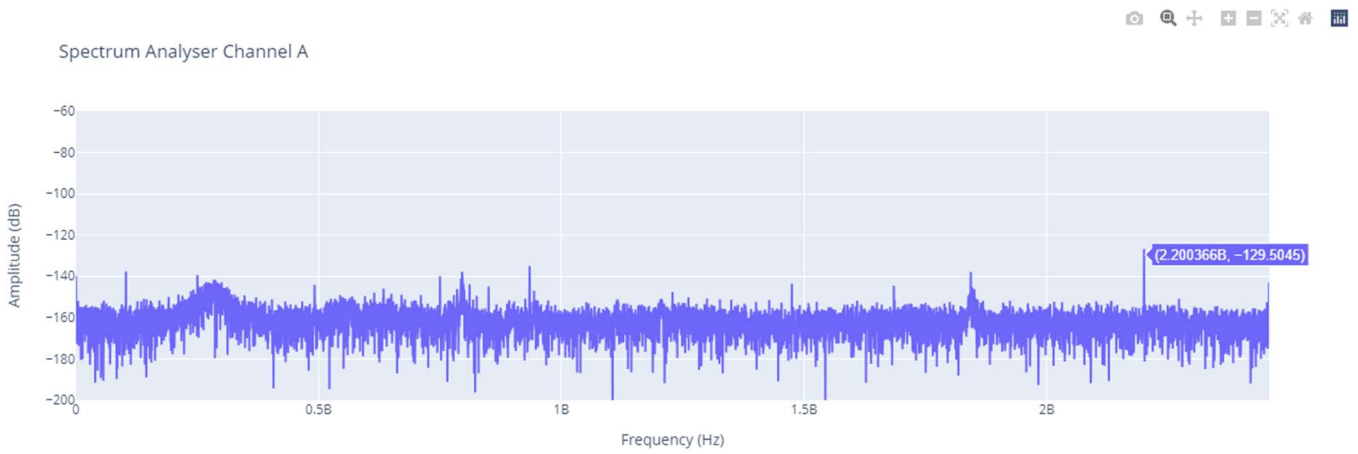
- LED 2 is ON.

2.6.4.2. Output Message

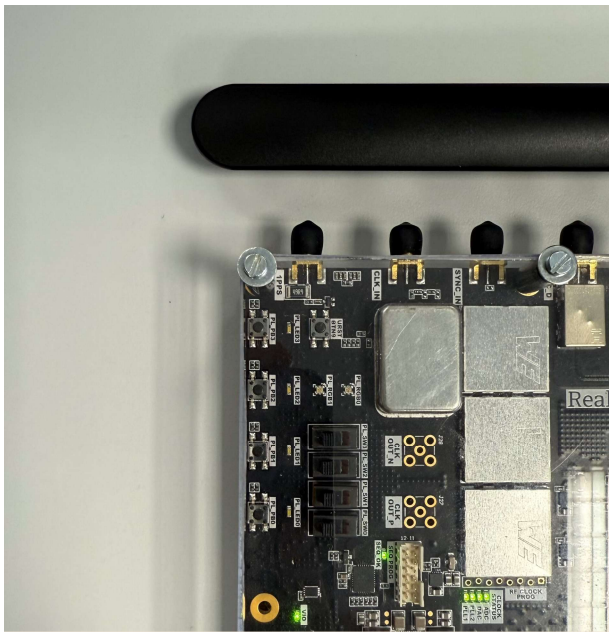
```
LED_2 is on
LED_2 is on
LED_2 is on
LED_2 is on
LED_2 is on
```

2.6.5. Frequency: 2.2 GHz

2.6.5.1. Spectrum Plot



- A peak at 2.2 GHz is visible.



- All LEDs are OFF.

2.6.5.2. Output Message

```
LED_all is off
LED_all is off
LED_all is off
LED_all is off
LED_all is off
```

3 Transmission and Receiver for RGB Control

3.1. Hardware Photo

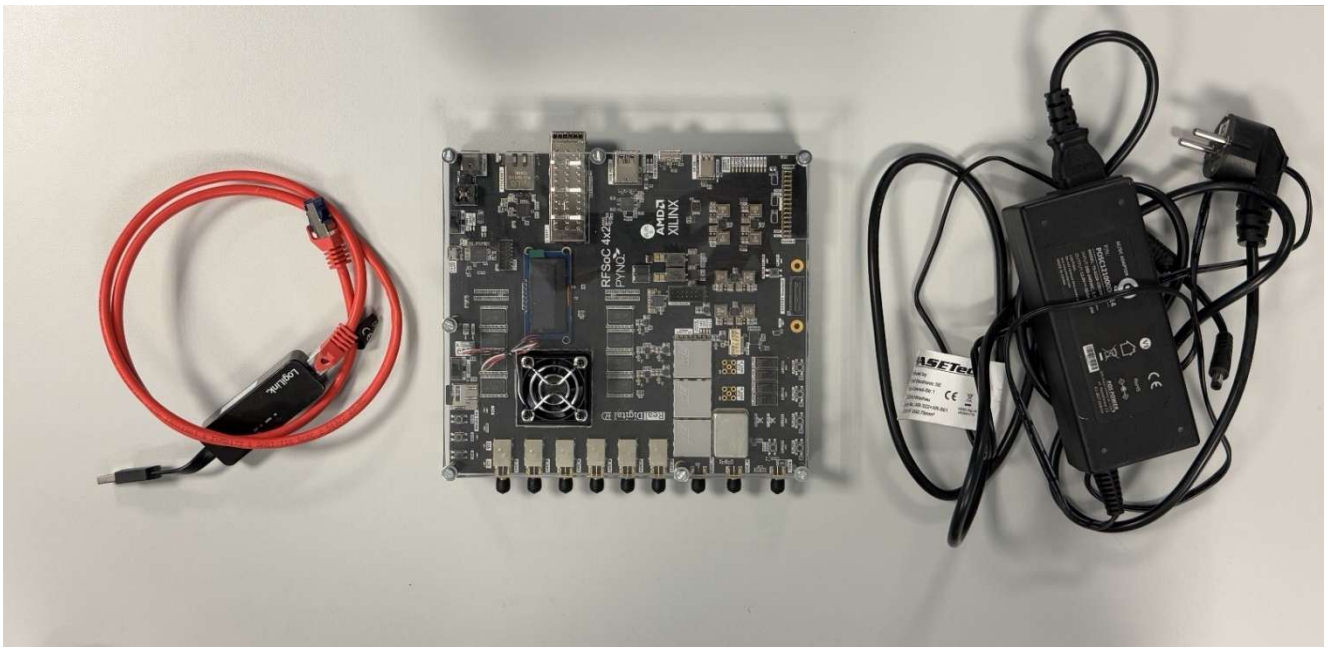


Figure 3.1: The RFSoC boards (transmitter and receiver) with their power supplies and Ethernet connections.

Figure 3.2: two antennas used for transmission and reception.

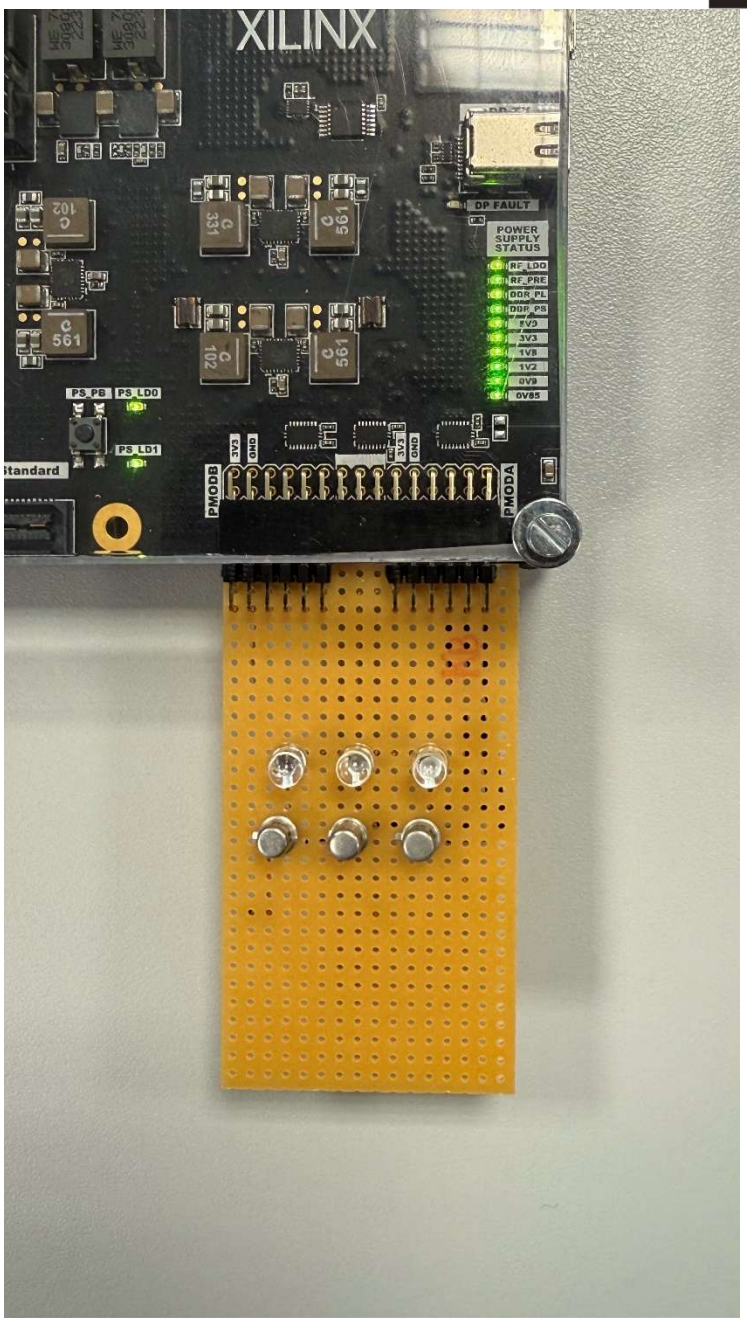


Figure 3.3: The PMOD circuit (PCB-mounted) connected to the receiver board, including the transistors, LEDs, and PMOD interface.

3.2. Hardware Setup

3.2.1. Transmitter Setup

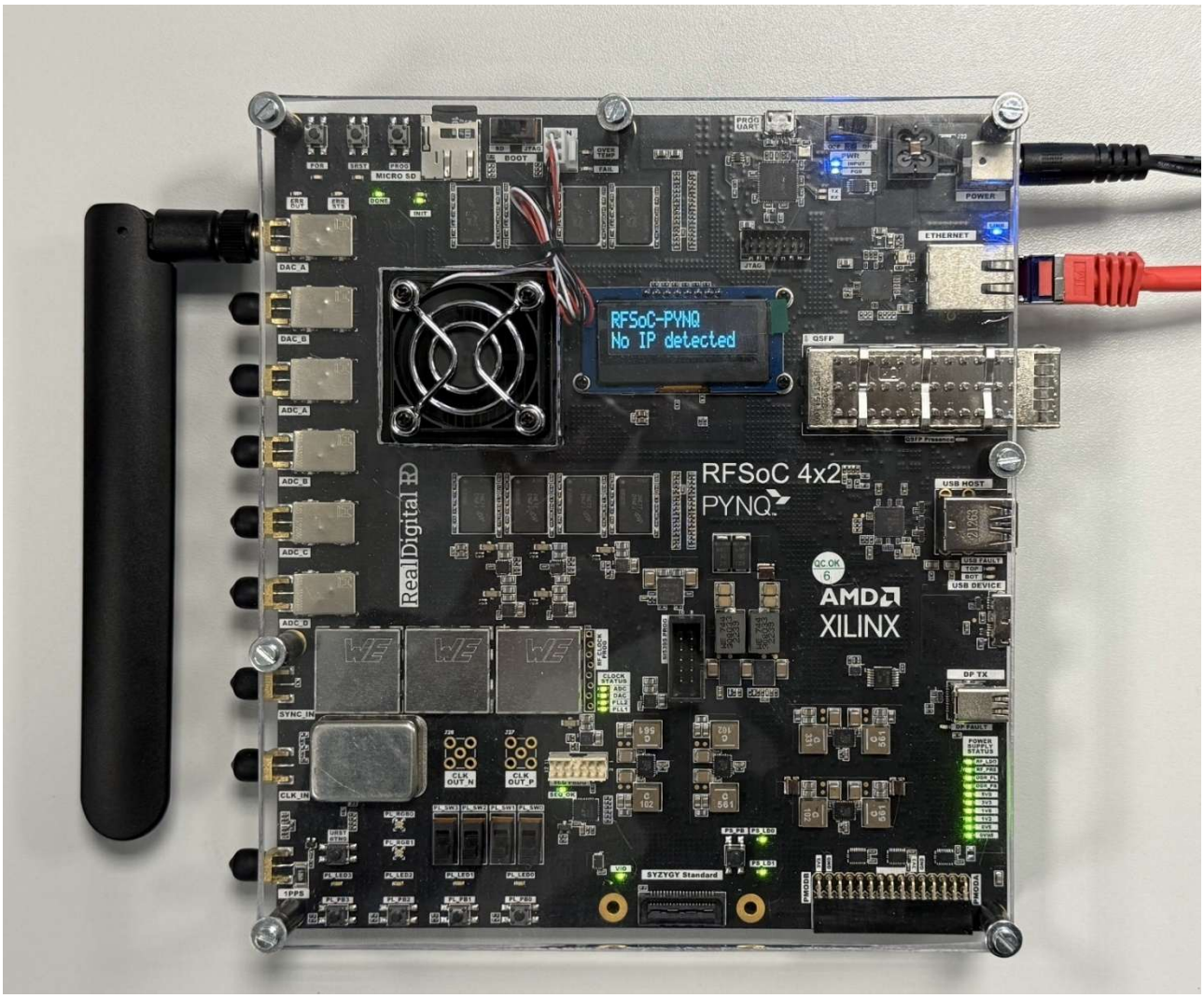


Figure 3.4: The Photo of Transmitter Setup

Explanation

- **Board Configuration:**
  - The transmitter RFSoC board is connected to a PC using an Ethernet cable for control through JupyterLab (accessed via http://192.168.2.99:9090/lab with the password xilinx).
  - Built-in buttons on the board are allows users to select specific RF signal frequencies.
- **Power Supply:**
  - The board is powered via a 12V adapter, ensuring stable operation.
- **Antenna Connection:**
  - An external antenna is connected to DAC_A for wireless signal transmission.

3.2.2.Receiver Setup

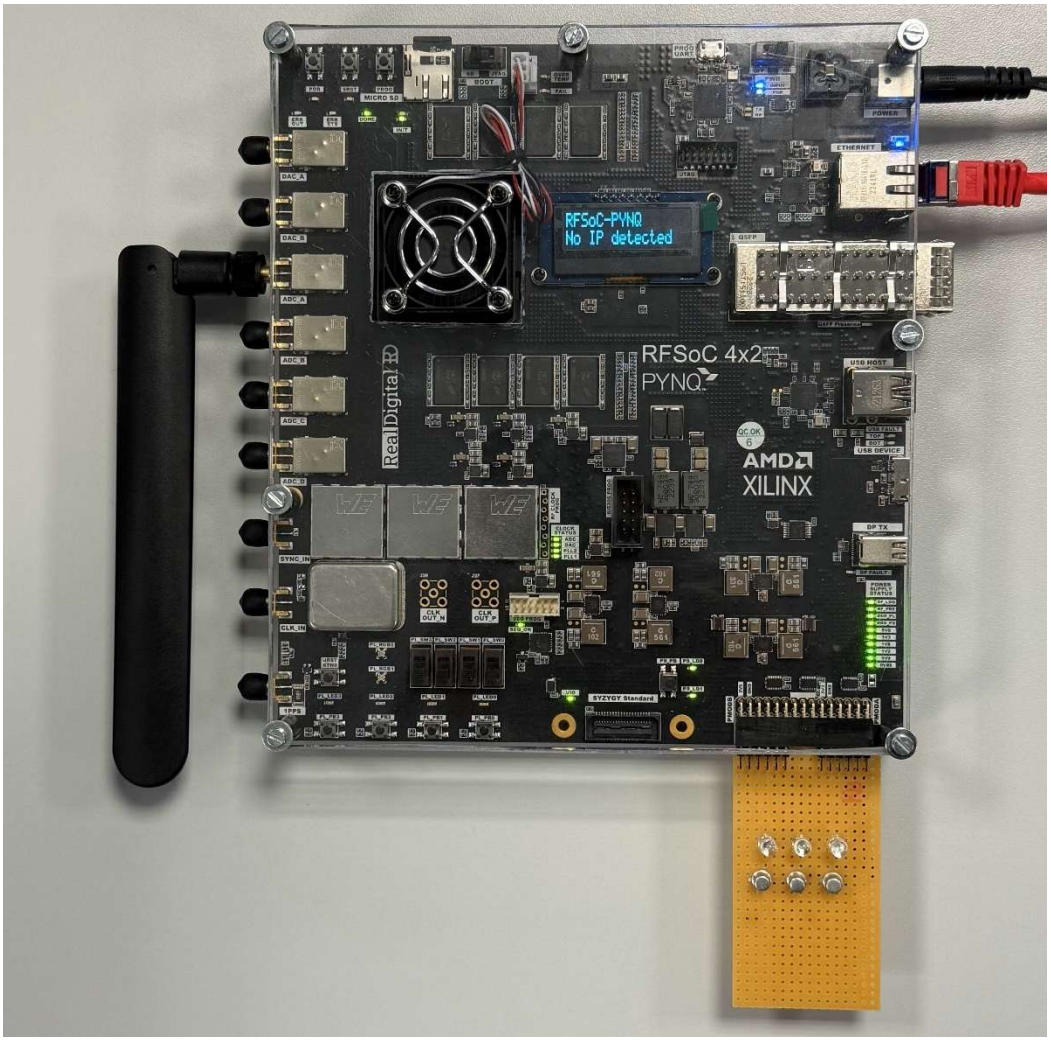


Figure 3.5: The RFSoC receiver board with its antenna, power supply, Ethernet connection, and connected PMOD circuit.

Explanation

- **Board Configuration:**
  - The receiver RFSoC board is connected to a PC via Ethernet to monitor the spectrum and control the RGB LEDs through JupyterLab.
  - A PMOD circuit with three transistors (2N2222) is used to control the Red, Green, and Blue LEDs based on the detected RF signals.
- **Power Supply:**
  - A 12V adapter powers the receiver board.
- **Antenna Connection:**
  - An external antenna is connected to ADC_A for wireless signal reception.

3.3. PMOD Circuit

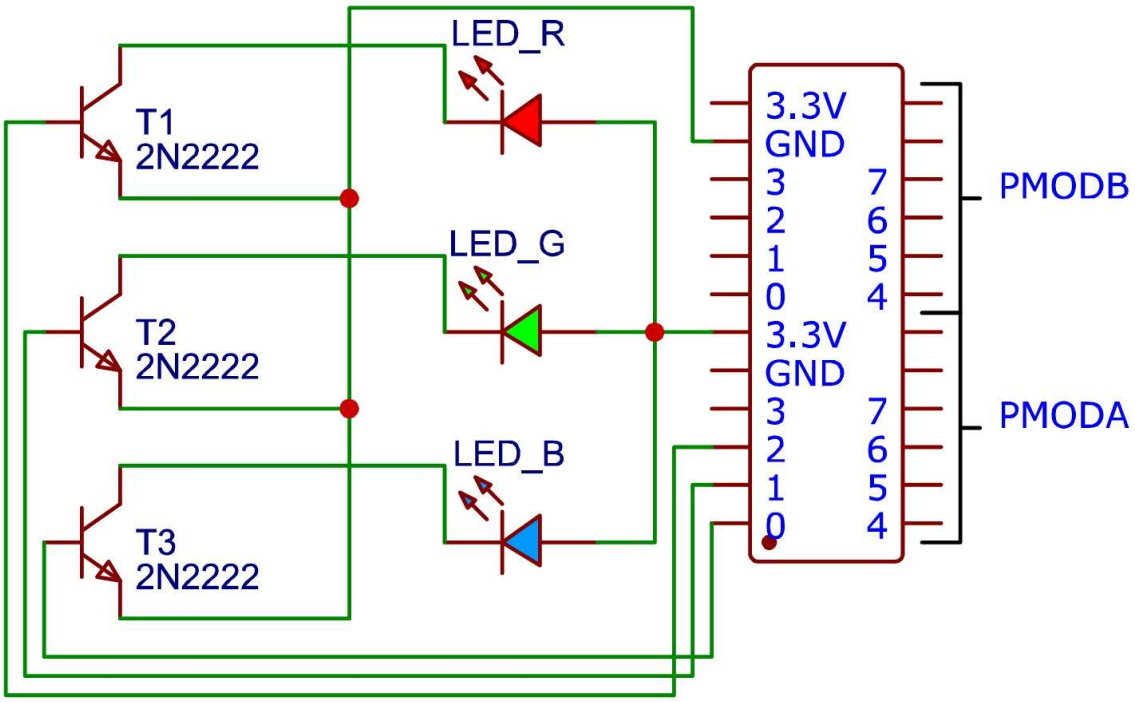


Figure 3.6: PMOD circuit Diagram

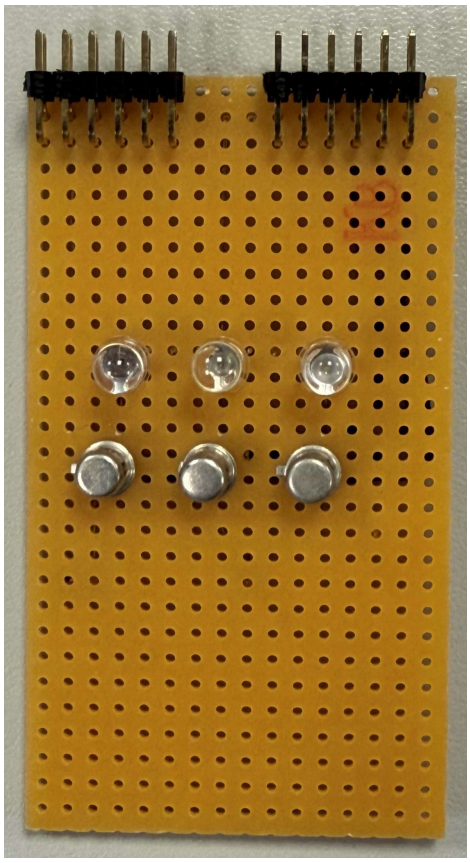


Figure 3.7: PMOD circuit.

Description

- **Components:**
  - **T1, T2, T3 (2N2222 Transistors):** These transistors act as switches to control the Red, Green, and Blue LEDs.
  - **LED_R, LED_G, LED_B:** Red, Green, and Blue LEDs connected to the collector terminals of the transistors.

3.3.1. PMOD Connection

- The PMOD circuit is connected to the following PMOD pins:
  - **Pin 0 of PMODA:** Controls the Blue LED.
  - **Pin 1 of PMODA:** Controls the Green LED.
  - **Pin 2 of PMODA:** Controls the Red LED.
  - **Pins 5 of PMODA :** Provide 3.3V power.
  - **Pins 4 of PMODB:** Provide ground.

3.3.2.Circuit Working

1. **Signal Control:**
   - Signals received from the PMOD pins activate the base of the respective transistor (T1, T2, T3), enabling current flow and lighting up the corresponding LED.
2. **Frequency Mapping:**
   - RF frequencies detected by the receiver board map to specific PMOD signals to control the LEDs:
     - **0.8 GHz:** Activates the Blue LED.
     - **1.2 GHz:** Activates the Green LED.
     - **1.8 GHz:** Activates the Red LED.
     - **2.2 GHz:** Turns off all LEDs.
3. **Current Regulation:**
   - The transistors ensure safe operation of the LEDs by regulating current flow.

3.4. Transmitter Code for RGB Control

3.4.1. Code Overview

[1]:
```
from pynq.overlays.base import BaseOverlay
from time import sleep

print('wait')

base = BaseOverlay('base.bit')
base.init_rf_clks()

def set_transmitter_channel(channel, enable, gain, frequency):
    channel.control.enable = enable
    channel.control.gain = gain
    channel.dac_block.MixerSettings['Freq'] = frequency

print('press now')

while True:
    if base.buttons[0].read():
        set_transmitter_channel(base.radio.transmitter.channel[1], True, 1, 800)
        print("Transmitting 0.8 GHz signal to turn ON Blue LED")
        sleep(0.2)

    elif base.buttons[1].read():
        set_transmitter_channel(base.radio.transmitter.channel[1], True, 1, 1200)
        print("Transmitting 1.2 GHz signal to turn ON Green LED")
        sleep(0.2)

    elif base.buttons[2].read():
        set_transmitter_channel(base.radio.transmitter.channel[1], True, 1, 1800)
        print("Transmitting 1.8 GHz signal to turn ON Red LED")
        sleep(0.2)

    elif base.buttons[3].read():
        set_transmitter_channel(base.radio.transmitter.channel[1], True, 1, 2200)
        print("Transmitting 2.2 GHz signal to turn OFF all LED")
        sleep(0.2)
```

Figure 3.8: Transmitter Code for RGB Control

This code configures the RFSoC board to transmit RF signals at specific frequencies when buttons are pressed. These frequencies correspond to controlling the RGB LEDs on the receiver.

3.4.2.Code Explanation

3.4.2.1. Imports and Initialization

```
from pynq.overlays.base import BaseOverlay
from time import sleep
```

- **pynq.overlays.base:** Provides access to the base overlay for RFSoC, enabling control of buttons, LEDs, and RF modules.
- **time.sleep:** Used to introduce delays between transmissions to ensure smooth operation.

```
base = BaseOverlay('base.bit')
base.init_rf_clks()
```

- **BaseOverlay('base.bit'):** Loads the base bitstream required to configure the RFSoC hardware.
- **base.init_rf_clks():** Initializes RF clocks, preparing the board for RF operations.

3.4.2.2. Function to Configure Transmitter

This function sets the transmitter's parameters:

```
def set_transmitter_channel(channel, enable, gain, frequency):
    channel.control.enable = enable
    channel.control.gain = gain
    channel.dac_block.MixerSettings['Freq'] = frequency
```

- **enable:** Activates or deactivates the channel.
- **gain:** Adjusts the signal strength.
- **frequency:** Sets the transmission frequency in Hz.

3.4.2.3. Main Loop for Button Monitoring

```
while True:
    if base.buttons[0].read():
        set_transmitter_channel(base.radio.transmitter.channel[1], True, 1, 800)
        print("Transmitting 0.8 GHz signal to turn ON Blue LED")
        sleep(0.2)

    elif base.buttons[1].read():
        set_transmitter_channel(base.radio.transmitter.channel[1], True, 1, 1200)
        print("Transmitting 1.2 GHz signal to turn ON Green LED")
        sleep(0.2)

    elif base.buttons[2].read():
        set_transmitter_channel(base.radio.transmitter.channel[1], True, 1, 1800)
        print("Transmitting 1.8 GHz signal to turn ON Red LED")
        sleep(0.2)

    elif base.buttons[3].read():
        set_transmitter_channel(base.radio.transmitter.channel[1], True, 1, 2200)
        print("Transmitting 2.2 GHz signal to turn OFF all LED")
        sleep(0.2)
```

- **while True:** Creates an infinite loop to continuously monitor button states.
- **Button 0: When pressed, it transmits 0.8 GHz.**
  - Action: This signal turns ON LED Blue on the receiver.
- **Button 1: Transmits 1.2 GHz.**
  - Action: Turns ON LED Green.
- **Button 2: Transmits 1.8 GHz.**
  - Action: Turns ON LED Red.
- **Button 3: Transmits 2.2 GHz.**
  - Action: Turns OFF all LEDs.

3.5. Output of Transmitter Code of RGB Control

When the transmitter code is executed, it generates RF signals corresponding to specific button presses. The observed outputs are:

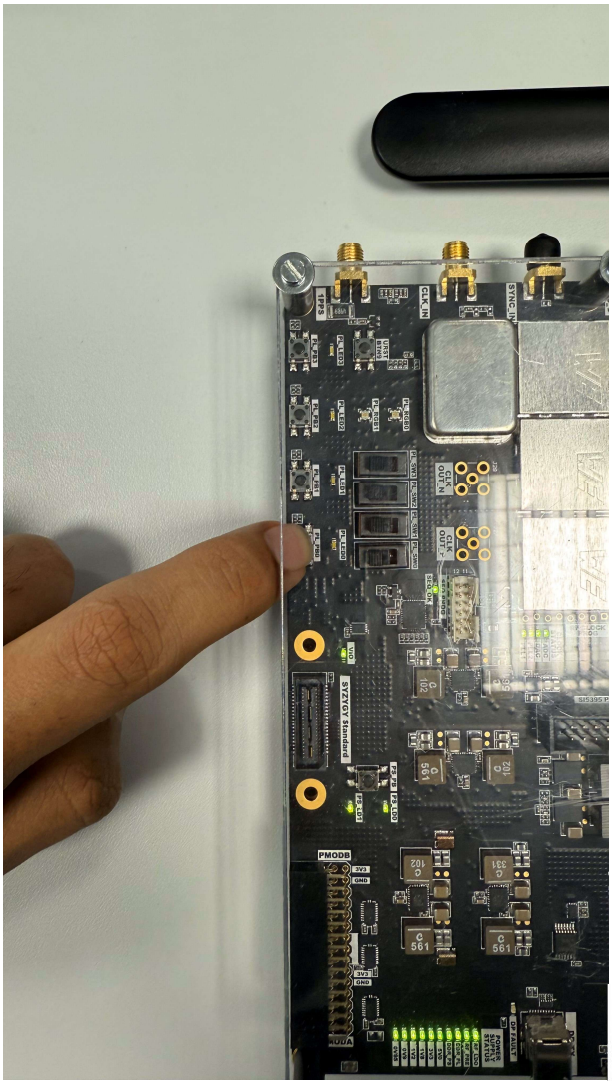
3.5.1. Initialization

```
wait
press now
```

Indicates the board is ready to respond to button presses.

3.5.2. Button Press

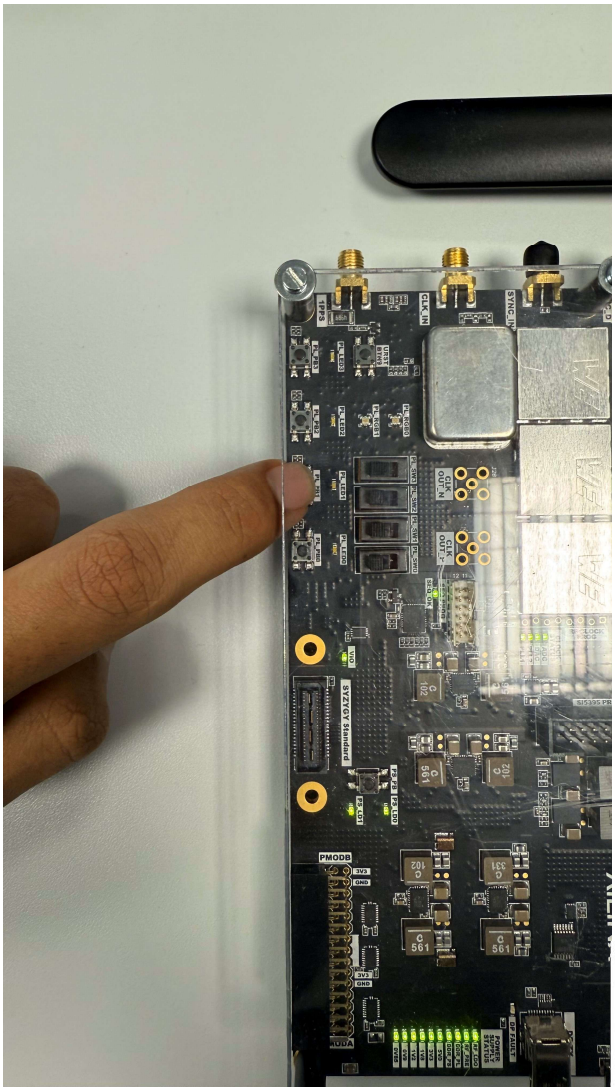
3.5.2.1. When Button 0 is pressed:



```
wait
press now
Transmitting 0.8 GHz signal to turn ON Blue LED
```

The transmitter sends an RF signal at 0.8 GHz.

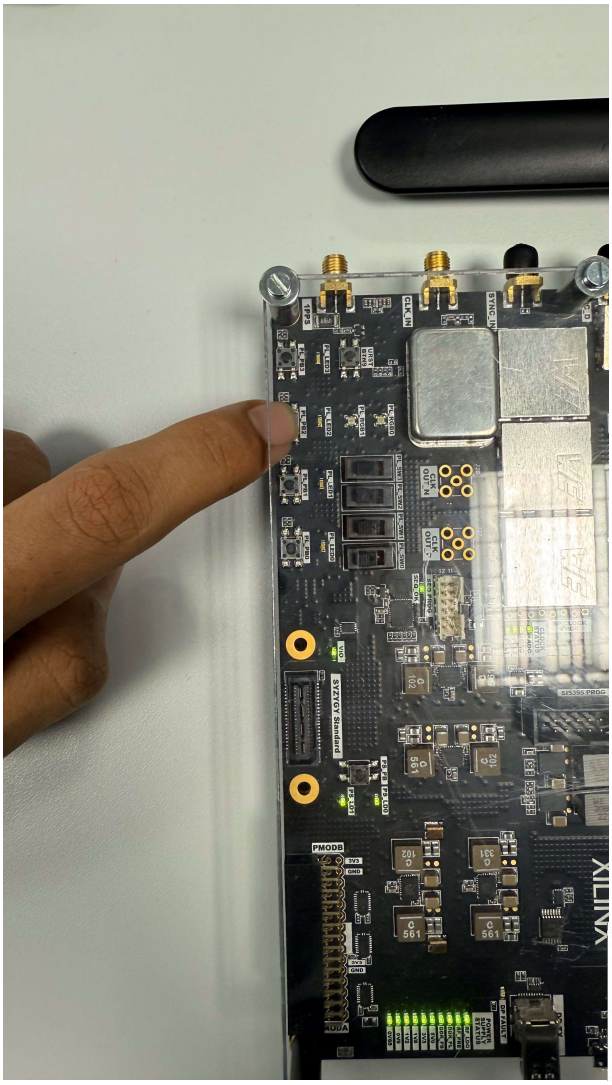
3.5.2.2. When Button 1 is pressed:



```
wait
press now
Transmitting 0.8 GHz signal to turn ON Blue LED
Transmitting 1.2 GHz signal to turn ON Green LED
```

The transmitter sends an RF signal at 1.2 GHz.

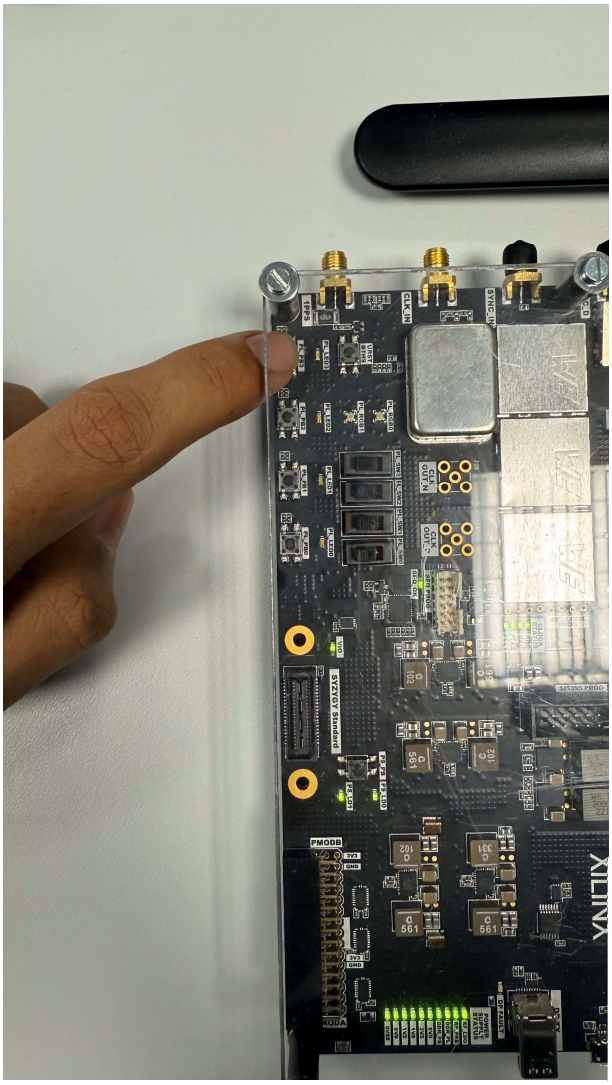
3.5.2.3. When Button 2 is pressed:



```
wait
press now
Transmitting 0.8 GHz signal to turn ON Blue LED
Transmitting 1.2 GHz signal to turn ON Green LED
Transmitting 1.8 GHz signal to turn ON Red LED
```

The transmitter sends an RF signal at 1.8 GHz.

3.5.2.4. When Button 3 is pressed:



```
wait
press now
Transmitting 0.8 GHz signal to turn ON Blue LED
Transmitting 1.2 GHz signal to turn ON Green LED
Transmitting 1.8 GHz signal to turn ON Red LED
Transmitting 2.2 GHz signal to turn OFF all LED
```

The transmitter sends an RF signal at 2.2 GHz.

3.6. Receiver Code for RGB Control

3.6.1. Code Overview

[3]:
```
import time
import numpy as np
import ipywidgets as ipw
from pynq.lib import pmod
from pynq.overlays.base import BaseOverlay
from rfsystem.spectrum_sweep import SpectrumAnalyser

base = BaseOverlay('base.bit')
base.init_rf_clks()

analysers = []
period, duty = 1000, 90
period1, duty1 = 1000, 90
number_samples = 12000
sample_frequency = 2457.6e6


b = pmod.Pmod_IO(base.PMODA, 0, 'out')
g = pmod.Pmod_IO(base.PMODA, 1, 'out')
r = pmod.Pmod_IO(base.PMODA, 2, 'out')

led_b = 0.8e9
led_b_min_tol = led_b/100 * 99.5
led_b_max_tol = led_b/100 * 100.5

led_g = 1.2e9
led_g_min_tol = led_g/100 * 99.5
led_g_max_tol = led_g/100 * 100.5

led_r = 1.8e9
led_r_min_tol = led_r/100 * 99.5
led_r_max_tol = led_r/100 * 100.5

led_off = 2.2e9
led_off_min_tol = led_off/100 * 99.5
led_off_max_tol = led_off/100 * 100.5

analysers.append(SpectrumAnalyser(channel = base.radio.receiver.channel[3],
                                  sample_frequency = sample_frequency,
                                  number_samples = number_samples,
                                  title = ''.join(['Spectrum Analyser Channel A ']), height = None, width = None))

ipw.VBox([analyser.spectrum_plot.get_plot() for analyser in analysers])
```

Figure 3.8: Receiver Code for RGB Control part 1

```
[*]: while True:
    for analyser in analysers:
        analyser.update_spectrum()

    led_b_f_min_index = np.argmin(np.abs(np.array((analyser.spectrum_plot._data.x)) - led_b_min_tol))
    led_b_f_max_index = np.argmin(np.abs(np.array((analyser.spectrum_plot._data.x)) - led_b_max_tol))

    led_g_f_min_index = np.argmin(np.abs(np.array((analyser.spectrum_plot._data.x)) - led_g_min_tol))
    led_g_f_max_index = np.argmin(np.abs(np.array((analyser.spectrum_plot._data.x)) - led_g_max_tol))

    led_r_f_min_index = np.argmin(np.abs(np.array((analyser.spectrum_plot._data.x)) - led_r_min_tol))
    led_r_f_max_index = np.argmin(np.abs(np.array((analyser.spectrum_plot._data.x)) - led_r_max_tol))

    led_off_f_min_index = np.argmin(np.abs(np.array((analyser.spectrum_plot._data.x)) - led_off_min_tol))
    led_off_f_max_index = np.argmin(np.abs(np.array((analyser.spectrum_plot._data.x)) - led_off_max_tol))

    for i in range(led_b_f_min_index, led_b_f_max_index):
        A = np.floor(analyser.get_spectrum()[i]).astype(int)
        if A == -125:
            print('LED_blue is on')
            b.write(1)
            g.write(0)
            r.write(0)

    for i in range(led_g_f_min_index, led_g_f_max_index):
        B = np.floor(analyser.get_spectrum()[i]).astype(int)
        if B == -125:
            print('LED_green is on')
            b.write(0)
            g.write(1)
            r.write(0)

    for i in range(led_r_f_min_index, led_r_f_max_index):
        C = np.floor(analyser.get_spectrum()[i]).astype(int)
        if C == -125:
            print('LED_red is on')
            b.write(0)
            g.write(0)
            r.write(1)

    for i in range(led_off_f_min_index, led_off_f_max_index):
        D = np.floor(analyser.get_spectrum()[i]).astype(int)
        if D == -125:
            print('LED_all is off')
            r.write(0)
            g.write(0)
            b.write(0)
```

Figure 3.9: Receiver Code for RGB Control part 2

This code configures the RFSoC board to monitor the RF spectrum, detect specific frequencies, and activate the RGB LEDs accordingly.

3.6.2. Code Explanation

3.6.2.1. Importing Libraries

```
import time
import numpy as np
import ipywidgets as ipw
from pynq.lib import pmod
from pynq.overlays.base import BaseOverlay
from rfsystem.spectrum_sweep import SpectrumAnalyser
```

- **time:** For delays.
- **numpy:** Used for numerical computations, e.g., spectrum analysis.
- **ipywidgets:** Supports interactive plots in JupyterLab.
- **pmod:** Provides access to the PMOD interface for controlling RGB LEDs.
- **SpectrumAnalyser:** Handles RF spectrum analysis for signal detection.

3.6.2.2. Initialization

```
base = BaseOverlay('base.bit')
base.init_rf_clks()
```

- **BaseOverlay('base.bit'):** Loads the base overlay.
- **base.init_rf_clks():** Initializes RF clocks.

3.6.2.3. PMOD Configuration

```
b = pmod.Pmod_IO(base.PMODA, 0, 'out')
g = pmod.Pmod_IO(base.PMODA, 1, 'out')
r = pmod.Pmod_IO(base.PMODA, 2, 'out')
```

Configures PMOD pins to control:

- **Pin 0: Blue LED.**
- **Pin 1: Green LED.**
- **Pin 2: Red LED.**

3.6.2.4. Frequency Definitions

```
led_b = 0.8e9
led_b_min_tol = led_b/100 * 99.5
led_b_max_tol = led_b/100 * 100.5

led_g = 1.2e9
led_g_min_tol = led_g/100 * 99.5
led_g_max_tol = led_g/100 * 100.5

led_r = 1.8e9
led_r_min_tol = led_r/100 * 99.5
led_r_max_tol = led_r/100 * 100.5

led_off = 2.2e9
led_off_min_tol = led_off/100 * 99.5
led_off_max_tol = led_off/100 * 100.5
```

- **led_b:** Frequency for Blue LED (0.8 GHz).
- **led_b_min_tol / led_b_max_tol:** Define ±0.10% tolerance for signal detection.
- **Similarly, the code defines frequencies and tolerances for Green (1.2 GHz), Red (1.8 GHz), and OFF (2.2 GHz).**

3.6.2.5. Spectrum Analysis

```
analysers.append(SpectrumAnalyser(channel = base.radio.receiver.channel[3],
                                  sample_frequency = sample_frequency,
                                  number_samples = number_samples,
                                  title = ''.join(['Spectrum Analyser Channel A ']), height = None, width = None))

ipw.VBox([analyser.spectrum_plot.get_plot() for analyser in analysers])
```

Initializes a spectrum analyzer for RF signal monitoring.

3.6.2.6. Main Loop for Spectrum Monitoring

```
while True:
    for analyser in analysers:
        analyser.update_spectrum()
```

Continuously updates the RF spectrum to detect signals.

3.6.2.7. Frequency Detection and LED Control

```
for i in range(led_b_f_min_index, led_b_f_max_index):
    A = np.floor(analyser.get_spectrum()[i]).astype(int)
    if A == -125:
        print('LED_blue is on')
        b.write(1)
        g.write(0)
        r.write(0)

for i in range(led_g_f_min_index, led_g_f_max_index):
    B = np.floor(analyser.get_spectrum()[i]).astype(int)
    if B == -125:
        print('LED_green is on')
        b.write(0)
        g.write(1)
        r.write(0)

for i in range(led_r_f_min_index, led_r_f_max_index):
    C = np.floor(analyser.get_spectrum()[i]).astype(int)
    if C == -125:
        print('LED_red is on')
        b.write(0)
        g.write(0)
        r.write(1)

for i in range(led_off_f_min_index, led_off_f_max_index):
    D = np.floor(analyser.get_spectrum()[i]).astype(int)
    if D == -125:
        print('LED_all is off')
        r.write(0)
        g.write(0)
        b.write(0)
```

- **Blue LED:** Lights up when 0.8 GHz is detected.
- Similar logic applies for:
  - **Green LED:** Detected at 1.2 GHz.
  - **Red LED:** Detected at 1.8 GHz.
  - **All LEDs OFF:** Detected at 2.2 GHz.

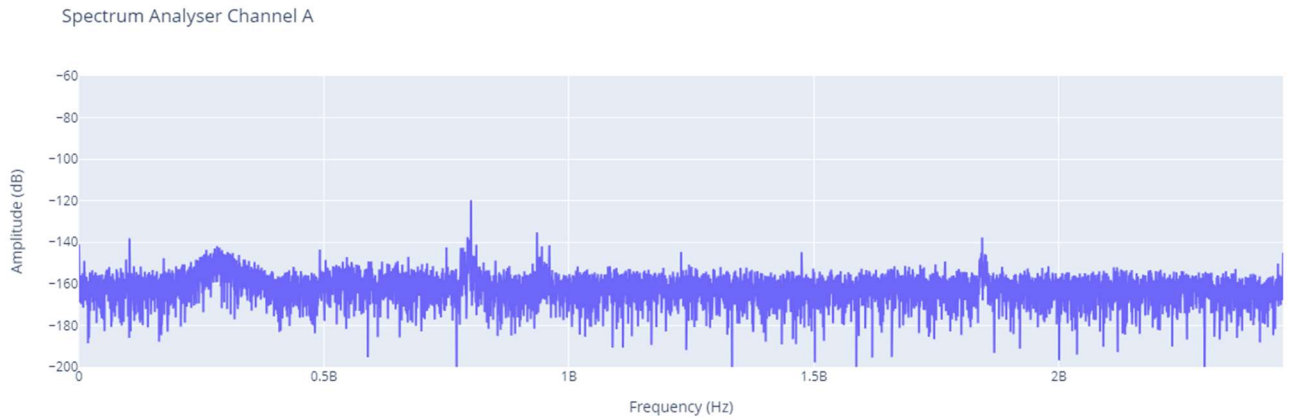
3.7. Output of Receiver Code for RGB Control

The receiver code monitors the RF spectrum for specific frequencies transmitted by the transmitter board. Upon detecting a valid frequency, it activates the corresponding RGB LED on the PMOD circuit. Below is a step-by-step breakdown of the receiver's output:

3.7.1. Interactive Spectrum Plot

3.7.1.1. Before Detection

- The spectrum analyzer continuously updates the RF spectrum plot within the while loop.
- The plot displays a real-time graph of signal strength (dB) versus frequency (GHz).
- The frequency peaks indicate detected RF signals.

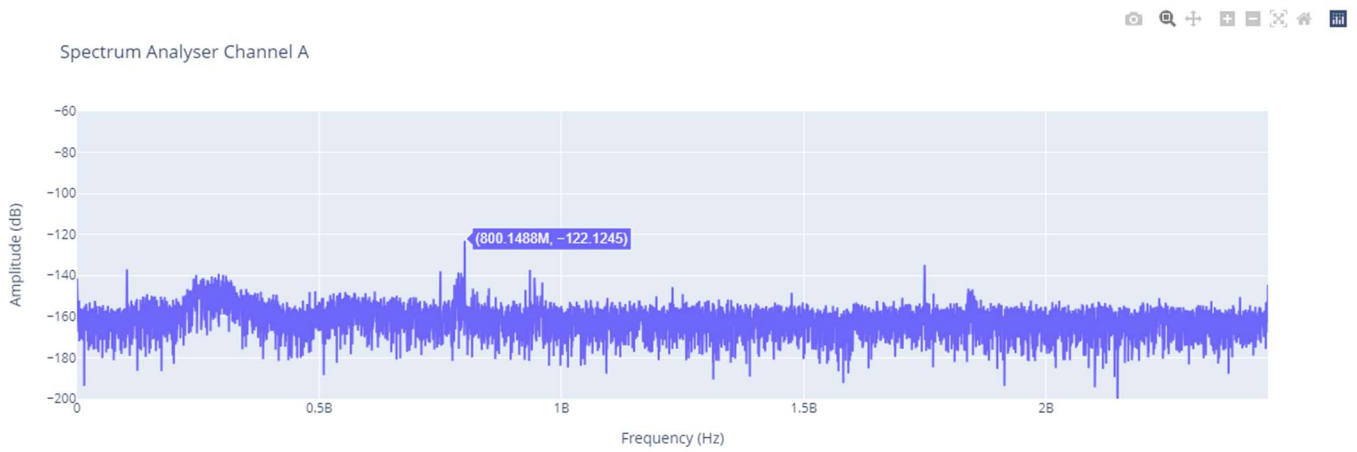


- the spectrum plot shows no significant peaks (background noise).

3.7.2. Frequency Detection and LED Activation

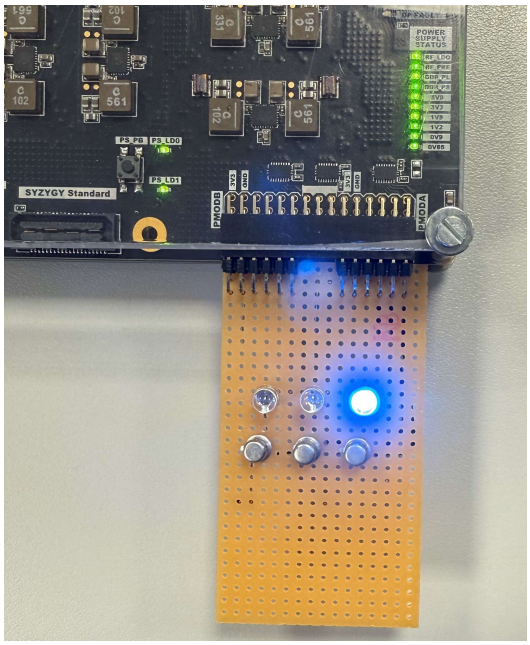
3.7.2.1. Frequency: 0.8 GHz

1. **Detection:**



- The spectrum analyzer identifies a peak at **0.8 GHz** (within ±0.10% tolerance).
- The program maps this frequency to the **Blue LED.**

2. **Actions:**



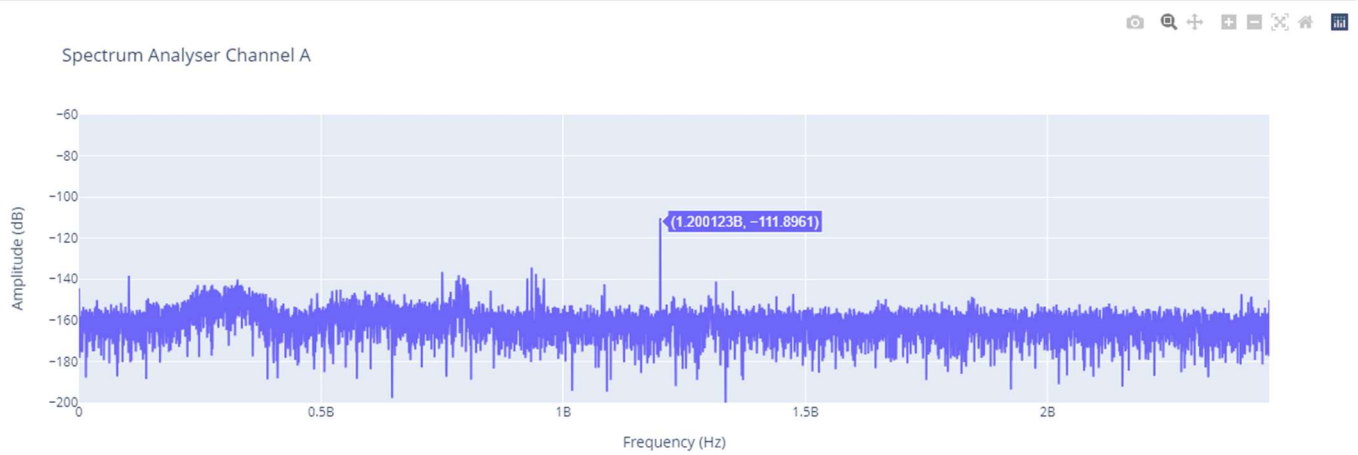
- The PMOD circuit activates the Blue LED (via **PMODA Pin 0**).
- The Green and Red LEDs remain OFF.

3. **JupyterLab Output:**

```
LED_blue is on
LED_blue is on
LED_blue is on
LED_blue is on
LED_blue is on
```

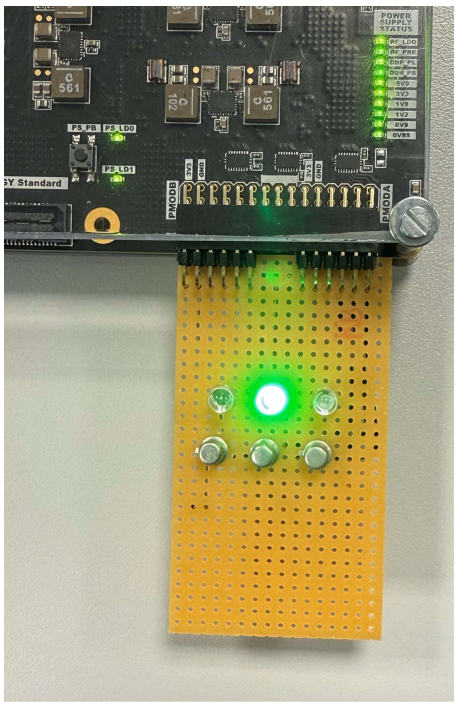
3.7.2.2. Frequency: 1.2 GHz

1. **Detection:**



- The spectrum analyzer identifies a peak at **1.2 GHz** (within ±0.5% tolerance).
- The program maps this frequency to the **Green LED**.

2. **Actions:**



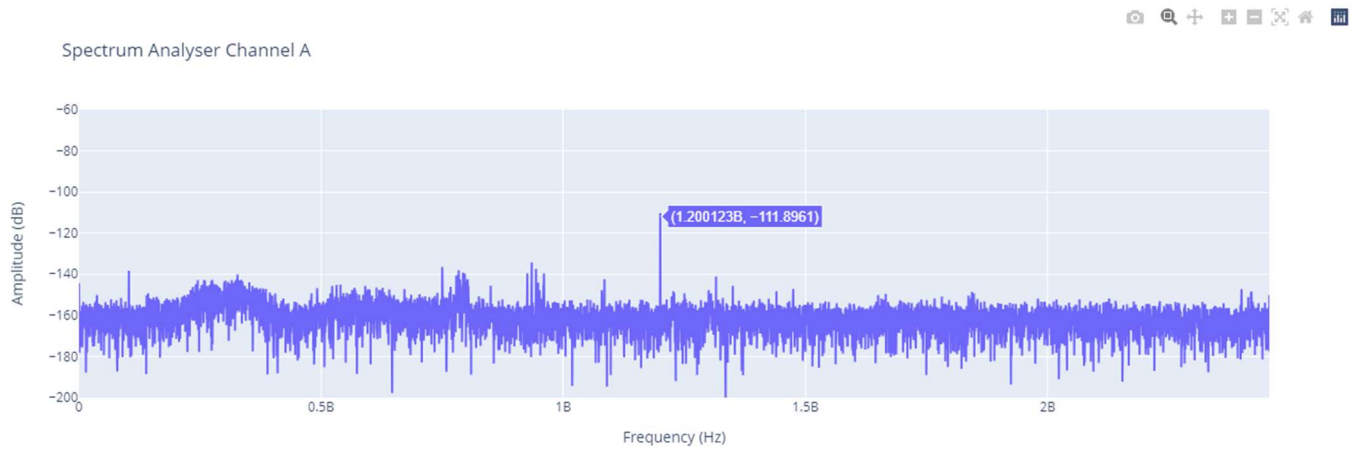
- The PMOD circuit activates the Green LED (via **PMODA Pin 1**).
- The Blue and Red LEDs remain OFF.

3. **JupyterLab Output:**

```
LED_green is on
LED_green is on
LED_green is on
LED_green is on
LED_green is on
```

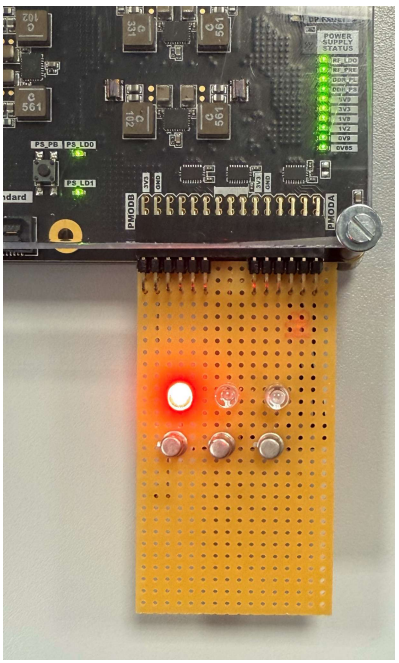
3.7.2.3. Frequency: 1.8 GHz

1. **Detection:**



   - The spectrum analyzer identifies a peak at **1.8 GHz** (within ±0.5% tolerance).
   - The program maps this frequency to the **Red LED.**

2. **Actions:**



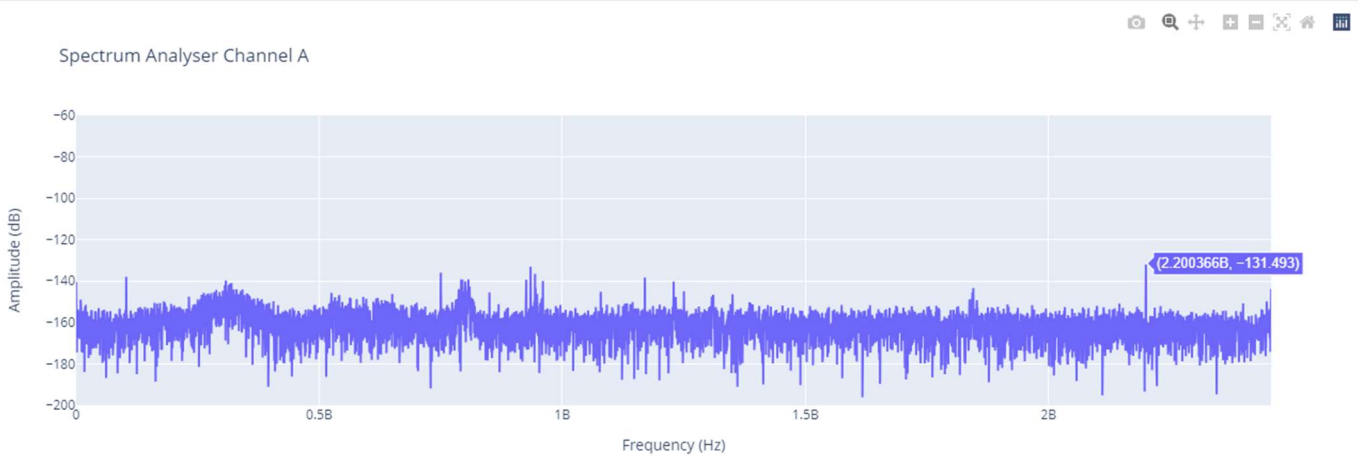
   - The PMOD circuit activates the Red LED (via **PMODA Pin 2**).
   - The Blue and Green LEDs remain OFF.

3. **JupyterLab Output:**

```
LED_red is on
LED_red is on
LED_red is on
LED_red is on
LED_red is on
```

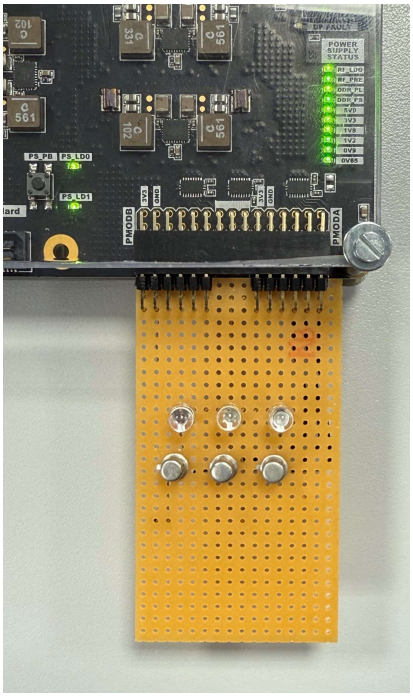
3.7.2.4. Frequency: 2.2 GHz

1. **Detection:**



   - The spectrum analyzer identifies a peak at **2.2 GHz** (within ±0.5% tolerance).
   - The program maps this frequency to the **OFF** state.

2. **Actions:**



   - The PMOD circuit deactivates all LEDs.
   - The Blue, Green, and Red LEDs turn OFF.

3. **JupyterLab Output:**

```
LED_all is off
LED_all is off
LED_all is off
LED_all is off
LED_all is off
```

4 Challenges and Solutions

4.1. Signal Interference

- **Challenges:**
  - Interference caused by nearby RF signals or environmental noise, affecting the reliability of LED and PMOD responses.
  - Limited precision in frequency tuning could lead to overlapping frequencies triggering incorrect LEDs.
- **Solutions:**
  - Antenna **adjustment** can improve the performance of signal transmission and reception
  - Implement robust **frequency tolerance ranges** in the receiver code
  - Use a **spectrum analyzer** to identify noisy frequency bands and avoid interference

4.2. RFSoC Resource Utilization

- **Challenge:**
  - Synchronization between the transmitter and receiver, especially in scenarios where LEDs and PMOD channels respond to frequency-based inputs.
  - Mapping the DAC and ADC channels to ensure correct signal transmission and reception.
  - Python integration issues during the control of hardware overlays and signal processing workflows.
- **Solution:**
  - Incorporate Python debugging techniques like detailed logs, breakpoints, and modularized code for isolating issues in transmitter-receiver communication.
  - Using documentation and a trial-and-error method to map the DAC and ADC channels for accurate signal transmission and reception.
  - Simplify code structures for modular debugging, ensuring ease of maintenance and scalability for future iterations.

4.3. Software Debugging

- **Challenge:**
  - Highlight debugging issues, such as:
    - Synchronization between transmitter and receiver.
    - Errors in RF signal generation and spectrum interpretation.
    - Python code-related issues in integration.
- **Solution:**
  - **Steps taken to debug:**
    - Using breakpoints and logs in Python.
    - Leveraging spectrum analyzer for real-time feedback.
    - Simplifying and modularizing code for easier debugging.

4.4. Signal Monitoring and Calibration

- **Challenge:**
  - Ensuring accurate real-time monitoring and calibration of PMOD output signals, including amplitude and frequency.
- **Solution:**
  - Used the Analog Discovery 2 Oscilloscope for precise signal measurement and iterative adjustments, enabling accurate DAC, ADC channel and PMOD mapping for reliable signal transmission and reception.

5 Applications

5.1. IoT Devices

5.1.1. Description

The project demonstrates the potential of RFSoC for enabling Internet of Things (IoT) devices. The ability to control LEDs wirelessly using RF signals exemplifies the integration of hardware and software for IoT applications. This functionality can extend to smart homes for controlling lighting systems, appliances, and other connected devices or to industrial IoT for automation.

5.1.2. Key Features and Use Cases

- **Smart Homes:** RF signals can trigger devices like smart bulbs or door locks.
- **Industrial IoT:** Use in assembly lines for remote monitoring and control.
- **Flexibility:** Ability to configure frequency settings dynamically to suit different devices.

5.1.3. Implementation from Project

- The transmitter code sends specific RF signals that are received and interpreted by the receiver code to control LEDs.

5.2. Wireless Sensor Networks

5.2.1. Description

Wireless sensor networks (WSNs) rely on robust communication channels to transmit data or receive control signals. This project's approach demonstrates scalable RF-based communication, suitable for sensor nodes in remote or inaccessible locations.

5.2.2. Key Features and Use Cases

- **Environmental Monitoring:** Sensors transmitting data like temperature or humidity.
- **Agriculture:** Remote irrigation control via RF signals.

5.2.3. Implementation from Project

- The use of RF signals to control LEDs showcases a prototype for wireless communication.
- The receiver identifies transmitted frequencies and maps them to specific actions (e.g., LED activation)

5.3. Educational Platforms for Embedded Systems

5.3.1. Description

This project offers a replicable setup to teach FPGA programming, RF communication, and signal processing. The integration of RFSoC and PYNQ provides a user-friendly environment for students and researchers.

5.3.2. Key Features and Use Cases

- **Academic Use:** Demonstrates hardware-software co-design.
- **Training Modules:** Practical exercises on RF signal generation and detection.
- **Accessible Development:** Python integration simplifies learning.

5.3.3. Implementation from Project

- Students can use the project setup to understand signal transmission and LED control.
- Jupiter notebooks allow easy interfacing and experimentation.

References

1. L.H. Crockett, D. Northcote, R. W. Stewart (Editors), *Software Defined Radio with Zynq UltraScale+ RFSoC*, First Edition, Strathclyde Academic Media, 2023. https://www.rfsocbook.com.
2. RFSoC User Guides. Available online: https://www.xilinx.com/support/documentation.html
3. PYNQ Documentation. Available online: https://pynq.readthedocs.io.
4. Python Libraries: NumPy, Matplotlib, ipywidgets. Documentation available at https://numpy.org and https://matplotlib.org.
5. Circuit Design Software https://easyeda.com/editor
6. Supporting textbooks and articles on RF design and IoT systems from IEEE Xplore and SpringerLink.