Development of a GPS Software Receiver on an FPGA for Testing Advanced Tracking Algorithms

by

W. Luke Edwards

A thesis submitted to the Graduate Faculty of
Auburn University
in partial fulfillment of the
requirements for the Degree of
Master of Science

Auburn, Alabama
August 9, 2010

Keywords: GPS, software receiver, FPGA, vector tracking, deep integration

Abstract

In this thesis, the development of an FPGA-based software GPS receiver with a special focus on advanced tracking algorithms is developed. The particular algorithms of note in this thesis are in a class known as vector tracking algorithms. Vector tracking GPS algorithms boast an increased immunity to interference and jamming and the ability to perform at low signal-to-noise (C/N_0) ratios. Addition of an inertial device to the vector tracking algorithm is known as deep integration and further boosts these benefits.

A trade study is presented that compares different hardware platforms for an embedded real-time system. An FPGA is chosen based on its ability to combine all of the necessary functions on a single device and its ability to seque a FPGA logic design to an application-specific integrated circuit (ASIC). Implementation details of each different component that constitutes a GPS receiver are given. In the single system, three soft-core microprocessors are synthesized on the FPGA to compute various components of the GPS algorithm, and their interfacing to other custom logic and to each other is described. The operation of each of the custom GPS logic modules is outlined in detail. Hardware resource utilization and computational timing results are also given. This thesis shows that a preliminary design of a real-time embedded GPS receiver capable of vector tracking is feasible, but there are more improvements to be made before deep integration is successful in real time.

Acknowledgments

I would like to first and foremost give every single ounce of credit to my Heavenly Father, the Lord Jesus Christ. He has been completely faithful to me in this process and throughout my life, and I have done absolutely nothing to deserve His grace in my life. My wife, Kristen, has been wonderful throughout the past two years as I have had to put many long hours into this research and thesis, and I could not have done it without her help and encouragement. I would like to thank my family and friends who have been so supportive, encouraging, and flexible with our strange schedules. I would like to particularly thank Dr. David Bevly for his time, encouragement, and mentoring while I have had the privilege of working under him. The financial support that I have received through Dr. Bevly has been such an incredible blessing, keeping me and my wife from having to take out further student loans. Other members of the GAVLAB have also been instrumental in the progression of this research. Dr. Nelson and Dr. Hung have also always been available for their technical expertise and career decisions, and I owe them both a huge debt of gratitude. I would like to particularly thank Ben Clark for his eagerness and willingness to join this project with me. I have personally never seen someone as driven, intellectual, systematic, and devoted to his friends and his work as I have of Ben. He has been an enormous blessing to me, in and out of work. Brad Dutton has also been an amazing source of knowledge dealing with FPGAs and their implementations. Much of this work would not have been possible without his help.

Table of Contents

List of Figures

List of Tables

Chapter 1
Introduction

1.1 Motivation

Since its inception, the Global Positioning System (GPS) has continued to become more and more ubiquitous. Applications in the commercial sector such as car and marine navigation devices, handheld hiking units, and running watches are commonly used by a number of different people to meet many different needs. Military uses include battlefield logistics, target tracking, and missile guidance. Research is currently being conducted that even uses GPS to guide unmanned vehicles through dangerous territory.

In most of these military applications, there is an undeniable need for a GPS receiver to supply precise, uninterrupted measurements that are immune to jamming. Relaying ground troop position coordinates and precise missile navigation are two notable instances that must be accounted for. GPS is a notably weak signal and is easily occluded by urban canyons and dense foliage. For these reasons and many others, it is important to have a GPS receiver in the field that can withstand these challenging scenarios.

1.2 Vector Tracking and Deep Integration

One method to deal with these difficulties is found in vector tracking algorithms. Vector tracking algorithms have been studied for the past two and a half decades [43]. The main advantages of vector tracking include an increased immunity to jamming, an ability to function in low signal-to-noise ratios, and, with inertial aiding, good performance in high dynamics. A GPS receiver that combines a vector tracking algorithm with an inertial measurement unit (IMU) is known as ultra-tight coupling or a deeply integrated (DI) system.

The inertial measurements used in DI systems increase the immunity to receiver dynamics and jamming even further. There has been a good deal of progress in the area of DI systems in the past few decades, but much of this success has been limited to post-processed data or non-embeddable real-time solutions. In order to use these very beneficial algorithms in the field, an embedded GPS receiver that supports vector tracking and deep integration must be designed.

1.3 Software Receivers

Since the late 1990s, researchers have studied the power and usefulness of a software-defined GPS receiver [3]. A software-defined GPS receiver, like any other software defined radio, is useful as a testing platform because it can be reconfigured any number of times in any number of ways. Defining the receiver implementation in software also allows explicit control over almost every signal passing through the system. A software platform like this opens up many opportunities to explore new algorithms and to test existing algorithms with little extra effort. Software receivers are normally run on a PC, DSP, or maybe a combination of both. In the past few years, there has been an increased interest in using the field-programmable gate array (FPGA) technology as a real-time software receiver platform [9, 29, 34].

1.4 Field-Programmable Gate Array

A field-programmable gate array (FPGA) contains many inherent qualities that make it an ideal platform for achieving this real-time performance. It offers the possibility of high parallelism, speed comparable to an application specific integrated circuit (ASIC), a large number of inputs/outputs (I/O), reprogrammability, and a great deal of design flexibility. An FPGA consists of programmable logic blocks (PLBs), input/output, and interconnects. At any time, an FPGA can be configured to have a specific system function. The system function is determined by activating some or all of the I/O pins, assigning certain logic functions to the PLBs, and using the interconnects to route information to/from the I/O and PLBs.

An FPGA was chosen as the implementation device for this software receiver because of its practicality as both a testing device and a final product. The main advantage of having an FPGA during testing is the fact that it is reconfigurable. Normally, one or many application-specific integrated circuits (ASICs) are used to build a commercial GPS receiver, but creating an ASIC for a design that needs to be reconfigured and verified multiple times is unrealistic and very costly. In fact, FPGAs are often used as a prototyping platform for designing ASICs. As a practical implementation device, an FPGA is a good choice due to its parallel processing capability, speed, and lack of need for a host PC or separate DSP for co-processing.

1.5 Contributions and Outline

Research in the general area related to this thesis has been mostly limited to seveal different categories. Some work describes the vector tracking and deep integration algorithms themselves [43]. Other research has been performed to compare the validity and performance gain of these algorithms [28, 35, 37]. A few different formulations of the deep integration techniques have been published along with their supposed performance benefits [5, 12]. Software receivers with their application to GPS began to be studied in the late 1990's [3]. There have since been numerous publications describing different GPS software receivers that are used for many different applications [7, 23, 25, 32, 22]. A few researchers have used an FPGA as a standalone hardware platform for a software receiver [34, 39, 13], and some have also used the model-based tools discussed later in this thesis [38, 8]. A very recent paper describes a real-time implementation of a deeply integrated GPS/INS device on a personal computer platform [26].

Besides the published works of the author [16, 17], there are currently no documented attempts of achieving real-time performance of these algorithms on an embedded platform. The goal of this thesis is to discuss the implementation details of a preliminary hardware/software architecture that uses a single FPGA to implement these advanced tracking algorithms. Hardware and software details of a hardware description language-based design will be presented. Timing performance and resource utilization will also be examined. This research is important because real systems which use vector tracking or deep integration are likely to be power, size, and cost-limited. A low-power, small-footprint, and cost-efficient receiver must be designed in order to meet these needs. However, designing embedded devices that meet these needs is complicated in themself, especially when these devices are constrained to real-time performance.

Specific contributions detailed in this thesis are a trade study of different hardware platforms to implement a real-time embedded GPS receiver, a prototype system built using an FPGA, and some preliminary timing and resource utilization results of the prototype system. Chapter 2 provides an overview of the GPS signal structure. Because of the nature of the vector tracking algorithm, the designer cannot take only the position and velocity output of a commercial GPS receiver. Instead, the designer must access the GPS signal immediately after it is sampled by a hardware front end's analog to digital converter (i.e. the bit-level). Chapter 3 describes the hardware and software processes of a typical GPS receiver. Each part of the receiver is duplicated in the proposed hardware platform, so it is important to have an understanding of these functions. Chapter 4 is a trade study of different potential hardware platforms that can be used to implement this embedded real-time receiver with a focus on the chosen platform (FPGA). Chapter 5 describes the prototype system which uses hardware description languages to model its fundamental GPS functions. The prototype uses a 32-bit microprocessor that is synthesized into the FPGA fabric to perform the receiver's baseband functions. This microprocessor is programmed using C and C++. Chapter 6 gives some preliminary timing results and a report of the resource utilization for the prototype. This chapter also summarizes the contributions of this thesis and makes suggestions for future work regarding this research.

Chapter 2

GPS Signal Structure

In order to design a full-fledged software GPS receiver, it is necessary to understand the GPS signal structure sent from each satellite vehicle (SV) and received by the user equipment.

2.1 Modulation Techniques

The GPS signal employs a modulation technique called Binary Phase Shift Keying (BPSK), which takes an RF carrier and either leaves it unmodified or reverses the phase of the signal by 180° based on a bipolar signal (±1) [24]. This effectively means that wherever the bipolar signal changes from a +1 to a -1 or vice versa, the RF carrier reverses its direction. Figure 2.1 shows an example of BPSK modulation with two bipolar signals, PRN code (defined later) and navigation data.

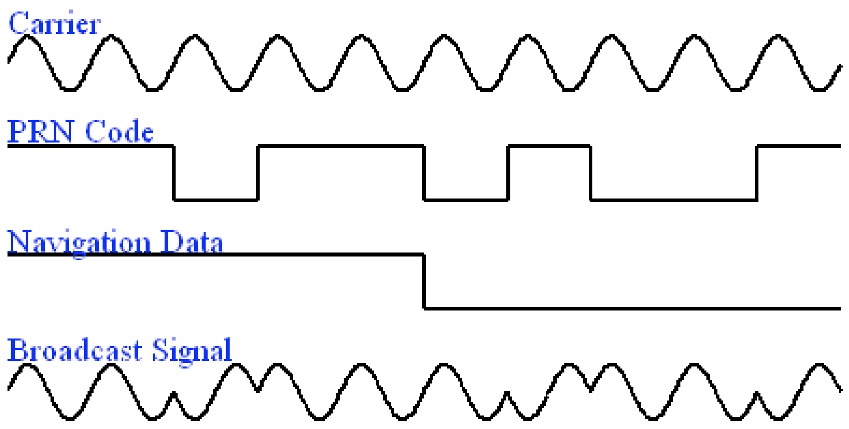

Figure 2.1: BPSK Modulation

In actuality, the GPS signal uses this same idea in a modulation technique called Quadrature Phase Shift Keying (QPSK). The 180° phase shift modulation is the same, but the difference lies in the fact that QPSK gives two separate outputs - the original modulated carrier (known as the in-phase component) plus another modulated carrier that is 90° out of phase from the first signal (quadra-phase) [24]. These can be thought of as sine and cosine outputs of a waveform generator. Figure 2.2 shows an example of QPSK modulation.

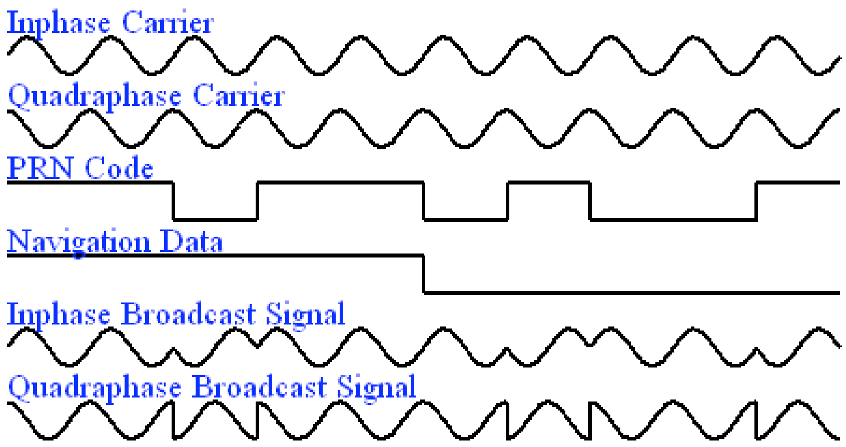

Figure 2.2: QPSK Modulation

The GPS signal structure relies heavily on a modulation technique known as Direct Sequence Spread Spectrum (DSSS). This extends the idea of BPSK by using a high-rate spreading signal known as a Pseudo-random noise (PRN) waveform to spread the signal over a wider bandwidth [30, 24]. Generation and properties of this PRN waveform will be discussed later, but it is important to note that the PRN sequence is periodic and must be known *a priori* in order to use. Each bit in the PRN sequence is called a chip, and the rate that the code is transmitted is known as the chipping rate . The purpose of DSSS is twofold [24]:

1. To make precise ranging possible

2. To allow multiple satellites to broadcast on the same frequency using a technique known as code division multiple access (CDMA)

CDMA is especially important to GPS because it allows the receiver to distinguish between the different satellites in view at any particular time [24].

2.2 Transmitted and Received Signals

Equation (2.1) gives the signal received by an antenna in a receiver (neglecting transmitted signal power).

$$s(t) = C(t)D(t)cos(2\pi f_{L1}t + \phi_{L1}) + P(Y(t))D(t)sin(2\pi f_{L1}t + \phi_{L1}) \\ +P(Y(t))D(t)sin(2\pi f_{L2}t + \phi_{L2}) \tag{2.1}$$

Frequencies f_{L1} and f_{L2} represent the L1 and L2 carrier frequencies (1575.42MHz and 1227.60MHz, respectively) that are in the dedicated GPS band frequency range [7]. Signals $C(t)$ (coarse acquisition) and $P(Y(t))$ (precise encrypted) are the aforementioned code spreading waveforms for the civilian and military signals, respectively. Because the GPS receiver designed in this thesis uses only the civilian band signal, the high frequency P(Y) code and L2 bands are filtered out and are no longer considered. After filtering, the data is converted into some intermediate frequency (IF). This process will be discussed in Section 3.1, but it is enough for now to understand that the incoming signal to the software receiver is in the form of Equation (2.2) (again, neglecting signal power) [30].

$$s_i(t) = C(t + \delta_i)D(t + \delta_i)cos(2\pi(f_{IF} + f_{dopp_i})t + \phi_i) \tag{2.2}$$

where i represents the ith satellite that is currently in view of the receiver.

The received signal looks very similar to the transmitted signal from each satellite with a few exceptions. The signal is still BPSK-modulated by the C/A code and data bit signals, but the carrier frequency has been downconverted to some intermediate frequency (f_{IF}) and is offset by a Doppler frequency that is unique for each satellite. This Doppler frequency offset is a function of the relative motion between the user and the particular satellite vehicle in question. This effect is well documented in many communications textbooks and is a very important concept for GPS for determining user velocity [11]. The term ϕ_i is the phase offset between the transmitted and received signal frequencies. The phase of the signal (and therefore the frequency) is usually tracked by a phase-locked loop (PLL) in the receiver hardware [7]. The term δ_i is the code phase of the C/A code. This quantifies the time-alignment of the received C/A code and is used for determining precision ranges to each satellite [24].

2.3 C/A Code Details

The coarse acquisition (C/A) code is a critical aspect of the GPS signal and must be explored in further detail. It is transmitted only on the L1 frequency at a "chipping rate” of 1.023MHz (each bit in the sequence is called a chip). The C/A code sequence repeats every 1023 chips, which corresponds to a sequence period of 1ms. The purpose of the C/A code is twofold [24]:

1. To allow multiple satellites to broadcast on the same carrier frequency

2. To acquire precise ranging information to each satellite

As discussed previously, the C/A code BPSK-modulates the GPS signal and is used as a direct sequence spread spectrum (DSSS) signal; that is, one of its purposes is to widen the bandwidth of the received signal. The C/A codes were carefully chosen from a family of pseudo-random noise (PRN) sequences known as Gold codes [19] because of the following two properties:

1. Autocorrelation with identical signal only when time-aligned (Figure 2.3) [7]

$$r_{kk}(m) = \sum_{b=0}^{1022} C_k(b)C_k(b+m) \approx 0 \qquad \text{for } |\text{m}|\neq 0$$

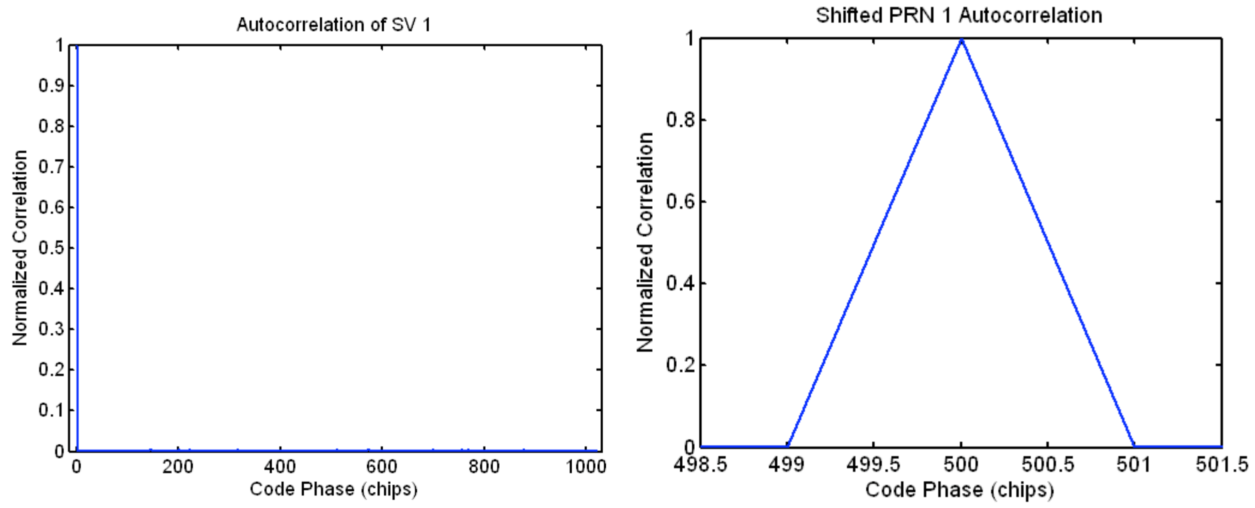

Figure 2.3: Autocorrelation of PRN 1 with no time shift (left) and 500-chip shift (right)

2. Cross-correlation with other Gold codes ≈ 0 (Figure 2.4) [7]

$$r_{jk} = \sum_{b=0}^{1022} C_j(b)C_k(b+m) \approx 0 \qquad \text{for all m}$$

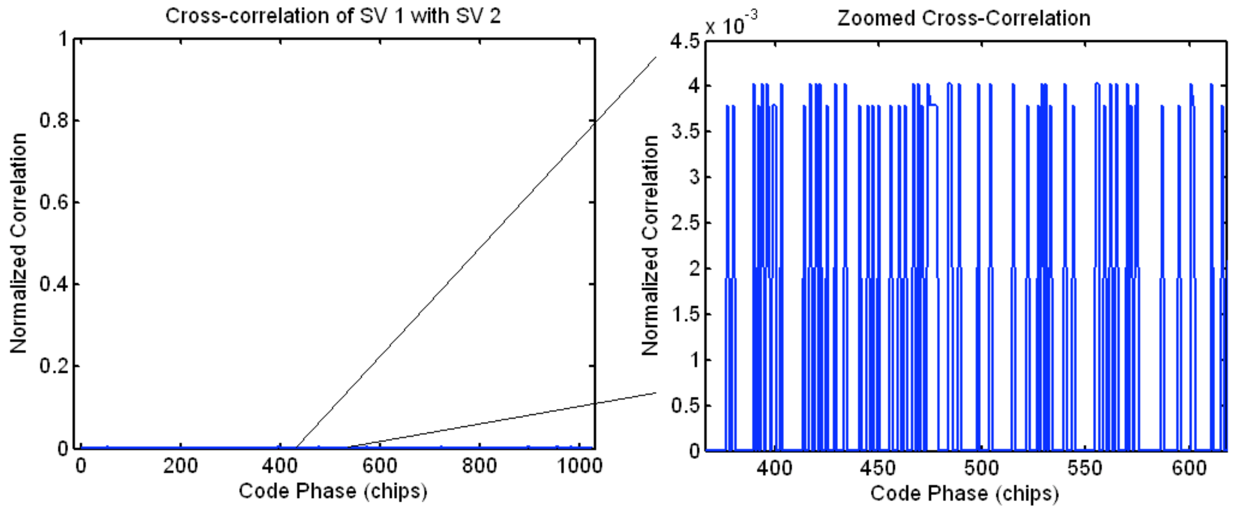

Figure 2.4: Cross-correlation of PRN 1 and PRN 2

This is very important for both of the C/A code purposes mentioned above. First, each of the satellites broadcasts a unique C/A code which is orthogonal to every other satellite's C/A code. Due to this orthogonality, a GPS receiver can determine which satellites are in view of the receiver with near certainty. This process, called signal acquisition, will be outlined in Section 3.2. These Gold code properties (along with DSSS) help to fulfill the second purpose of the C/A code: precise ranging. "De-spreading" the C/A code is accomplished by multiplying the incoming signal by the C/A code in a process called signal tracking that will be described in Section 3.3. When "de-spreading" the C/A code to read off the data bits, the code must be perfectly time aligned; that is, the code phase must be correct between the received signal and the generated replica code [24]. Once the code phase is known, it can be compared to every other visible satellite's code phase to see which satellite is closest to the receiver and compute a position accordingly. This process will be outlined in the position calculation section (Section 3.4).

2.3.1 C/A Code Generation

As mentioned before, each satellite's C/A code is unique. However, because they all belong to the same family of Gold codes, they use the same generation scheme. The C/A code generator employs two linear feedback shift registers (LFSRs) called G1 and G2. Each of these shift registers is 10-bits wide and generates a repeating 1023-chip long pattern. The linear feedback portion of G1 and G2 is characterized by the polynomials in Equations (2.3) and (2.4) which correspond to the "tap settings" for that shift register [7].

$$f_{G1}(x) = 1 + x^3 + x^{10} \tag{2.3}$$

$$f_{G2}(x) = 1 + x^2 + x^3 + x^6 + x^8 + x^9 + x^{10} \tag{2.4}$$

Each of these "taps" is fed into an exclusive-or (XOR) gate which is then fed back into the first position of the shift register. This concept is illustrated for G1 in Figure 2.5.

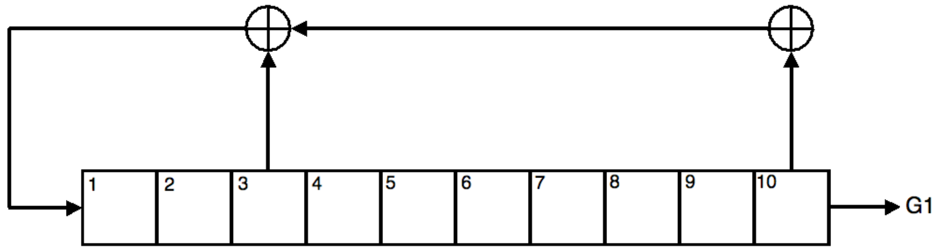

Figure 2.5: Shift register generator for G1 ($G1 = 1 + x^3 + x^{10}$)

Test pattern generation for built-in self test (BIST) platforms use this same technique to generate pseudo-random test patterns [1].

G1 and G2 both output their respective 1023-chip sequences repeatedly, so in order to create unique sequences for each satellite, G2 is shifted by a unique number of chips and then modulo-2 added (using an XOR operation) to G1. This can be done one of two different ways. First, the full 1023-chip G1 and G2 sequences can be generated and then modulo-2 added together, or second, a selection of two different registers called the phase selection can be applied to G2 and then modulo-2 added to G1 [7]. The first option can be chosen when hardware resources can be sacrificed to enjoy simplicity. The second option can be employed when logic space (or program memory) must be conserved, but it comes with a higher degree of complexity. A third option is to generate all of the C/A sequences *a priori*; this uses the largest amount of resources but requires the least amount of algorithmic complexity. The second option is most commonly done to save resources, and is depicted in Figure 2.6.

2.4 Data Bits

As mentioned before, data bits transmitted by each satellite BPSK-modulate the outgoing signal. The purpose of these data bits is to give clock corrections and precise orbital information called ephemeris data which allow a receiver to calculate the satellite's position and velocity at any given time [30, 7, 24]. These data bits are transmitted every 20ms (50Hz)

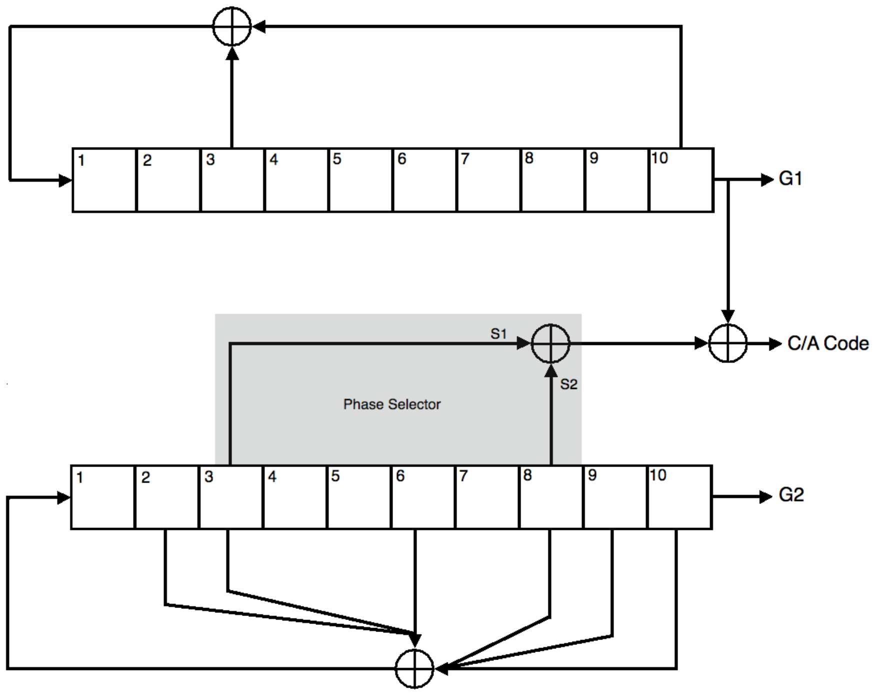

Figure 2.6: Entire C/A code generator

and each bit begins exactly at the start of a C/A code sequence. A data bit is 20 times longer than the 1ms C/A code sequence.

The data structure is organized into fifty frames with five subframes each. Each of the subframes contains 300 bits organized in ten 30-bit words. Since the data rate is 50Hz, the entire data structure takes 12.5 minutes to complete before cycling through again [7]. Subframes 1, 2, and 3 repeat the same information for all 50 frames. A single frame of the GPS data structure is pictured in Figure 2.7.

Subframe 1 contains satellite clock correction terms (to correct for relativistic effects) and satellite health information [24]. Subframes 2 and 3 contain information regarding that particular satellite's orbital parameters. This is known as the ephemerides or ephemeris data. This includes harmonic data and other orbital parameters that, when processed, yield a very accurate solution of the satellite's position and velocity. Knowing this data with high precision will, in turn, yield a high precision user position. The other two subframes contain information known as almanac data [24]. This holds information about the entire

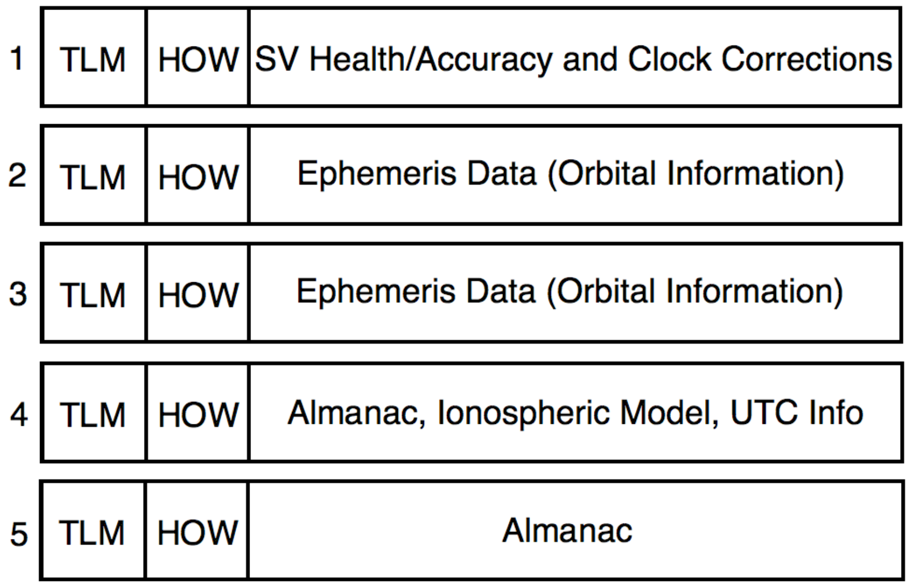
1
TLM
HOW
SV Health/Accuracy and Clock Corrections
2
TLM
HOW
Ephemeris Data (Orbital Information)
3
TLM
HOW
Ephemeris Data (Orbital Information)
4
TLM
HOW
Almanac, Ionospheric Model, UTC Info
5
TLM
HOW
Almanac

Figure 2.7: Navigation Data Message

constellation of satellites and rough estimates of where each of them are at a given point in time. This knowledge is often used by GPS receivers to facilitate a warm start, whereby the receiver begins with some knowledge of which satellites will be in view and their approximate locations when the receiver is initially powered on, making it possible to obtain a solution more quickly. This is one method of reducing the time to first fix, or TTFF.

Each subframe begins with a telemetry (TLM) and handover word (HOW). The TLM contains a known 8-bit sequence called the preamble that is used to synchronize the frame data and also to determine which satellite is closest in distance to the receiver [7]. It is because the TLM is repeated every subframe that the GPS receiver does not take 12.5 minutes to initialize. The HOW contains a truncated version of the time of week (TOW) and a few other flags. The time of week is used to measure when particular data frames were sent from the satellite.

2.5 Utilization of the GPS Signal

Many of the concepts about the GPS signal structure have been discussed in this chapter. The main purpose of a GPS receiver is to give accurate position and velocity of the user. In order to do this, the GPS signal must be properly processed by the user equipment. In order to demodulate the data bits in each signal, three important signal elements must be known [3, 24].

1. Satellites currently in view at the receiver antenna
2. Code phase of each C/A code for satellites in view
3. Doppler frequencies of each satellite in view

If, for a particular satellite in view, the code phase and Doppler frequencies are tracked perfectly, the data bits in the signal become accessible. Once these data bits are accessible for at least four satellites, a position solution can be calculated. The next chapter outlines the detailed hardware and processing modules of a typical GPS receiver that accomplish the above tasks.

Chapter 3

Overview of a Typical GPS Receiver

3.1 RF Front End

As mentioned in Section 2.2, the L1 (civilian band) GPS signal is transmitted at a carrier frequency of 1.57542 GHz, but signal processing at such a high frequency is very difficult with current technology [7, 3]. For this reason, a hardware device known as a GPS front end is used to shift the L1 frequency signal down to a more manageable rate. This is accomplished in the front end with a series of filters, downconverters, and an analog-to-digital converter (ADC). The purpose of filtering is to attempt to eliminate all of the unusable portions of the incoming signal. That is, since the usable portion of the signal lies within a particular bandwidth and the antenna is not ideal, the filter selects only that part of the signal that contains GPS data and attenuates the rest [7]. The goal of downconversion is to take the aforementioned high frequency L1 signal and shift it into an intermediate frequency (IF) that can be handled by the ADC. The ADC takes digital samples of the analog signal that will be used by a subsequent processing unit.

The hardware front end provides data and clock signals to whatever component that interfaces it. The clock that is provided is the sampling clock at which the data bits are sampled. The data signals are the digital bits on the output of the ADC, and the width of the data depends on how many bits are used to quantize the data. For this research, 2-bit quantization is used and follows a standard sign and magnitude convention as shown in Table 3.1[40].

Due to the nature of the GPS signal, it is often advantageous to only use one-bit quantization (outputs ± 1) [46]. Even with the signal downconverted to some intermediate frequency, a large amount of signal processing is still required. Using only one bit reduces

Table 3.1: Observed output values on hardware front end

Sign	Magnitude	Output Value
0	0	+1
0	1	+3
1	0	-1
1	1	-3

the incoming data rate by 100%, requires smaller memories, and uses less resources when executing arithmetic operations. However, two-bit quantization gives a higher degree of accuracy for reconstructing the original signal (an effective 1.41dB gain) [42]. This can potentially lead to higher signal integrity and therefore more accurate results.

There are two main hardware front end chips which are popular in the software GPS receiver community. The first and older of the two is the Zarlink GP2010/2015 chip family. This uses 2-bit quantization (sign/magnitude) with an intermediate frequency of 4.309 MHz and a sampling clock of 5.714 MHz [52, 53]. This particular front end is useful for digital signal processors and software receivers that require a low sampling rate. In contrast, one of the more modern front end chips is the SiGe 4110L/4120L. This chip employs 2-bit quantization at an IF of 4.092 MHz and a sampling frequency of 16.3676 MHz, although these values are configurable to suit different applications [40]. The 4110L is touted as a high sensitivity GPS-only device while the 4120L claims to be the world's first GPS/Galileo ready receiver [40]. This research uses the SiGe 4110L as the hardware front end.

It can be noted that any front end chip that uses more than one data bit, such as the SiGe 4110L used in this research, can reduce the signal bandwidth by taking only the sign bit of the quantized data. This effectively reduces the signal from a multi-bit ADC to a one-bit ADC.

3.2 Acquisition

Once the high frequency signal has been converted to digital samples by the front end, the software receiver begins the next stage of the process; this stage is called signal acquisition. The purpose of signal acquisition is three-fold [30]:

1. Find which satellites are in view from the current GPS antenna position
2. Obtain a rough estimate of code phase for observed satellites
3. Obtain a rough estimate of Doppler frequency for observed satellites

These rough estimates are used in the subsequent stage of signal processing (tracking). There are many different approaches to finding these rough estimates [7]. First, a serial search over all possible code phases and Dopplers can be performed. This can take the longest amount of time, but has a very low complexity. A second method called a parallel frequency space search acquisition method is also available. This searches all of the frequencies simultaneously (using a Fourier transform) and searches each code phase serially. Because of the Fourier transform, implementation is more complex than the purely serial approach. The third and final option is called parallel code phase search acquisition. This method searches all of the code phases simultaneously by using multiple Fourier transforms, while the Doppler frequencies are searched serially. This approach is much more complicated than the other two, but is the most time-efficient. Due to the resource intensity of this method, much research has been done to mitigate the complexity of these algorithms [15, 39].The first and third acquisition options will be discussed in this thesis. The second option is not discussed because this method is not typically seen in software receivers; this is perhaps because the frequency resolution is poor compared to the other two methods [7].

The acquisition search space consists of at least 1023 code phases (one for each chip) and several kHz of Doppler offset. For an illustrative purpose, half-chip code phase spacing (totaling 2046 possibilities) and a Doppler bin that spans ± 5000 Hz from the center (IF) frequency in increments of 200 Hz is used (for a total of 51 different total Doppler frequency bins) on data collected from a front-end data recorder. Figure 3.1 shows a plot of the entire search space for satellite 31 using the serial search. An obvious peak at a Doppler of +800Hz and code phase of about 820 chips can be seen. The z-axis is the output value of the integrate and dump operation for 1ms of data.

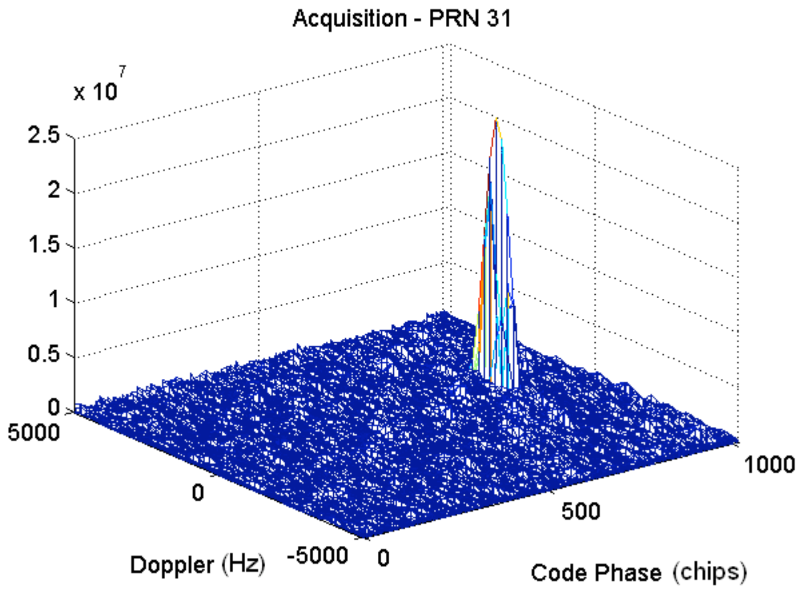

Figure 3.1: 3D plot of acquisition results

3.2.1 Serial Search Acquisition

The first acquisition method is known as serial search acquisition [24]. The reason it is called a serial search is because, given no additional information, each possible code phase and Doppler must be independently searched. The code phase is typically searched in half-chip intervals (totaling 1023*2 = 2046 total code phase bins). The Doppler frequency must be searched in predefined increments around the intermediate frequency. The Doppler search increment spacing is based on expected signal to noise ratio (C/N_0, a measure of how much a signal is corrupted by noise); the worse the expected C/N_0, the smaller the frequency increments that must be searched [24]. Also, the range on either side of the IF that must be searched depends on the expected user dynamics. The higher the receiver dynamics, the higher the potential Doppler effect [11]. In high dynamic situations, a typical Doppler search increment is about 10kHz around the center frequency. However, for this example, a normal dynamic situation is assumed, and therefore a range of ±5 kHz will be used with a 200Hz increment. This corresponds to 51 total Doppler bins ($2\frac{5000}{200}+1$). Therefore, in order to search every code phase and Doppler possibility sequentially, a total of 2046*51=104,346 different correlations must be performed.

This is a straightforward approach to the signal acquisition problem illustrated in the block diagram found in Figure 3.2. The incoming signal is multiplied by the generated C/A code which is then multiplied by the in-phase and quadra-phase portions of the local oscillator. Over a predetermined period (called the predetection time), the generated signal is integrated by both phases of the oscillator. In order to get an accurate picture of signal strength, both sides are squared and then summed together [7]. At this point, the output is compared to some threshold that determines whether or not a satellite is detected at that code phase and Doppler [24].

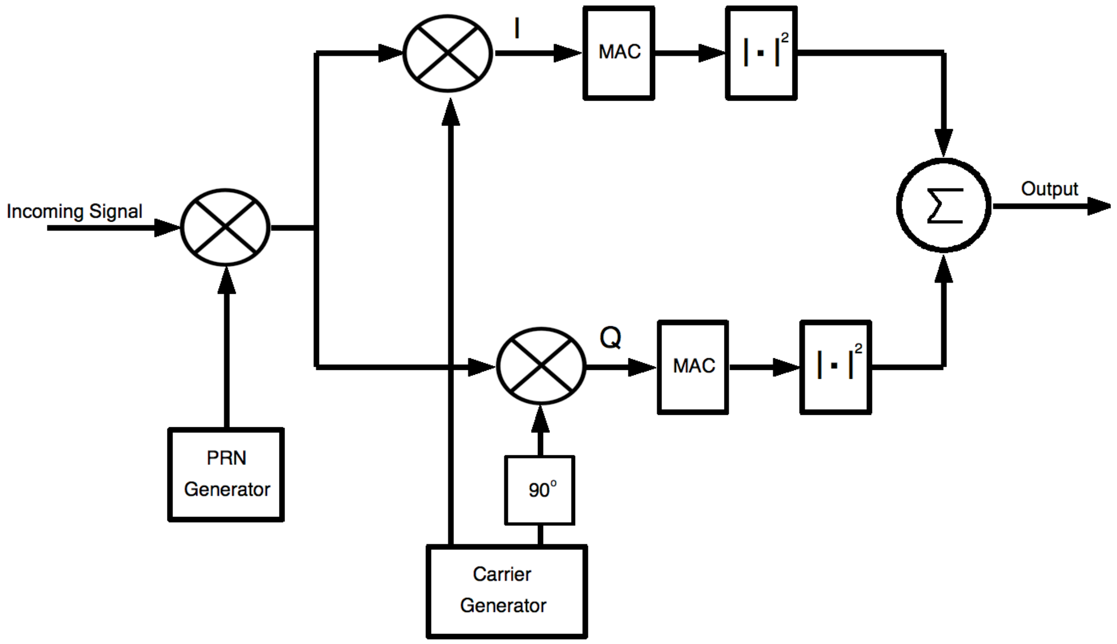

Figure 3.2: Serial search acquisition block diagram

If only one of these serial search modules is used, it can take a considerable amount of time to find the correct code phase and Doppler combination. One potential technique to improve acquisition time is to use multiple signal correlators. For example, instead of multiplying a single code phase by a particular Doppler frequency, three different code phases can be used, potentially improving the acquisition speed by three times. Many modern receivers are cited to use thousands of correlators in order to improve signal acquisition time [18]. Another technique that can improve the complexity of the correlators is to use only the sign bit of the incoming signal (1-bit ADC). This requires far fewer hardware resources than a 2-bit multiplier.

Serial search acquisition is mainly used by application-specific integrated circuit (ASIC) or field-programmable gate array (FPGA) GPS devices. This is because the parallel processing capabilities of these types of hardware can easily implement a large array of correlators at relatively little hardware cost. The more advanced parallel code phase search acquisition algorithm is usually done on microcontroller-driven software receivers.

3.2.2 Parallel Code Phase Search Acquisition

The parallel code phase search acquisition approach reduces the total search space and computation time as compared to the serial search method, but this comes at the cost of overall complexity [7]. In the serial search, every half-chip must be searched separately, resulting in 2046 total separate code phase bins. As the name suggests, the parallel code phase search acquisition approach searches all of the code phases simultaneously. This effectively means that the number of search iterations is reduced by a factor of 2046, which is a drastic improvement. In the example above, the code phases are completely searched every iteration, and only the 51 Doppler bins must be independently checked. This amounts to a maximum total of 51 different combinations.

This functionality is made possible by computing this cross-correlation in the frequency domain [7]. Figure 3.3 is a block diagram of this functionality. A fast Fourier transform

(FFT) is used to convert time-domain signals to the frequency domain. This method takes the discrete fast Fourier transform of both the carrier-wiped input (the incoming raw signal modulated by the carrier replica) and the C/A code, conjugates the C/A code branch, and then multiplies the two together. An inverse fast Fourier transform is used to shift the entire system back to the time domain. This method of cross-correlation in the frequency domain is well understood in digital signal processing [31]. Once the system is back to the time domain, the absolute value of the output is squared to get an accurate estimate of signal power. This signal power is compared to a threshold in order to determine if a signal is present at the Doppler being search [24]. The output will peak at the proper code phase that the signal lies within.

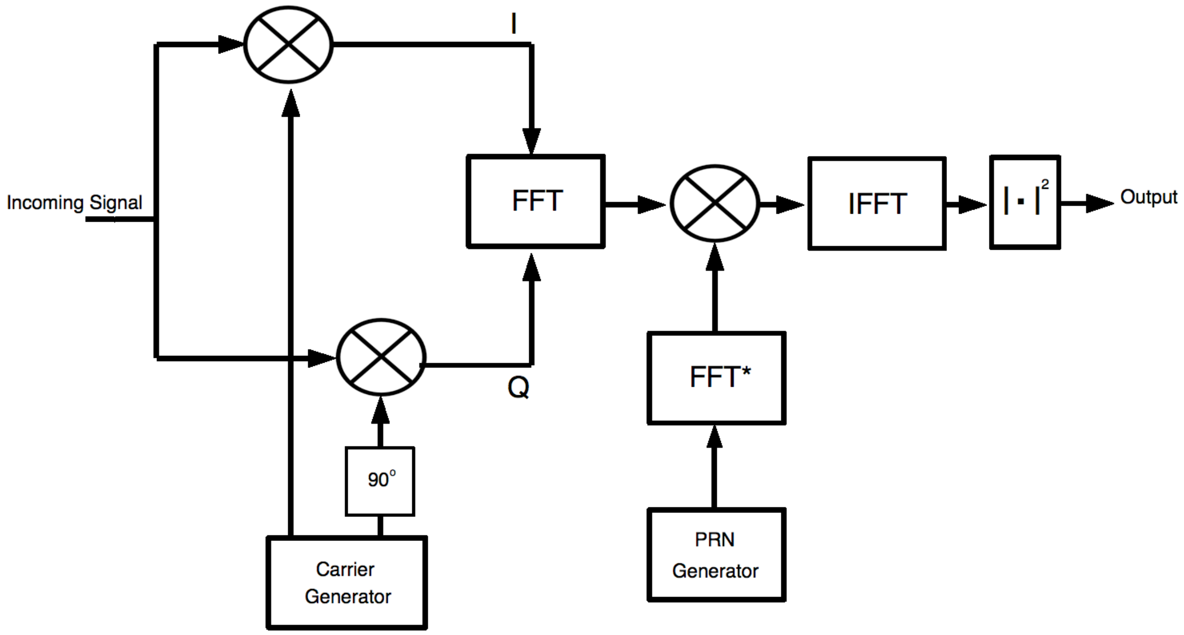

Figure 3.3: Parallel code phase search acquisition

Because the fast Fourier transform is a discrete signal processing technique, the size of the FFT is based on the number of samples in the predetection time. In a system with a sample clock of 16.3676 MHz, there are about 16,368 samples per millisecond. In ASIC or FPGA design, the ability to perform a fast Fourier transform on a single 16,368-sample input is very hardware-demanding because there needs to be a large number of hardware multipliers

[31]. This acquisition method requires two fast Fourier transforms, one inverse fast Fourier transform, and other control hardware; therefore it is a very resource intensive approach in these types of hardware. In contrast, this method is very well suited for software receivers that are run on microprocessors. The ability to store large amounts of data in memory and then operate on them sequentially is very beneficial for these types of systems.

3.3 Signal Tracking

The purpose of the tracking stage of the GPS receiver is to fine-tune the Doppler frequency and code phase of the incoming signal for a particular satellite [7]. In doing this, the C/A code modulation is removed, the carrier is wiped off, and the data bits become accessible. This information is all that is needed to determine the pseudorange (discussed in Section 3.4) and therefore the user position.

3.3.1 Basics of a Phase-Locked Loop

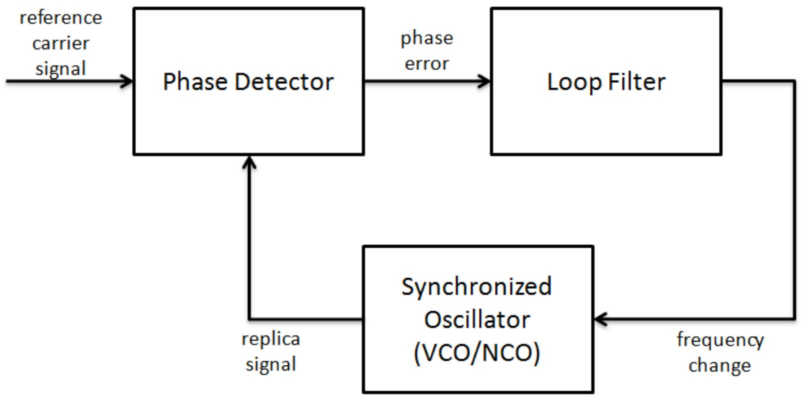

Figure 3.4: Basic PLL block diagram

The phase-locked loop (PLL) shown in Figure 3.4 is the foundation of a tracking loop and includes three basic parts [6]. The first part of the PLL is a synchronized oscillator. This oscillator generates a replica of the system's carrier (or intermediate) frequency based on its inputs. In practice, this oscillator is usually voltage-controlled (taking an analog voltage as its input) or numerically-controlled (using discrete data points as the input). The next part of the phase-locked loop is the phase detector. This component compares the reference signal with the output coming from the synchronized oscillator. Comparing the two involves multiplying the replica and reference signals together. If the phase and frequency match, the carrier wave is essentially "wiped off" except for a high frequency component (which will be filtered out). This can be seen from the trigonometric product-to-sum identity in Equation (3.1).

$$cos(\theta)cos(\psi) = \frac{1}{2}[cos(\theta - \phi) + cos(\theta + \phi)] \tag{3.1}$$

At this point, the phase error in the signal has been detected and the high frequency duplicate must be filtered out. This is usually a low-pass filter, taking multiple error signals from the phase discriminator as its input and filtering out the high frequency noise to give a more accurate picture of the error [7]. In practice, this is accomplished by integrating the signal over a certain period of time, amounting to a series of multiply and accumulate operations. Once the signal has been filtered, the outputs are sent to the loop filter. The loop filter, as previously mentioned, converts this error into a form that the synchronized oscillator understands. Based on the phase error, the synchronized oscillator will increase or decrease its frequency to better match the reference. This closes the loop and provides precise frequency information to the navigation processor.

3.3.2 Costas Loop

Recall from Section 2.1 that both the ranging codes and the data message use QPSK modulation. This means that anytime the bit changes from a 1 to a 0 (or a 0 to a 1), both sinusoidal signals shift in phase by 180°. If a normal phase-locked loop were used, the synchronized oscillator would suddenly receive information that the signal that it is attempting to replicate suddenly changed by 180°[30]. This would cause very erratic behavior when trying to match the phase.

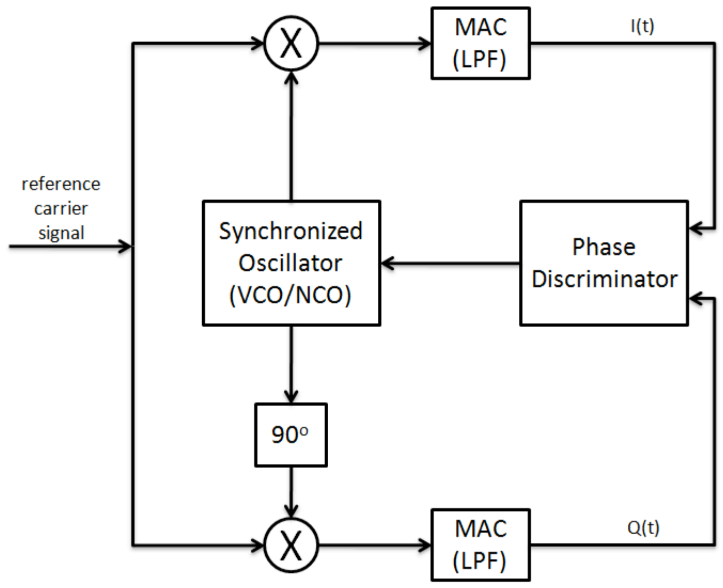

Figure 3.5: Costas loop block diagram

Because of this potential 180° shift, a modified PLL called a Costas loop is used as shown in Figure 3.5. The Costas loop splits the incoming signal into two branches (spaced 90° apart) [6]. The two branches are called the I (in-phase) and Q (quadrature) branches. Each branch contains independent components of a PLL, but they are joined together at the phase discriminator. The carrier replica is multiplied by the reference and then low-pass filtered in both branches. The outputs of each of the low-pass filters in each branch are sent into an advanced loop filter known as the phase discriminator. This part is responsible for calculating the error in phase between the replica and real signal using both branches of the Costas loop. There are a number of different phase discriminators that could be used (see Table 3.2), and each of them has advantages and disadvantages in terms of complexity and accuracy. In practice, the most commonly used discriminator is the 2-quadrant arctangent function [24].

For GPS, a predetection integration period of 1ms to 20ms is used. This means that data is collected and integrated for that length of time. Choosing the integration period depends on how well the signal is currently being tracked. Loosely-locked signals favor the shorter

Table 3.2: Common Costas loop discriminators [24]

Name	Discriminator	Output Error
Classic Analog	$Q_P I_P$	$sin(2\phi)$
Decision-directed	$Q_P sign(I_P)$	$sin\phi$
Tangent	$\frac{Q_P}{I_P}$	$tan\phi$
2-Quadrant Arctangent	$atan(\frac{Q_P}{I_P})$	ϕ

predetection integration times so that the filter can react to sharp changes, while tightly tracked signals favor the longer predetection integration times. Once the integration takes place, both branches of the Costas loop send the final result to the arctangent discriminator that was discussed above. The output of the carrier phase discriminator must be converted into a form that the synchronized oscillator understands. This depends on the discriminator output and the type of oscillator that is being used.

3.3.3 Basics of a DLL

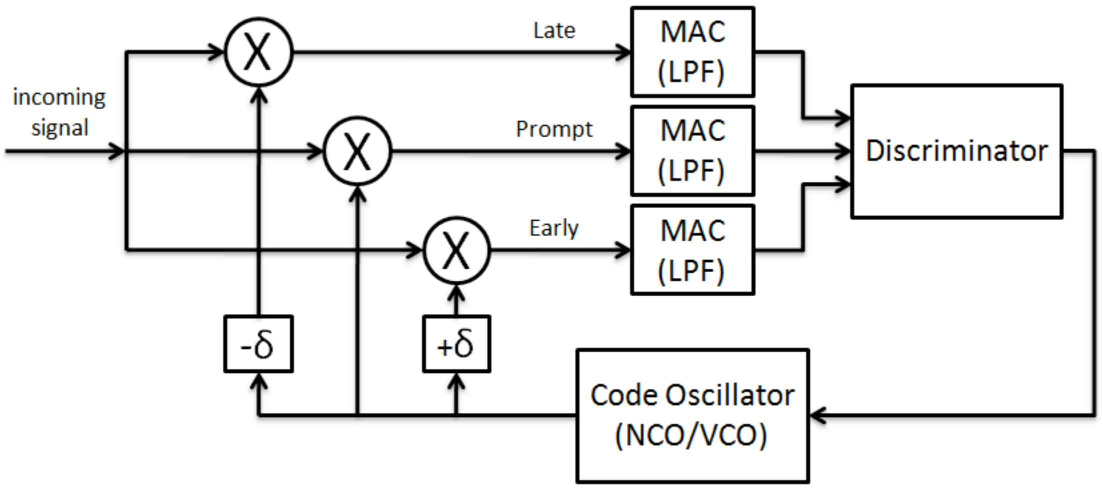

Figure 3.6: Basic delay-locked loop block diagram

A delay-locked loop, or DLL, is a modified version of a PLL and is used to track the code phase of the C/A code. A picture of a DLL that is used in GPS code tracking loops can be found in Figure 3.6. The DLL operates as follows. First, the received Gold code is split up into three branches - prompt, early, and late. The prompt branch provides the best estimate of the phase/delay of the Gold code in the received signal. The early branch is spaced δ chips in front of the prompt branch, and the late branch is spaced δ chips behind the prompt branch. The variable δ is called the correlator spacing and carries a typical value of 1/2 chip. These three replicas are created for both the in-phase (I) and quadrature (Q) branches, resulting in a total of six replicas.

Recall from Section 2.3 that the autocorrelation of the Gold codes is at its maximum when the reference and replica Gold codes are time-aligned. From that center point, the correlation reduces linearly in magnitude until it reaches almost pure noise. This triangular-shaped correlation is illustrated in Figure 3.7. The early, prompt, and late outputs of the DLL are also shown in the figure for purposes of illustration.

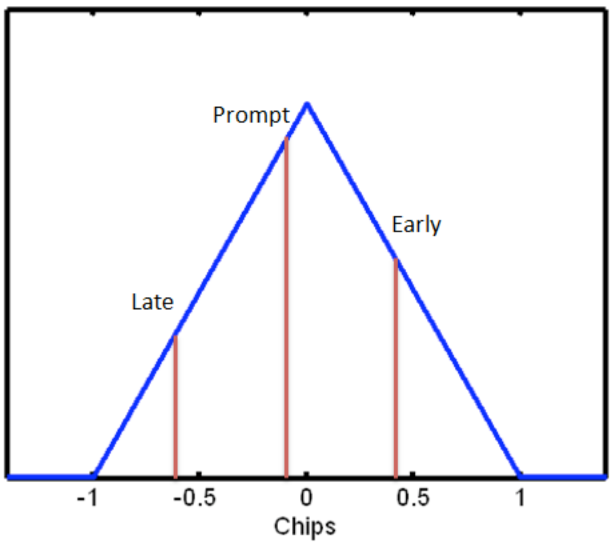

Figure 3.7: Triangular autocorrelation function with early, prompt, and late outputs

The DLL, using the six replicas, nominally spaced 1/2 chip apart, serve to track the maximum peak of that triangular-shaped correlation. Because of this triangular shape, the early and late replica must essentially be equal in order for the prompt replica to match up with the correlation peak [7]. There are a number of different types of code phase discriminators that perform this function, and they each have their advantages and disadvantages in terms of complexity and accuracy. Table 3.3 outlines several of the most commonly used code phase discriminators used in GPS.

Table 3.3: Common Code Phase Discriminators [7]

Name	Type	Discriminator
Early-Late	Coherent	$I_E - I_L$
Early-Late Power	Non-Coherent	$(I_E^2 + Q_E^2) - (I_L^2 + Q_E^2)$
Normalized Early-Late Power	Non-Coherent	$\frac{(I_E^2+Q_E^2)-(I_L^2+Q_E^2)}{(I_E^2+Q_E^2)+(I_L^2+Q_E^2)}$
Dot Product	Non-Coherent	$I_P(I_E - I_L) + Q_P(Q_E - Q_L)$

The output of these discriminators must usually be normalized and then converted to a form that controls the rate of the Gold code generator. This rate is controlled by a local oscillator (either NCO or VCO). The DLL follows the same predetection time parameters as discussed in Section 3.3.2.

3.3.4 Entire Tracking Loop

For GPS, the carrier-tracking PLL (Costas loop) and code-tracking DLL must be combined into a single structure. This structure has the form of Figure 3.8. Combined, the carrier frequency of the system is tracked in conjunction with the code phase [7]. Keeping a lock on each of these enables the GPS receiver to obtain the data bits needed to calculate GPS position.

3.4 Position Solution

After first finding a rough estimate of the Doppler frequency and code phase, the tracking stage makes fine adjustments to those estimates and the final output is the data bits in the signal. Using these data bits and the properties of how they are sent allows a receiver to determine its position and velocity.

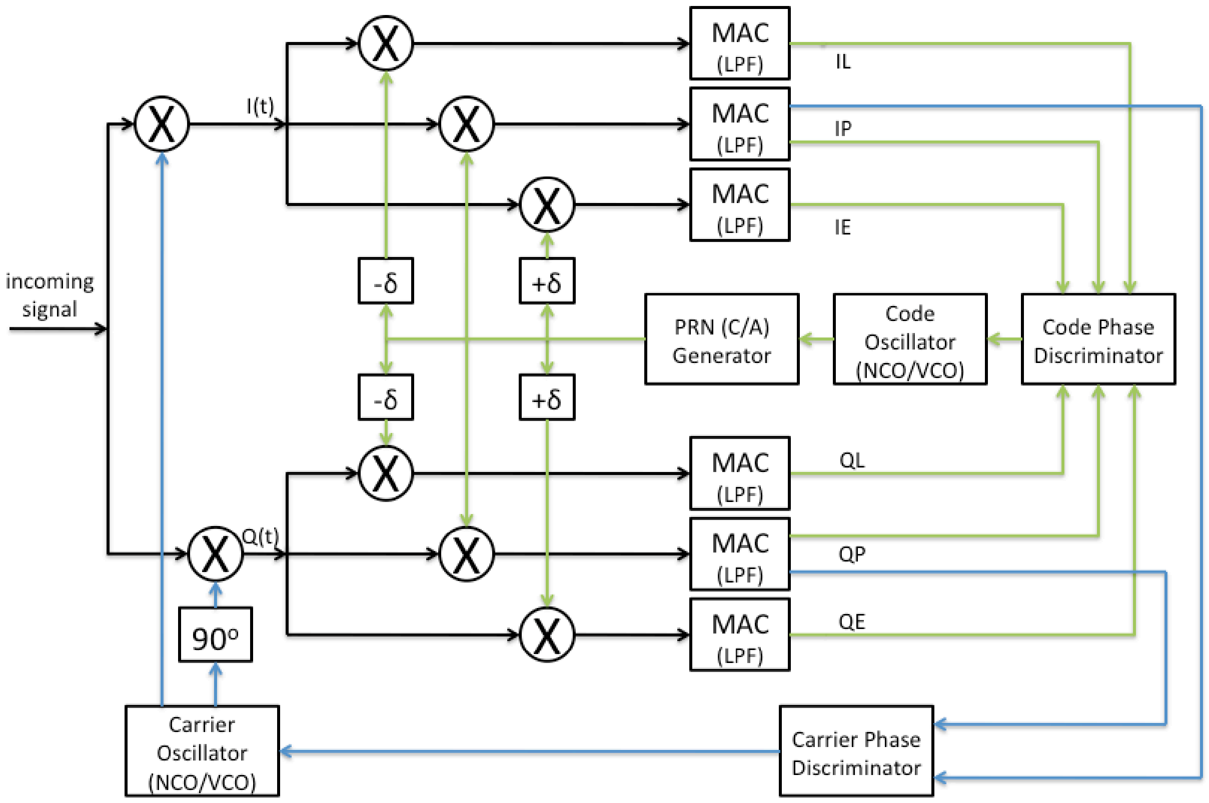

Figure 3.8: Combined tracking loop with carrier [blue] and code [green] feedback

3.4.1 Calculating Pseudoranges

A pseudorange is the line of sight vector from a satellite to the receiver. The reason that it is known as a "pseudo" range is because a clock bias term is also found in the range measurement [43].

Only a few items within the data message need to be found in order to calculate a correct pseudorange from each satellite. At the beginning of each subframe in the telemetry (TLM) word is what is called a preamble [7]. The preamble is 10001011 (or 01110100 depending on how the data bit polarity is defined). At the rising edge of the first bit of the preamble, the clock reading is captured in a variable called z-count [43]. Each satellite is synchronized with each other to a common GPS time, so the data messages are being transmitted almost simultaneously. This means that the time it takes the signal to reach the user depends on how far away each satellite is from the user. These times can be compared by analyzing the z-count of each satellite [43]. This distance is the pseudorange measurement. A nominal transit time for a signal to reach a user from a satellite is about 68ms (0.068 seconds) [7]. Assume i_1 is the time that the closest satellite has logged its z-count, i_2 is the corresponding time to the next closest SV, and so on. The raw pseudoranges can therefore be calculated by the formulas in Equation (3.2). Note that f_S is the sampling frequency of the hardware front end (16.3676MHz for the front-end used in this thesis).

$$\begin{aligned} \rho_{1,raw} &= 0.068c \\ \rho_{2,raw} &= (0.068 + \tfrac{i_2 - i_1}{f_S})c \\ &\vdots \\ \rho_{N,raw} &= (0.068 + \tfrac{i_N - i_{N-1}}{f_S})c \end{aligned} \tag{3.2}$$

There is also a clock correction factor that comes from the information found in subframe 1. Using this clock correction factor T_{corr}, the raw pseudoranges can be corrected by Equation (3.3) and then used to compute the user position.

$$\rho_{i,corrected} = \rho_{i,raw} + cT_{i,corr} \tag{3.3}$$

3.4.2 Calculating User Position from Pseudoranges

As mentioned, a pseudorange is the estimated geometric range from a user to the satellite plus a clock bias. The pseudorange is represented by Equation (3.4).

$$\rho_i = \sqrt{(x_i - x_u)^2 + (y_i - y_u)^2 + (z_i - z_u)^2} + ct_u + \nu \tag{3.4}$$

In this equation, ρ_i is the pseudorange from the satellite to the user; x_i, y_i, and z_i are the satellite i's geometric locations; x_u, y_u, and z_u is the user's geometric position, t_u is the receiver clock bias, and ν is noise.

The satellite vehicle's x, y, and z positions can be determined by processing the ephemeris in the satellite's data message. Since the receiver does not know its position, an estimated pseudorange (or prediction of the pseudorange, denoted by $\hat{\rho}$) must be calculated as seen in Equation (3.5). In this equation, $\hat{x}_u$, $\hat{y}_u$, $\hat{z}_u$, and $\hat{t}_u$ denote the predicted position and clock bias states of the receiver.

$$\hat{\rho}_i = \hat{r}_i + c\hat{t}_u = \sqrt{(x_i - \hat{x}_u)^2 + (y_i - \hat{y}_u)^2 + (z_i - \hat{z}_u)^2} + c\hat{t}_u \tag{3.5}$$

A well-known curve-fitting algorithm called least squares is used to determine the user's position [30]. Typically, if no additional information is known, a receiver would begin by guessing a user position, such as {x, y, z} = {0, 0, 0} in an earth-centered, earth-fixed (ECEF) coordinate frame - the center of the earth. The least squares algorithm uses this initial position and then identifies the error in the initial guess. This error correction is applied to the initial guess, and the algorithm is repeated until a very small correction is required. The typical least squares setup holds the form of Equation (3.6):

$$error = H * unknowns \tag{3.6}$$

In the case of GPS, there are four unknowns: x, y, z, and the clock bias, t. In order to solve for a position, at least four satellites must be tracked. In any case where more than four satellites are being tracked, the system becomes overdetermined and least squares is used to solve for user position and clock bias. The H matrix is the system of linearized pseudorange equations to each satellite, where $a_{p,N}$ denotes the line of sight unit vector from satellite i to the user in the p direction. The unknowns and H matrix are combined more clearly in Equation (3.7) and the least squares solution is solved in Equation (3.8).

$$\begin{bmatrix} \hat{\rho}_1 - \rho_1 \\ \vdots \\ \hat{\rho}_N - \rho_N \end{bmatrix} = H \begin{bmatrix} \delta\hat{x}_u \\ \delta\hat{y}_u \\ \delta\hat{z}_u \\ \delta\hat{t}_u \end{bmatrix} = \begin{bmatrix} \frac{x_1 - \hat{x}_u}{\hat{r}_1} & \frac{y_1 - \hat{y}_u}{\hat{r}_1} & \frac{z_1 - \hat{z}_u}{\hat{r}_1} & -1 \\ \vdots & \vdots & \vdots & \vdots \\ a_{x,i} & a_{y,i} & a_{z,i} & -1 \end{bmatrix} \begin{bmatrix} \delta\hat{x}_u \\ \delta\hat{y}_u \\ \delta\hat{z}_u \\ \delta\hat{t}_u \end{bmatrix} \tag{3.7}$$

$$\begin{bmatrix} \delta\hat{x}_u \\ \delta\hat{y}_u \\ \delta\hat{z}_u \\ \delta\hat{t}_u \end{bmatrix} = (H^T H)^{-1} H^T \begin{bmatrix} \hat{\rho}_1 - \rho_1 \\ \vdots \\ \hat{\rho}_i - \rho_i \end{bmatrix} \tag{3.8}$$

Each new pseudorange measurement allows for further user position accuracy and also tracks position and velocity changes for dynamic receivers. Equation (3.9) shows the recursive least squares equation that is run to obtain a new measurement, updating the old measurement with the most current set of data.

$$\begin{bmatrix} \hat{x}_{u,N} \\ \hat{y}_{u,N} \\ \hat{z}_{u,N} \\ \hat{t}_{u,N} \end{bmatrix} = \begin{bmatrix} \hat{x}_{u,N-1} \\ \hat{y}_{u,N-1} \\ \hat{z}_{u,N-1} \\ \hat{t}_{u,N-1} \end{bmatrix} + (H^T H)^{-1} H^T \begin{bmatrix} \hat{\rho}_1 - \rho_1 \\ \vdots \\ \hat{\rho}_i - \rho_i \end{bmatrix} \tag{3.9}$$

Other more complex methods such as weighted least squares and Kalman filters are used to compute user position instead of least squares, but they shall not be discussed in this thesis.

3.4.3 Determining User Velocity

User velocity cannot be determined by taking $\frac{du}{dt}$ (where du is the change in user position) because the noise in the position measurements is amplified far too much [43]. Instead, user velocity is determined by measuring the Doppler shift on the carrier signal for each satellite or the change in phase with time ($\frac{d\phi}{dt}$). Equations (3.10) and (3.11) give the user velocity.

$$d_i = \frac{c(f_{dopp} - f_{L1})}{f_{L1}} + \begin{bmatrix} V_{x,i} & V_{y,i} & V_{z,i} \end{bmatrix} \begin{bmatrix} a_{x,i} \\ a_{y,i} \\ a_{z,i} \end{bmatrix} \tag{3.10}$$

$$\begin{bmatrix} \delta\dot{x}_u \\ \delta\dot{y}_u \\ \delta\dot{z}_u \\ \delta\dot{t}_u \end{bmatrix} = (H^T H)^{-1} H^T \begin{bmatrix} d_1 \\ \vdots \\ d_N \end{bmatrix} \tag{3.11}$$

where d_i is the effect of the Doppler with respect to user position and $V_{p,i}$ is the satellite velocity in the p-direction. The satellite velocities are obtained from the data message (Section 2.4) by processing the satellite ephemeris data. A pseudorange-rate is directly related to the user velocity and satellite velocities. A pseudorange-rate is the rate of change of the pseudorange with respect to time. Therefore, there exists a pseudorange-rate for each satellite that is being tracked by the receiver. The pseudorange-rate is the line of sight velocity (including clock terms) of the satellite relative to the user.

3.5 Vector Tracking and Deep Integration Algorithms

Vector tracking algorithms designed for use with the global positioning system (GPS) have been studied by the navigation community for the past two and a half decades [5, 12, 20, 26, 28, 33, 35, 43]. The goal of this research is to exploit the advantages that vector tracking offers such as increased immunity to interference and jamming and the ability to perform at low signal-to-noise (C/N_0) ratios [36]. These advantages increase the reliability and robustness of a normal GPS receiver and have many practical applications in an environment that is typically challenging for normal GPS receivers.

Vector tracking functions by exploiting the general principle by which GPS receivers operate. Typical GPS receivers determine position and velocity by tracking the phases and frequencies of the received signals from available satellites individually [24]. Vector tracking combines the position and velocity determination with the signal tracking for all available satellites into a single step [36]. This combination is usually accomplished with an extended Kalman filter (EKF). Using the estimates from the filter, the phase and frequency of each visible satellite can be predicted for the next iteration of the EKF. Vector tracking loops generally have a greater immunity to receiver dynamics than typical scalar loops, and the additional aiding of inertial sensors bolsters this immunity even further. The fusion of the vector tracking algorithm with inertial sensors is known as Ultra Tightly Coupled (UTC) or Deeply Integrated (DI) systems [33].

In order to give an accurate description of how vector tracking differs from traditional methods, a traditional GPS receiver must first be discussed. A typical GPS receiver consists of four mostly independent parts as discussed in previous sections. The first part is the radio frequency (RF) front end which mixes the very high frequency satellite signal down to an intermediate frequency (IF) that can be handled by modern hardware. Second, an acquisition module uses a search algorithm to find which satellites are currently in view and get a rough estimate of their code phase and Doppler frequency. The next step is called tracking, where the estimates of code phase and Doppler frequency for each visible satellite are determined at a much finer resolution. The final step is to take the outputs of each of the tracking loops and compute a position, velocity, and time (PVT) solution. Vector tracking receiver architectures differ from traditional architectures in the tracking and navigation solution steps [36].

3.5.1 Traditional Receiver Operation

Traditional GPS receivers utilize scalar tracking loops to track each satellite in view as described in Section 3.3. Scalar loops, as the name implies, treat each loop as a single, independent entity. There is no feedback or sharing of information between any of the scalar tracking loops and no statistical correlation between channels. The outputs of these tracking loops are the pseudorange and pseudorange-rates for each of the satellites. These outputs are used as the input to the navigation solution stage, and using each of these individual measurements, the receiver computes a position. Figure 3.9 is a block diagram of the traditional receiver architecture using scalar tracking loops.

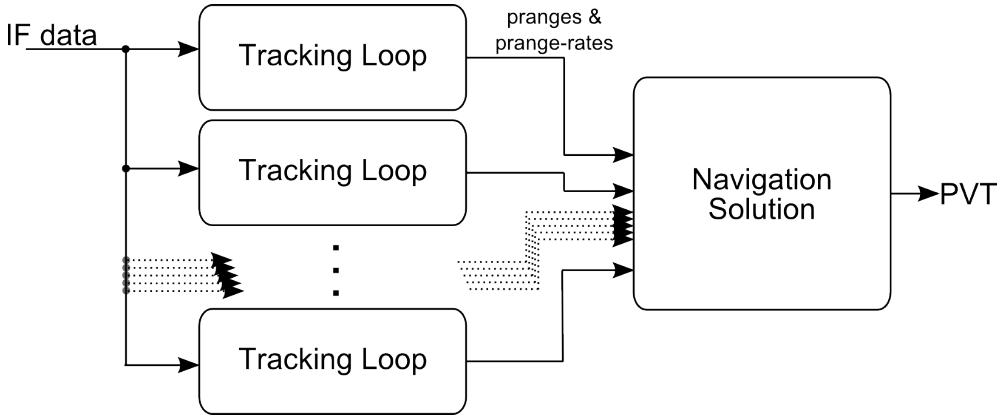

Figure 3.9: Block diagram of the traditional receiver scalar tracking loop architecture

3.5.2 Vector Tracking Receiver Operation

Vector tracking receivers take a different approach than traditional receivers. In the vector tracking algorithm, the tracking and navigation solution steps are combined in a single step, usually accomplished by an extended Kalman filter (EKF). Vector tracking, as its name implies, considers all of the satellites in aggregate to obtain a navigation solution [36]. The vector tracking approach that will be considered in this research is a vector delay frequency locked loop (VDFLL) in an unfederated (centralized) architecture. This means that there is one central EKF that predicts both the code phase (pseudorange) and frequency (pseudorange-rate) for each satellite in view. Figure 3.10 is a block diagram of the vector tracking (VDFLL) architecture that is used in this system.

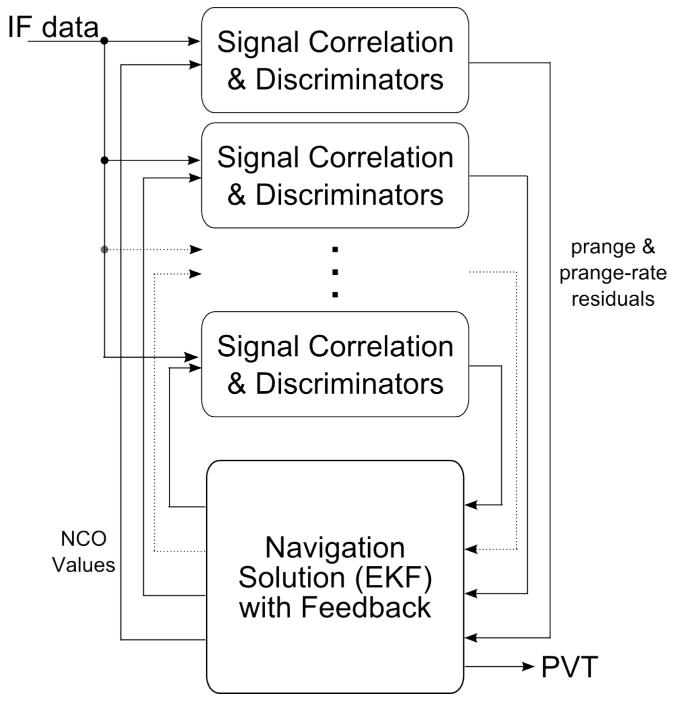
IF data
Signal Correlation & Discriminators
Signal Correlation & Discriminators
Signal Correlation & Discriminators
prange & prange-rate residuals
NCO Values
Navigation Solution (EKF) with Feedback
PVT

Figure 3.10: Block diagram of the vector tracking receiver loop architecture

Notice that instead of separate tracking loops, the only operation that is done independently is the signal correlation. The outputs of these correlators, after being passed through a discriminator, are the pseudorange and pseudorange-rate residuals. These residuals are the measurements used for the centralized filter. The EKF used in the position state VDFLL are the errors in the receiver's position, velocity, and time (PVT) rather than the PVT itself, so the actual PVT is managed separately. The error states of the EKF used in this thesis are:

$$\begin{bmatrix} \delta x \\ \delta \dot{x} \\ \delta y \\ \delta \dot{y} \\ \delta z \\ \delta \dot{z} \\ \delta t \\ \delta \dot{t} \end{bmatrix}$$

3.5.3 Vector Tracking Algorithm Details

This section outlines some implementation details of the vector tracking implementation used in this thesis. This implementation was originally outlined by Robert Crane from L3 Interstate Communications [12]. It utilizes an extended bank of signal correlators that are positioned around the nulls in the power density function of the C/A code to get an accurate picture of noise power.

Because the vector tracking architecture for this research has been chosen as a VDFLL, where both the code phase and carrier Doppler frequency are predicted, the carrier phase is not locked. Because the carrier phase is not locked, no data bits can be backed out, so the vector tracking receiver must be initialized with a PVT solution, ephemeris data, GPS time, Doppler frequencies, and code phase information.

There are two sets of processes that are performed in the vector tracking algorithm. The first is the extended Kalman filter loop that completes at a rate of 50Hz (20ms period), and the second set of processes yields the data and measurements for each individual satellite.

The second set of processes is mostly unique to vector tracking (denoted by *) and many are unique to this particular vector tracking formulation by Crane (denoted by $^\$$) [12]. For each satellite, the following operations are performed:

- update satellite position
- get pseudorange and transit time
- populate pertinent rows in C matrix with line of sight unit vectors to current satellite
- predict Doppler frequency*
- predict code phase*
- calculate noise variance$^\$$
- get signal amplitude, noise power, and $C/N_0{}^\$$
- calculate measurement covariance matrix R$^\$$
- use discriminators to calculate measurements$^\$$

Specifics for implementation of the Crane method can be found in [12].

3.5.4 Predicting Doppler Frequency and Code Phase

Calculating the Doppler frequency for each satellite is based on the definition of the Doppler effect. First, the velocity of the satellite relative to the receiver is measured using Equation (3.12).

$$LOS_{vel} = a_{x,y,z} \cdot (sv_{vel} - \dot{\hat{x}}) \tag{3.12}$$

Then, the definition of the Doppler effect is used. It is adjusted by the measurement of the receiver clock drift (t_d) and then shifted from the L1 transmit frequency (f_{L1}) to the intermediate frequency (f_{IF}), as shown in Equation (3.13).

$$f_{if+dopp} = \frac{1 - LOS_{vel}}{c + t_d} f_{L1} - f_{L1} + f_{IF} \tag{3.13}$$

The code phase is predicted by using the fact that all satellites broadcast the preamble of the data message simultaneously. After pseudoranges to the satellites are initially found based on the scalar tracking solution, this range can be converted into a time by multiplying by the sampling period. This is the time difference between the satellite sending the preamble and it being received by the user. Once this time is found, it is compared to the elapsed time that the system actually took based on the previous pseudorange. The difference between these two times is used to adjust the NCO of the PRN generator, and thus the code phase is predicted.

3.5.5 Deep Integration

Most of the research that has been done in the vector tracking field has included measurements from some inertial sensor such as an IMU [35]. A system that combines both vector tracking and an inertial sensor is called ultra-tight coupling or deep integration. The addition of an inertial device increases the receiver's immunity to dynamics by being able to track changes in position and velocity between vector tracking filter updates. Since the vector tracking formulation uses an extended Kalman filter, many of the Doppler and code phase predictions are performed using the time-propagated position. If there is a significant outage in the GPS signal (and thus no measurement update), the predicted position, velocity, and time is propagated forward in time using the designated motion model until it comes back into view. The effects of aiding a receiver with an inertial device to help bridge these outage gaps is well documented [24].

Adding an inertial sensor is not without its complications. If an inertial measurement unit (IMU) is added to the system, there must be several additional states in the navigation

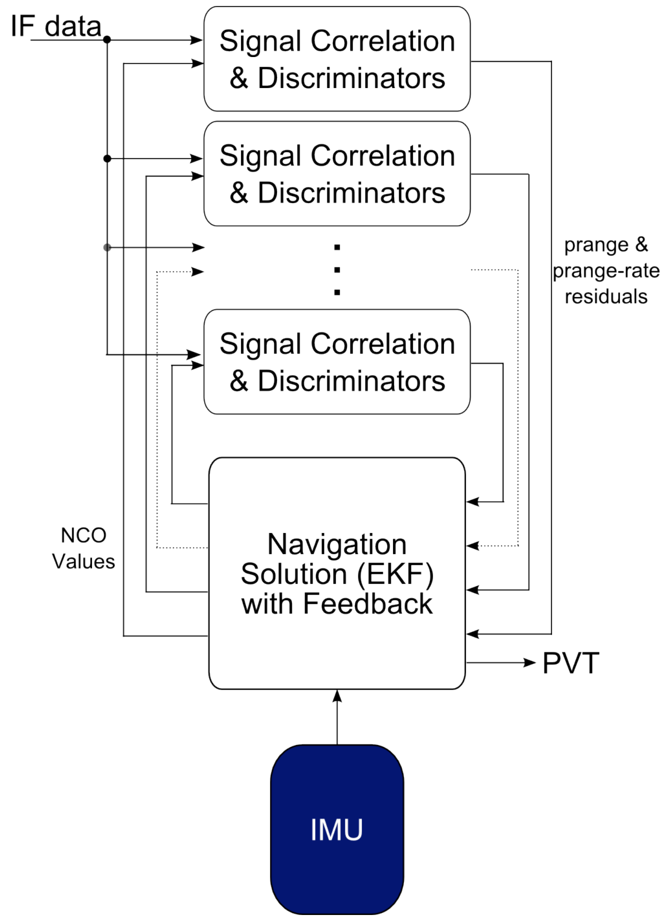
IF data
Signal Correlation & Discriminators
Signal Correlation & Discriminators
Signal Correlation & Discriminators
prange & prange-rate residuals
NCO Values
Navigation Solution (EKF) with Feedback
PVT
IMU

Figure 3.11: Block diagram of an ultra-tightly coupled/deeply integrated system

filter which track the error dynamics of the IMU. Because the update rate of the IMU is typically very fast (~50-800Hz), tracking these parameters can become burdensome on an embedded system. The IMU measurements must also be synchronized in time with the GPS data, as unmodeled errors can be introduced if the two are not synchronized [24]. This synchronization usually involves a one-second timing pulse from the GPS (known as the pulse per second, PPS) and a fairly large amount of memory space to buffer several samples of IMU data so that it can be lined up correctly with the GPS clock.

As mentioned, the addition of an IMU adds numerous states to the central Kalman filter. The 17 states include position, velocity, attitude (roll, pitch, and yaw), gyro biases, accelerator biases, clock bias, and clock drift.

Chapter 4

Software Receiver Platform Trade Study

The concept of extending the software defined radio to the GPS industry began in the late 1990's [3]. Software receivers provide several advantages over hardware-only implementations. First and most importantly, software receivers are reconfigurable. They can theoretically be reprogrammed any number of times in any number of ways. This is extremely important in the navigation research community, where developing and testing new algorithms is paramount. The ability to reconfigure a working GPS receiver to incorporate some research interest advances the field far more rapidly than being constrained to third party hardware or software.

A second benefit to software receivers is the fact that any part of the GPS process is configurable. For example, in a hardware-only receiver, a dedicated chip might perform signal acquisition, but it functions as a black box. Only the inputs and outputs are visible. In contrast, a software receiver might be responsible for performing the entire signal acquisition process, so any point within the algorithm can be viewed or potentially modified for improvements.

Software receivers also provide the potential for increased portability and modularity. Since the implementation is in software, there might be a considerable number of choices for a hardware platform to run the receiver. This means that the development of a software receiver might be done on a desktop computer, but subsequent iterations of the same software might be able to be extended to a hand-held personal digital assistant (PDA) device. In this context, modularity can be imagined by assuming that there are more advanced processing techniques that require the processing power of the desktop but can easily be removed for the PDA implementation.

Recent developments in the GPS community have shown that there are three main hardware platforms that are typically used in designing software receivers: the microprocessor, digital signal processor (DSP), and field-programmable gate array (FPGA). Software receivers can also sometimes use different combinations of these platforms in a single receiver.

4.1 Microprocessor

A microprocessor is a hardware platform that is generally not designed with a particular purpose in mind [47]. Because of this, many are known as general purpose processors (GPPs). Personal computers such as desktops and laptops are built using GPPs. Microprocessors typically operate at much higher frequencies than the other hardware platforms that will be discussed, and can therefore process a very large amount of instructions in a short amount of time. Single core microprocessors sequentially execute a single instruction at a time. Sometimes microprocessors implement a concept called threading where multiple different software processes (threads) give the appearance of running simultaneously by constantly switching back and forth between the processes [41]. The ability to program a microprocessor is quite flexible, and does not typically require special training. Common programming languages such as C and C++ can be run on a microprocessor.

Microprocessors cannot be used alone, as they have only a very small amount of memory on the chip itself. Instead, they must use external RAM or ROM to load program instructions and data [47]. Also, since they are widely labeled as general purpose, they do not specialize in the heavy mathematics operations that are necessary for a software receiver. Depending on the number of cores in the microprocessor, the high frequency data correlation can be difficult to accomplish. Furthermore, they are potentially expensive. Unless custom hardware is designed to include only the pieces of the design which are necessary (the microprocessor, memory, and input/output), any additional units must be purchased at roughly the cost of a motherboard (which can vary tremendously based on the amount of processing power needed).

There are many implementations of GPS software receivers that use microprocessors. The OpenSource GPS project uses a small interfacing hardware front end to a personal computer via USB and can perform in real time [25]. Dennis Akos and Kai Borre developed a software receiver in MATLAB based on the seminal doctoral research of Akos [7]. Other research is being done to identify the benefits that a multi-core processor could provide [23].

4.2 Digital Signal Processor

A digital signal processor (DSP) is an integrated circuit which specializes in, as the name suggests, digital signal processing applications. One of the most notable of these specializations is the ability to perform a multiply-accumulate (MAC) operation in a single clock cycle (usually). Current high performance DSPs operate at speeds around 1GHz, and some contain ARM microprocessor cores inside of them for control logic and other general purpose functions.

DSPs are designed to support some degree of parallelism, but still must execute its software sequentially like a microprocessor. External memory is also required to store the data and instructions. Unless they have an embedded microprocessor, they are not typically as good at general purpose logic. Some DSPs can be programmed with common languages such as C or C++, but others require knowledge of a specialized set of instructions (such as SHARC) [47].

DSPs have been used in recent research efforts for L1/L2 software receivers [32]. An even more specialized DSP known as a graphics processing unit (GPU) has also been used in building a GPS software receiver [22].

4.3 Field-Programmable Gate Array

An FPGA is a programmable integrated circuit which consists of programmable logic blocks (PLBs), input/output (I/O), and interconnects as shown in Figure 4.1. At any time, an FPGA can be configured to have a specific system function. The system function is determined by activating some or all of the I/O pins, assigning certain logic functions to the PLBs, and using the interconnects to route information to/from the I/O and PLBs.

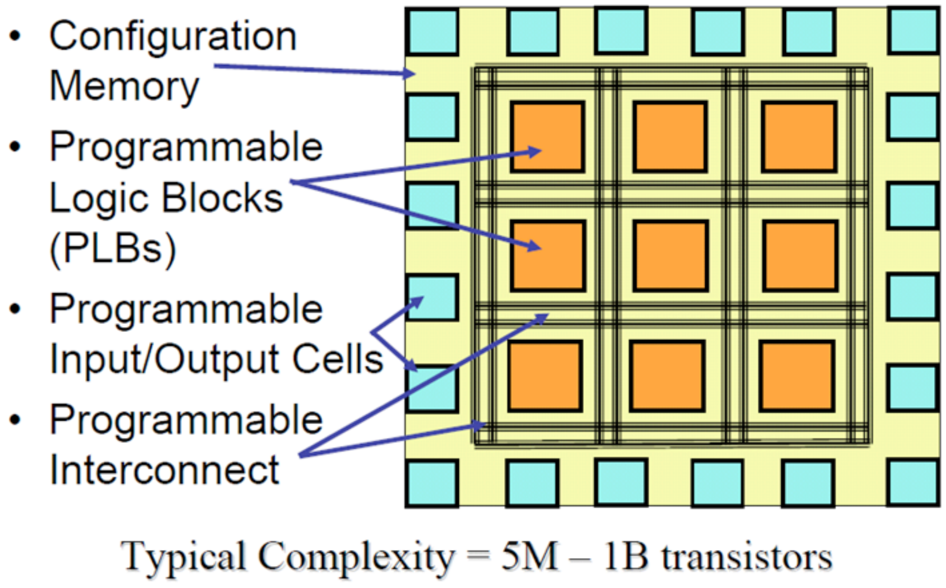
• Configuration Memory
• Programmable Logic Blocks (PLBs)
• Programmable Input/Output Cells
• Programmable Interconnect
Typical Complexity = 5M – 1B transistors

Figure 4.1: FPGA overview [44]

4.3.1 FPGA as a Real-time Platform

FPGAs have many advantages that make them a very valuable choice as a software receiver platform. There are multiple vendors such as Altera and Xilinx, and every company produces a wide array of different FPGAs with different capabilities and specifications. Aside from having many different options to choose from, FPGAs are known for their speed due to fact that an FPGA is pure hardware. Its computational performance has been found to closely resemble (but not match) that of an ASIC of the same functionality [27].

An FPGA does not execute one statement of code after another in a sequential fashion, but instead can perform multiple functions simultaneously. This is similar to the idea of “threading” in computer science terms, except that where a single microprocessor handles each thread in turn to make the functionality only seem simultaneous, the parallel operations in an FPGA actually are simultaneous. This improvement in speed, brought about by this parallelism, makes it a very good option for handling multiple correlators, as the vector tracking architecture used in this research requires. An FPGA is also completely reconfigurable; this is a very important property in reducing a project's design time and is, of course, a necessity for a software receiver.

The FPGA is not without its drawbacks. On the more powerful FPGAs, power consumption is a concern. Many FPGA manufacturers are attempting to alleviate this problem by releasing low-power versions of their most popular designs [2]. An FPGA footprint can range from quite small to very large, and are often many times larger than their ASIC counterparts would be. This is because of all of the extra logic that is required to make the FPGA reprogrammable and the fact that an ASIC only includes as many gates as it needs and no more, whereas an FPGA might have many unused gates in a design. Another drawback to development with FPGAs is the designer accessibility. FPGAs must be programmed with hardware description languages (HDLs), which are generally less accessible than more commonly known languages such as C and C++.

An FPGA is very well equipped to perform high frequency operations such as the high frequency correlation found in a GPS receiver's tracking loops [21]. It is not, however, equipped to execute large amounts of matrix mathematics as GPS and vector tracking demand because the sizes of the matrices are constantly changing and the FPGA hardware must remain fixed. Arithmetic functions are also expensive in terms of hardware. Valuable logic resources are used up to implement a multiplication that may happen only once. For these reasons, it is almost a necessity to combine the FPGA with an embedded microprocessor [21]. This can potentially add a considerable amount of hardware interfacing unless the FPGA has a microprocessor already embedded into the fabric or there is enough room to synthesize one. The latter options are available in almost all modern FPGAs from vendors such as Xilinx or Altera.

4.3.2 System on Chip Design

One of the most advantageous parts of the FPGA platform is its sufficiency as a standalone device. The combination of all logic (including microprocessors), memory, I/O, timing, etc. on a single integrated circuit is known as system on chip (SoC) design. All of these individual pieces can be synthesized into a single FPGA, yielding very tight integration between the different parts of a system. Because the parts are so tightly woven together, many of the difficulties of interfacing external memories and processors can be avoided. Using a single FPGA, a designer can combine I/O, memory, intermediate frequency (IF) processing, and baseband processing on a single chip.

Recently, the system on chip approach using FPGAs has become more of a reality. FPGA manufacturers have recognized that there are specific functions that the developers who use their products are utilizing repeatedly. Because of this, there has been a significant increase in the number of "extra" features that an FPGA has embedded within its programmable logic blocks (PLBs) and interconnects. These extras include block RAMs (BRAMs), 32-bit PowerPC microprocessors, multi-gigabit Ethernet interfaces, PCI Express, and DSP slices [49]. Each of these extras are actually embedded within the FPGA fabric itself so no extra logic is used in achieving this functionality. If, however, there is a need for another function that is not embedded within the FPGA, Xilinx and many other third parties offer Intellectual Property (IP) solutions that synthesize commonly-used functions into logic on the FPGA. One of the most important flexibilities of an FPGA is its ability to synthesize a microprocessor using its PLBs. Many FPGA vendors offer customized, modular 32-bit soft-core microprocessors. Microprocessors supporting the ARM architecture are also available in both hard and soft cores.

Combining the programmable logic and embedded features present in an FPGA makes it a very appealing platform when total control is a priority. However, this flexibility is not without some very concerning limitations. The most notable limitation is the power and speed of the hard and soft-core processors used for baseband processing. The soft-core microprocessor used in this research, Xilinx's Microblaze, runs at an effective speed of around 200 MHz. Compared to GPPs which can currently achieve speeds around 5 GHz, the speed capability of the Microblaze is very limited.

4.3.3 Prototyping ASIC Design

The FPGA platform is also very useful when moving from a research-based software receiver to an actual hardware platform in the field. In a high quantity design where speed, low power, size, and cost are ideal, the use of an FPGA as a prototyping platform is very beneficial [10]. FPGAs are programmed using hardware description languages (HDLs) such as VHDL and Verilog. These same languages are used in the process of designing application-specific integrated circuits (ASICs); therefore, much of the hardware design and testing has already been completed when designing the FPGA prototype, which drastically reduces the amount of development time when designing an ASIC.

Many FPGA vendors also offer cost-effective alternatives to developing ASICs. The FPGA vendors take a design, strip out all portions of the FPGA which provide programmability or are unused, and craft a non-programmable gate array that features ASIC-like performance at a reduced price compared to a full FPGA. Two of the main vendors, Xilinx and Altera, offer this service in the forms of EasyPath and HardCopy, respectively [50, 4].

Chapter 5
Receiver Architecture

The goal of the research in this thesis is to develop a platform that supports both traditional GPS (scalar tracking) and vector tracking/deep integration in real time. The inputs to this system would be GPS data after it has been converted to some intermediate frequency via an RF front end and inertial data from an IMU. The proposed platform has been implemented on a Xilinx ML506 development board that contains the Virtex-5 SXT50T FPGA, volatile/non-volatile memories, and several different I/O options [48]. The proposed hardware solution is shown below in Figure 5.1. The system presented in this chapter is the second revision of this prototype and was published in [17]. Details about the first prototype were published in [16] and are given in Appendix 7.3.

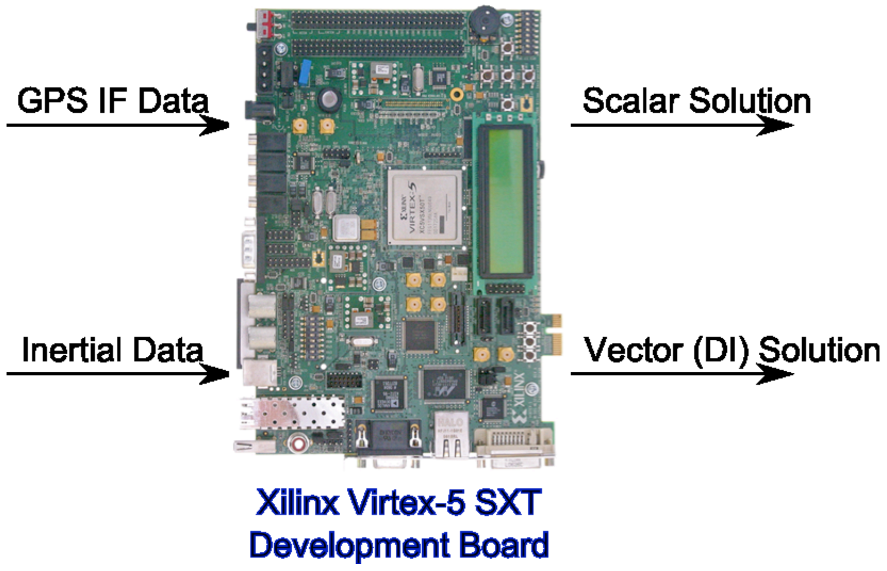

Figure 5.1: Proposed hardware solution for combining scalar and vector/deeply integrated GPS/INS

The proposed platform takes advantage of the system on chip approach as previously discussed. The FPGA is capable of performing both the high frequency operations and the low frequency operations in the same chip. The high frequency domain contains modules that perform signal acquisition, scalar tracking, and vector tracking loops. A counter is used to synchronize the acquisition and tracking loop blocks. Each channel is implemented individually and can be duplicated a number of times to create a modular "N-channel" system. The baseband operations are performed by a 32-bit soft-core microprocessor from Xilinx called the Microblaze. There are three independent Microblaze processors which operate on three distinct portions of the receiver. Instruction and data memory for each Microblaze is located in the block RAM (BRAM) of the FPGA and requires no external memory. Communication between the processors is achieved using BRAM and mailboxes. Communication between the high-frequency and baseband domains is achieved using a high-speed data bus using processor memory mapping techniques. Data is output to a PC for debugging by serial (RS232) communication. The RF front end is interfaced using I/O pins on the FPGA. Data is input from the IMU via serial (RS232) communication. Each of the aforementioned parts of the proposed receiver are detailed in the remainder of this section. A detailed block diagram of the different hardware functions is shown in Figure 5.2.

This thesis will not focus on the hardware architecture outside of the FPGA. It can be assumed that a GPS front end has downconverted the GPS L1 C/A signal to an intermediate frequency (IF) and that an inertial measurement unit with a pulse per second synchronization input communicates with the FPGA via an RS232 connection.

5.1 Software Architecture

The proposed platform takes advantage of the FPGA's ability to synthesize multiple soft-core processors. Since the platform supports both scalar and vector tracking loops, a Microblaze processor is synthesized for each of these independent functions, and another Microblaze is synthesized to read and parse IMU data from a serial port. It is important to note that each of the Microblaze processors can be replaced by any hard-core or external 32-bit microprocessor that supports memory mapped I/O. It is even possible (and perhaps

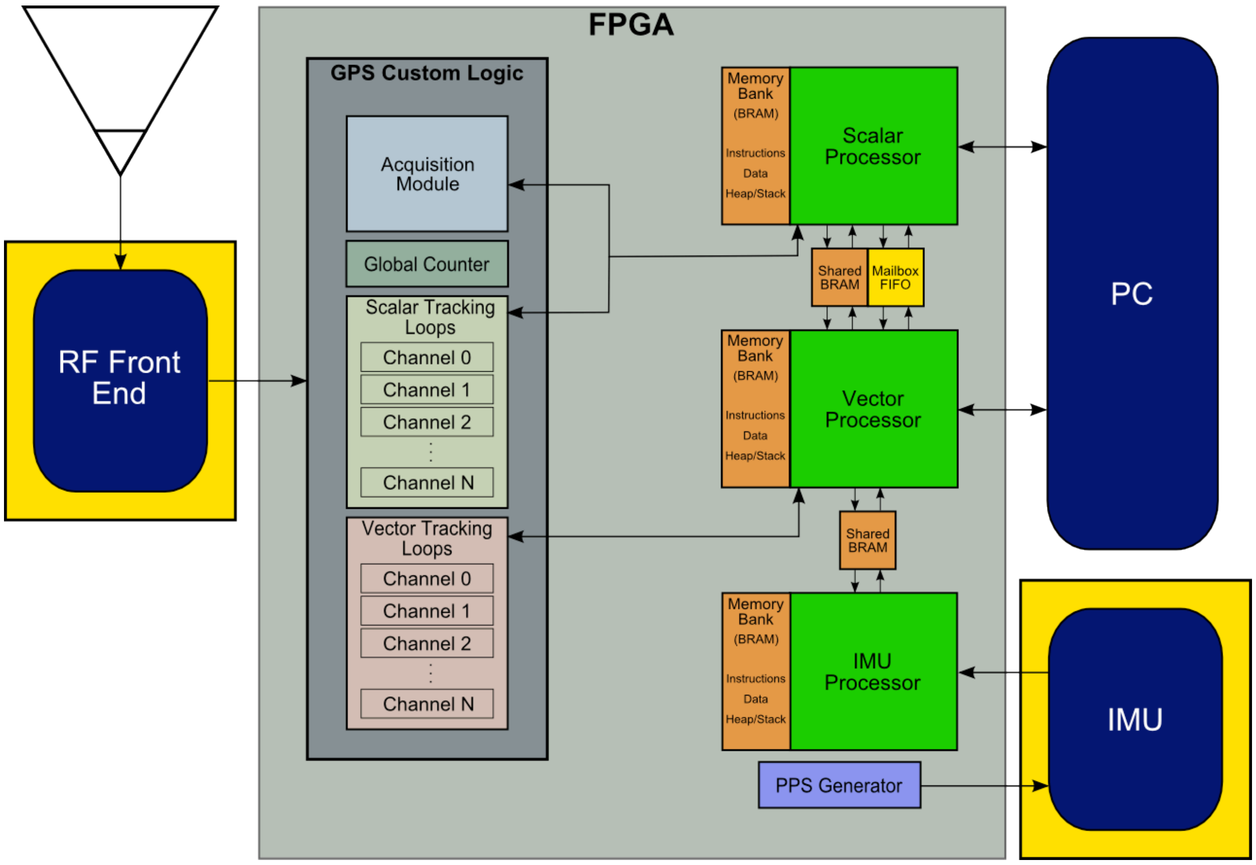

Figure 5.2: Block diagram of proposed system

desirable) to replace all of the synthesized Microblazes with a single, more powerful external microprocessor. However, since the goal of this research is to achieve real-time performance using a system on chip approach, all three soft-core processors are used. Figure 5.3 is a diagram of all three of the processors and the interprocessor communication mechanisms used (as will be discussed in Section 5.1.4).

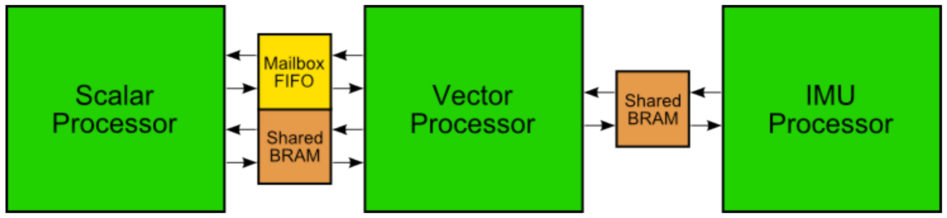

Figure 5.3: Block diagram of all processors and their interprocessor communication mechanisms

5.1.1 Scalar Processor

The scalar processor performs all of the baseband functions for a traditional GPS receiver such as signal acquisition, tracking loop feedback, bit synchronization, pseudorange determination, and PVT calculation. For the serial search acquisition (discussed later), this includes cycling through which PRN to search, the Doppler offset at which to search, and determining the correlation peak to next peak ratio (CPPR) to find visible satellites. For the scalar tracking loops, this includes the normalized early minus late discriminator, the loop filter (adjustable for different pull-in ranges), and feeding back code and carrier NCO values for each channel. Each of these operations is performed using fixed-point arithmetic for high throughput. Bit synchronization is also accomplished on this processor after the initial transients of the loop filter have settled out. This processor can also determine pseudoranges, decode ephemeris data, and solve for a user's PVT vector.

This processor is almost entirely interrupt-driven. The acquisition module interrupts the processor at 1ms intervals to provide the correlator outputs, the code phase, and the Doppler frequency being searched. Also, each individual tracking loop interrupts the processor at 1ms intervals. The interrupting information is the tracking loop correlator values from both the in-phase and quadrature branches which must be processed within interrupt handlers.

With regard to the vector tracking functionality of the proposed platform, the scalar processor's main role is to provide an initial PVT to the vector processor. The discriminators of the vector processor force the solution to be initialized using a starting PVT that is close to the receiver's true PVT. The vector processor depends on the scalar processor for this information.

5.1.2 Vector Processor

The vector processor performs all of the baseband functions for a vector tracking receiver or a deeply integrated GPS/INS receiver. (It is important to note that the vector processor can do either GPS-only vector tracking *or* GPS/INS deep integration; it is not possible to do both solutions independently on the same processor.) These baseband functions include the discriminators for each channel, updating the extended Kalman filter (EKF) measurements, and then computing the time and measurement updates for the EKF. In the current formulation, each channel that is currently being tracked is processed asynchronously. This is because of the arithmetic simplifications that can be made to the EKF which reduce the size of a matrix inversion from 8x8 (vector tracking) or 17x17 (deep integration) down to a 2x2 matrix, which is trivial. This comes, however, at the cost of generating the state transition matrix repeatedly and subtracting out previous channels' corrections from each new measurement. It has yet to be determined whether or not this tradeoff is superior in terms of computation time versus a synchronous measurement update.

The vector processor combines data from each of the active channels in a centralized EKF to determine new NCO values that must be fed back to each individual channel. This processor is interrupted by each channel every 20ms. The processor's ability to meet real-time deadlines is a function of the speed at which computing the EKF measurement update can occur and the number of channels in the system.

5.1.3 IMU Processor

The IMU processor has a very simple job. It is responsible for reading the IMU data stream from a serial port, parsing the IMU data received, and supplying the measurements of acceleration and rotation rates from the IMU to the vector processor. It can be noted that even though it is not done here, the IMU processor can easily supply these values to the scalar processor to potentially achieve loose, close, or tight GPS/INS coupling.

5.1.4 Interprocessor Communication

Communication between these processors is very simple. Between the scalar and vector processors, as shown in Figure 5.3, a BRAM and a mailbox provide all of the communication needed. The mailbox is structured as a FIFO memory and passes simple messages to the vector processor. These messages tell the vector processor when the PVT is initially ready, when the scalar processor has a loss of lock, and other miscellaneous information. The mailbox is also very useful because it can trigger interrupts on the vector processor. For instance, when the initial PVT has been solved for, an interrupt can trigger the vector processor to begin accepting interrupts from each of its channels. A BRAM between these two processors contains information such as the PVT vector and ephemeris data for each of the visible satellites. Any other information that needs to be shared between the two processors can be placed in this BRAM.

Between the vector and IMU processors, only a BRAM is needed. This BRAM stores the accelerations and gyro values from the IMU. There is no need for a mailbox because no messages need to be passed; the vector processor simply uses the values in the BRAM at each measurement update of the deep integration EKF to update the state transition matrix with accurate inertial information.

5.2 Hardware Architecture

Custom GPS logic has been developed for performing all of the high frequency operations of the GPS receiver. The subsequent sections will describe the different high frequency components of the proposed platform in detail. Each of the following modules has been implemented in VHDL (very-high-speed integrated circuit hardware description language).

5.2.1 Acquisition

The acquisition module that is used is a serial search acquisition. This means that each code phase and Doppler frequency is searched independently for each satellite. This is in contrast to many of the popular acquisition algorithms such as the parallel code phase search acquisition. The reason for choosing the serial search algorithm is due to the fact that it is completely scalable; as many (or as few) correlators as desired may be used in the acquisition process. This allows the user to control the number of frequencies and code phases to search in any given iteration. If a parallel code phase search algorithm had been used instead, the resulting acquisition module might be many times faster, but it requires two fast Fourier transforms (FFTs) and one inverse FFT. Depending on the sampling frequency of the hardware front end, this could be a considerable amount of logic space required to perform these three FFTs. Instead, using the scalable serial search acquisition allows a designer the flexibility to determine tradeoffs in resource usage versus speed of acquisition. If a faster acquisition time is desired, more correlators can be used (and vice versa). Figure 5.4 is a block diagram of the acquisition module.

A single BRAM is used as a ROM to store each of the 32 PRN sequences. This is because there is no need for a code NCO, and each of the PRNs can be easily selected by setting the number of the PRN to the topmost 5 bits of the PRN ROM address. Note that the correlator outputs of the acquisition module are fed into a register bank. This register bank is considered part of the memory mapped space of the scalar processor. Memory mapping is discussed in Section 5.2.6.

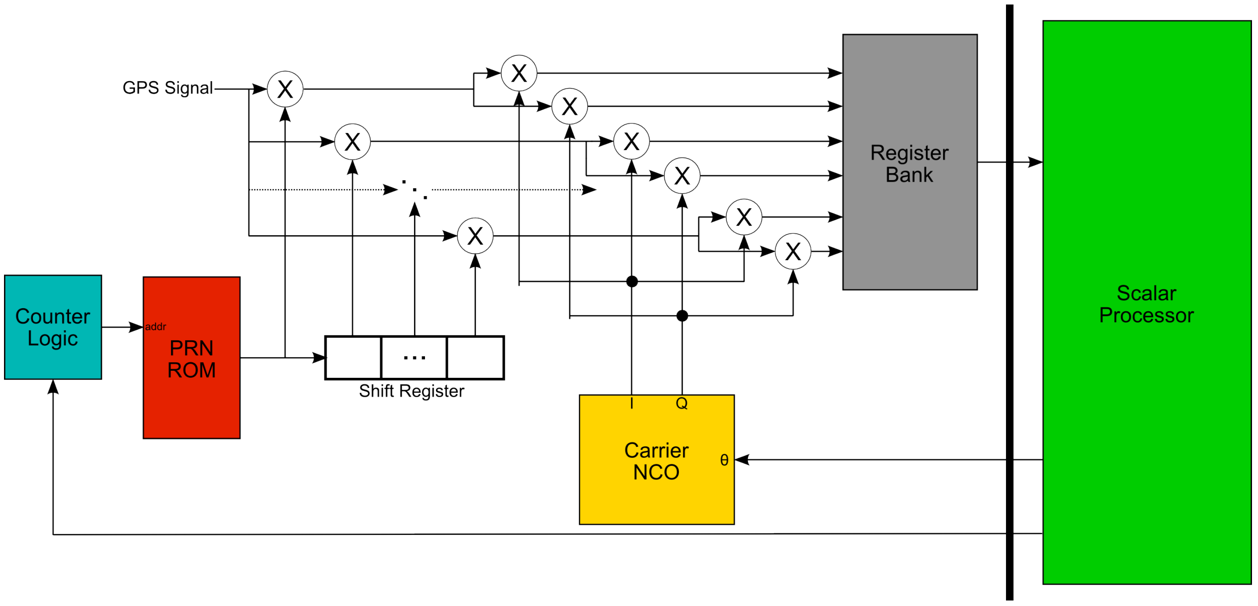

Figure 5.4: Block diagram of serial acquisition module

5.2.2 Scalar Tracking Channels

Each scalar tracking loop is a feedback loop. A carrier NCO is used to wipe off the carrier IF for each channel, and a code NCO is used to drive a PRN generator. This PRN generator generates early, prompt, and late samples which are used to wipe off the code sequence from each signal. The PRN generator also contains the functionality to trigger an interrupt every time it begins its sequence. Since the sequence repeats at a rate of 1kHz, an interrupt is triggered at the scalar processor every millisecond. Each of the correlator outputs is placed in a register bank which is read by the scalar processor as memory mapped I/O. The scalar processor performs the code and carrier discriminators, processes the data in a loop filter, and then feeds back the new code and carrier NCO values. Figure 5.5 is a block diagram of a single scalar tracking channel.

5.2.3 Vector Tracking Channels

Each vector tracking loop is also a feedback loop, though the feedback term differs from the scalar tracking loop. A carrier NCO is used to wipe off the carrier (IF) frequency for each channel (known as carrier wipeoff), and a code NCO is used to drive a PRN generator.

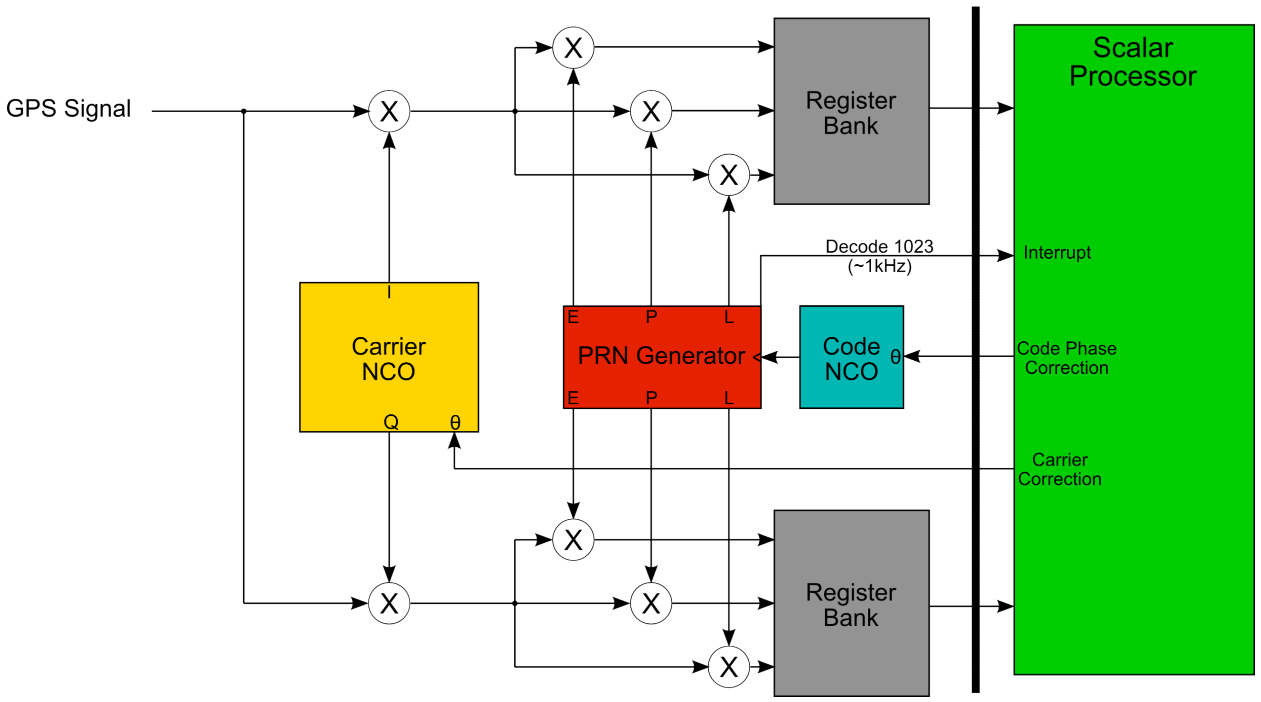

Figure 5.5: Block diagram of scalar tracking loop/channel

The PRN generator generates the typical early, prompt, and late samples and also several other delayed samples that correspond to nulls in the autocorrelation function per the discriminators used in [12]. The PRN generator also generates a signal every millisecond, but in order to reduce the vector processor's computational load, the vector processor is only interrupted every 20ms (a full data bit), so a counter is used to only generate interrupts at that rate. The correlator outputs are fed into a register bank which can be read by the vector processor as memory mapped I/O. Figure 5.6 is a block diagram of a single vector tracking channel.

The vector tracking/deep integration tracking channel is very similar to the scalar tracking channel. This might be confusing to the reader because of the inherent differences in the scalar and vector tracking formulations. However, it is important to note that the difference comes in the baseband processing: the scalar processor computes independent NCO feedback based solely on that channel's correlator outputs, whereas the vector processor determines the feedback values based on the combination of each of the channels measurements within

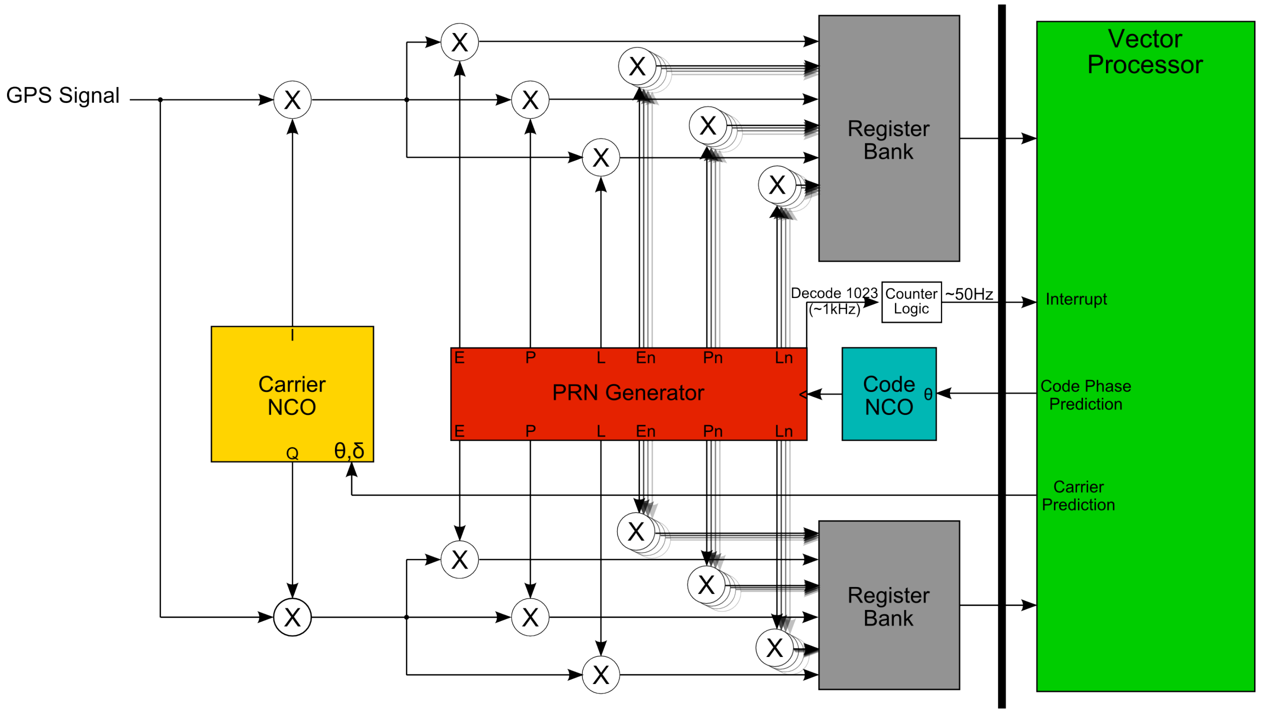

Figure 5.6: Block diagram of vector tracking channel

the EKF. It is also important to note that the vector tracking/deep integration solution needs independent tracking loops only if the scalar and vector solutions will operate simultaneously. Otherwise, the scalar tracking correlators may be used for the vector tracking solution. Since the desired goal for this research is to have the two solutions operate independently, separate correlators are required.

5.2.4 GPS Counter

A counter is used to synchronize the acquisition and tracking modules. This is accomplished by clocking the counter at the front end sampling frequency, f_S, and counting up to $round(\frac{f_S}{1000})$ (i.e. $round(\frac{16.3676MHz}{1000}) = 16368$ for this research), the approximate number of data samples in a single millisecond. When the acquisition module detects a peak, the code phase is logged. When initializing the scalar tracking loop, the code phase value is placed in a register. When this register matches the current GPS counter, the PRN generator is initialized and the tracking loop is started. A similar handoff from scalar to vector tracking is used.

5.2.5 PPS Generator

Many IMUs allow synchronization with GPS measurements by inputting a pulse per second (PPS) timing strobe. This pulse forces the IMU to capture and deliver measurements that are synchronized in time with the GPS measurements and prevents a complicated time synchronization mechanism (such as storing multiple measurements, discarding some, and realigning on a GPS update). In the hardware platform designed in this thesis, this pulse is generated by a hardware linear feedback shift register (LFSR) counter. This counter uses the 100MHz reference clock to count up to 50 million and then toggle the PPS register value. This generates a 1Hz square wave with a 50% duty cycle.

5.2.6 Memory Mappings

Memory mapped I/O is a very widely used I/O mechanism for modern microprocessors. Because a 32-bit processor can index 2^{32} bytes of data, much of that address space would typically not be used up by instruction and data memories. Mapping peripherals into the processor's memory space allows the processor to use predefined memory read and write instructions to access peripherals in the custom logic space of the FPGA. In the scalar and vector processors described above, the instruction and data memories (including the stack and heap memories) are physically placed in BRAMs. The processor peripherals such as the interrupt controllers, mailboxes, shared BRAMs, etc., are located in the FPGA fabric, and their I/O values are placed in either BRAMs or slice registers. The custom GPS peripherals mentioned above place their correlator values into register banks and their corresponding NCO values into registers. When designing the system, each of these different subsystems that the processor must read from or write to are located in the processor's memory map.

Figure 5.7 shows a sample memory map for the vector processor. Note that there is plenty of space to add more peripherals if desired.

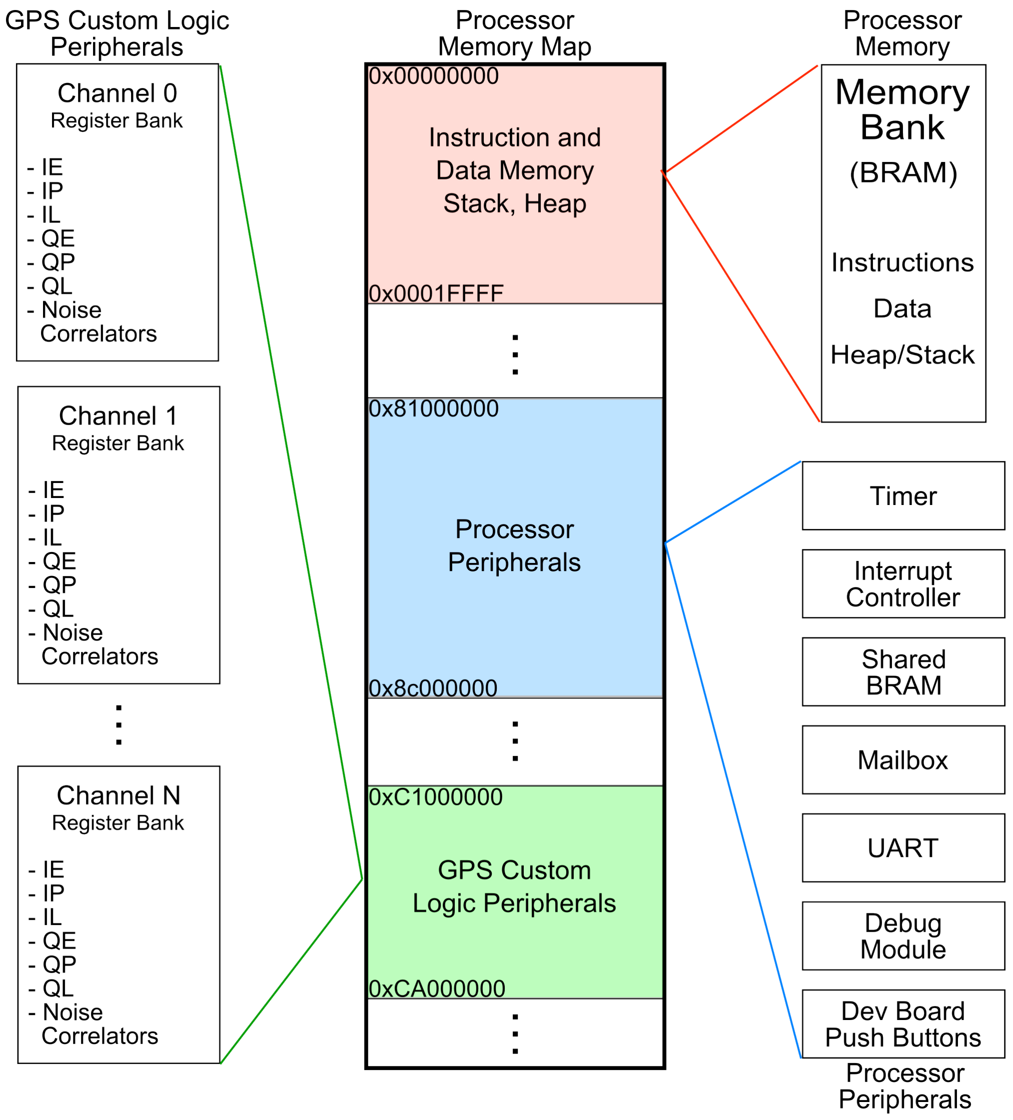

Figure 5.7: Memory map for the vector processor

5.2.7 System I/O and Clock Domains

Figure 5.8 highlights the system I/O and different clock domains that the proposed platform uses. The IF data is read in from the GPS front end at a rate of f_S (16.3676MHz for this research). The given example uses 2-bit (sign and magnitude) quantization. The scalar and vector processors can output debug information to a PC using serial RS232 communication, and the IMU processor also reads RS232 data from the IMU. There is a JTAG interface on the FPGA that allows for debugging and programming. A 3-volt PPS signal is generated in the FPGA logic and is available for the IMU to use for time synchronization. The custom GPS logic modules are all clocked at a rate of f_S, and the baseband soft-core processors and PPS logic are clocked by an off-chip 100MHz reference oscillator. Note that even though the Microblazes are clocked at 100MHz, their 5-stage pipelines give an effective clock rate of around 200MHz.

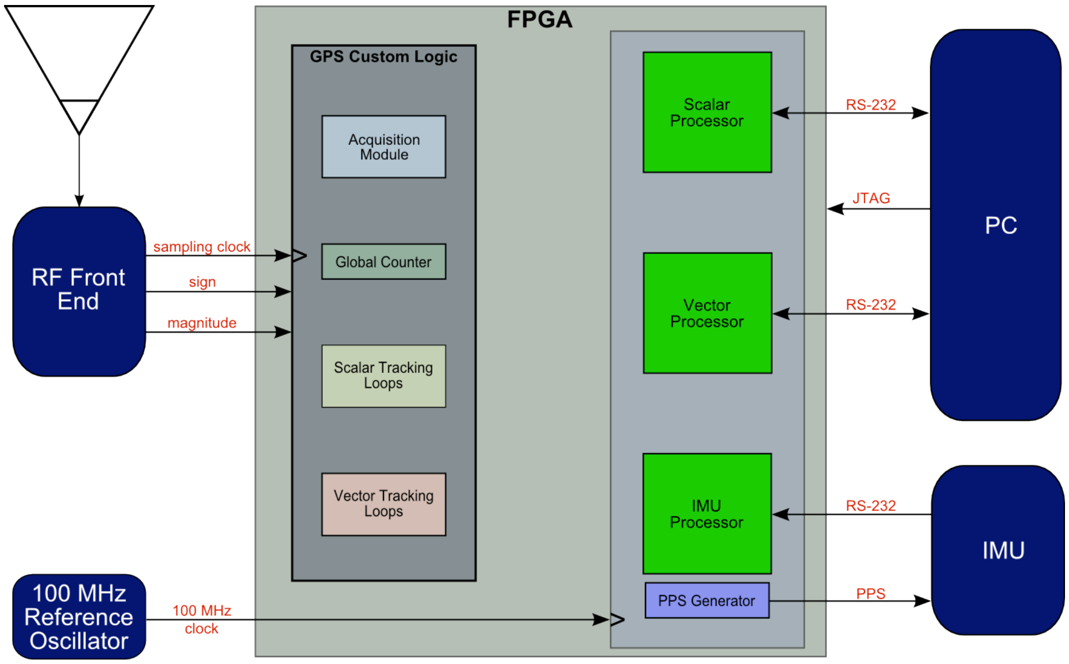

Figure 5.8: Block diagram of the FPGA I/O and clock domains

The hardware and software proposed in this chapter is one architecture chosen from a multitude of options. An advantage of this architecture is that the modularity of the design allows the proposed receiver to be scalable so that the system can be tailored to a particular FPGA or purpose. The separate processors allow the receiver to be used for research in scalar tracking only, scalar and vector tracking, or scalar and/or vector with inertial aiding. The possibilities of this platform are very promising for future research goals.

Chapter 6

Computational Results

6.1 Hardware Results

Table 6.1 is a listing of the amount of resources used for each of the different modules of the system proposed in Chapter 5 and their maximum operating frequency. Note that the custom GPS peripherals (acquisition module, scalar tracking channels, and vector tracking channels) have a much higher resource usage than a standalone version of the same module would have. This is because of the extra logic required to format and deliver their outputs and inputs to the processors via a high-speed data bus. The total system described in Table 6.1 uses 8 tracking channels for both the scalar and vector solutions.

Table 6.1: Resource utilization for 8-channel system using Virtex-5 SXT50T FPGA

Module Name	**Slice Registers**	**Slice LUTs**	**BRAMs**	**DSP48Es**	**Maximum Frequency**
Acquisition module	551	388	2	12	165 MHz
Scalar tracking channel (single)	580	506	1	6	183 MHz
Vector tracking channel (single)	930	722	1	24	182 MHz
GPS counter	68	72	0	0	310 MHz
Scalar processor	1963	2236	16	5	177 MHz
Vector processor	2274	2571	64	5	179 MHz
IMU processor	2063	2426	8	5	179 MHz
Total (8 channels, with all peripherals)	20104	19305	111	267	N/A

6.2 Software Results

The acquisition module performs very well. A serial search algorithm was employed using MATLAB to verify the analytical results. Figure 6.1 shows a comparison of the MATLAB and ModelSim instances of the search algorithm on the same data. This data is searched for satellite 31 using the serial search algorithm. The correlation peak is at the same code phase but at different correlation heights. This is due to the fixed-point nature of the FPGA. Quantization along with truncation of the correlator decimal values results in a slightly lower peak than the full-resolution MATLAB implementation.

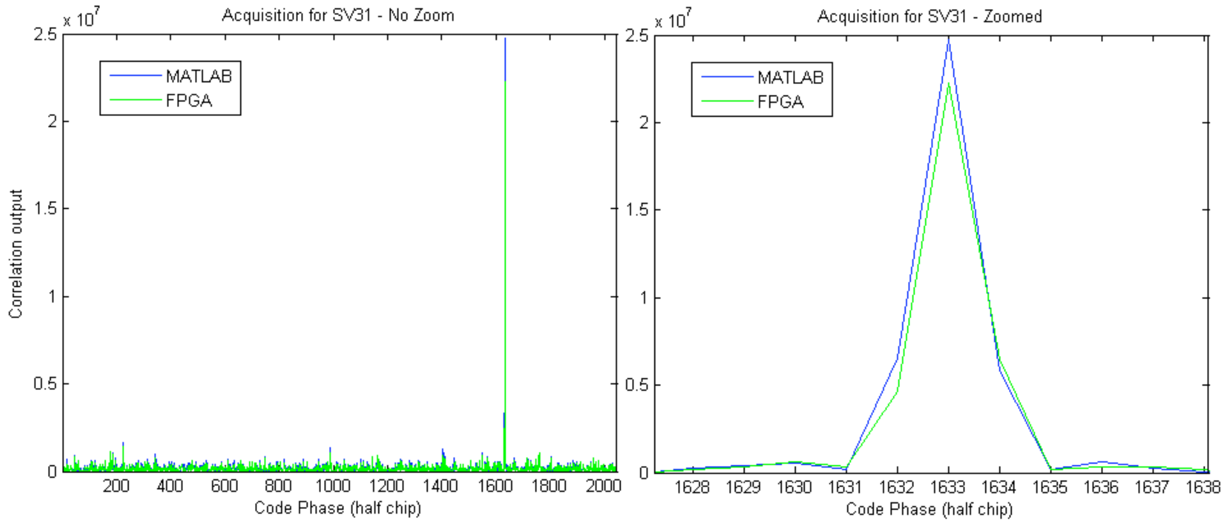

Figure 6.1: Serial search acquisition module results in MATLAB and FPGA (using ModelSim)

A simulation of the scalar tracking loops also performs well. Figures 6.2 through 6.4 show a particular instance of a code DLL and carrier PLL settling to zero after an initial Doppler frequency offset and the subsequent correlator outputs. Output IP is the in-phase prompt output, which clearly shows the incoming data bits of the system.

The vector processor performs each of the measurement updates asynchronously. Since each tracking channel interrupts the processor every 20ms, the vector processor must be able to perform a measurement update in 20ms / number of channels in the system. Since the measurement updates are computed asynchronously, the amount of process time per

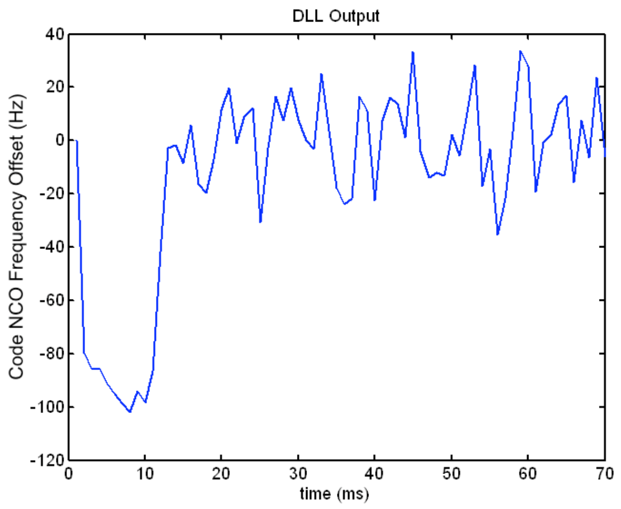

Figure 6.2: Scalar processing simulation - delay locked loop output

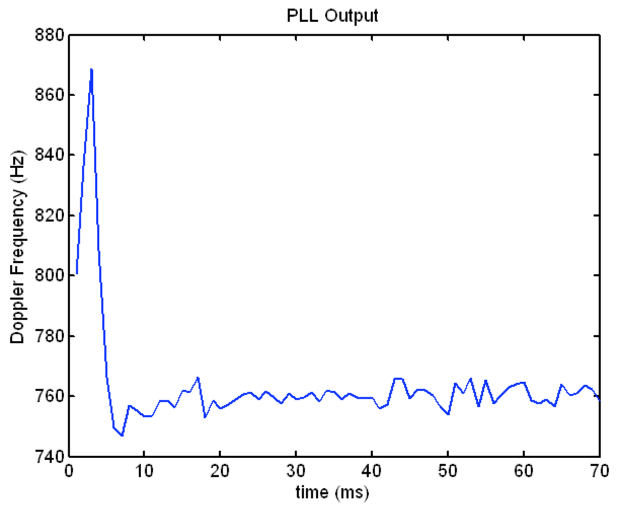

Figure 6.3: Scalar processing simulation - phase locked loop output

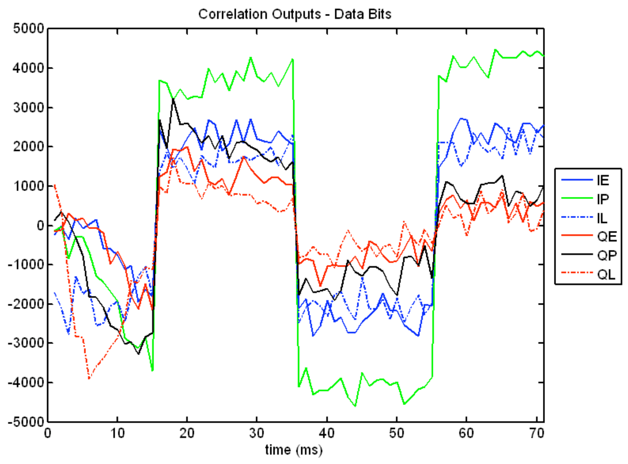

Figure 6.4: Scalar processing simulation - correlator outputs

channel depends on the total number of channels. Table 6.2 shows the computation time for a 6-channel receiver.

Table 6.2: Vector processor real time feasibility analysis using 6-channel receiver baseline

Algorithm	Number of states	Process time per channel	Real-time?
Vector tracking (GPS only)	8	1.36 ms	Yes
Deep Integration (GPS + IMU)	17	28 ms	No

Table 6.3 gives a more detailed description of the time requirements for vector tracking, and a graphical representation of the same data is shown in Figure 6.5. These tables show the process time per channel and the total time for all channels together (neglecting overhead).

Table 6.4 gives a more detailed description of the time requirements for deep integration using asynchronous measurement updates, and a graphical representation of the same data is shown in Figure 6.6.

In the proposed system's current state, GPS-only vector tracking is possible in real time, but deep integration is not. Chapter 7 will address some of the options for achieving this desired real-time performance.

Table 6.3: Vector tracking (asynchronous) timing results

Number of channels	Process time per channel	Process time (all channels)	Real time?
1	1.0703 ms	1.0703 ms	Yes
2	1.1052 ms	2.2104 ms	Yes
3	1.1593 ms	3.4781 ms	Yes
4	1.2145 ms	4.8582 ms	Yes
5	1.2688 ms	6.3444 ms	Yes
6	1.3233 ms	7.9398 ms	Yes
7	1.3766 ms	9.6364 ms	Yes
8	1.4319 ms	11.4552 ms	Yes
9	1.4853 ms	13.3678 ms	Yes
10	1.5317 ms	15.3170 ms	Yes

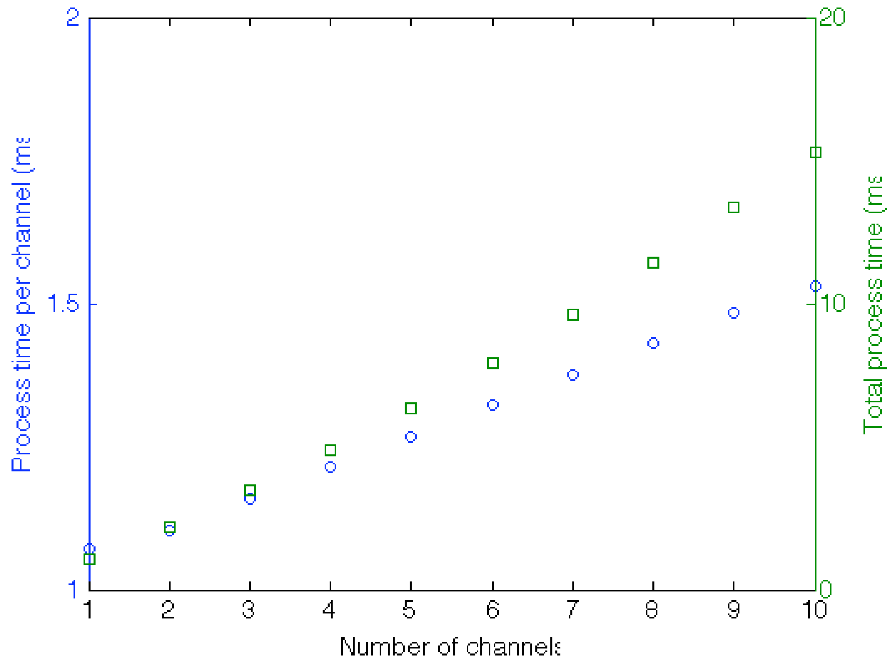

Figure 6.5: Graphical representation of (asynchronous) vector tracking timing results

Table 6.4: Deep integration (asynchronous) timing results

Number of channels	Process time per channel	Process time (all channels)	Real time?
1	9.7844 ms	9.7844 ms	Yes
2	13.1740 ms	26.3481 ms	No
3	16.9487 ms	50.8461 ms	No
4	20.7137 ms	82.8546 ms	No
5	24.4829 ms	122.4146 ms	No
6	28.2501 ms	169.5004 ms	No
7	32.0253 ms	224.1770 ms	No
8	35.7993 ms	286.3947 ms	No
9	39.5055 ms	355.5496 ms	No
10	43.3255 ms	433.2550 ms	No

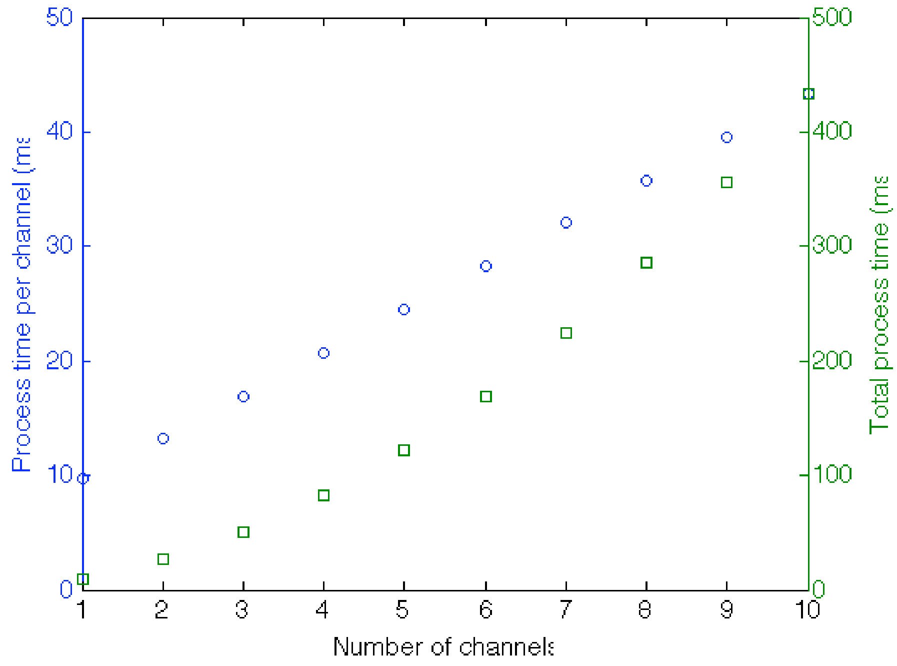

Figure 6.6: Graphical representation of (asynchronous) deep integration timing results

Chapter 7
Conclusions and Future Work

The purpose of this thesis was to document the design of a preliminary real-time embedded system capable of running advanced tracking algorithms. A trade study of the different possible hardware platforms for this system was detailed in Chapter 4, and the FPGA was chosen based on its prowess as a standalone platform (system on chip) and natural segue to ASIC design. In Chapter 5, the detailed hardware and software architectures were outlined for the preliminary design; this chapter highlighted the system on chip aspect of the FPGA. Chapter 6 provides an analysis of the computational results for both the hardware and software modules of the system proposed in this thesis. It has been shown that the design fits with much room to spare on a Virtex-5 SXT50T FPGA, but it also shows that the system's performance is limited by the soft-core processor's computational power. In Chapter 7, an outline will be given for how to address the current system's inability to perform deep integration in real time. In summary, the contributions of this thesis are:

- A trade study of different possible hardware platforms for a real-time embedded GPS receiver capable of running advanced tracking algorithms in real time (Chapter 4)
- Detailed hardware and software architectures for a proposed real-time embedded GPS receiver on an FPGA, capable of running advanced tracking algorithms (Chapter 5)
- Computational results of the preliminary design of the real-time embedded GPS receiver, where vector tracking is shown to be real-time capable but deep integration is not (Chapter 6)

7.1 Conclusions based on Computational Results

Using an FPGA as a platform for a combined scalar and vector tracking solution supporting deep integration is very appealing. It gives the designer control over every signal in the entire system, and every part of the system (with the exception of the RF front end, IMU, and reference oscillator) is on the FPGA itself. The FPGA is capable of supporting I/O, memory, high frequency processing, and baseband processing in the same chip. However, this comes at a loss of computational power for the baseband operations. In Chapter 6, it was shown that the 8-state vector tracking formulation performs in real time but the 17-state deep integration currently does not.

There are many different options that might be useful in achieving this real-time performance. Currently, all of the baseband processing is done in floating point arithmetic. The Microblaze processors support a hardware floating point unit, but it is still about 4 times slower than fixed-point operations [51]. Also, more functions that are currently being done sequentially in the vector processor can be offloaded onto a peripheral in the FPGA logic. There might also be several ways to reduce the number of states in the EKF. If the application can be constrained to two dimensions, several states can likewise be eliminated. Finally, if all of these potential solutions still do not achieve real-time performance for deep integration, a study might be done to explore other external chips such as a DSP or GPP to do the baseband processing.

The ability to integrate the high frequency modules and baseband processors into the same device does offer many advantages. It is especially useful as a prototype for ASIC production because very accurate timing information can be obtained since everything is on the same chip and the design tools are from the same vendor. It is also very flexible in that many different types of microprocessors may be synthesized into the hardware to test out the effectiveness of different platforms. However, the FPGA is often far more expensive than some very powerful microprocessors. Since the FPGA is far more useful as a high frequency device than as a baseband processor, interfacing the FPGA with external memories and external processors might be advantageous. A much smaller, cheaper, and less powerful FPGA can be purchased and then interfaced with these external devices to obtain a very powerful system that is less expensive than an FPGA-only option. As a research platform and low-quantity product, this approach might be superior, but as a precursor to ASIC design, the system on chip approach may be more advantageous.

7.2 Proposed Low-Cost, High Performance Solution

As discussed in the previous section, the potential exists to explore the option of offloading the baseband operations for the scalar, vector, and IMU processing to an off-chip DSP or GPP. This separation into different hardware pieces destroys the potential for a standalone ASIC design, but the results of using a smaller, less powerful FPGA in conjunction with a powerful external processor stands to gain a huge cost (and likely performance) benefit. This section will outline a proposed solution along with cost analysis for production.

7.2.1 Separating High Frequency Operations from Baseband

In Chapter 5, each hardware piece was independently implemented and tested. The high frequency custom GPS operations, such as most of the acquisition module, scalar tracking loops, and vector tracking loops, are modular, and therefore many of these can be simply used as modular blocks when implementing a larger receiver. These high frequency operations are very well-handled by the FPGA, where high clock frequencies are natural. Communication between the high frequency components and the baseband processors is accomplished via memory banks and memory-mapped input/output, as discussed in Section 5.2.6. The baseband processors are three distinct soft-core microprocessors with three separate purposes: scalar (traditional receiver) processing, vector (and deep integration) processing, and the inertial measurement device processing. Each of the processors communicates with their own high-frequency components using the aforementioned memory mapping and also passes information among each other using shared block RAM and FIFO mailboxes.

Because the interface between the high-frequency components and the baseband processors is a simple memory interface, these components can be completely separated into different pieces of hardware. The high frequency components can remain on the FPGA and can write into either its block RAM or an external memory. The baseband processors can become three external processors (DSPs or GPPs). More effectively, they can be placed on a single, more powerful DSP or GPP that handles all of the baseband functions concurrently. This processor can interface with the block RAM of the FPGA via the FPGA's I/O pins, or it can read from the same external memory that the FPGA writes to. Figure 7.1 shows the logical separation discussed in this section compared to the architecture of the receiver prototype presented in this thesis.

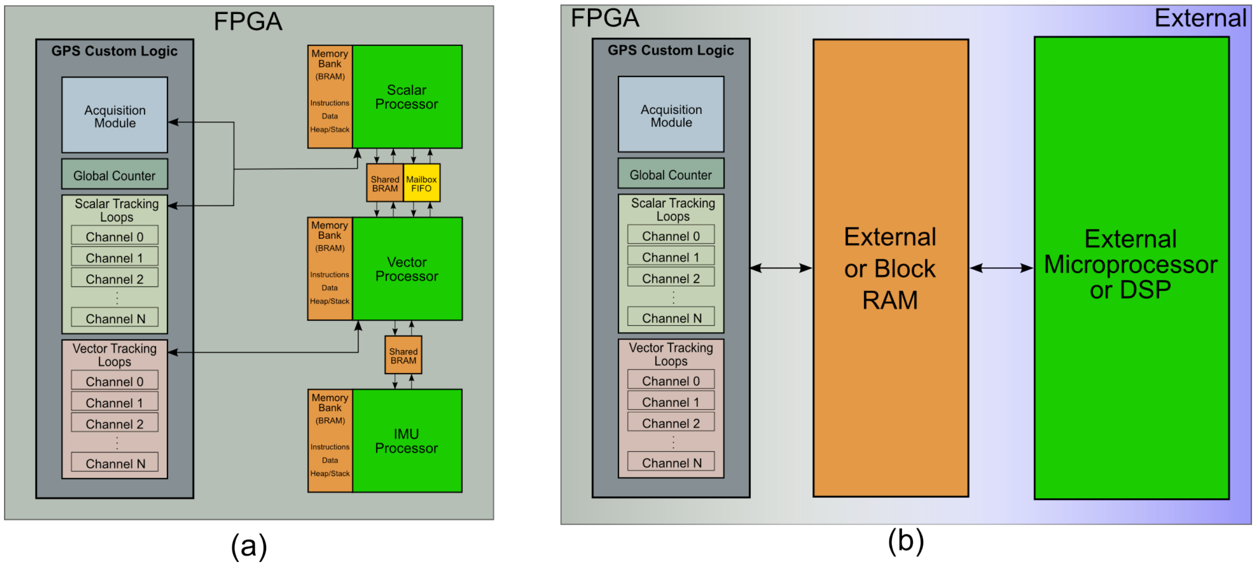

Figure 7.1: Comparison of (a) fully integrated design as described in thesis versus (b) proposed segregated system

7.2.2 Finding a Replacement FPGA

If the only components that are on the FPGA are the high-frequency modules, this leaves a wealth of free space on the FPGA platform used in this work (Virtex-5 SXT). This particular platform has a very high relative cost. At the moment of this writing, a standalone version of this FPGA costs close to $1000 [14]. It is important to synthesize only the components needed onto a smaller FPGA to determine the potential cost of a system. These components, as mentioned, are the high-frequency modules such as signal acquisition, the scalar tracking channels, and the vector tracking channels. In order to gauge the size of FPGA needed, these components, along with a memory controller and block RAM (if interfacing memory is on the FPGA itself) must be synthesized on an FPGA. The method of determining the smallest FPGA is to verify that the aformentioned components completely fit inside the FPGA and that the maximum speed of each component is appropriate. For example, the acquisition and tracking loop components must perform at at least the front end sampling frequency, and the block RAM must perform at an appropriate speed to interface with the external processor.

A top-level hierarchical model has been implemented which takes the following inputs: GPS clock, GPS data, GPP clock, GPP address, GPP data to write, and GPP write enable. The top layer model includes a single declaration of the acquisition module, 16 scalar channels, and 16 vector channels. The choice to have 16 channels of each might seem excessive, but in case future work calls for the inclusion of WAAS satellites or other satellite systems, this choice might help to prevent the potential problem of choosing an FPGA that is too small to satisfy future research needs. The system that has been synthesized for testing is not complete. For the moment, a temporary placeholder memory controller has been used that will loosely mimic the behavior of a full controller. A block diagram of the model to be synthesized is shown in Figure 7.2.

The above system will likely fit onto a Spartan-3A DSP FPGA. This FPGA costs close to \$80, which is about 92% less than the original FPGA (about \$1000). This system would, however, incur extra costs such as external memory and processor(s).

7.2.3 Replacing Microblazes with an External Processor

There are a number of options for replacing the Microblaze soft-core processors with a powerful DSP or GPP. In Chapter 4, a number of the benefits and potential drawbacks to

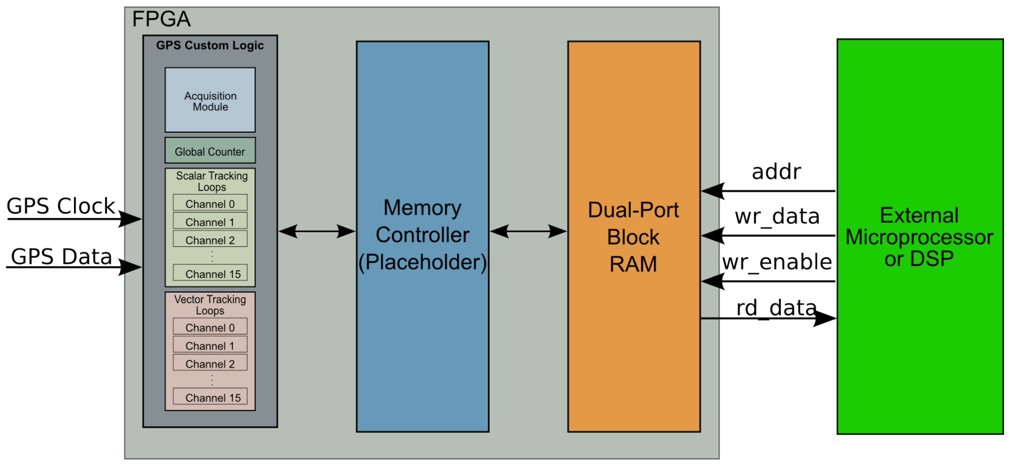

Figure 7.2: Block diagram of the proposed hardware changes

each of these platforms was discussed. Because the current system is still being used as a research platform, it is wise to explore processor options that are available as a development kit. There has been a considerable interest in the field of embedded applications which uses the ARM Cortex processor architecture. This ARM architecture is avaliable as synthesizable code that can be implemented on any logic device. ARM processors are very well documented and can be found in a number of development kits. However, the speed of these processors varies considerably, and it is difficult to tell whether or not a given device will supply enough processing power for this job without further investigation. These processors do include a floating point unit.

Recently, researchers have used the TMS320C6455 DSP from Texas Instruments to create an L1/L2 software receiver in full, including all high-frequency operations [32]. This DSP operates at 1.2GHz and has development software support for interfacing with FPGAs [45]. The starter kit costs $595, and the chip itself costs about $300 [14]. A lower-speed version of the same processor can be found for under $170, which might also be suitable. A drawback with development on this DSP is that it is fixed-point only. Floating point operations can be accomplished with software, but much higher performance would necessitate conversion from floating to fixed point operation.

7.2.4 Other Potential Replacement Options

If available, a Virtex-5 FXT device can be used instead. This platform has the embedded PowerPC processors which run at 550MHz, which is over twice the speed of the Microblaze. If the processor performance has a linear relationship with the processing time, this 550MHz platform is still potentially not fast enough. Instead, a combination of the PowerPC processors and fixed point math could potentially achieve deep integration in real time.

Another potential solution which would be very beneficial for this particular research platform is to take advantage of an FPGA's built-in "extras" such as a PCI Express controller, Ethernet PHY, USB, or any other physical interface that can interface with a standard PC. This way, development on a GPP can take place without specialized hardware. The Virtex-5 series of FPGAs, including the one used in this thesis, offer a PCI Express controller that can be interface with a PCI Express-capable PC. This way, the Virtex-5 can be placed inside of the PC and development can take place with no extra hardware costs.

7.3 Future Work

This thesis has given the details of an embedded system that can perform vector tracking in real-time but not deep integration. This issue should be addressed by the potential solution given in Section 7.2. Once the system is capable of closing the deep integration loop in real time, the system should be interfaced with a the SiGe 4110L hardware front end and tested to verify that the signal integrity is maintained. It would be particularly useful to create a method of testing the proposed system with real data without having to use FPGA modeling software such as ModelSim; therefore, a software add-on which allows raw GPS and IMU data to be read from a text file and then computed in "pseudo" real time. This would ensure algorithm integrity on actual FPGA logic instead of pure software simulation. Once the system is completed, it can be used to test many advanced GPS algorithms such as fault detection exclusion (FDE) and differential GPS. It would be particularly useful to study the effects on quantization and computational burden mitigation on these algorithms for deployment in real systems.

Bibliography

[1] M. Abramovici and C.E. Stroud. BIST-based test and diagnosis of FPGA logic blocks. *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, 9(1):159 –172, feb 2001.

[2] Peggy Abusaidi, Matt Klein, and Brian Philofsky. Virtex-5 FPGA System Power Design Considerations. White paper, Xilinx, Inc, February 2008.

[3] Dennis M. Akos. *A Software Radio Approach to GNSS Receiver Design*. Ph.d dissertation, Ohio University, 1997.

[4] Altera. ASIC Transceiver ASIC design, ASICs. Internet, May 2010.

[5] Don Benson. Interference Benefits of a Vector Delay Lock Loop (VDLL) GPS Receiver. In *63rd Annual Meeting of the ION*, Cambridge, MA, April 2007. ION.

[6] Roland E. Best. *Phase-Locked Loops: Design, Simulation, and Applications*. McGraw Hill, New York, NY, 5th edition, 2003.

[7] Kai Borre, Dennis Akos, Nicolaj Bertelsen, Peter Rinder, and Soren Holdt Jensen. A Software-Defined GPS and Galileo Receiver: Single-Frequency Approach. 2006.

[8] H.S. Cho, S.H. Ira, and G. Jee. A FPGA-based software GPS Receiver Implementation Using Simulink and xilinx system generator. In *Proceedings of ION GNSS the 18th International Technical Meeting of the Satellite Division, Long Beach, CA, The Institute of Navigation, Inc*, pages 234–240, 2005.

[9] Stewart Cobb. FPGA-Optimized GNSS Receiver Implementation Techniques. In *Proceedings of the ION GNSS*, Savannah, GA, September 2008. ION.

[10] RC Cofer and Ben Harding. *Rapid System Prototyping with FPGAs: Accelerating the Design Process*. Newnes, 2005.

[11] Leon W. Couch. *Digital and Analog Communication Systems*. Pearson Prentice Hall, 7th edition, 2007.

[12] Robert Crane. A simplified method for deep coupling of GPS and inertial data. In *Proceedings of the National Technical Meeting of the Institute of Navigation*, San Diego, CA, 2007. ION.

[13] Viswanath Daita. Behavioral VHDL Implementation of Coherent Digital GPS Signal Receiver. Master's thesis, University of South Florida, 2004.

[14] Digi-Key. Digi-Key Catalog. Internet, June 2010.

[15] F. Dovis, F. Maurizio, and M. Pini. Efficient signal acquisition and tracking for a real time GPS/Galileo software receiver. pages 2071 –2075, oct. 2008.

[16] L. Edwards, M. Lashley, and D. Bevly. FPGA implementation of a vector tracking gps receiver using model-base.

[17] Luke Edwards, Benjamin Clark, and David Bevly. Implementation Details of an Embedded Deeply Integrated GPS/INS Navigation system. In *IEEE/ION Position Location and Navigation Symposium 2010*, Palm Springs, CA, May 2010.

[18] Neil Gerein and Michael Olynik. NovAtel's GIOVE Monitoring Receiver. Technical report, NovAtel, 2008.

[19] R. Gold. Optimal binary sequences for spread spectrum multiplexing (Corresp.). *Information Theory, IEEE Transactions on*, 13(4):619 – 621, oct 1967.

[20] Donald E. Gustafson, John R. Dowdle, and Jr. John M. Elwell. United States Patent 6,331,835, December 2001.

[21] LaMacchia M. Hasan, M. S. Designing the Joint Tactical Radio System (JTRS) Handheld, Manpack, and small form fit (hms) radios for interoperable networking and waveform applications. *IEEE*, 2007.

[22] Thomas Hobiger, Tadahiro Gotoh, Jun Amagai, Yasuhiro Koyama, and Tetsuro Kondo. A GPU based Real-time GPS Software Receiver. *GPS Solutions*, 14(2), March 2009.

[23] Todd Humphreys, Jahshan Bhatti, Thomas Pany, Brent Ledvina, and Brady O'Hanlon. Exploiting Multicore Technology in Software-Defined GNSS Receivers. In *22nd International Technical Meeting of the Satellite Division of The Institute of Navigation*, Savannah, GA, September 2009. ION GNSS.

[24] Elliot D. Kaplan and Christopher J. Hegarty, editors. *Understanding GPS: Principles and Applications*. Artech House, 2nd edition, 2006.

[25] Clifford Kelley, J Cheng, and J Barnes. Open Source Software for Learning about GPS. In *15th Int. Tech. Meeting of the Satellite Division of the U.S. Inst. of Navigation*, Portland, Oregon, September 2002.

[26] Stefan Kiesel, Andreas Maier, Ulrich Seiller, and Gert Trommer. Preliminary Simulation Results of a Deeply Coupled GPS/INS System for high dynamics. In *IAIN*, 2009.

[27] Ian Kuon and Jonathan Rose. Measuring the Gap Between FPGAs and ASICs. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 26(2):203–215, February 2007.

[28] Matthew Lashley and David Bevly. Comparison of Traditional Tracking Loops and Vector Based Tracking loops for weak gps signals. In *Proceedings of the ION NTM*, pages 789–795, San Diego, CA, January 2008. ION.

[29] Yong Li, Peter Mumford, and Chris Rizos. Performance of a Low-cost Field Reconfigurable Real-time GPS/INs integrated system in urban navigation. In *Proceedings of the IEEE*. IEEE, 2008.

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

[31] Sanjit K. Mitra. *Digital Signal Processing: A Computer-Based Approach*. McGraw-Hill Education, New York, NY, 3rd edition, 2006.

[32] Brady O'Hanlon, Todd Humphreys, M Psiaki, and P Kitner. Development and Field Testing of a DSP-Based Dual-Frequency Software gps receiver. In *22nd International Technical Meeting of the Satellite Division of The Institute of Navigation*, Savannah, GA, September 2009. ION GNSS.

[33] Thomas Pany, Roland Kaniuth, and Bernd Eissfeller. Deep Integration of Navigation Solution and Signal Processing. In *Proceedings of the ION ITM*, pages 1095–1102. ION, 2005.

[34] K.J. Parkinson, A.G. Dempster, P. Mumford, and C. Rizos. FPGA based GPS Receiver Design Considerations. In *International Symposium on GPS/GNSS*, volume 5, pages 70–75, Hong Kong, December 2006. ION.

[35] Mark Petovello and Gerard Lachapelle. Comparison of Vector-Based Software Receiver Implementations with applications to ultra-tight gps/ins integration. In *Proceedings of the ION ITM*, pages 1790–1800. ION, 2006.

[36] Mark Petovello, Matthew Lashley, and David Bevly. What About Vector Tracking Loops? *GNSS Solutions*, pages 16–21, May/June 2009.

[37] MG Petovello, C. O'Driscoll, and G. Lachapelle. Carrier Phase Tracking of Weak Signals Using Different Receiver Architectures. In *Proceedings of ION National Technical Meeting, San Diego, CA, Institute of Navigation*, pages 781–791, 2008.

[38] Shankararaman Ramakrishnan, Grace Xingxin Gao, David De Lorenzo, Dennis Akos, Todd Walter, and Per Enge. Design and Analysis of Reconfigurable Embedded GNSS Receivers Using model-based design tools. In *Proceedings of the ION GNSS*, Savannah, GA, September 2008. ION.

[39] C. Sajabi, C.-I.H. Chen, D.M. Lin, and J.B.Y. Tsui. FPGA Frequency Domain Based GPS Coarse Acquisition Processor Using fft. pages 2353 –2358, april 2006.

[40] SiGe. *SE4120L PointCharger GNSS Receiver IC - Preliminary Information*. SiGe Semiconductor, August 2006.

[41] Abraham Silberschatz, Peter Baer Galvin, and Greg Gagne. *Operating Systems Concepts with Java*. John Wiley & Sons, Inc., Danvers, MA, 7th edition, 2007.

[42] J J Spilker. *Digital Communication by Satellite*. Prentice Hall, Englewood Cliffs, NJ, 1995.

[43] James Spilker. *Global Positioning System: Theory and Applications*, volume 1, chapter GPS Signal Structure and Theoretical Performance, pages 78–81. American Institute of Aeronautics and Astronautics, 1996.

[44] Charles Stroud. Overview of FPGAs. Internet: http://www.eng.auburn.edu/ strouce/class/elec4200/FPGAoverview.pdf, October 2009.

[45] TexasInstruments. TMS320C6455 DSP Starter Kit (DSK). Internet, June 2010.

[46] James Bao-Yen Tsui. *Fundamentals of Global Positioning System Receivers: A Software Approach*. John Wiley & Sons, Inc., 2000.

[47] Wayne Wolf. *Computers as Components: Principles of Embedded Computer System Design*. Morgan Kaufmann, San Francisco, CA, 2005.

[48] Xilinx. ML505/ML506/ML507 Evaluation Platform User Guide. Internet, October 2009.

[49] Xilinx. Virtex-5 FPGA User Guide. Internet, September 2009.

[50] Xilinx. EasyPath FPGAs. Internet, May 2010.

[51] Xilinx. MicroBlaze Processor Reference Guide. Internet, May 2010.

[52] Zarlink. GP2021 12 Channel GPS Correlator Datasheet. http://www.zarlink.com/zarlink/gp2015-datasheet-sept2007.pdf, August 2005.

[53] Zarlink. GP2015 GPS Receiver RF Front End Datasheet. http://www.zarlink.com/zarlink/gp2015-datasheet-sept2007.pdf, September 2007.

Appendix - Software Receiver using Model-based Tools

The purpose of this appendix is to outline the original attempt at a software receiver that would perform vector tracking in real time. This first attempt was also a prototype and was shown to be able to perform the vector tracking loop closure within the real-time limitations [16]. This attempt used a set of software tools that are essentially a third party blockset that is available to MATLAB's Simulink software. This gives the software used, and therefore the implemented approach, the nomenclature "model-based."

Model-based Tools

In order to speed up development time for digital signal processing (DSP) applications, Xilinx has created a software suite known as DSP Tools. This software is integrated into The Mathworks' MATLAB and Simulink software, so that prior knowledge of programming hardware definition languages (HDLs) is not necessary to implement a working design. System Generator is part of the DSP Tools package, and it functions by simply adding a set of blocks to the Simulink blockset library that can be used to create system models. These system models are then synthesized into logic designs on an FPGA.

These models can be simulated in software, providing a cycle-true simulation of the given system function. System Generator also introduces the concept of hardware co-simulation, whereby a known input can be loaded into the MATLAB workspace and then passed to the FPGA as an input; the FPGA executes its system function on this input data and then sends user-selected outputs back to the MATLAB workspace. This is a very useful testing platform, especially for DSP applications. Using Xilinx's provided software tools, development time can be drastically reduced. More information and some additional examples using these model-based tools can be found in [38].

Handling Vector Tracking Computational Burdens

All of the benefits that vector tracking boasts such as increased jamming and interference immunity comes at a high computational cost. In this appendix, only a few of these are discussed. First, for the particular formulation of the vector tracking architecture used in this appendix, many additional correlators are needed. In a platform such as a microprocessor, each operation must be performed sequentially. Therefore, adding any additional correlators consumes a large quantity of time. Second, vector tracking architectures use an extended Kalman filter that involves keeping track of and operating upon large matrices. Matrix operations in an FPGA are not implicit, and therefore a different approach is needed.

PRN Generation for Multiple Correlators

There are several different formulations of the vector tracking architecture, and each of them include computational drawbacks. The method used in this thesis uses the discriminators and noise variance determination as described by Robert Crane in [12]. The approach by Crane involves a bank of noise correlators to obtain noise power levels. The number of additional correlators can vary, but the authors have determined that past an additional 18 correlators, the benefit of adding any more is negligible. These additional correlators are the early, prompt, and late outputs of the same pseudo-random noise (PRN) sequence shifted by a value that is centered on nulls in the autocorrelation function of the PRN sequences.

There is a relatively simple solution to this problem that can be easily realized with the model-based tools discussed above. One way that a PRN code can be generated is by modulo-2 adding maximal-length sequence G1 with a shifted copy of another maximal-length sequence, G2 [43]. Each of these sequences is 1023 bits long, and can fit compactly into a few FPGA lookup tables (LUTs). One thing to notice is that it is much more efficient to place the 1-bit values (0 and 1) of G1 and G2 into LUTs and then convert that number to 2-bit ($\pm$ 1) using an exclusive NOR gate and concatenating a 1 to the least significant bit. A very similar solution to this problem can be realized using two of the 36K block RAMs and the same addressing scheme. There are also a large amount of delays that might typically take quite a few memory elements to implement in a DSP or PC, but using an FPGA, about half of the lookup tables can function as 32-bit shift registers [49], so very little logic is needed for this method. The integrate and dump operations are simply multiply and accumulate operations, so only a single DSP48E is needed for each of the 20ms integrate and dump operations. Figure A.1 shows this process using only a single noise correlator.

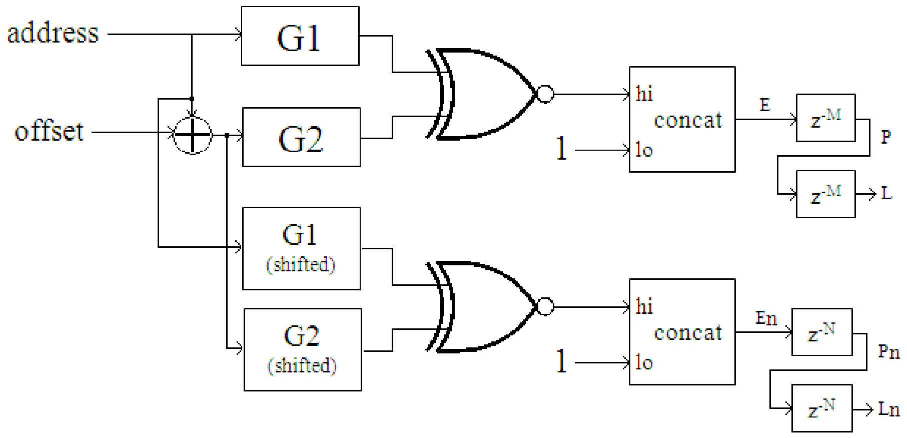

Figure A.1: Block diagram of PRN generation for normal and noise correlator banks

As mentioned before, the delays listed here correspond to nulls in the autocorrelation function. These values give a very good estimate of the noise power in the system, which is critical for determining signal amplitude and C/N_0. The values given in Figure A.2 correspond to a 1023-bit PRN signal being indexed by a counter that counts from 0 to 16367. This number comes from using a sampling frequency f_s of 16.3676MHz, so that one millisecond of data resides in close to 16368 samples of data. The values of the delay reflects this f_s value. The ROM shift values shown in Figure A.2 are the circular shift offsets for each 1023-bit ROM that place the "prompt" outputs on the center of the autocorrelation nulls. Figure A.2 shows this technique done using the model-based tools from Xilinx using three noise PRN ROMs (wrap checking not included).

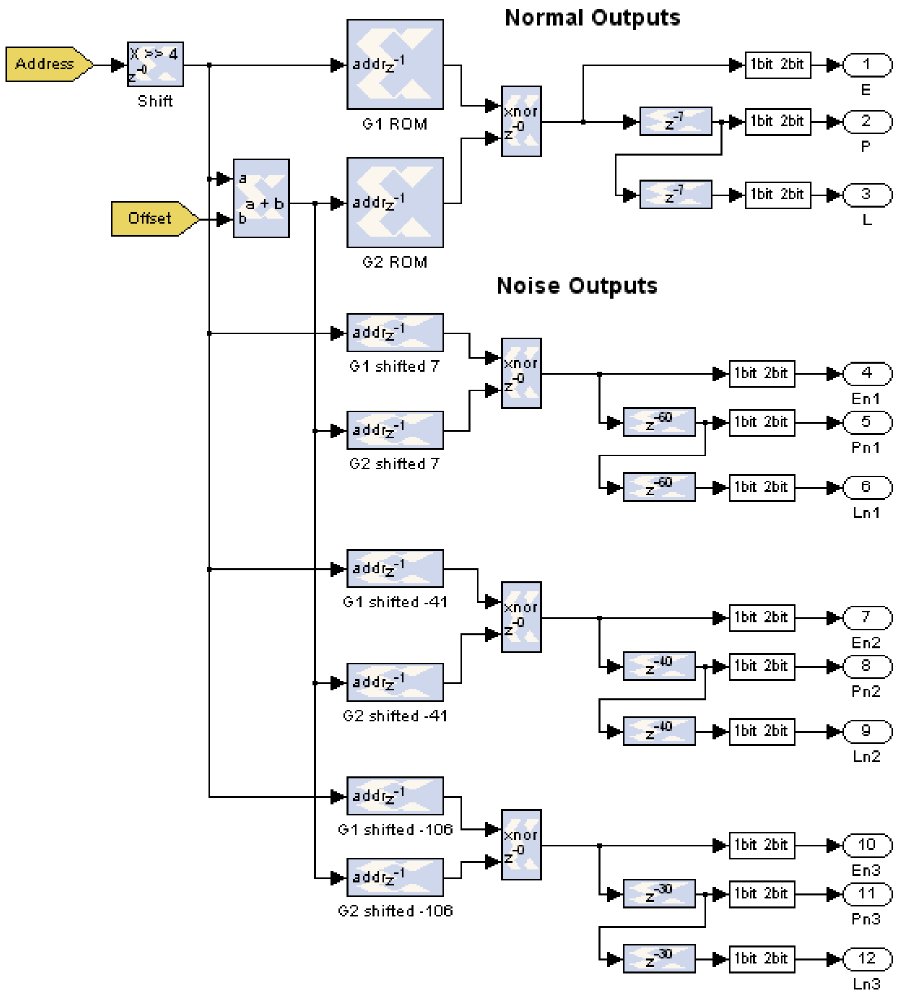

Figure A.2: PRN generation for normal and noise correlator banks using model-based tools

Matrix Math on the FPGA

Regardless of which formulation of the vector tracking architecture is used, an extended Kalman filter is used as the system's state estimator. The EKF is inherently matrix-intensive. Its equations will not be listed here, but Table A.1 shows the size of each of the matrices used in the EKF, where M is the number of states in the system and N is the number of satellites that are being tracked. This information gives the reader an idea of how large the code space might be if implemented in fixed point or floating point math.

Table A.1: Functions and sizes of matrices used

Matrix Name	**Description**	**Size**
Q	Process noise covariance	M×M
A	State transition matrix	M×M
P	Error covariance	M×M
C	Observation matrix	2N×M
K	Kalman gain	M×2N
$\hat{x}$	State vector	M×1
y	Measurement vector	2N×1
R	Measurement covariance	2N×2N

The most computationally expensive part of the EKF is the matrix inversion, and in the case of vector tracking, the size of the inverted matrix is the same as that of R, 2N×2N, so one of the major computational bottlenecks is the number of channels that the receiver architecture uses. Unlike a microprocessor, an FPGA does not consider input and output (I/O) in terms of memory mappings; this behavior must be explicitly defined within the logic itself. Also, although hardware reconfiguration of an FPGA at runtime is possible, it is not likely that a matrix inversion algorithm can be easily expanded in hardware from a 12×12 matrix to a 16×16 matrix. Even if this could be done, it is not realistic in terms of resource usage, especially when inversion is considered alongside all the other matrix arithmetic in the EKF. A microprocessor is almost completely necessary to accomplish this.

As mentioned previously, FPGA developers have begun to place embedded 32-bit microprocessors within the FPGA fabric. Unfortunately, the Virtex-5 SXT family does not include any of these hard-cored processors. However, a 32-bit processor called a MicroBlaze can be synthesized within the FPGA logic which includes the same functionality. This processor can be programmed in assembly, C, or C++ code, so existing algorithms can be easily ported to the FPGA platform. In this research, a custom matrix library was written in C to perform the EKF operations with fast performance and efficient memory management.

The use of both an embedded processor and the model-based tools was anticipated by the FPGA developers, so there is a simple interface for exchanging information between them. In fact, because a microprocessor's I/O is memory-mapped, any information needed to or from the model-based tools is mapped to a memory location on the processor. Loading or sending information is as simple as reading from or writing to a memory location. It is important to note that the MicroBlaze can only operate at speeds up to around 200MHz. The embedded PowerPC processors operate much faster at 550MHz, and any off-chip DSP or microprocessor can be used here as well. The MicroBlaze was chosen because of the relative ease in interfacing it with the rest of the FPGA logic, and might eventually be replaced when the algorithm's real-time requirements supersede the ability of the MicroBlaze. Figure A.3 shows a block diagram of the GPS system on the FPGA.

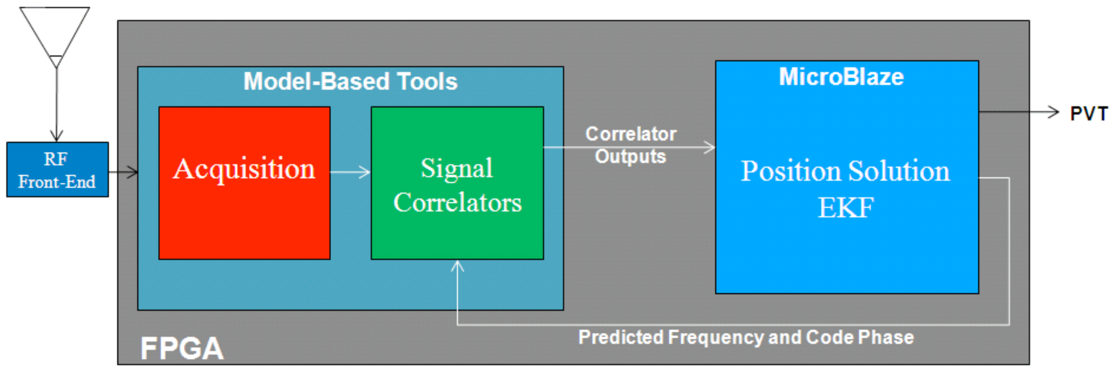

Figure A.3: Block diagram of FPGA GPS receiver using model-based tools

Results

FPGA Resource Usage

One of the most important considerations when determining a suitable FPGA to use for a particular application is the amount of resources required to perform the desired operation. In the solution described in the previous section where the PRN sequences for the normal and noise correlators are generated, sequences G1 and G2 and their shifted copies are packed nicely into lookup tables (LUTs), and the delays are shown to have been implemented efficiently with LUTs acting as 32-bit shift registers. Table 7.3 shows the FPGA resource usage needed to implement this behavior. This uses the model from Figure A.2 to obtain these results. There were no block RAMs or DSP48Es used in this design.

	Used	**Device Total**
Slice Registers	25	32640
Slice LUTs	162	32640
LUTs used as Shift Registers	12	162

Table A.2: Resources used for PRN generation using presented technique

We shall consider a vector tracking correlation channel to be the PRN generation, numerically controlled oscillators (NCOs), and integrate and dump operations. A single vector tracking correlation channel requires the resources as displayed in Figure A.4.

Including the MicroBlaze synthesized processor as discussed above and including an additional seven channels, an eight-channel receiver can easily fit on the hardware platform, as shown in Figure A.5. Notice that this does not include the acquisition step or the scalar loops. The size of these other modules will not be listed in this appendix, but they have been presented in [38] for a similar FPGA platform.

Real-time Requirements

The data used for this research was recorded using a NordNav IF recorder with a sampling frequency f_s of 16.3676 MHz and an IF of 4.1304 MHz. One millisecond of sampled

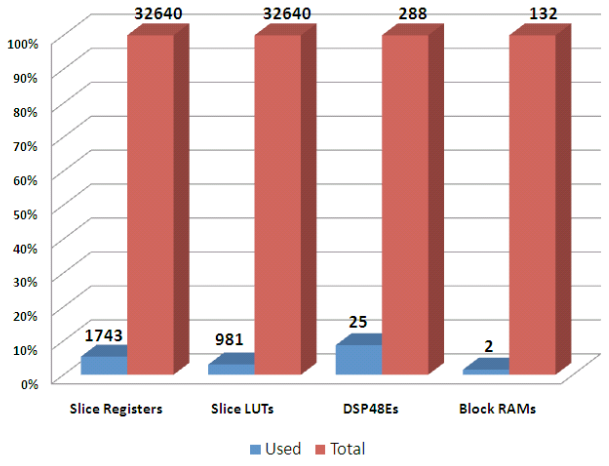

Figure A.4: Single vector tracking correlation channel resource usage

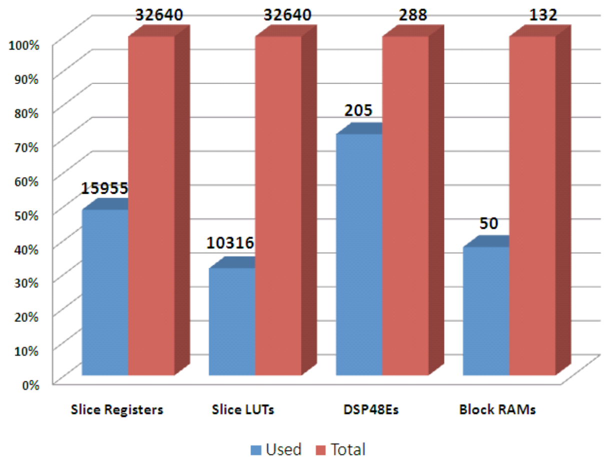

Figure A.5: 8-channel vector tracking receiver resource usage (without acquisition module)

data (16367 or 16368 samples) from the NordNav was collected at sampling frequency f_s and placed in an FPGA block RAM. This block RAM is read at the FPGA system clock of 100MHz and then waits on the next millisecond of data to be placed into the RAM. Using this method, several memory elements are saved and thus the 20 ms integrate and dump is the total of twenty 1 ms integrate and dumps.

Each iteration of the EKF must be completed in 20 ms, so in order to perform in real time, all calculations must be performed in that 20 ms window. Table A.3 represents each of the tasks that must be accomplished by a 6-channel vector tracking receiver and a comparison of the amount of time taken by the FPGA and MATLAB implementations. The MATLAB implementation of the signal correlation uses a mex-function (written in C) for increased speed. These tests were performed on a 2GHz dual core PC with 2GB of RAM. The FPGA system performance here is limited by the speed of the MicroBlaze and could be improved upon using a PowerPC processor or an external processor. This performance could also be improved by placing the C/N_0 estimation and discriminators into the FPGA fabric instead of the MicroBlaze. However, this would save only a small amount of time, as the most prominent sources of latency are the EKF iterations and satellite PVT calculation. The $*$ in Table A.3 denotes that there is some period of time that must elapse for the final millisecond of correlation to be done on the FPGA, but there is no additional processing that must be done on the MicroBlaze besides retrieving this data and converting it to a usable type. Notice that even a C (MEX) implementation of the signal correlation is many times slower than the FPGA.

Conclusions

In this appendix, a GPS vector tracking software receiver that is capable of performing in real time is presented. Some of the computationally expensive elements of vector tracking are considered and their potential solutions are discussed. This appendix focused on the use

Function	**FPGA Time Required**	**MATLAB Time Required**
Retrieve and Convert Correlator Outputs (×6)	0.411 ms (×6)	Not Required
Compute Satellite PVT (×6)	1.3194 ms (×6)	0.1336 ms (×6)
Predict Code Phase and Doppler (×6)	.00278 ms (×6)	0.1394 ms (×6)
Signal and Noise Correlators (×6)	0.1637 ms*	115.573 ms (×6)
Discriminators and C/N$_0$ Calculation (×6)	0.1644 ms (×6)	0.1538 ms (×6)
EKF Iterations	3.8111 ms	0.6825 ms
Miscellaneous Setup	0.2119 ms	21.222 ms
Total	**15.5722 ms**	**717.9035 ms**

Table A.3: Comparison of time elapsed between FPGA and MATLAB/C implementation

of model-based tools to solve these problems, but these solutions can also be extended to hardware description languages for maximum resource utilization.

A field-programmable gate array has been shown to be a very capable hardware resource for prototyping and designing GPS software receivers because of their ability to perform many operations in parallel and its many extra embedded components. Model-based design tools were shown to speed up development time by providing an accessible design process to algorithm developers and receiver designers.

The real-time requirements for vector tracking were met with a 6-channel system, but adding many more channels will not be able to meet this requirement. The limiting factor is the speed of the MicroBlaze microprocessor, so either an embedded PowerPC device or an external processor must be used if more channels are required.