FPGA-BASED REAL-TIME GPS RECEIVER

A Design Project Report
Presented to the Engineering Division of the Graduate School
of Cornell University
in Partial Fulfillment of the Requirements for the Degree of
Master of Engineering (Electrical)

by
Adam M. Shapiro

Project Advisor: Bruce R. Land

Degree Date: January, 2010

Abstract

Master of Electrical Engineering Program

Cornell University

Design Project Report

Project Title: FPGA-based Real-time GPS Receiver

Author: Adam M. Shapiro

Abstract:

A survey of the available open-source resources in the field of satellite navigation systems reveals that there is a shortage of robust, easily reconfigurable code. Most existing designs are either intended only for a specific use and therefore too difficult to modify for general-purpose release, or commercially sold and not openly available. With this situation in mind, we are currently working with in collaboration with the Cornell GNSS Research Group to develop and build a mobile FPGA-based GPS receiver. Our intent is to create a general purpose receiver which is easily modifiable, highly modular, and completely open-source.

The design consists of two major components, implemented on separate Altera Cyclone II FPGAs in Verilog. The primary component is a hardware GPS receiver, capable of receiving, tracking, and processing multiple satellite transmissions of the L1 civilian GPS signal in real time. The secondary component is a combined navigation controller and graphics processing unit designed to calculate a navigation solution in real-time and provide a human-readable graphical interface. Once completed, this freely available general purpose receiver should prove useful to the Cornell GNSS Research Group as well as the satellite navigation research community as a whole.

Report Approved by

Project Advisor: Date:

Executive Summary

This project involved the design and development of an open source GPS receiver on an FPGA. The receiver was developed in such a way as to be completely customizable, allowing a user to easily reconfigure it for most applications involving currently-available civilian GPS technology. In addition, the receiver was designed with further development in mind by standardizing interfaces so that new algorithms, designs, and hardware modules could be quickly integrated into the existing platform.

The GPS receiver was developed and tested in the Verilog hardware description language for an Altera Cyclone II FPGA. The hardware was modeled and simulated using a MATLAB receiver emulator, and the results were then compared with the those of the hardware receiver. Initial testing has been successful, and has shown much promise. The hardware is capable of acquiring and tracking a number of GPS satellites in parallel in a variety of environmental conditions. Hardware design optimization has made the receiver resource-efficient enough to greatly exceed initial expectations of tracking capabilities, particularly the number satellites that can be tracked simultaneously. Further development will provide data extraction and satellite observable reporting functionality, allowing a user to navigate using the receiver in real-time.

In addition to the development of the hardware receiver, a significant portion of this project involved the creation of a system for quickly developing large-scale embedded systems projects. A methodology and set of guidelines were developed that aid in the quick production of clean, understandable, functional code. The guidelines set forth rules for hardware development, coding style, and testing methods, all of which aid the system integration process and greatly reduce development time. A set of software tools was developed which enable quick generation of reusable, reconfigurable Verilog hardware. The tools have been tested in the development of the hardware receiver, as well as in external projects, and have proven to be an extremely useful and valuable resource.

All of the hardware, software, and tools developed for this project have been released to the public domain under the GNU General Public License, version 2.

Contents

1 Introduction

The expansion of the open-source software community has prompted many researchers to look for publicly available solutions before turning to more costly alternatives, often with remarkable success. However, the satellite navigation community seems to suffer from a dearth of quality open-source code. This is in part due to the fact that most satellite navigation research is carried out by government or commercial organizations, which are often reluctant to release design data under public license. In an attempt to make a valuable contribution to the open-source resources available to the Global Positioning System (GPS) community, we have been working with the Cornell Global Navigation Satellite Systems (GNSS) research group to develop and build a mobile GPS receiver.

Commercial and academic GPS receivers are often designed in software for ease of modification and production. Writing software affords a number of benefits, including ease of debugging, modularity, and expandability. It is often considerably easier to prototype a system in software, either on a CPU or a DSP, than it is to develop an equivalent system in hardware. That being said, satellite tracking is a largely parallelizable process, and the serial nature of software makes it difficult to easily take advantage of this fact. One of the primary goals of this project is to design a modular, parallel architecture for GPS signal tracking in hardware, while still maintaining much of the modularity and expandability attributed to software. At the same time, we have developed a system of design and set of tools to ease the development of such large-scale hardware systems in the future.

Presented here is a discussion of the design and development of the hardware GPS receiver developed for this project. Basic concepts of GPS navigation and signal tracking are reviewed, followed by an enumeration of several issues and challenges involved in such a design, as well as the strategies developed to address those issues. Finally, preliminary receiver results and future goals for the project are discussed, as well as a description of the tools and procedures developed to aid in the production of such a complex system on an FPGA in Verilog.

1.1 Design Alternatives

The initial attempt at building this receiver produced tracking channels in which hardware was dedicated to a single satellite at a time. Additionally, the multipliers required to calculate vector magnitudes were contained within each channel instead of within the tracking loops, further reducing the amount of shared resources. This design, while fully functional, was extremely inefficient. A single channel, capable of tracking only one satellite, took approximately 3,500 FPGA logic elements (where each element consists of a single flip-flop, a 4-input logic lookup table, and an arithmetic carry chain) and a number of hardware multipliers to implement. This design severely limited the extendibility of the receiver. The large number of logic elements dedicated to each channel meant that the receiver would only be able to support a maximum 8 parallel channels.

An original project design iteration involved including hardware for a navigation controller and graphical interface on the same FPGA as the receiver. This plan was quickly changed however, in favor of a data and observables packaging system. Continuing the generic design methodology, by packaging the data into a standardized format, the receiver is able to be connected to any compliant navigation controller, or directly connected to a computer, for use.

2 The Global Positioning System

2.1 GPS Signal Description

Satellite navigation systems work by triangulating a user's location from known-location reference transmitters. In the case of GPS, a constellation of 32 satellites orbits the Earth in a Medium Earth Orbit (MEO), with a height range of approximately 20,000 km to 26,000 km. [5] Each satellite transmits a data message, providing a receiver with orbital parameters that can then be used to determine the satellite location. Once the locations are known for each of the available satellites in view, the receiver can then determine its location by calculating distances to each satellite from the time-of-flight of the data bits for that satellite's message.

The currently-available civilian GPS signal, called the Coarse Acquisition (C/A) code, is broadcast by each satellite in the constellation on the GPS L1 frequency (1575.42 MHz). This signal contains a data message with the broadcasting satellite's orbital parameters (ephemerides), an atmospheric correction model, and a set of coarse satellite positioning parameters for all satellites, known as the almanac. The broadcast signals are encoded with a Code-Division Multiple Access (CDMA) scheme such that all satellites can broadcast simultaneously on the same frequency. The signal is then mixed with the L1 carrier frequency and broadcast.

The CDMA encoding scheme uses pseudorandom binary sequences known as Gold codes. Gold codes are maximum-length sequences with very good cross-correlation characteristics, which allow for easy signal separation of individual signals by a receiver. A data stream encoded with a given code (referred to by its pseudorandom number code, or PRN code) can be transmitted at the same time as another, and later recovered by correlating it with that same code. The codes used for the L1 C/A code are 1023 bits long and transmit at a rate of 1.023 MHz. Code bits are referred to as chips in order to distinguish them from data bits, as the code bits carry no useful information.

2.2 Satellite Acquisition

Acquisition is the process by which the receiver determines which satellites are in view such that it can track them and begin to navigate. To track a satellite's transmitted signal, a receiver must remove both the carrier frequency and the PRN code from the signal, so that the remaining signal only contains the transmitted data bits (plus noise). There are therefore two pieces of information that a receiver needs to know in order to track a given satellite: the carrier frequency and the code position, as explained below.

As discussed in Section 2.1, all satellites transmit the civilian-intended C/A code with carrier frequency equal to the GPS L1 frequency. The observer, however, does not always see the signal exactly modulated by the L1 frequency, but slightly shifted by the satellite's Doppler shift. Doppler shift is the change in carrier frequency that arises from the radial velocity of the satellite relative to the receiver. A satellite moving towards the receiver will appear to have a carrier frequency slightly greater than the true carrier frequency (L1), whereas a satellite moving away will appear to have a carrier frequency slightly less than the true carrier. This means that, in order to demodulate the signal to removing the carrier frequency (baseband wipe-off), the receiver must determine the Doppler shift of the satellite.

Using a the satellite orbital velocity and the satellite orbit apogee and perigee distances from the center of the Earth,, one can calculate the maximum and minimum expected Doppler shift for any satellite in view. A satellite directly overhead (90 degrees elevation) has zero velocity in the direction of the receiver, while a satellite at the horizon (0 degrees elevation) has a significant velocity component in the direction of the receiver. The Doppler shift, therefore, reaches a maximum magnitude at the horizon, and zero when exactly overhead. Taking into account satellite motion, a satellite coming into view over the horizon will have a positive Doppler shift, while a satellite setting at the horizon and leaving the view of a receiver will have a negative Doppler shift. The maximum expected Doppler shift can be calculated to be approximately 6,000 Hz.

In addition to generating the carrier, the receiver must be able to generate the CDMA code for the selected satellite in order to remove that from the data signal (code wipe-off). The autocorrelation properties of the Gold codes are such that if the generated replica code is offset by one or more chips, the autocorrelation value will always be less than 100 (whereas with zero offset, the autocorrelation reaches a maximum of 1023). Effectively, if the replica code is misaligned with the transmitted code by a full chip or more the received signal will look like noise and will not be recoverable. The acquisition process must therefore search for and determine the current position of the sequence.

When a receiver is first powered on it has no a priori knowledge of the state of the GPS constellation. It does not have any knowledge of where it is located or what the current time is. This type of startup is called a cold start. Without any additional information, the receiver must search for all possible PRNs, checking all possible Doppler and code shifts, until satellites are found.

A warm start involves using prior knowledge to reduce the search space, speeding up the acquisition process. Each satellite transmits a set of parameters, called the almanac, which coarsely describe the long-term orbits of all satellites in the constellation. Combining this information with knowledge of the approximate receiver location and time, the receiver can determine which satellites should be in the sky and what their approximate Doppler shifts are. It can then search only on these parameters. Almanac-aided acquisition can speed up the acquisition process significantly (more than 90% in some cases).

2.3 Signal Tracking

Since the satellites are constantly moving in their orbits, the distance between any given satellite and a receiver is always changing. As the distance between the satellite and receiver change, so does the time at which code bits arrive at the antenna. Similarly, changes in relative velocity of the satellite with respect to the receiver cause the carrier frequency to change over time, as discussed in Section 2.2. Once acquired, therefore, a satellite must be tracked over time to maintain synchronization of the locally-generated carrier and code replicas. This is done in two steps. First, the local code replica is upsampled to the incoming sampling rate. The code is tracked by delaying or advancing the local replica by fractions of a chip in order to maximize the received signal strength. Second, the carrier is tracked by adjusting the local carrier frequency (Doppler shift) to eliminate frequency or phase offset between the received signal and the local replica.

To track the code, three local code replicas are generated for each PRN. Slightly advanced (early) and delayed (late) replicas are generated, in addition to the matched version (prompt). After correlating the incoming signal with each of the three replicas, the receiver can determine if the code must be adjusted, by how much, and in which direction by comparing the three received signal strengths using a Delay-Locked Loop (DLL). It should be noted that the early and late codes must not be offset by more than one half of a chip from the prompt code. If they are offset by more than half of a chip, they might correlate to a different starting PRN chip for a large tracking error, resulting in an inability to recover the code.

To track the carrier, the receiver generates an in-phase carrier replica and a quadrature replica, which is phase shifted from the in-phase carrier by 90 degrees. The receiver then uses a Frequency-Locked Loop (FLL) or Phase-Locked Loop (PLL) to adjust the Doppler shift as needed to obtain the desired result. An FLL attempts to lock in-phase and quadrature accumulations of the signal in place, such that the angle between two subsequent accumulations when plotted in the imaginary plane does not change quickly. When the frequency of the replica and the incoming signal match, the angle will not change from one accumulation to the next. A PLL attempts to reduce the quadrature component to zero, locking the in-phase component to the incoming signal, which subsequently matches both the frequency and phase of the signal.

2.4 Data

Once the carrier and code have been wiped off of the incoming signal, the remaining information, disregarding any noise, is the encoded data message broadcast by the satellite. Data bits are transmitted at a rate of 50 bits/second. The satellites' data broadcasts are divided into frames, which are each further divided into five subframes. Each subframe contains ten 30-bit words. An entire frame takes 30 seconds to transmit.

Data bits are encoded as ± 1, and bit transitions can thus be recovered by looking for 180 degree phase shifts (negations) of the incoming carrier signal. Each satellite uses an atomic clock to synchronize its bit transitions with all of the other satellites. Each subframe begins with a 30-bit telemetry word, which contains a fixed 22-bit preamble, followed by a telemetry message and accompanying parity bits. [5] A receiver can use the preamble to locate the start of a subframe prior to decoding the data message.

The subframes from a given satellite provide ephemeris data required to locate the satellite's orbital position, satellite constellation health information, atmospheric modeling parameters used to correct for positioning errors introduced by ionospheric and tropospheric delays, and an almanac of long-term, coarse satellite orbital parameters for the entire constellation.

2.5 Observables

Navigation using GPS involves triangulating the receiver location in three dimensions relative to the positions of the satellites. To do this, the receiver needs to know how far it is from each satellite used for navigation. Additionally, it must be able to determine to high precision where the satellites are in their orbits, which requires precise knowledge of the current time. While there are only three positional degrees of freedom that must be resolved to locate a receiver, the unknown mismatch between the receiver and satellite clocks adds a fourth unknown that must also be solved in order to navigate. A receiver must therefore track a minimum of four satellites to navigate without additional assistance. Once a signal is tracked, there are three observable parameters that the receiver can extract that provide it with this information.

The first observable, pseudorange, is perhaps the most intuitive of the three. The range to a given satellite is given by the time from transmission of a given bit to reception of that bit, multiplied by the speed of light. Pseudorange is the distance to a satellite measured in this fashion, plus or minus some additional error sources. Since the transmission time is not known by the receiver, this value cannot be directly measured. Because the bit transitions for all satellites occur at the same time (see Section 2.4), however, the distance of one satellite to the receiver can be measured relative to another satellite. By selecting one satellite as a reference point, the relative distances to all other satellites can be measured. The difference between the relative ranges and the true ranges is simply the range to the reference satellite. This unknown distance is the first difference between pseudorange and true range, and is usually thought of as a receiver clock bias.

Next, though the satellites have highly-precise clocks, each has a slight drift which adds additional error to the range calculation. This drift is tracked closely by the GPS control segment, modeled as a quadratic error, and broadcast as part of the ephemeris data to the receiever.

If the transmitted signals travelled through vacuum to the receiver, the measured transmission times would exactly equal the distances to the satellites given the speed of light. The signals do not travel through vacuum, however, and are delayed by any medium they pass through including the ionosphere and troposphere, as well as any ground-based objects they may meet on the way to the receiver. An atmospheric model, which can be used to correct for some signal delay in order to improve navigation accuracy, is broadcast as part of the data message send with the C/A code.

Adding all of these error sources together results in the receiver-perceived distance, pseudorange, as follows

$$P^S = \rho^S + c(\delta^S - \delta_R) + \epsilon^S \tag{1}$$

where P^S is the pseudorange for satellite S, ρ^S is the true range to satellite S, δ^S is satellite S's clock offset, δ_R is the receiver clock error (including the unknown reference satellite range), and ϵ^S is additional error introduced by atmospheric propagation, multipath interference, and other sources. Note that δ_R is common to all satellites, while ϵ^S is satellite-depedent. Because pseudorange is defined by the signal data bit transitions, it is only as accurate as the upsampled code chipping rate (sampling rate) multiplied by the speed of light.

The next observable, Doppler shift, is the most easily-found observable. It is tracked and directly reported by the carrier tracking loops. Doppler shift represents the velocity of the radial receiver with respect to the satellite. The measured Doppler shift includes this value and error terms due to the clock drift rates of both the satellite and the receiver. The Doppler shift, measured in Hz, is given by

$$f_D = \frac{f_{L1}}{1 + \dot{\delta}^S} \left[\frac{-\hat{\rho}^T(\vec{v}^S - \vec{v}) + c\dot{\delta}_R}{c + \hat{\rho}^T(\vec{v}^S - \vec{v})} - \dot{\delta}^S \right] \tag{2}$$

where f_D and f_{L1} are the Doppler shift and the L1 frequency of 1575.42MHz respectively, $\hat{\rho}$ is the unit vector to satellite S from the receiver location in Earth-Centered, Earth-Fixed (ECEF) coordinates, $\vec{v}^S$ is the satellite velocity vector in ECEF, $\vec{v}$ receiver velocity in ECEF, and c is the speed of light in meters per second. $\dot{\delta}^S$ and $\dot{\delta}_R$ are the derivatives of the satellite and receiver clock offsets, respectively.

The final observable, carrier phase, represents the range to the satellite defined by the number of carrier cycles between it and the receiver. Using a PLL, the receiver can keep track of the number of carrier cycles that elapse during a given accumulation period. The range to the satellite can then be determined with an accuracy on the order of the L1 wavelength (approximately 0.19 m). However, because phase range is a relative measurement (the receiver measures changes in phase, not total phase), the true range to the satellite is unknown initially. Resolving the initial number of cycles between the receiver and the satellite is an issue known as the integer ambiguity problem, and is an active topic of research.

Methods for determining the receiver location and velocity, as well as the current time, are not discussed in this paper. Focus will instead be placed on satellite acquisition and tracking.

3 Implementation

This section discusses the implementation details of the hardware receiver. The receiver has been developed in conjunction with Cornell Electrical and Computer Engineering de-

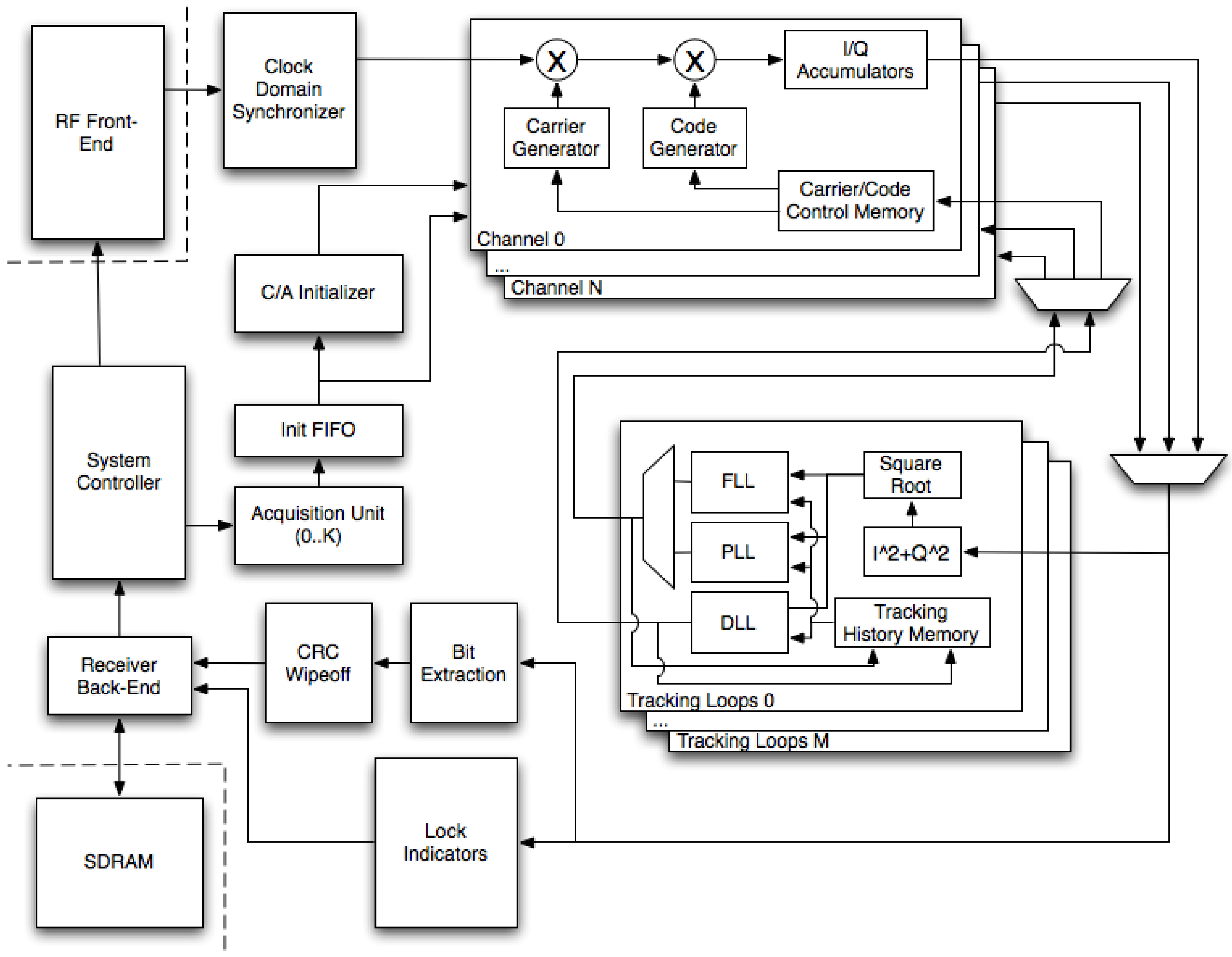

Figure 1: Receiver high-level diagram.

partment's GPS courses (ECE 4150 and ECE 5840), microcontroller and embedded design courses (ECE 4760 and ECE 5760), and the Cornell GNSS Research Group.

3.1 Receiver Overview

This project involved developing a modular GPS receiver in Verilog for an Altera Cyclone II FPGA. The receiver was designed to contain all necessary functionality in hardware to acquire and track satellites, and package observables and extracted satellite data in an easily consumable form for navigation. The hardware is additionally outfitted with a Maxim MAX2741 GPS L1 front-end module with a built-in 3-bit analog-to-digital converter. [3]

The hardware receiver high-level design is shown in Figure 1. The system consists of a set of channels, each capable of tracking multiple satellites in parallel, a set of tracking loops shared among all channels, a set of acquisition units and accompanying channel initialization hardware, a Nios II CPU acting as a receiver back-end and control interface, and data extraction and signal lock detection/indication hardware. An external RF front-end with initial filtering, demodulation stages, and analog-to-digital converter (ADC) can be attached to the receiver via external I/O pins.

All of the hardware for this project was designed to be as generically modular as possible. Receiver functionality was divided among a set of independent modules, and standard interfaces were defined to pass required information between them. In this way, a given functional hardware module could be easily replaced with another without requiring significant modification. This also involved developing a set of tools for automatic code generation, so that the receiver could be reconfigured quickly and easily by changing a small set of global parameters at the top level. This allows the receiver to be quickly modified for different applications, as well as for use with different RF front-ends. The tools developed for this project are discussed in Section 4.

3.2 Channels

In a GPS receiver, each channel is responsible for tracking an assigned satellite over time. A single channel consists of a carrier replica generator, an upsampled code generator, and in-phase and quadrature signal accumulators. Carrier generation is performed using a Direct Digital Synthesis (DDS - see Section 3.3) unit tuned to the carrier frequency ($f_{L1} \pm f_D$) and sine/cosine lookup tables. The replica PRN code is generated by a C/A code upsampler, which uses a DDS unit referenced on the incoming sample clock to generate a 1.023 MHz clock for the C/A code generator. The C/A upsampler generates the early code bit, and delays the bit as required to subsequently generate the prompt and late code bits. The incoming data words are mixed with the generated carrier and code, and accumulated for a a user-specified accumulation period (typically the period is between 1 and 20 ms).

The accumulators and carrier/code generation units take a significant amount of FPGA hardware resources. In order to increase the number of satellites that can be tracked at a time, each channel is pipelined and slotted so that it is capable of performing accumulations for as many satellites as there are cycles between samples. Operating at 200 MHz with a 16.8 MHz sample clock, this means that a given channel can track up to 11 satellites at a time.

Figure 2 shows the hardware implementation of a slotted channel unit, excluding pipelining required for calculation timing and memory data retrieval. A slotted channel contains three memory banks used to store information for its slots. One memory bank holds the tracking control parameters (Doppler shift and code chipping rate DDS phase increments) returned by the tracking loops after each accumulation. A second memory bank holds all of the internal register values for each piece of channel hardware (C/A upsampler, carrier generator DDS, and sample counter) for each slot. The final memory bank holds the in-phase and quadrature accumulator values for the early, prompt, and late subchannels for each slot.

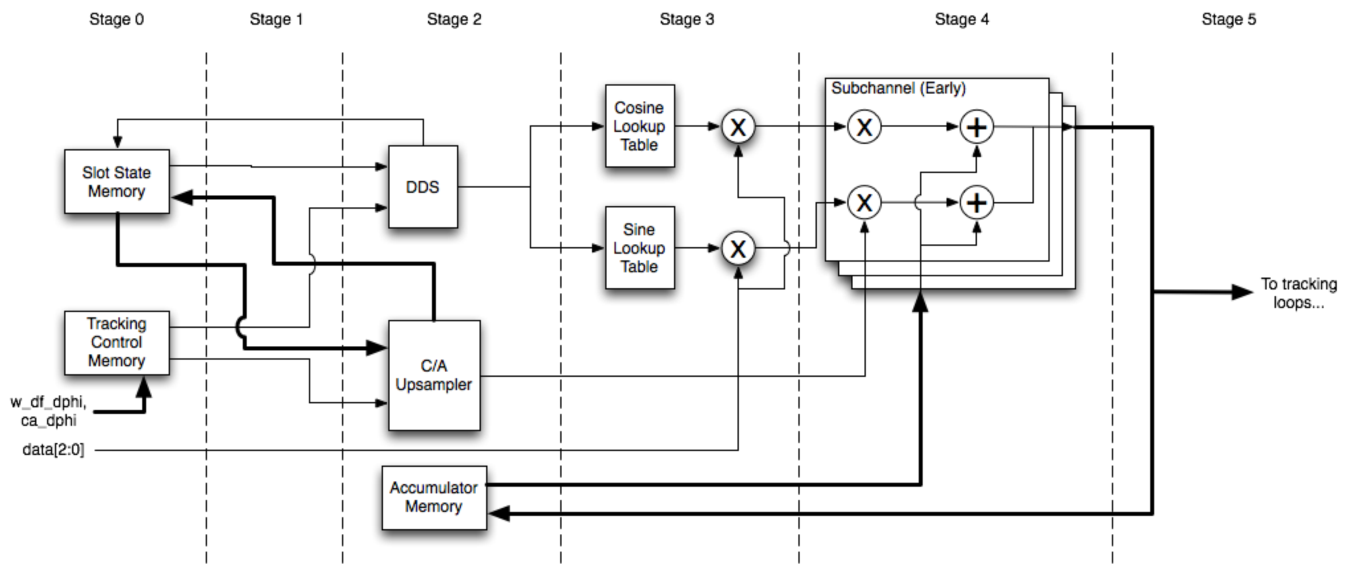

Figure 2: Slotted channel pipeline structure.

3.3 Direct Digital Synthesis

Direct Digital Synthesis, or DDS, is a method of generating arbitrary frequency signals entirely digitally from a given reference frequency. A DDS unit is comprised of an N-bit adder and associated accumulator, either in flip-flops or RAM, an incoming reference clock, and an M-bit phase increment value. The output is an O-bit signal, where $O < N$, taken from the most significant bits of the accumulator output. On each cycle of the reference clock, the accumulator value is increased by the phase increment value. The amount of increase controls the rate at which the accumulator resets (overflows), and subsequently the rate at which the output signal changes.

The phase increment for a desired output frequency is given as

$$\Delta\phi \sim 2^N \frac{f}{f_{ref}}$$

where f is the target output frequency and f_{ref} is the reference frequency. Note that because $\Delta\phi$ is an integer value, the frequency resolution of the DDS unit is restricted by the accumulator width and not all target frequencies will be exactly reproducible.

The first step in choosing design parameters for a DDS unit is to choose the desired frequency resolution for a given reference frequency. The resolution is the amount (in Hz) that the output frequency will change for an increment of the phase increment value. The frequency resolution for a given accumulator width is

$$\Delta f = \frac{f_{ref}}{2^N}$$

and so, for a desired frequency resolution the required accumulator width is

$$N = \left\lceil \log_2 \left(\frac{f_{ref}}{\Delta f_{res}} \right) \right\rceil$$

Next, the maximum phase increment value can be determined from the maximum output frequency necessary for the DDS to produce.

$$\Delta\phi_{max} = 2^M \frac{f_{max}}{f_{ref}}$$

Finally, the required phase increment width is

$$M = \lceil \log_2 (\Delta\phi_{max}) \rceil$$

3.4 Acquisition Units

Each acquisition unit is designed to perform a brute-force acquisition for a specified PRN. Each unit is outfitted with a C/A code generator, an acquisition controller, and a user-specifiable number of accumulation units. Each accumulation unit is connected to a dedicated carrier generator DDS unit and accompanying demodulation hardware.

When performing a search, the acquisition controller assigns Doppler shift bins to each accumulation unit, where the bin spacing is specified by the user at compile time. Accumulations are then performed by demodulating the signal using the assigned Doppler shift, and correlating it with the local code replica for all possible code shifts. The accumulation vector magnitude (see Section 3.5) is then computed, and the peak value is stored along with its corresponding Doppler and code shifts. Once all Doppler and code shifts have been tested, the peak magnitude is compared to a predefined carrier-to-noise threshold to determine if a satellite has been located. A second threshold is defined which enables the accumulation unit to terminate prematurely if a local maximum peak has been located above a certain carrier-to-noise ratio.

$$\begin{aligned}
L_{\text{Single Shift}} &= f_s T_{C/A} \\
L_{\text{Single Bin}} &= (f_s T_{Acc}) L_{\text{Single Shift}} = f_s^2 T_{Acc} T_{C/A} \\
L_{\text{Full PRN Search}} &= \frac{2 f_{D,max}}{\Delta f_{Bin}} L_{\text{Single Bin}} = \frac{2 f_{D,max} f_s^2 T_{Acc} T_{C/A}}{\Delta f_{Bin}} \\
L_{\text{AcquisitionLength}} &= \frac{L_{\text{Full PRN Search}}}{N_{Accumulators}} = \frac{2 f_{D,max} f_s^2 T_{Acc} T_{C/A}}{N_{Accumulators} \Delta f_{Bin}} \qquad (3)
\end{aligned}$$

The total length of time required to perform a brute-force acquisition for a single satellite depends on the size of the Doppler shift search space, the frequency spacing of the Doppler bins, and the number of upsampled chips in a single C/A code repetition, which itself depends on the sampling rate. The Doppler search space is divided among all available accumulators, potentially reducing the search time greatly. The number of cycles required to perform acquisition for a single PRN is given by Equation 3.

Parameter	Value	Description
f_{sys}	200 MHz	System clock frequency
f_s	16.8 MHz	Front-end sampling rate
$T_{C/A}$	1 ms	C/A code repetition period
L_{Acc}	3	Acquisition accumulation length
$f_{D,max}$	6000 Hz	Maximum Doppler shift value
Δf_{Bin}	200 Hz	Doppler bin width
$N_{Accumulators}$	3	Number of acquisition accumulators

Table 1: Acquisition configuration parameters.

Using the current hardware configuration parameters defined in Table 1, a complete search of Doppler and code space for a single PRN takes approximately 1 minute 15 seconds. Note that T_{Acc} is in units of code repetitions ($T_{Acc} = L_{Acc}T_{C/A}$). A serialized search of the entire GPS constellation at this rate would take over 45 minutes, necessitating the instantiation of multiple acquisition units, as well as the use of smart search techniques such as warm start (almanac-aided) search space reduction, to speed up the process.

Separating the acquisition process from the tracking hardware provides a number of advantages. First, rather than being restricted to three sets of accumulators (early, prompt, and late), an acquisition unit can support however many accumulators are needed to balance search performance with hardware usage. Second, a single C/A code upsampler can be shared across all accumulators, without requiring additional hardware necessary to generate the prompt and late codes needed for tracking. Third, the acquisition process can be entirely separated from the reset of the receiver, allowing different acquisition methods to be connected and tested easily without the need to modify the tracking hardware.

Memory Bank Acquisition requires performing correlations on at least 1 ms of sample data sets as quickly as possible. In order to take advantage of the large number of cycles between samples, a memory bank is connected to each acquisition unit in order to feed sample data for accumulations at the system clock speed. A single memory bank records samples, noting the start time of the first sample, for the length of a single acquisition, at which point it is placed into playback mode. It then replays the sample data at full speed, marking the start and end of the frame.

In order to avoid stale acquisition data as a result of a locked memory bank in playback mode, two memory banks are instantiated for the acquisition process. While one memory bank feeds samples to the acquisition units, the other records samples until the accumulation length has been met. The memory banks are then swapped and the first bank begins recording data. By doing this, the accumulation results are never older than the length of a single acquisition accumulation, which ensures that the located Doppler and code shifts will not have changed by much prior to the beginning of tracking.

Initialization Once a satellite is acquired, its Doppler and code shifts, as well as the PRN for the satellite, are queued in a FIFO for tracking initialization. A dedicated C/A code upsampler then seeks through the C/A code until the located code shift has been reached. The state of all of the internal C/A upsampler registers and the located Doppler shift are then passed to the a channel and queued for initialization into the next-available slot.

3.5 Tracking Loops

The receiver tracking loops actively adjust the carrier frequency and code chipping rate in order to account for changes in the transmitted signal characteristics over time. When an accumulation is completed by a given channel, its accumulated in-phase (I) and quadrature (Q) values are queued for tracking processing. Tracking updates are processed in a first-come, first-served order and returned to the channel through the appropriate tracking control memory.

When an accumulation becomes available, the I and Q values are squared, summed, and then passed to a fixed-point square root module (see Section 3.5.1). The resulting values are the vector magnitudes of the early, prompt, and late I and Q values when plotted in the imaginary plane. Next, the channel's tracking history is retrieved from memory and passed, along with the magnitudes and the I/Q values, to the tracking loops. Finally, the resulting control values are stored in memory.

Because the results take multiple cycles to become available, a number of samples at the beginning of each acquisition are correlated using old tracking control parameters. At the target system frequency of 200 MHz, and assuming a tracking update delay of 120 cycles, this results in an update delay of approximately 10 samples per accumulation, or roughly 0.06% of a C/A code repetition. This error value is low enough to be ignored for most design purposes.

In the following subsections, the I and Q values are considered as a vector in the imaginary plane with the Q values plotted on the imaginary axis (this definition of the imaginary plane for I/Q vectors is hereafter referred to as I/Q space). The magnitude of the vector directly corresponds to the received signal strength. The I/Q vector and magnitude are defined in Equation 4.

$$\begin{aligned} \overline{IQ} &= \begin{bmatrix} I \\ Q \end{bmatrix} \\ \|\overline{IQ}\| &= \sqrt{I^2 + Q^2} \end{aligned} \tag{4}$$

3.5.1 Square Root

To calculate the square roots required by the tracking loops, a module was developed to calculate a fixed-point integer approximation to the square root of an input number using an iterative algorithm requiring only bit-shifting, addition, and bitwise logical operations. The iterative algorithm implemented is based on the Friden algorithm, dating back to mechanical calculators. [1] Rather than using a Newton-Rapshon scheme to calculate the square root, which requires division and an unknown number of iterations to converge to the required precision, the Friden-based algorithm uses a binary-adapted "sum of the odd integers" approach to estimate the square root of an input number in a number of iterations approximately equal to the bit width of the input number. It relies on the fact that the nth partial sum S_n of the sequence

$$s = 1, 3, 5, 7, ..., 2k + 1, 0 \leq k \leq \infty$$

can be written as

$$S_n = \sum_{k=0}^{n-1} 2k + 1 = p^2$$

where p is a positive integer. By iteratively subtracting multiples of elements of the sequence s and comparing the divisor and remainder, successive digits of the square root can be computed using bit shifts instead of multiplication and division.

3.5.2 Delay-Locked Loop (DLL)

The delay-locked loop tracks the PRN code for a given satellite by maximizing the vector magnitude (signal strength) of the prompt subchannel with respect to the early and late subchannels. The DLL is a first-order proportional controller, which adjusts the code chipping rate based on the difference between the early and late accumulations. When the local code replica exactly matches the transmitted code, the early and late codes will have the same vector magnitude.

The new chipping rate is found by first computing the theoretical peak correlation amplitude as

$$Amplitude = \frac{\|\overline{IQ}_{early}\| + \|\overline{IQ}_{late}\|}{2 - chips_{EML}} \tag{5}$$

where $chips_{EML}$ is the fractional number of C/A code chips between the early and late codes. Next, the code shift in chips is calculated by normalizing the difference between the early and late magnitudes by the amplitude as

$$\tau' = \frac{\|\overline{IQ}_{early}\| - \|\overline{IQ}_{late}\|}{2Amplitude}$$

This shift is then upsampled from the nominal chipping rate to the incoming sample rate.

$$\begin{aligned}\tau'_{up} &= \tau' \frac{f_s}{f_{C/A}} \\ &= \frac{\|\overline{IQ}_{early}\| - \|\overline{IQ}_{late}\|}{\|\overline{IQ}_{early}\| + \|\overline{IQ}_{late}\|} \left(\frac{(2 - chips_{EML}) f_s}{2 Amplitude f_{C/A}} \right) \end{aligned} \tag{6}$$

Finally, the chipping rate is calculated by applying a proportional gain term to Equation 6 and adding the correction to the nominal chipping rate as

$$f_{C/A,k+1} = f_{C/A}(1 + K\tau')$$

The output of the receiver DLL is the change in chipping rate for the next accumulation, converted to a DDS phase increment referenced on the incoming sample clock. The reported phase increment is shown in Equation 7.

$$\begin{aligned}\Delta\phi_{C/A,k+1} &= \Delta f'_{C/A} \frac{2^{N_{C/A}}}{f_s} \\ &= K \frac{2^{N_{C/AAcc}}}{f_s} f_{C/A} \tau' \\ &= K \left(\frac{(2 - chips_{EML}) 2^{N_{C/A}}}{2 Amplitude} \right) \frac{\|\overline{IQ}_{early}\| - \|\overline{IQ}_{late}\|}{\|\overline{IQ}_{early}\| + \|\overline{IQ}_{late}\|} \end{aligned} \tag{7}$$

Carrier-Aiding Because the satellite is moving with respect to the receiver, the chipping rate changes slightly with the Doppler shift. This change must be accounted for when calculating the new chipping rate, or the code will drift over time and eventually lose track. The additional chipping rate contribution from the Doppler shift is

$$\Delta f_{aid} = \frac{\omega_{df,k+1} f_{C/A}}{2\pi f_{L1}}$$

Converting the contribution to a DDS phase increment, and noting that the Doppler shift reported by the carrier tracking loops is a carrier DDS phase increment (see Section 3.5.3), results in an aiding contribution of

$$\Delta\phi_{aid} = \Delta\phi_{df,k+1} \frac{f_{C/A}}{2\pi f_{L1}} \frac{2^{N_{C/A}}}{f_s} \frac{2\pi f_s}{2^{N_{Carrier}}}$$

Finally, simplifying the result gives a carrier contribution of

$$\Delta\phi_{aid} = \Delta\phi_{df,k+1} \frac{2^{N_{C/A} - NCarrier} f_{C/A}}{f_{L1}} \tag{8}$$

Fixed-Point Computation The DLL calculations are all performed using fixed-point arithmetic, with varying fractional resolution for each coefficient. The three coefficients defined for τ', $\Delta\phi_{C/A,k+1}$, and $\Delta\phi_{aid}$ from Equations 6, 7, and 8 are

$$\begin{aligned} A_{DLL} &\approx \left(\frac{(2 - chips_{EML})f_s}{2Amplitude f_{C/A}}\right) 2^{Shift_{DLL,A}} \\ B_{DLL} &\approx K\left(\frac{(2 - chips_{EML})2^{N_{C/A}}}{2Amplitude}\right) 2^{Shift_{DLL,B}} \\ C_{DLL} &\approx \left(\frac{2^{N_{C/A}-N_{Carrier}} f_{C/A}}{f_{L1}}\right) 2^{Shift_{DLL,C}} \end{aligned}$$

Multiplications are performed as integer operations using the above-defined constants (computed at compile time), and then left-shifted by the appropriate amount.

Fixed-Point Truncation In order to reduce computational complexity, the early and late vector magnitude sum and difference values are truncated to a smaller width before performing any additional computation. Because the amplitude of the accumulation values varies with the signal strength, shifting by a fixed amount can result in an unnecessary loss of precision. Instead, the values are shifted such that the most significant bit of the greater value becomes the most significant bit of the result. This allows maximum precision for changing orders of signal amplitude.

First, the position of the MSB for the vector magnitudes is calculated as

$$\begin{aligned} index_{diff} &= \lceil \log_2(\|\overline{IQ}_{early}\| - \|\overline{IQ}_{late}\|)\rceil - 1 \\ index_{sum} &= \lceil \log_2(\|\overline{IQ}_{early}\| + \|\overline{IQ}_{late}\|)\rceil - 1 \end{aligned}$$

where the index values are referenced from 0. For efficiency, the base-2 logarithm is computed in hardware using a priority encoder to locate the highest-order bit.

Next, the truncation amount is defined as

$$\begin{aligned} index &= max(index_{sum}, index_{diff}) \\ shift &= \begin{cases} index - (width - 1) & : index > width - 1 \\ 0 & : else \end{cases} \end{aligned}$$

where *width* is the desired result bit-width. The input values are then truncated by the specified shift amount.

3.5.3 Frequency-Locked Loop (FLL)

The frequency-locked loop tracks a given satellite's carrier frequency by adjusting the Doppler shift until the local replica is aligned with the incoming signal. This is done by driving the change in rotation angle between the current I/Q vector for the prompt subchannel, $\overline{IQ}_{prompt,k}$, and the previous I/Q vector, $\overline{IQ}_{prompt,k-1}$.

The cross-product of the two vectors in I/Q space is

$$\overline{IQ}_k \times \overline{IQ}_{k-1} = \|\overline{IQ}_k\| \|\overline{IQ}_{k-1}\| \sin(\Delta\theta)$$

where $\Delta\theta$ is the rotation angle, or the angle between the two vectors. Note that for efficiency's sake, the prompt subscript has been omitted from all equations.

Using a small angle approximation, $\sin(\Delta\theta) \approx \Delta\theta$, the rotation angle can be defined as

$$\Delta\theta \approx \frac{\overline{IQ}_k \times \overline{IQ}_{k-1}}{\|\overline{IQ}_k\| \|\overline{IQ}_{k-1}\|}$$

Finally, substituting the definition of the vector cross-product yields

$$\Delta\theta \approx \frac{Q_k I_{k-1} - I_k Q_{k-1}}{\|\overline{IQ}_k\| \|\overline{IQ}_{k-1}\|} \tag{9}$$

The FLL uses a second-order control loop to adjust the Doppler shift of the local carrier replica. The outputs of the control loop are Doppler shift and Doppler shift drift rate (the derivative of Doppler shift). The two parameters are derived from the rotation angle defined in Equation 9 as

$$\omega_{df,k+1} = \omega_{df,k} + \dot{\omega}_{df,k} T + s B_{FLL} \Delta\theta \tag{10}$$

$$\dot{\omega}_{df,k+1} = \dot{\omega}_{df,k} + s A_{FLL} \Delta\theta \tag{11}$$

where T is the tracking accumulation period in seconds, A_{FLL} and B_{FLL} are FLL gain coefficients dependent on the desired loop bandwidth, and s is a sign value corresponding to the type of front-end baseband mixing used as follows

$$s = \begin{cases} -1 & : \text{high-side mixing} \\ 1 & : \text{low-side mixing} \end{cases}$$

Fixed-Point Computation As with the DLL, the FLL calculations are all performed using fixed-point arithmetic, with varying fractional resolution for each coefficient (see Section

3.5.2). The coefficients defined in Equations 10 and 11 for A_{FLL} and B_{FLL} include scaling factors to convert the rotation angle in units of radians to DDS phase increment units. Additionally, the coefficients include the loop bandwidth-dependent gain terms discussed in Section 3.5.3. The fixed-point coefficients are defined as

$$A_{FLL} \approx \left((1.89 \Delta f_{loop})^2 \frac{2^{N_{C/A}}}{2\pi f_s} \right) 2^{Shift_{FLL}}$$
$$B_{FLL} \approx \left(1.89 \sqrt{2} \Delta f_{loop} \frac{2^{N_{C/A}}}{2\pi f_s} \right) 2^{Shift_{FLL}}$$

where Δf_{loop} is the desired loop bandwidth in Hz. Note that, unlike the DLL, the FLL fixed point coefficients use the same fixed-point resolution (see Section 3.5.2).

In addition to the above constants, the accumulation period multiplication in Equation 10 is performed as a fixed point operation. A separate fixed-point resolution is defined for the period constant in the hardware implementation.

Fixed-Point Truncation To reduce computational complexity, the I/Q values and vector magnitudes are truncated using the method described in Section 3.5.2. For the FLL, the vector magnitudes are used to set the scaling point for truncation.

3.5.4 Phase-Locked Loop (PLL)

A phase-locked loop adjusts the frequency of the local carrier replica to not only match the frequency of the incoming signal, but the phase as well. This is done by minimizing the quadrature component such that the entire signal accumulation is contained within the in-phase component. A phase-locked loop is not currently implemented in the hardware receiver. Plans exist to implement one in the near future.

3.6 Data Extraction

C/A code data bits are transmitted at a rate of 50 bits per second. The bits, taking the values -1 and 1, modulate the carrier signal and appear as 180 degree phase changes. With each tracking update the phase of the carrier signal is checked. If a phase flip is detected the previously reported bit value is inverted. If not, the previous bit value is reported again. This results in a bit stream upsampled at the tracking accumulation rate.

In hardware, symmetry properties in the imaginary plane can be exploited in order to avoid calculating inverse trigonometric functions to determine the angle between I/Q vectors. Generally, a 180 degree phase flip results in sign changes for both the I and Q values. This can be tested efficiently using an exclusive-or comparison of the values' sign bits. This rule cannot be used exactly as stated near the real or imaginary axes however, since the signal is affected by noise causing fluctuations of I and Q values (variation of the phase change from exactly 180 degrees). When either the I or Q value is small, as determined by a predefined threshold, only the sign bit of the other value is tested for a transition.

3.7 Lock Indicators

Lock indicators are used to signal when a given slot's tracking results are ready for use, and to what degree they can be used to navigate. There are four lock indicators used for each satellite: code, carrier, bit, and frame.

Code lock indicates a match of the local code replica to the transmitted code from the satellite. This is done by simply comparing the carrier-to-noise ratio at each accumulation to a lock threshold. If the signal strength is higher than the specified value, it is assumed that the two codes are aligned. Signals that do not obtain code lock may not be suitable for accurate navigation as their code phases (and therefore pseudoranges) may be incorrect.

Carrier lock is used to indicate when the carrier frequency has been located and tracked to within a reasonable precision. This is determined by comparing the angle between the current and previous I/Q vectors (corrected for 180 degree phase shifts due to bit transitions) to a specified threshold. An angle less than the threshold indicates a near-match of the frequency of the local carrier replica to the signal carrier. When the two frequencies match exactly the angle will not change from one accumulation to the next. Doppler shift and carrier phase measurements for a given satellite should not be weighed heavily when navigating if they do not obtain carrier lock. Loss of carrier lock can be an indication that a signal is no longer being tracked, and should be responded to accordingly.

Bit lock occurs when an empirically-found statistical bit transition rate threshold is met by the signal, indicating that data is being observed. A moving window is used to count the number of bit transitions in a short period. If the number of transitions observed is greater than or equal to the expected threshold, the data stream is assumed to be visible.

Since data bits transition at a slower rate than bit values are reported, the bit stream must be downsampled before it can be processed. Moving window counts are kept for each possible transition bin, where the number of bins is equal to the tracking accumulation period divided by the C/A bit rate, and the bin with the greatest count value is assumed to be the correct stream transition point. The data stream is then downsampled from that point, resulting in a data stream at the transmitted rate. Until bit lock is obtained, the data stream for a given satellite cannot be downsampled properly, and therefore cannot be decoded.

The final indicator, frame lock, is used to indicate when the start of a C/A code subframe has been located. This is done by searching for the subframe preamble (a fixed sequence of bits defined in the GPS interface specification) in two or more consecutive subframes. If the preamble is located in the data stream, separated by exactly one subframe (300 bits), the beginning of the frame has been located and the contained data can now be decoded. [5]

3.8 Observables Extraction

As discussed in Section 2.5, GPS receivers typically report three observable values: pseudorange, Doppler shift, and carrier phase. These values are updated with each tracking update.

Pseudorange, or code phase range, represents the distance to the satellite from the receiver antenna. Because the satellites are synchronized, pseudorange can be measured relative to one satellite, selected as a reference, as the difference in arrival times of the satellites' pseudorandom codes. In order to do this, the first measurement for the reference satellite must be reported with a zero value. The time of the reference transition can then be extrapolated out by exactly one code length and used to find future pseudorange values for all tracked satellites. Any error in the extrapolated reference time value will appear in the navigation solution as a receiver bias or drift. This value can, in turn be fed back to the receiver for further correction if desired.

Doppler shift represents the rate of change of the range from the receiver to the satellite. Doppler shift is tracked directly by the carrier tracking loops (FLL and PLL), and can simply be reported when a new value becomes available. Doppler shift can be used to determine receiver velocity to a good degree of accuracy. In this receiver, Doppler shift is reported as a DDS phase increment, and must be converted to a more user-friendly unit (either Hz or rad/s) prior to use.

Like pseudorange, carrier phase also represents the distance to the satellite. Pseudorange, which is calculated based on the start of the pseudorandom code, can have a resolution only as good as a fraction of the size of a C/A code chip in meters (generally around 3-6 meters). [6] Carrier phase, on the other hand, is referenced on transitions that are much closer together, resulting in a ranging error on the order of the L1 wavelength (approximately 0.19 meters). Carrier phase is determined by recording the number of carrier cycles between accumulations.

3.9 Clock Domain Synchronization

Clock domain synchronization involves synchronizing digital data and control signals across hardware boundaries where two clocks with different frequencies exist. In the case of this receiver, it was necessary to synchronize the incoming sample data with the high-speed system clock. Without this step, the setup and hold times for the receiver hardware cannot be guaranteed and will cause unexpected results. For instance, if an incoming data bit is high is not synchronized the output could take on any of three states: high, low, or metastable (a voltage between the CMOS high and low voltage thresholds).

A synchronizer is a device designed to eliminate the output metastability that occurs when the input of a flip-flop is not asserted for a long enough time before the clock edge to meet the setup requirement. Figure 3 shows the schematic for a single-bit synchronizer. The input signal (in the source clock domain) is connected to a flip-flop clocked on the destination clock. The output of that flip-flop can take on any state, as described previously. The output is

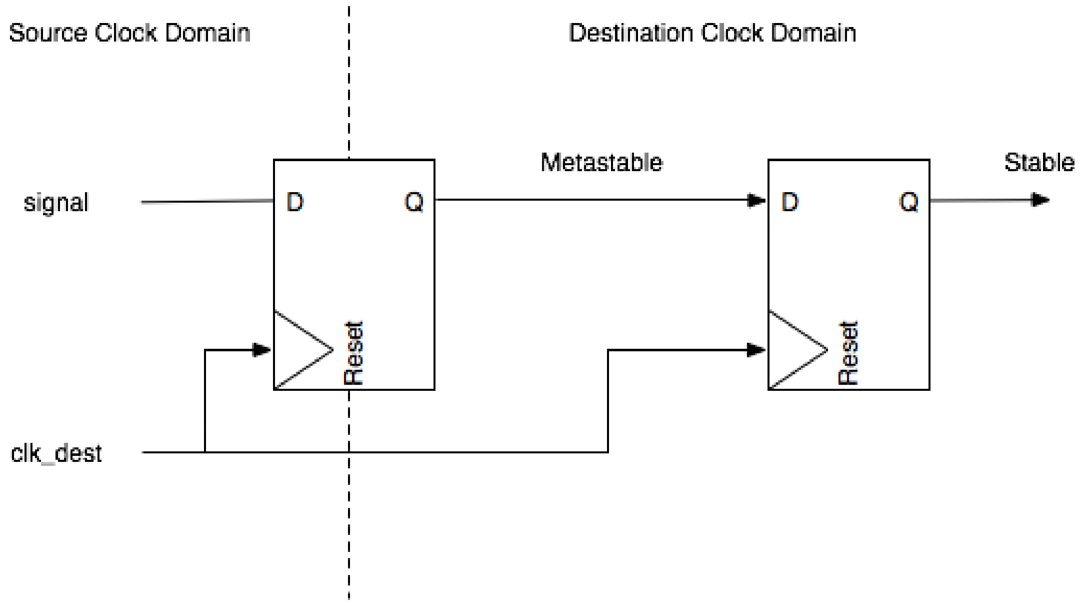

Figure 3: Clock domain synchronizer.

then fed to the input of a second flip-flop, again clocked on the destination clock. Because the output is asserted, in whatever state it takes, for a whole cycle, the setup requirement of the second flip-flop is met and the result will always be stable.

This method works well when synchronizing individual signals; however, when multiple signals or multi-bit signals are required it faces data incoherence issues. Because the intermediate wire can be metastable, the voltage level of which depends on the exact setup time and physical parameters of the circuit, the output of the synchronizer might lag an input change by an additional cycle. The different lengths of signal routing paths for multi-signal synchronization means that some signals might lag others by one system clock cycle.

The solution to this problem, shown in Figure 4, is to use a mux-recirculation synchronizer. [7] Rather than send each signal through a separate dual flip-flop synchronizer, a mux-recirculation synchronizer synchronizes a single enable signal, then uses that signal to enable flip-flops to capture the other incoming bits. In the case of the receiver, the enable signal is the incoming sample clock, which is then strobed for one system clock cycle to provide a latch enable signal for the sample data bits. The downside to this technique is that it requires the incoming data to be stable for at least three system clock cycles to capture the data signals. This may not be possible in some applications.

4 Tools and Testing

In order to allow for easily modular code development, a number of hardware and software tools had to be designed throughout the course of this project. Code generation tools were developed to enable quick code modification, easy adjustment for calculation or hardware

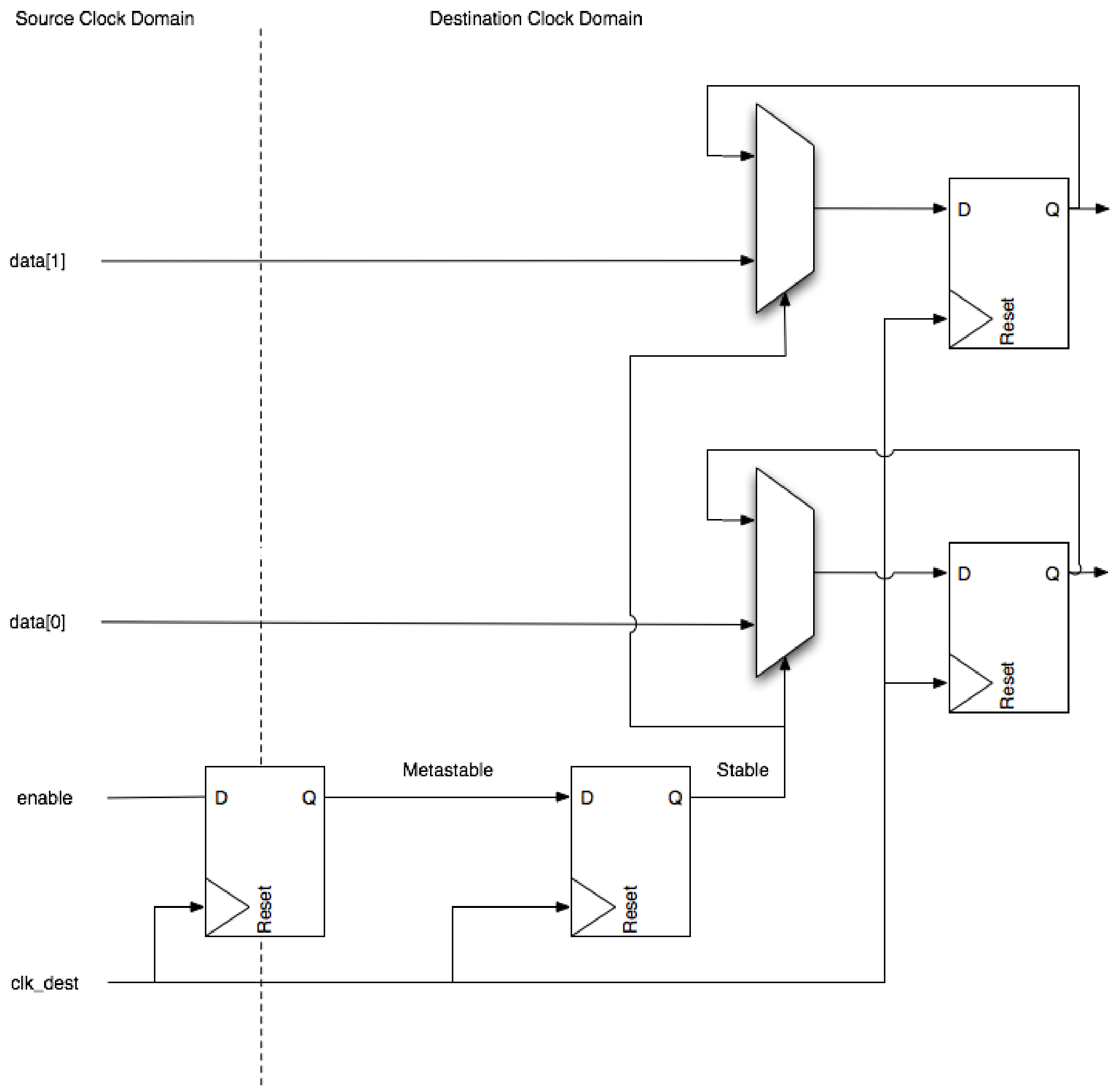

Figure 4: Mux-recirculation synchronizer.

changes, including changing of signal widths, compile-time determination of required signal widths and bit ranges, optimizing design efficiency on the fly, and repeatable testing with real-time data log playback capabilities. Additional software tools were created to generate controllable data logs, and to test and tune fixed-point tracking algorithms prior to developing hardware components.

4.1 Verilog Defines Parser

The Defines Parser, or defparser, is a Java utility designed to automatically generate Verilog include files from a set of interdependent definitions. For each variable defined in the input files, defparser generates a Verilog macro by expanding the provided value expression, recursively expanding dependencies as needed. Values can be specified using a set of mathematical expressions and supported functions, as well as values from any other macros defined in the scope. This allows for quick modification of signal widths based on maximum representable values, wire bit ranges, fixed-width Verilog constants, and other useful parameters. Additionally, calculations can be maintained in floating-point for maximum precision, and converted to fixed-point or integer format when generated. This allows for the automatic generation of accurate fixed-point constants based on a smaller set of global definitions.

In Verilog, all macros are available within a global scope from the point at which they are defined. This means that a macro defined in one file will be available in all subsequently compiled files. An additional feature of defparser is the ability to generate a file containing Verilog undefine directives for each macro specified in the input files. This file can then be included at the beginning of a Verilog file in order to clear the macro replacement table and eliminate this problem completely.

Finally, defparser can be used to find overlapping macro names, warning the user of potential macro redefinitions, which might otherwise cause unexpected errors.

A future goal for defparser is the ability to generate dependancy files for a supplied input file. Such files could then be included in a project make flow, causing header files to be recompiled automatically whenever a required source file is changed.

4.2 Verilog Preprocessor

The Verilog Preprocessor is a set of Python scripts used as a means to perfom code generation and in-line arithmetic for code generalization beyond the ability of the Verilog language. Due to the effort to keep Verilog code as general as possible, there were many instances in which code generation was required that could not be performed using Verilog `generate` statements and other built-in options. The preprocessor allows a developer to use the full power of the Python programming language to perform these operations. The developer simply writes a Verilog module as per usual, but has the option to insert Python code at any time using a series of predefined delimiters: `<?` and `?>` for begin and end, respectively. When the preprocessor script is executed on this file, the Python code within these delimiters is executed in a Python environment, and the code surrounded by the delimiters as well as the delimiters themselves are replaced by the output of the Python code in the output file. This affords many opportunities to the user which are not available in the Verilog language; for example, a developer might use the preprocessor to automatically generate the `case` statement conditions for a lookup table at compile-time, based on a set of macros or parameters defined by the user. This task is not possible to do on-the-fly in Verilog code alone, and traditionally is done manually, making the hardware significantly less general and introducing room for error if the system parameters change without the code being modified.

In addition to executing Python code and printing an output file, the preprocessor parses the input file to locate any defined macros or header files. All header files are parsed recursively, moving "up the tree" until a top-level header file is reached, and all defined macros are stored in a dictionary object at runtime. This allows the developer to use predefined Verilog macros as variables in embedded Python code, which is useful for such tasks as controlling program flow via if/else statements or setting the number of iterations a loop is to perform.

Future goals for the preprocessor include support for concise Python code insertion (for example, abbreviating print statements for common expressions such as wire or input declarations) and improved parsing of module instantiation, to support recursive preprocessing of source files. These improvements will both streamline preprocessor code creation and ease the insertion of the preprocessor into project make flow.

4.3 High-Speed Ethernet Data Feed

In designing the hardware receiver, it quickly became clear that a system was needed to provide repeatable testing capabilities for development. Developing and debugging the individual components of the receiver from a live, uncontrollable, and noise-affected front-end data source was not feasible. The sample rate used in the hardware receiver, required due to the C/A code chipping rate, necessitated the design of a high-speed hardware data feed system. The bandwidth required by the sample data rate is

$$\begin{aligned} f_s &= 16.8\text{ Msamp/sec} \\ \mathrm{N}_{\text{Sample}} &= 3\text{ bits (sign/magnitude)} \\ BW &= f_s \mathrm{N}_{\text{Sample}} = 50.4\text{ Mbps} \end{aligned}$$

This bandwidth is too large for most common (simple) communication schemes – particularly those easily interfaceable to a computer without the addition of custom hardware. The solution is to feed the data over Ethernet, which is capable of bandwidth up 100 Mbps.

The DE2 development board developed by Terasic Technologies for the Altera Cyclone II FPGA contains a Davicom DM9000A Ethernet controller. The DM9000A provides bidirectional hardware Ethernet functionality, designed to interface to a processor using a 16-bit data bus. [2] A hardware module was developed in Verilog to interface to and control the DM9000A at full rate. The module generates the required Ethernet control clock, issues the required reset sequence, handles received interrupts, and reads from/writes to asynchronous data FIFOs such that Ethernet connectivity can easily be added to hardware operating at any clock speed.

The received Ethernet frame data is then passed to a packet processor, which filters the packet stream by EtherType, discarding any unwanted packets. After filtering, the remaining data is placed into a FIFO and read out into a sample stream generating buffer. The system generates a local sample clock, and data samples are removed from the buffer on each cycle and passed to the hardware.

A C utility was developed in conjunction with the hardware data feed, which is capable of transmitting data from a binary file using a user-specified burst size and bit rate. When combined, the data feed hardware and utility allow generated or recorded data logs to be played back into the receiver for easily-repeatable testing.

4.4 Data Generation

In order to test various aspects of the receiver independently of others, it was required to design a data set generation tool capable of producing sample data using specified parameters, that could then be passed to the receiver as needed. The data log generation utility was built in MATLAB and is capable of creating a sample set of any desired length with a specified PRN, Doppler shift, and code shift. The utility can additionally inject noise into the signal, and generate Doppler ramps over time, for additional testing as needed.

The resulting log data can be used in MATLAB for further testing, or can be saved as a log file for transfer to the hardware. A MATLAB MEX function was written in C, capable of saving a sample data vector to a binary file in either the hardware or software receiver supported formats. The hardware format involves packaging 3-bit, sign/magnitude sample values into 8-bit words. The software format requires that the values be scaled to two bits (sign/magnitude), then packaged 16 samples at a time.

4.4.1 Fixed-Point Implementation

MATLAB scripts were developed to perform the fixed-point calculations in the exact sequence in which they are performed in the receiver, including all truncation and loss steps, such that the receiver tracking outputs could be functionally verified. The test scripts report the floating-point truth values, the non-truncated fixed-point values, and the truncated fixed-point values. These values can be generated using measured accumulation and tracking history values reported by the receiver and then be compared to the hardware implementation results to locate errors.

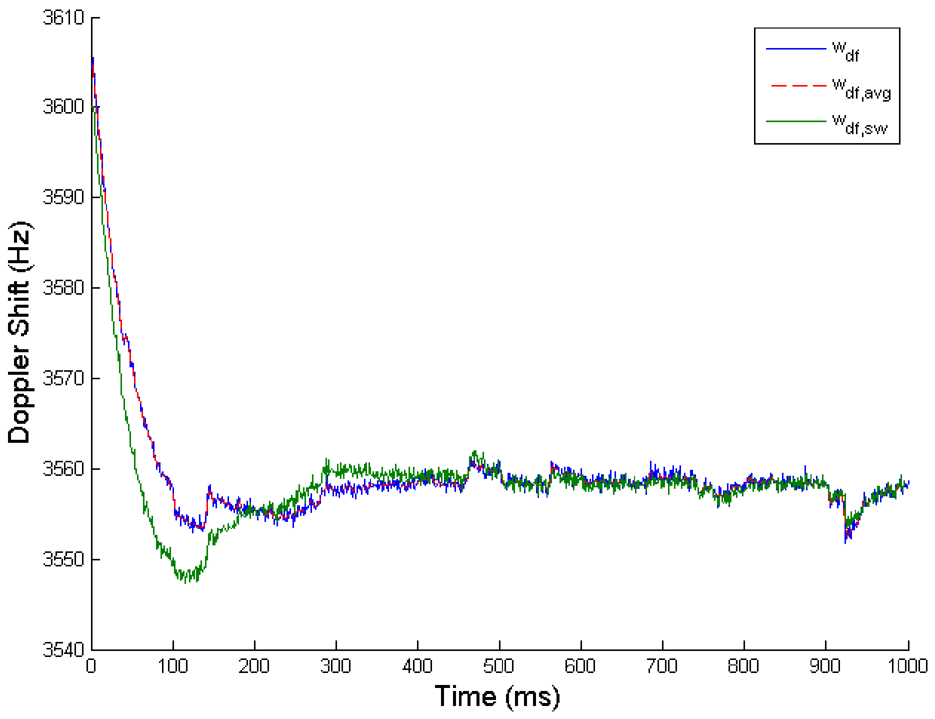

Figure 5: Doppler shift plot tracking a real satellite in noisy, low gain conditions.

4.5 MATLAB Software Testbench

The tracking scripts described in Section 4.4.1 were also modified to operate within the Cornell-developed software GPS receiver. By saving a generated log in the software-supported format (see Section 4.4), the software receiver could be used to test tracking loop calculations, try various discriminators, and tune fixed-point resolutions to provide the best possible results for the most optimized conditions.

5 Results

The current working receiver is capable of acquiring and tracking multiple satellites at a time. The acquisition unit performed quite well and, using 3 ms correlations, was able to acquire satellites using both the generated signals with injected noise and real data logged from the hardware front-end. Though it does take a considerable amount of time to search the entire Doppler and code space, the early completion threshold allows acquisition to complete early quite often and begin tracking much quicker.

Signal tracking using the fixed-point tracking loops has proven to be capable as well. The FLL is able to converge from a large distance away, including as far as 400 Hz away in some test cases, which is significantly larger than the maximum distance that will ever be required

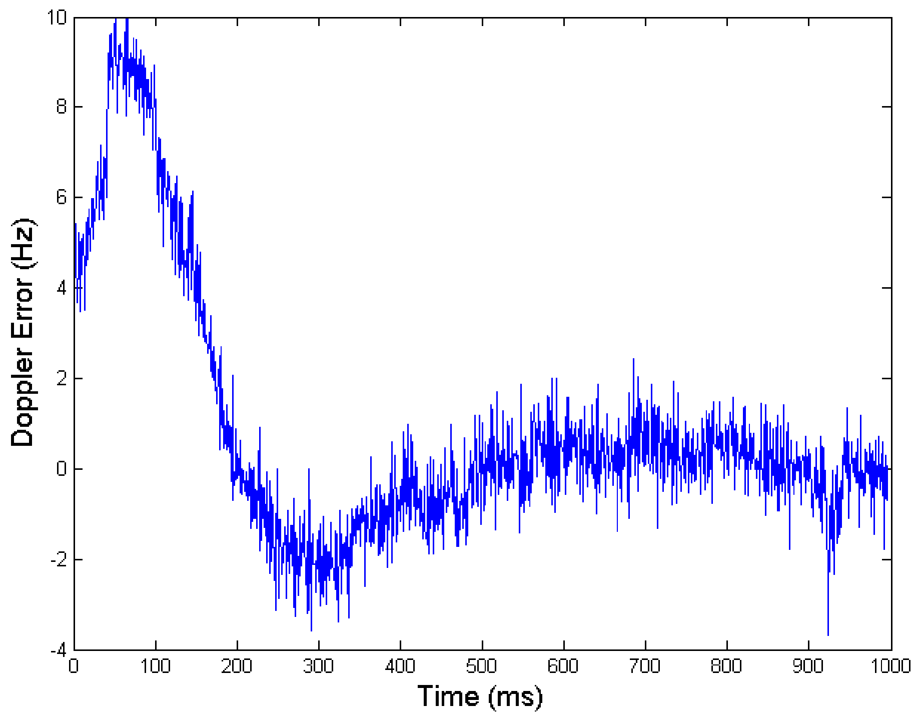

Figure 6: Hardware Doppler shift error compared to the MATLAB software receiver.

due to the Doppler bin spacing. Figure 5 shows the FLL carrier tracking results reported when tracking on real data with low front-end gain, resulting in low input dynamic range. Despite the poor input conditions, the receiver is able to converge quickly to the carrier frequency.

The blue plot in Figure 5 shows the hardware output, while the green plot represents the shows the results generated by the MATLAB software receiver. After an initial difference, likely due to code tracking and initialization differences, the hardware and software receivers converge and report almost identical results. Figure 6 shows the difference between the hardware and software Doppler shifts over time.

Even more remarkable, the high sampling rate and subsequent high code upsampling rate allow the DLL to converge and lock onto the code from very large chipping offsets. Figure 7 illustrates this ability of the DLL. In this case, the initial code start was offset by 16 upsampled chips (out of approximately 16.4 upsampled chips per code chip). Despite the extremely low correlation values, the channel is able to converge on the true code shift in approximately 20 ms, while simultaneously converging on the unknown Doppler shift. It is important to note that this convergence was made possible by using noise-free, generated data with only single satellite broadcasting.

Figure 8 shows a plot of the I/Q accumulation vectors over time while tracking a satellite using data recorded from a GPS antenna. The 180 degree transitions in the imaginary plane, caused by satellite data bit transitions, are clearly visible over time. The large radius

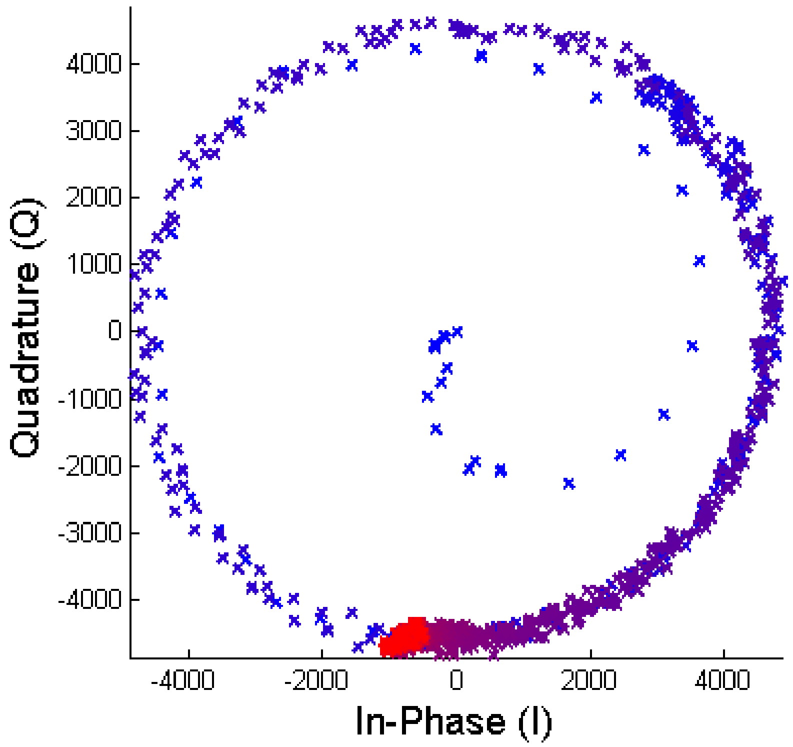

Figure 7: I/Q spiral from DLL convergence after a large initial offset on noise-free data.

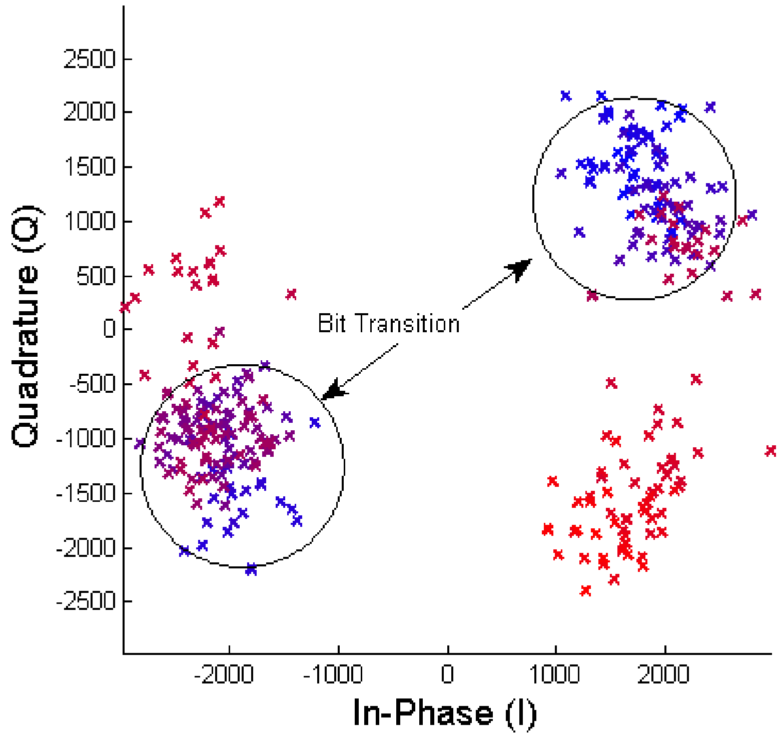

Figure 8: I/Q plot of a real GPS signal depicting visible bit transitions.

of the accumulation clusters is caused by signal noise and reduced dynamic range from low front-end gain.

5.1 Hardware

The hardware fits quite well onto the Altera Cyclone II FPGA used for this project. The FPGA has a total of $\sim$ 33k logic elements, 483,840 bits of on-chip memory, and 70 9-bit hardware multipliers. The three major receiver components, the channel, the acquisition unit, and the tracking loops take approximately 1,500, 3,000, and 6,000 logic elements respectively. A single memory bank uses approximately 193,500 bits of memory for sample data storage. Combined with a Nios II microprocessor used for data packaging, the current system uses approximately 34% of the total available resources.

By sharing the largest hardware component, the tracking loops, across all hardware channels, the receiver is capable of implementing quite a large number of parallel satellite tracking slots for very little added hardware usage. This leaves the majority of the remaining hardware for additional acquisition accumulators to speed up the acquisition process.

The modular design of the system makes it very easy to change as needed to accommodate new designs, hardware, and application-specific parameters. By carefully and meticulously using Verilog macros and parameters, most all of the hardware can be entirely reconfigured by modifying a small set of globally-defined constants in a single header file. Similarly, precision of fixed-point calculations used in the tracking loops can be easily changed as needed, simply by changing the desired fractional width definition constants.

5.2 Issues

The acquisition unit currently uses a fixed memory bank, as opposed to swapping between two memory banks to refresh stale data. Because of this, code and Doppler drift accumulate during the acquisition process and make it more difficult to initialize tracking after long searches. This is particularly true for weaker signal strength satellites, including low-eleveation satellites. Plans are in place to implement memory bank switching in another MEng project in the near future (see Section 6), enabling proper real-time acquisition.

The receiver does not currently contain lock indication or data extraction functionality. These functions have been planned however, and will be added to the hardware in the near future.

Due to time constraints, not all hardware was able to be optimized to run at the target system clock speed of 200 MHz. The results of this are increased acquisition time and slightly decreased tracking resolution. The time to completion for acquisition of a single PRN is currently approximately 2.5 minutes when an early-termination does not occur. Tracking updates take approximately twice as long to complete, with respect to number of samples missed, than originally intended. This additional delay, however, results in a total loss of approximately 23 samples, or 0.13% of the total accumulation size.

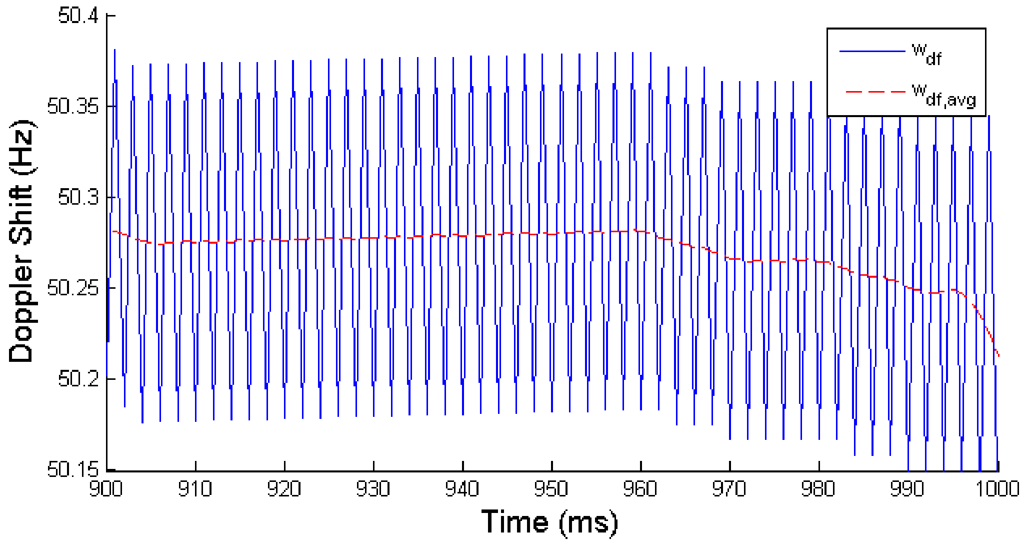

Figure 9: Carrier tracking limit cycle with a ±0.1 Hz oscillation.

Figure 9 shows a noticeable limit cycle, manifested as high-frequency Doppler shift jitter, that exists when tracking carrier using the hardware FLL. Using generated data with no noise injection and a constant Doppler shift, the tracked Doppler shift varies by up to about ±0.2 Hz peak to peak in steady state. This limit cycle is likely a result of three factors: the nonlinearity introduced by the small-angle approximation used in Equation 9, truncation error due to the fixed-point calculations, and transport delay introduced by the computation time required by the tracking loops.

6 Future Goals

This project is to be completed in two parts. The first part includes the major functional components of the receiver itself as described in this paper. The second part includes completion of the receiver, including data extraction and decoding, observables extraction, and the addition of a number of features including, but not limited to, WAAS navigational corrections, warm-start acquisition, a Kalman filter-based navigation solution implemented in C, and a graphical user interface.

The receiver hardware was developed primarily by Adam Shapiro, with assistance from Tom Chatt, in the Fall of 2009. The completion of the receiver and addition of features will be completed by Tom Chatt, with assistance from Adam Shapiro, in the Spring of 2010.

7 Acknowledgments

I would like to thank Professor Bruce Land for his endless support and advice. His help in the development of the receiver has been invaluable, and his unending knowledge and stories have kept me well educated, focused, and very entertained.

I would also like to thank Professor Paul Kintner, Professor Mark Psiaki, Brady O'Hanlon, and the Cornell GNSS Research Group for their help and support.

Finally, I would like to thank the Cornell University Department of Electrical and Computer Engineering for a terrific education and all of the opportunities to learn and grow it provided.

8 Appendix

All source code, tools, and documentation for this project are available on the web at the following URL: `http://cu-hw-gps.googlecode.com`. This project is open to the public, and is licensed under the GNU General Public License, version 2.

References

[1] Jack W. Crenshaw. Integer square roots. `http://www.embedded.com/98/9802fe2.htm`, 2008.

[2] Davicom Semiconductor, Inc. *DM9000A Ethernet Controller Datasheet*, May 2006.

[3] Maxim Integrated Products. *MAX2741 Integrated L1-Band GPS Receiver Datasheet*, January 2005.

[4] Pratap Misra and Per Enge. *Global Positioning System: Signals, Measurements, and Performance*. Ganga-Jamuna Press, 2 edition, 2006.

[5] Navstar GPS Space Segment. *Navstar Global Positioning System Interface Specification (IS-GPS-200), Rev. D*, March 2006.

[6] Trimble. How gps works. `http://www.trimble.com/gps/sub_phases.shtml`, 2009.

[7] Saurabh Verma, Ashima S. Dabare, and Atrenta. Understanding clock domain crossing issues. `http://www.embedded.com/columns/technicalinsights/205202134?_requestid=744`, 2007.

[8] J. F. Wakerly. *Digital Design: Principles and Practice*. Prentice Hall, 4 edition, 2006.