FPGA based GPS Receiver

6.111 Fall 2020

Final Project Report

Pedro Sales, Osvy Rodriguez

Table of Contents

Motivation

The Global Positioning System (GPS) is a satellite-based navigation system developed and maintained by the U.S. Department of Defense. GPS delivers worldwide coverage with a constellation of thirty two satellites and counting.

Because of its free, open, and dependable nature, GPS has become an essential element of global information. Nowadays, the GPS navigation system is present in almost every technology from cell-phones, to ships and planes, and even smartwatches that can track our fitness activities. By providing services such as location, navigation, tracking, mapping, and timing; GPS has the ability to boost productivity across the globe and even save human lives. For example, numerous applications such as banking systems, financial markets, and communication networks rely on precise time synchronization. Not to mention, that GPS preserves humans lives by preventing traffic accidents, and assisting search and rescue efforts.

Overview

For our 6.111 final project, we decided to implement an FPGA based GPS receiver to perform real time GPS localization. This involves acquiring the raw antenna signal with a software defined radio, detecting GPS signals, tracking satellites, and finally using the timings to solve for a geographical location. Due to time constraints and some challenges that we faced during the implementation process, the current system uses a Teensy to transmit GPS data downloaded from the internet. Then, the system detects GPS satellites, tracks their signals and decodes the Navigation Data. Finally, the time information contained on the Navigation Data is used to display the day of the week, hour, and minutes.

The project was inspired by the GNSS-SDR project [1]. Existing "Software" GPS receiver implementations using FPGAs have been demonstrated using a mixture of an FPGA and a co-processor [2][3].

Project Goals

Baseline

- Detect satellites in line of sight
- Obtain UTC time from satellite

Desired Functions

- Get satellites information (doppler shift, difference time of arrival, NAV data)
- Track a set of at least four satellites

Stretch goals

- Feed satellites' information into a Teensy to solve for receiver location (x,y.z coordinates)
- Implement location solver module on the FPGA
- Display receiver location on LCD

System Block Diagram

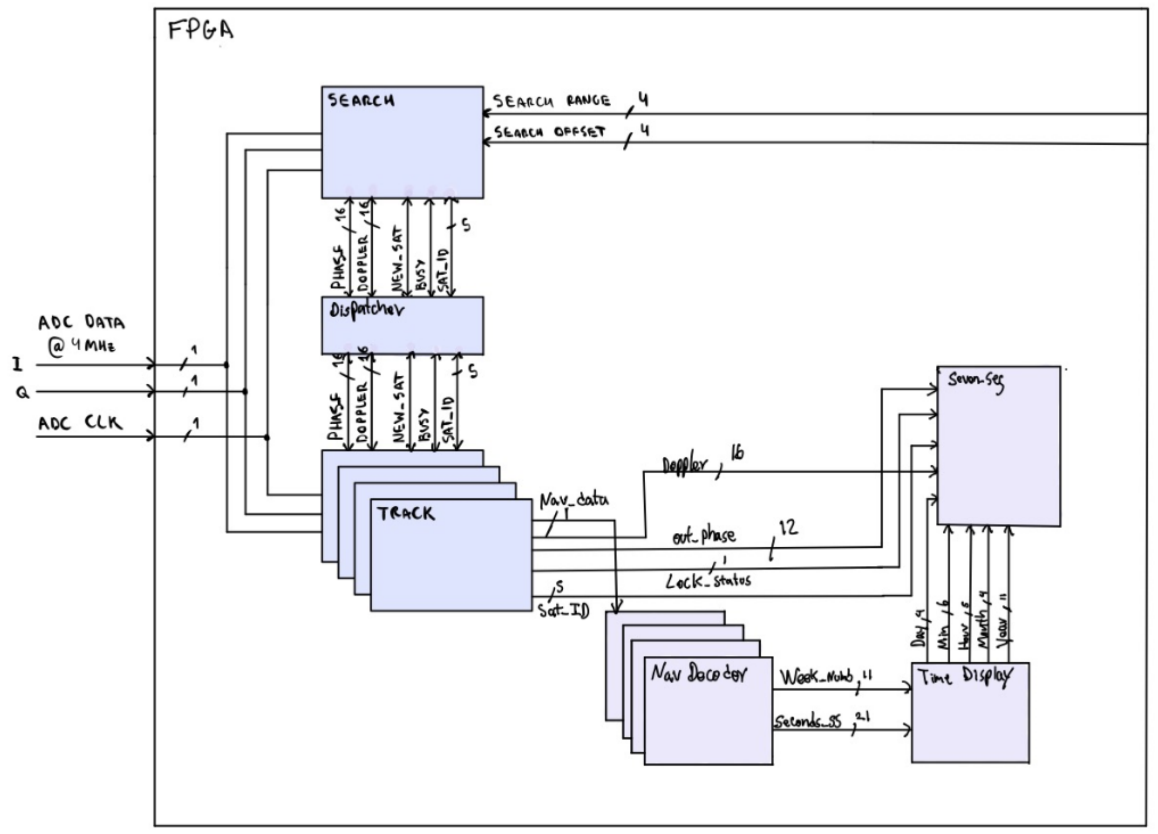


Fig1. Top level Block Diagram

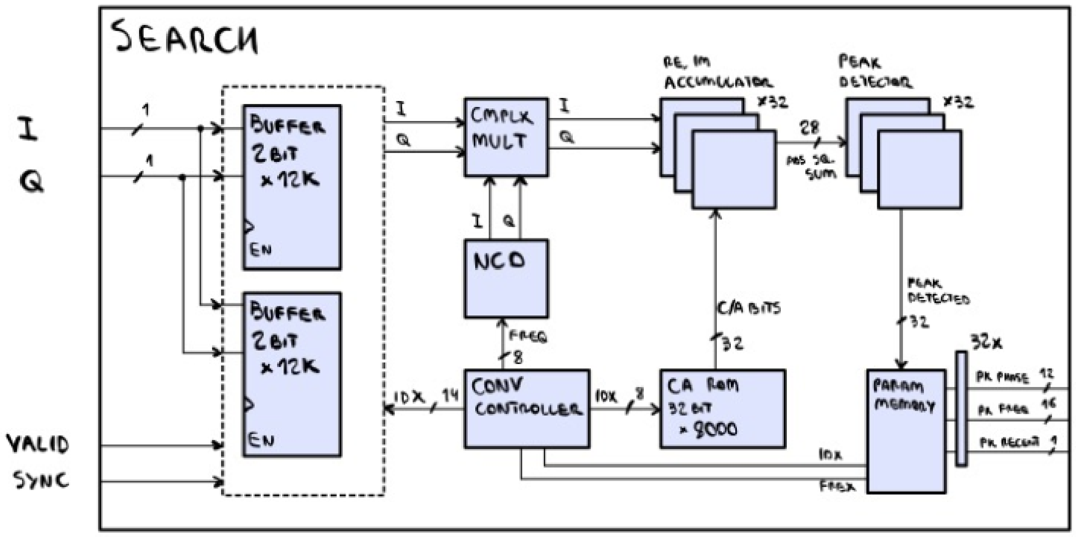


Fig2. Search Module Block Diagram

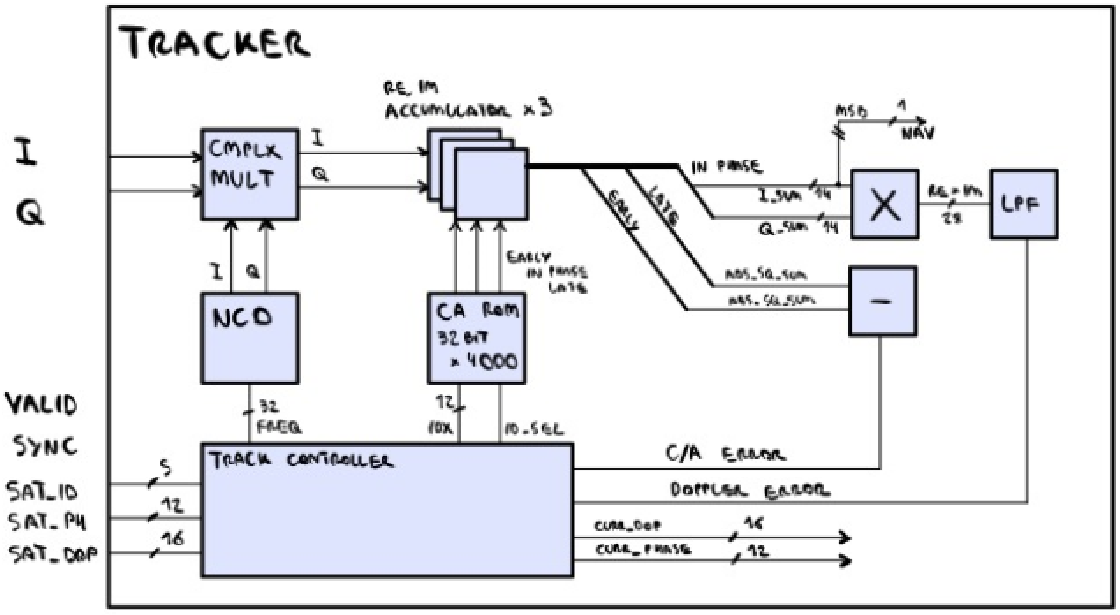


Fig3. Tracker Module Block Diagram

Implementation

I. Teensy as a GPS transmitter

Due to difficulty in getting the software defined radio to output not clean signals, and for repeatability during testing, we decided to use a Teensy to simulate the SDR. It sends an in-phase, out-of-phase, as well as clock sync signal to the FPGA, exactly the same signals the SDR would send. The connection schematic is the following:

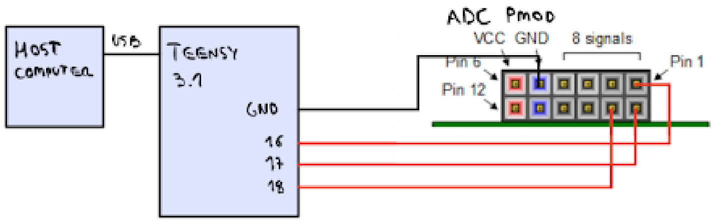


Fig 4. Connection between Teensy and FPGA PMOD port. The ADC port was used because of its lack of 100Ohm series resistors. Since the sample frequencies are ~4Mhz, these would have been problematic.

II. GPS search (td_search.sv)

The first step in the synchronization process is to perform a coarse alignment of the incoming GPS signal with the locally generated signal. In this state the replica signal is brought within one code chip time of the incoming signal. This module takes an input the superimposed satellite's signals (in-phase and quadrature components) and outputs the C/A code phase and carrier signal doppler shift once the correlation has overcome a threshold value, which indicates that a satellite has been found.

Since satellites are always transmitting the C/A code, when the receiver begins collecting data phase of the C/A code is unknown. In this module, we perform a correlation over 8000 samples for a total 2ms of data with a local code stored in memory. In addition, to the code phase, the carrier wave frequency is also estimated. For this, the module tests a total of 100 doppler shifts (with frequency bins of 200Hz) to map a doppler range of +-10KHz. The cost of performing this two dimensional search space is of 8000*4000*100=3.2 billion operations, which at 100Mhz takes 64 seconds, which is acceptable considering the fact that we perform the acquisition of all satellites in parallel, searching the whole space takes 32 seconds.The spreading code has a high correlation value only when it is aligned perfectly in time with itself.

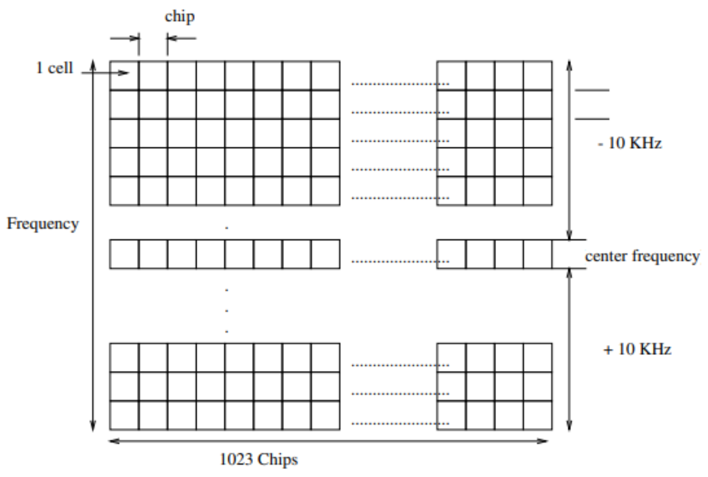


Fig4. Two Dimensional Search Space

We initially implemented the search module using FFTs, i.e. performing the time domain convolutions as multiplications in the frequency domain. However this approach was somewhat problematic since frequency domain multiplication corresponds to *circular* convolution in the time domain, and we were being affected by time aliasing, so convolution was not exactly what we wanted. To get a true convolution, we would need to add padding, i.e. increase the FFT size. Even with an 8192 FFT, the bram usage was pretty high, and it became increasingly difficult to debug. Simulations would take longer to launch than straight up compiling to hardware (sad, but true). And running simulations took forever. This made optimizing bitwidths a lengthy process, since sometimes we needed to take not the top bits, nor the bottom bits of the output, but somewhere in the middle and it was a bit of an art.

III. Handoff Controller (in top_level.sv)

The handoff controller receives an output from the GPS search module, and dispatches this information (satellite id, doppler shift, phase shift) to one of the available tracker modules. It also checks whether the satellite is already being tracked or not, since we do not want to track the same satellite twice. This code resides in the top-level module, as it is basically some glue logic that needs to interface with both the search and tracker modules. The operation is the following:

- Gets a 32 bit array indicating whether a satellite was detected at position i.
- Chooses the first satellite in this array, gets its ID and compares this against a mask of the currently tracked satellites.
- If the mask passes, ie. we are not tracking the satellite yet, it dispatches the information to the first available tracker. If no more trackers are available (all have their “busy” signal set to 1), it discards the data.

IV. GPS tracker (x8) (gps_tracker.sv)

After the signal acquisition process, a fine synchronization is needed. The goal of this process is to maintain the local oscillator in synchrony with the incoming signal to within half a chip time. This module takes as inputs the GPS raw signal, the satellite ID, the C/A phase, and the carrier doppler from the search module. As outputs, it produces the satellite ID, an updated C/A code phase and carrier frequency doppler, as well as a locked state and the navigation data. The main components of this module are an Early-Late Gate Synchronizer, a Costas Loop, a Numerically Controlled Oscillator(NCO) and a FSM.

The FSM starts in the WAITING_INFO state until it receives a new satellite from the search module. Then, it enters the WAITING_SYNC state that ensures that c/a code is properly referenced. Finally, it reaches the TRACKING state where the satellites are being tracked.

The Early-Late Gate Synchronizer is a delay locked loop that multiplies the incoming GPS signal with two versions of the C/A code, which are delayed and advanced by t = Tc/2, where Tc is the chip time. The cross correlated values are then subtracted in order to produce the error signal. We subtract the early error Pe from the late error Pl, so that if we are delayed we increase the phase (Pl-Pe > 0). Then, the error signal is passed through a loop filter that produces an updated C/A code that is in phase with the incoming signal. This PLL does not require a local oscillator for the acquisition codes because the C/A codes for all satellites have been previously generated and stored in memory.

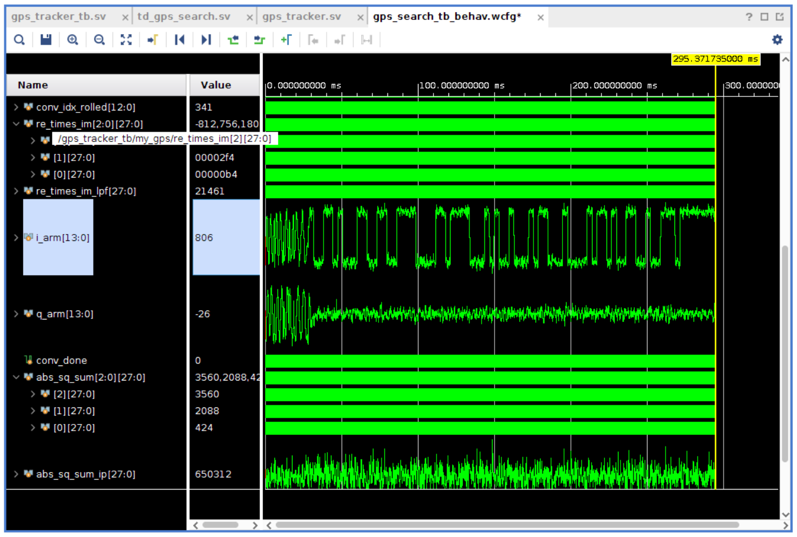
$$c_E(t) = c(t + \triangle t + 0.5T_c)$$

$$c_P(t) = c(t + \triangle t)$$

$$c_L(t) = c(t + \triangle t - 0.5T_c)$$

Fig5. Early, Late, and Prompt C/A codes

The Costas loop, also known as the carrier tracking loop, uses the prompt C/A code and two locally generated copies of the carrier wave. The first copy has the same phase as the received carrier, while the second copy has a +90° phase shift. It multiplies the incoming signal with the prompt C/A code and performs a correlation over 1ms for each of the carrier wave copies. Then, the two correlated outputs are low pass filtered and multiplied in order to obtain the carrier phase error which is then fed to the local sine/cosine generator. Normally, this feedback would operate as in a phase-locked loop, changing the local oscillator frequency. We found that this feedback was not enough for a PLL lock, so we also introduced an explicit phase feedback to the local oscillator. That way, in the presence of an error on the q arm of the Costas Loop, the local oscillator changes both its frequency and its phase, achieving lock within 100ms.



Name	Value
conv_idx_rolled[12:0]	341
re_times_im[2:0][27:0]	-812,756,180
[1][27:0]	00002f4
[0][27:0]	00000b4
re_times_im_lpf[27:0]	21461
i_arm[13:0]	806
q_arm[13:0]	-26
conv_done	0
abs_sq_sum[2:0][27:0]	3560,2088,42
[2][27:0]	3560
[1][27:0]	2088
[0][27:0]	424
abs_sq_sum_ip[27:0]	650312

Fig 6. Tracker locking onto NAV data. The Costas Loop maintains one arm near zero, while the other one has the data encoded on the sign bit. Note that the timescale in the simulation is not correct. In real time, the lock takes around 100ms, which corresponds to 100 iterations of the PLL.

V. Nav decoder (x8) (nav_decoder.sv)

When the Costas loop is locked, the upper branch of the PLL outputs the demodulated navigation data. This module obtains the navigation data from the tracker module as an input and outputs the time in two different ways. First, it outputs the number of seconds since midnight Sunday and secondly, it outputs the number of weeks since January 6th 1980. Note that since this is a given as a 10 bit number, there is a rollover every 19.6 years.

In order to decode the navigation data, there are a sequence of steps to be taken. First, the module waits until the NAV data stabilizes, and finds a data edge to sync to. Bits are then extracted from First the module finds the beginning of the TLM word by performing correlation operation with an 8-bits known preamble pattern (1000 1011). Once a match has been found then the positions of the TLM and HOW words are known and the desired data can be decoded. Then, we look for in the first subframe for the start of the time of week (17 bits long) and week number (10 bits long) information, we decode each of the bits and finally check that each word passes the 6-bit parity bit check. If this is the case then the time information has been successfully decoded.

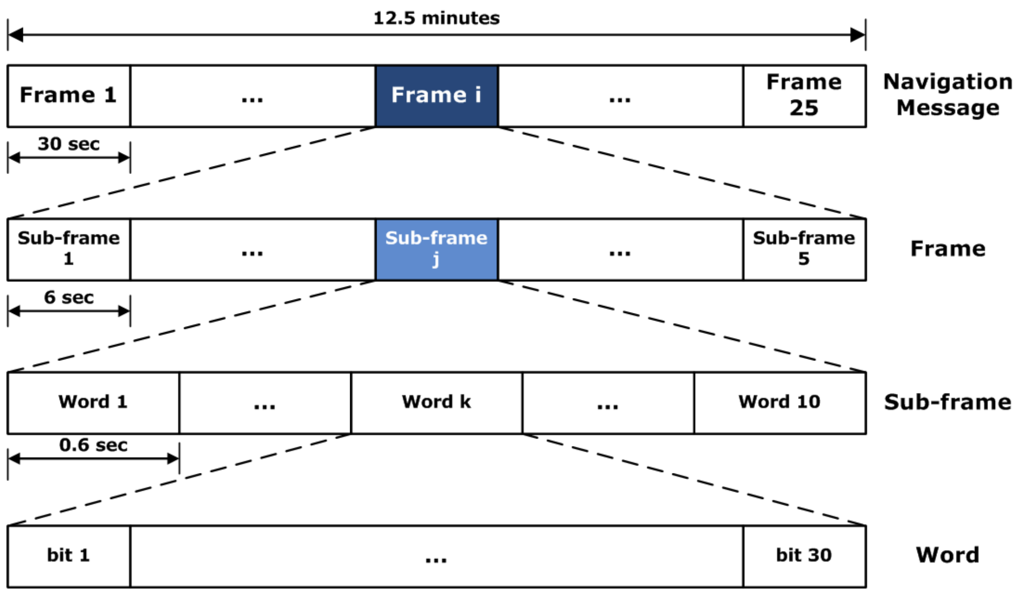


Fig7. NAV Data is transmitted as a series of frames, which are composed of subframes. Most of the relevant information for a position lock can be obtained from a single subframe. Multiple subframes are required for almanac information, which usually helps GPS receivers find satellites in the future.

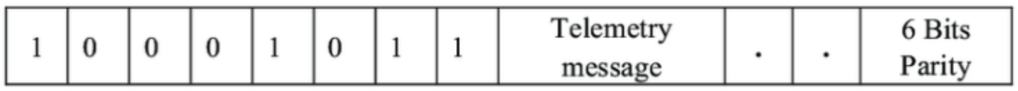


Fig8. Structure of NAV Message Word 1, which includes the preamble. This is transmitted at the start of every subframe (every 6s)

VI. Time display (time_display.sv)

This module's goal is to show in the seven segments display the time of the week information. It accepts as inputs the time of the week from the navigation decoder module and it outputs the time of the week in the following format.

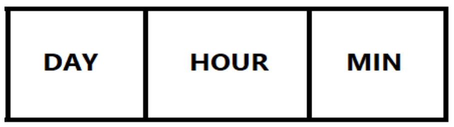


Fig9.. Display Format

Similarly, this module gets as input the week number from the navigation decoder module and it outputs the year and month.

The implementation is relatively straight forward. The modules utilizes two counters to keep track of the number of minutes and hours. Aslo, It utilizes two additional counters to keep track of the year and month. It can perform the desired computations within microseconds.

Lessons Learned & Advice

- While operating in the frequency domain might make convolutions operation easier, one should take care of circular convolution. In order to avoid the circular or spatial-aliasing result, we simply have to pad the two signals with zeros out to a DFT size that is large enough to contain the linear convolution result—the result of ordinary non-circular convolution.
- When developing signal processing modules, writing a version of the pipeline in a high level programming language can prove extremely useful. This allowed us to understand which signal processing components we really needed, and made debugging numerical errors a lot easier. Once a floating-point simulation works, coding a bit-perfect simulation is also a valuable tool, as it allows for step-by-step comparison with Verilog output. One must be conscious of bit widths and careful with overflows, since a Python3 int, for example is unbounded!. Using numpy with explicit int data types can help mitigate this inconsistency.

Future Work

- Decoding orbital information: At this point the implementation is able to decode the navigation data. However, we are only displaying time information. As a next step we would like to combine the orbital information and along with the time information for a minimum of four satellites and feed it into an equation solver to determine the receiver location.
- Determine receiver's location: Once the orbital and time information have been obtained the next logical step would be to implement a location solver in our FPGA. This is not a straightforward task since the system of equations is not linear.
- Finally, one really could feature would be to display the receiver coordinates in a display or be able to pinpoint the user in a map.

Conclusions

Overall, we are proud to have successfully implemented a FPGA based GPS receiver. We were able to detect the satellites in "line of sight" and accurately track the signal in order to demodulate the Navigation data. The highlighted objectives of the projects were achieved; however, as mentioned in the future steps section there is still additional work that needs to be done in order to get the receiver location.

Appendix A - Verilog Code

Refer to the github repo:
https://github.mit.edu/psales/fpga-gps-receiver/tree/master/tests/td_search/td_search.srcs
This includes code and testbenches for all modules.

Appendix B - Non-Verilog Code

Our github repo also contains several supporting code we used during the testing phase. These include:

- Jupyter notebooks: these were used to test the SNR on the sample signals. Implementing the search and tracker module on Python also allowed us to get a good understanding of the signal processing pipeline before attempting to implement it on logic. Considering the amount of numerical debugging we had to do in Python, this proved to be a huge time saver. The modules contained here are:
    - fft_search: a FFT-based search function to find satellite doppler and phase.
    - td_search: a time-domain implementation of the search function. We used this during development and testing of the verilog module, and got the verilog and Python modules to output bit-consistent results.
    - tracker: an implementation of the tracker module. It initially used floating point values for testing, and then it was converted to operate on integers. We used this to fine tune the Costas Loop parameters, which were then used on the Verilog version of the module.
    - translator: this contains various functions, including a C/A code generator, .coe file generators, as well as some code that thresholds the sample data and saves it to a file which can then be easily read by a verilog testbench.
- Teensy: this is a small Arduino project that sends serial data captured from the USB. It uses direct port manipulation to avoid delays due to function calls. It is by no means perfect, but it does the job and is capable of accepting samples at around 1-2Msps. There is also a python (slow) and C (faster) data sender that runs on the computer and sends data to the Teensy over USB.

Appendix C - SDR

We initially planned to use the system together with a software defined radio, so we could capture incoming GPS signals in real time. We chose the HackRF Software Defined radio for its low cost, high sampling rate and clock stability (1ppm needed for GPS applications).

This SDR also features a CPLD that implements some glue logic between the ADC and the microcontroller that then communicates with the host computer. Conveniently, some unused CPLD pins are exposed on a pin header. We planned to modify the open source logic running on the CPLD so that it would replicate the ADC outputs on this header. As mentioned in the main section, we only require 1 bit precision on the in-phase and out-of-phase RF lines. The modification we planned would have compared the incoming signal with the average value, and then output a sign bit for both lines, as well as a sample clock for synchronization.

Before proceeding with modifications, we did some tests with the SDR connected to a host computer. We used the open source program Gqrx (https://gqrx.dk/), which gives a waterfall plot around the center frequency. This was useful in determining the correct gain settings for the analog frontend. We then recorded some test signals at 1575.420MHz using the hackrf-tools command line utility[1]. Note that for GPS signal reception, one must enable the bias-t (option in hackrf-transfer command) which presents a DC voltage on the SMA connector to power the LNA on the GPS antenna.

We obtained some signals and then processed them using both GNSS-SDR (see appendix D), as well as our Python implementation of the search module (see appendix B). We found that the spectrum we obtained was much more noisy than the sample signal, and that the range of the signal only covered a small range of the 8bit ADC resolution. While we only needed the MSB of the signal, looking at the full signal range was useful in determining whether we were getting a good reading or if it was dominated by noise. We confirmed that the noise was higher than the sample we got off the internet by performing an FFT and seeing much higher wideband frequency content.

GNSS-SDR was able to find some satellites, but it was never able to attain lock and decode a full NAV data subframe, presumably due to noise. We also ran the code through our python receiver and confirmed that the SNR after correlation with the correct C/A code phase and doppler shift was only around 3. Considering that this corresponds to the peak C/A correlation with the average over the other phases, getting a clean peak with good isolation was not possible.

At some point, we decided to try with a new antenna and obtained slightly better results. We were not able to run many more tests as we left campus shortly after. We ultimately decided to use recorded data intended for the GNSS-SDR project for repeatability[2].

Appendix D - GNSS-SDR

During our SDR signal testing, and to evaluate the sample signal we downloaded, we used GNSS-SDR. This is an open source project implementing a software defined GPS receiver on a computer. It takes as input an SDR signal stream or a pre recorded sample. It detects satellites,

[1] see more at: https://github.com/mossmann/hackrf/wiki/Software-Support, http://manpages.ubuntu.com/manpages/hirsute/en/man1/hackrf_info.1.html

[2] sample data available at https://gnss-sdr.org/my-first-fix/

tracks them, decodes NAV data, and finds a position solution. It also presents useful information about the satellites found in the signal and their doppler shift.

We used this as a benchmark to see if the signals the SDR was capturing were good enough. If this program was unable to solve for a location, it was unlikely our FPGA implementation could.