

FPGA-BASED SOFTWARE GNSS RECEIVER DESIGN
FOR SATELLITE APPLICATIONS

SURABHI GURUPRASAD

A THESIS SUBMITTED TO THE FACULTY OF GRADUATE STUDIES
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF
MASTER OF SCIENCE

GRADUATE PROGRAMME IN EARTH AND SPACE SCIENCE AND ENGINEERING
YORK UNIVERSITY
TORONTO, ONTARIO

FEBRUARY 2015

© SURABHI GURUPRASAD, 2015

ABSTRACT

Recent trends suggest GNSS receiver technology has tremendous scope for satellite applications such as radio occultation, precise orbit determination and reflectometry. GNSS receivers developed for satellite applications have several additional hardware and software specifications when compared to their terrestrial counterparts. Spaceborne receivers are characterised by low power requirements, high processing speed and radiation resistant electronic components. Such sophisticated receivers, also called hardware GNSS receivers, are fabricated for specific applications and hence lack design flexibility. On the other hand, a software GNSS receiver allows easy design modifications without any hardware component replacement. Software receivers employ reconfigurable hardware elements called Field Programmable Gate Arrays (FPGAs). Hardware designs can be implemented or modified in FPGAs using Hardware Description Language (HDL).

In this research, a low-power, low-cost software GNSS receiver has been designed and developed using a combination of a microprocessor and FPGA (System-on-Chip or SoC). The developed software GNSS receiver is capable of detecting GPS satellites, tracking these signals and computing receiver position estimates. Efficient task partitioning is achieved by implementing operations in both the FPGA and the microprocessor. Also demonstrated is the improvement of processing speed by 20% when certain GNSS receiver operations are performed in the FPGA instead of the microprocessor.

ACKNOWLEDGEMENTS

I would like to express my sincere gratitude towards my supervisor Professor Sunil Bisnath, for providing me the opportunity to be a member of his research team and learn under his highly appreciated guidance. I would also like to thank him for the continuous support, motivation and expertise he provided throughout this thesis.

I also extend my sincere thanks to Professor Regina Lee, for providing valuable suggestions and insights during the study for this research. I also offer my deepest gratitude to Professor Janusz Kozinski for providing me a marvelous platform to learn and prosper. I also thank him for his constant enthusiasm and support during this thesis.

Last but not the least, I am immensely grateful towards my mother, father and brother for the endless love, support and inspiration they have provided me all these years. This work has been possible only because of them.

Table of Contents

| | |
|--|------|
| ABSTRACT | ii |
| List of Tables | vii |
| List of Figures | viii |
| List of Acronyms | x |
| List of Symbols | xi |
| Chapter One: Introduction | 1 |
| Overview of GNSS Constellations..... | 2 |
| Software GNSS Receivers | 4 |
| Field Programmable Gate Array | 5 |
| Software GNSS Receiver for Satellite Applications..... | 7 |
| Problem Statement | 9 |
| Thesis Objectives | 9 |
| Thesis Outline | 10 |
| Chapter Two: Software GNSS Receiver Using System On Chip | 12 |
| Brief History of Software GNSS Receivers | 12 |
| GPS Signal Characteristics..... | 16 |
| C/A-Code..... | 16 |
| Navigation Data..... | 19 |
| Software GNSS Receiver Architecture | 20 |
| Acquisition..... | 21 |
| Tracking..... | 26 |
| Pseudorange Measurements and Navigation Solution | 32 |
| System-on-Chip Technology..... | 34 |
| Chapter Three: Receiver Implementation..... | 39 |
| Software Implementation | 39 |
| Input Data Format..... | 39 |

| | |
|--|----|
| Acquisition Algorithm..... | 40 |
| Tracking Algorithm..... | 42 |
| Navigation Solution Algorithm..... | 48 |
| Hardware Implementation..... | 50 |
| Processor Selection..... | 50 |
| Embedded ARM..... | 52 |
| Software/Hardware Partitioning..... | 53 |
| FFT Core..... | 56 |
| HPS-FPGA Interconnect..... | 60 |
| Parallel Input/Output Peripherals..... | 61 |
| FIFO Core..... | 63 |
| Chapter Four: Receiver Performance Tests And Analysis..... | 66 |
| Test Setup..... | 66 |
| Signal Acquisition..... | 67 |
| Acquisition Result..... | 69 |
| Code Profiling Results..... | 71 |
| Hardware FFT versus Software FFT..... | 73 |
| Signal Tracking..... | 81 |
| Visual Inspection of In-Phase and Quadrature Signal..... | 81 |
| Code Phase Error..... | 82 |
| Carrier Frequency Error..... | 83 |
| Doppler Frequency..... | 84 |
| Navigation Solution..... | 85 |
| Chapter Five: Conclusions And Recommendations..... | 88 |
| Conclusions..... | 88 |
| Receiver Performance..... | 89 |
| Software/Hardware Partitioning Efficiency..... | 90 |
| Power Requirements..... | 92 |
| Recommendations for Future Work..... | 93 |
| Sufficient FPGA Resources..... | 93 |

| | |
|------------------------------------|----|
| Improve Positioning Algorithm..... | 94 |
| Multi-Constellation Receiver | 94 |
| Modular Design | 95 |
| References..... | 96 |

List of Tables

| | |
|---|----|
| Table 1: Navigation message frame structure..... | 20 |
| Table 2: Comparison of acquisition search methods..... | 22 |
| Table 3: Hexadecimal representation of data and its equivalent integer representation..... | 40 |
| Table 4: Comparison of target design and recent software receiver design specifications..... | 51 |
| Table 5: Comparison of available FFT IP configurations. | 57 |
| Table 6: 8-bit real and imaginary bit assignment..... | 62 |
| Table 7: Binary and hexadecimal representation of integer data samples..... | 62 |
| Table 8: Position errors..... | 86 |

List of Figures

| | |
|---|----|
| Figure 1: GNSS constellation - orbit comparison..... | 3 |
| Figure 2: Processing rate versus flexibility (Baracchi-Frei, 2010)..... | 6 |
| Figure 3: Types of software receivers..... | 7 |
| Figure 4: Processing power versus flexibility (Dovis et al., 2005)..... | 14 |
| Figure 5: Incoming signal demodulation (Borre et al., 2007)..... | 19 |
| Figure 6: Ideal software GNSS receiver..... | 21 |
| Figure 7: Parallel Code Phase Search Algorithm (Borre et al., 2007)..... | 26 |
| Figure 8: Functional block diagram for tracking (Petovello et al., 2008)..... | 27 |
| Figure 9: Costas loop (Borre et al., 2007)..... | 28 |
| Figure 10: Code tracking correlation with ‘early’, ‘late’ and ‘prompt’ replicas (Borre et al., 2007)..... | 31 |
| Figure 11: Delay Lock Loop for code tracking (Borre et al., 2007)..... | 32 |
| Figure 12: Processor performance vs. flexibility..... | 35 |
| Figure 13: Basic architecture of a System-on-Chip..... | 38 |
| Figure 14: Acquisition algorithm implemented using Parallel Code Phase Search method..... | 41 |
| Figure 15: Tracking with multiple channels (upto ‘n’)..... | 43 |
| Figure 16: State transitions using POSIX threads (Butenhof, 1997)..... | 44 |
| Figure 17: Tracking loop timeline (Gleason and Gebre-Egziabher, 2009)..... | 46 |
| Figure 18: Tracking algorithm implemented with multiple processing threads..... | 47 |
| Figure 19: Simultaneous channel tracking and receiver position computation using POSIX threads..... | 48 |
| Figure 20: Navigation Solution: Processing steps for receiver position computation..... | 49 |
| Figure 21: Pseudorange measurement between satellite and receiver..... | 50 |
| Figure 22: Highlighted region (FFT operations) implemented in the FPGA..... | 55 |
| Figure 23: Altera IP Core Implementation in the FPGA..... | 56 |
| Figure 24: Data and control signal interface for MegaCore FFT IP..... | 59 |
| Figure 25: HPS-FPGA Bridges (Altera SoC Embedded Design Suite User Guide, Altera)..... | 60 |
| Figure 26: Address map of PIO pins..... | 61 |

| | |
|--|----|
| Figure 27: PIO data flow direction between FPGA and HPS..... | 62 |
| Figure 28: SCFIFO control and data interface signals..... | 64 |
| Figure 29: Final hardware design implemented in the FPGA using FFT and FIFO cores. | 64 |
| Figure 30: Final acquisition algorithm using FPGA components for FFT and IFFT operations..... | 65 |
| Figure 31: Cyclone V SoC board and desktop computer interface signals. | 67 |
| Figure 32: Input sample data with four amplitude levels of -3, -1, 1 and 3..... | 68 |
| Figure 33: C/A-code for PRN 1 with two amplitude levels of -1 and 1. | 68 |
| Figure 34: Acquisition: List of detected satellites along with signal parameters like coarse frequency, fine frequency, magnitude and code phase of the incoming signal (console screenshot). | 69 |
| Figure 35: Acquisition plot – signal magnitude versus satellite number..... | 70 |
| Figure 36: Satellite acquisition - Doppler frequency versus satellite number. | 71 |
| Figure 37: Gprof results enlisting the percent of execution time required by software functions..... | 72 |
| Figure 38: FFT of C/A code for PRN 1 using software FFT function. | 74 |
| Figure 39: FFT of C/A code for PRN 1 using Hardware FFT function. | 74 |
| Figure 40: FFT of sampled GPS data using software FFT function..... | 75 |
| Figure 41: FFT of sampled GPS data using hardware FFT function..... | 75 |
| Figure 42: Product of FFT outputs (I component)..... | 76 |
| Figure 43: Acquisition – comparison of results obtained when software FFT, scaled and unscaled hardware FFT functions are used..... | 77 |
| Figure 44: Software/hardware synchronization - SignalTap waveform comparing hardware and software clock cycles | 79 |
| Figure 45: Comparison of I and Q components of the incoming signal..... | 81 |
| Figure 46: I component containing navigation data with bit transitions. | 82 |
| Figure 47: Discriminator output - code phase error varying over time. | 83 |
| Figure 48: Carrier frequency error varying over time. | 83 |
| Figure 49: Doppler frequency for PRN 8 varying gradually over time..... | 85 |

List of Acronyms

| | |
|---------|--|
| AMBA | Advanced Microcontroller Bus Architecture |
| ASIC | Application Specific Integrated Circuit |
| AXI | Advanced eXtensible Interface |
| C/A | Coarse/Acquisition |
| CDMA | Code Division Multiple Access |
| DSP | Digital Signal Processor |
| DTB | Device Tree Blob |
| FFT | Fast Fourier Transform |
| FLL | Frequency Lock Loop |
| FPGA | Field Programmable Gate Array |
| GLONASS | Global'naya Navigatsionnaya Sputnikova Sistema |
| GNSS | Global Navigation Satellite Systems |
| GPS | Global Positioning System |
| HDL | Hardware Description Language |
| HOW | Handover Word |
| HPS | Hard Processor System |
| IFFT | Inverse Fast Fourier Transform |
| IP | Intellectual Property |
| KissFFT | Keep It Simple Stupid FFT |
| LED | Light Emitting Diode |
| LEO | Low Earth Orbit |
| LPF | Low Pass Filter |
| LW | lightweight |
| MEO | Medium Earth Orbit |
| NCO | Numerically Controlled Oscillator |
| OS | Operating System |
| PC | Personal Computer |
| PIO | Parallel Input/Output |
| PLL | Phase Lock Loop |
| POSIX | Portable Operating System Interface |
| PRN | Psuedo-Random Noise |
| RF | Radio Frequency |
| SCFIFO | Single Clock First In, First Out |
| SoC | System-on-Chip |
| TLM | Telemetry |
| TOW | Time of Week |
| UART | Universal Asynchronous Receiver/Transmitter |
| UTC | Coordinated Universal Time |

List of Symbols

| | |
|-----------|--|
| f_{L1} | Operating frequency of GPS signal: 1575.42 MHz |
| f_{L2} | Operating frequency of GPS signal: 1227.60 MHz |
| P_C | Signal power of C/A-code |
| C_k | C/A-code sequence |
| D_k | Navigation message sequence |
| P^k | P(Y)-code sequence |
| P_{PL1} | P(Y)-code signal power at L1 frequency |
| P_{PL2} | P(Y)-code signal power at L2 frequency |
| I | In-phase signal component |
| Q | Quadrature phase signal component |
| E | 'Early' signal component |
| L | 'Late' signal component |
| P | 'Prompt' signal component |
| ρ | Geometrical range |
| dt_i | Receiver clock error |
| dt^k | Satellite clock error |

Chapter One: Introduction

Global Navigation Satellite System (GNSS) receivers are currently used for various applications such as navigation, geodetic science, atmospheric science and other space applications. Even though the vast majority of users are land-based, trends over the past decade suggests that there is tremendous scope for application in the aviation and space industry. Receiver specifications vary greatly depending on usage. As an example, receivers built for space application draw low power and have mass and size constraints that significantly differ from their terrestrial counterparts.

Receivers designed to suit only certain categories of applications are called Application Specific Integrated Circuit (ASIC)-based receivers. ASIC-based receivers use hardware components for signal processing and are hence called hardware receivers. Such receivers do not permit any design or functional changes once they are fabricated. Aforesaid design inflexibility can be a hindrance when new algorithms are to be implemented as modifications would require a re-fabrication of the receiver from scratch, increasing costs and development time. As a solution to this problem, software GNSS receivers were introduced. Dennis Akos (1997) described the first complete Global Positioning System (GPS) software receiver (Borre et al., 2007).

A software receiver typically consists of a processor responsible for all the system functions. Any design changes leading to algorithm modifications can be made easily in

software, without requiring any hardware changes (unlike ASIC-based receivers). This feature is of great benefit for space applications especially as software upgrades can be made when the receiver is in the field. The only drawback of such an implementation is that the computational load on the processor is very high and at times beyond its capability, leading to an undesirable processor slowdown. As a work around to this problem, hardware components known as Field Programmable Gate Arrays (FPGAs) are used alongside processors. These components, although hardware, are software reconfigurable components, hence maintaining the definition of a “software receiver”.

Specific to space applications, hardware receivers are still preferred due to their higher processing power. However, with the tremendous advancement made in the field of software receiver technology over the past decade, FPGA-based software GNSS receivers seem promising and may in time be the preferred option.

Overview of GNSS Constellations

GNSSs are constellations of satellites that allow users on Earth and space to use the timing information transmitted by them to determine user position and time. The constellations that are operational are the GPS developed by the U.S. and the Global'naya Navigatsionnaya Sputnikova Sistema (GLONASS) developed by Russia. China is in the process of expanding its regional navigation system BeiDou to a global system and the European Union is developing Galileo. Amongst all of these constellations, by far the most widely used is GPS.

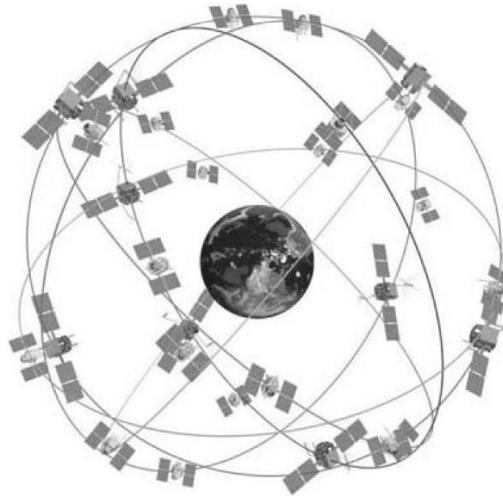


Figure 1: GPS constellation (Kaplan and Hegarty, 2006)

GPS was initiated by the U.S. Department of Defense in 1973 and has more than 24 operational Medium Earth Orbit (MEO) satellites. Figure 1: GPS constellation Figure 1 depicts the GPS constellation. GPS transmits navigation data on three different frequencies L1 (1575.42 MHz), L2 (1227.6 MHz) and L5 (1176.45 MHz). The L1 signal consists of the Coarse/Acquisition (C/A)-code, which is a unique sequence of bits also known as a pseudo random noise (PRN) code (Kaplan and Hegarty, 2006). At the receiver end, satellite signals can be identified by recognizing a particular PRN sequence. Each of these PRN sequences highly correlate only when matched with the same aligned sequence and does not correlate with any other satellite's PRN. Since all the GPS satellite signals operate at the same frequency, data are transmitted using Code Division Multiple Access (CDMA). CDMA allows several signals to be transmitted in the same frequency channel while distinguishing each one of them using unique code modulations. The satellite signal also

consists of the P(Y)-code; however, this code is encrypted and not available for civilian applications. Most importantly, GPS signals carry navigation and other timing information that is used to compute the user position.

Software GNSS Receivers

As mentioned earlier, satellite signals carry vital navigation data. GNSS receivers are used to process these satellite signals and decode the underlying navigation message to compute various parameters such as position, velocity of the receiver and obtain timing information. These receivers can be used for a multiple frequency signals (L1, L2, L5) or even multiple constellations (GPS, GLONASS, etc.).

Until the late 1980s, GNSS receivers completely relied on hardware components. Over the next few decades, with significant advancement in computing power of CPUs and processors, it became possible to implement, if not all, but some of the receiver components in software. This development led to reduction in receiver size and mass while maintaining functionality. Consequently, development costs were also lowered and algorithm testing became more efficient. It became possible to implement design changes even after receivers were placed in the field, significantly reducing development time. The flexibility demonstrated by software receivers is one of the main reasons they have generated much research interest in the academic and industrial communities.

However, using software receivers has its own set of challenges. A pure software receiver must single-handedly take responsibility of the complete signal processing activity.

However, the increased processing load lowers the performance of the receiver. Also, since software receivers are based on generic processors, the power consumption of the chipset may be higher than the hardware receivers. Research to improve software GNSS receiver performance is ongoing and has given rise to many design alternatives as discussed in Chapter 2.

Field Programmable Gate Array

The increased software computation load is a serious problem due to which alternate processing methods must be considered. Traditional hardware receivers (ASIC-based), have excellent performance in terms processing speed and have low power requirements. A major drawback however is that the design cannot be changed once the ASIC is fabricated. Also, since the ASIC is a specialised chip, its development cost is also high (Ma et al., 2004).

More recently, Field Programmable Gate Arrays (FPGAs) have been introduced as a substitute for ASICs. Even though FPGAs have lower computational capabilities than ASICs, an FPGA is a software reconfigurable type of hardware. This feature allows the user/developer to make hardware design changes in software using Hardware Description Language (HDL) without making external hardware changes. Software upgrades resulting in hardware changes is extremely beneficial and developer-friendly as design modifications can then also be implemented post-development of the receiver (Ma et al., 2004). With the introduction of new GNSS signals, such a feature becomes very valuable.

FPGAs consist of large arrays of logic blocks which can be used to compute multiple functions making FPGAs capable of performing several tasks in parallel. This feature is of great value as GNSS signal processing can then be performed in parallel channels in contrast to sequential processing in software (processors) therefore speeding up the overall process.

As shown in Figure 2, FPGAs offer optimum flexibility and processing rate. Recent trends suggest the use of a processor along with FPGA, enhancing the receiver flexibility further (Hein et al., 2006). Several FPGA manufacturers have noted this added benefit and now FPGAs are available with embedded processors, also referred to as a System-on-Chip (SoC). SoCs offer single chip solution with low power requirements, easy integration and better performance. They have software (such as embedded processors) and hardware components (such as FPGAs) that interact to complete assigned tasks (Ben Salem et al., 2008).

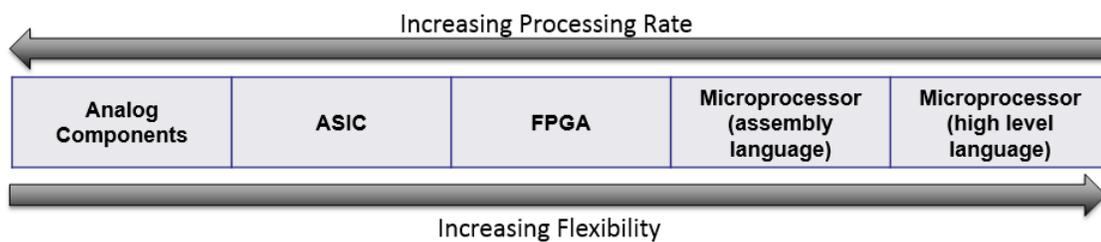


Figure 2: Processing rate versus flexibility (Baracchi-Frei, 2010).

Figure 3 illustrates the most common types of software GNSS receivers used. PC-based post-processing of sample GNSS signals is performed generally for algorithm testing

(Gleason and Gebre-Egziabher, 2009). Real-time software receivers can also be PC-based, however more interest lies in their application as embedded systems. Software receivers can also be based purely on FPGAs. As mentioned earlier, although they illustrate high task parallelism, it is more efficient to use them alongside a processor.

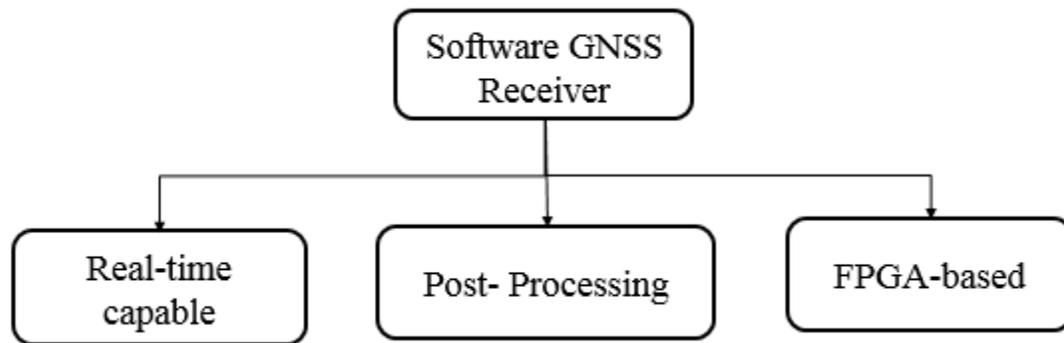


Figure 3: Types of software receivers.

Software GNSS Receiver for Satellite Applications

GNSS receivers hold many promising applications for Low Earth Orbit (LEO) satellites. In addition to providing positioning and tracking information of the LEO satellites, GNSS receivers can also serve as an instrument to determine various geodetic and atmospheric parameters (Montenbruck et al., 2008). Some of the applications are precise orbit determination, ocean altimetry, tide height determination in coastal regions, determination of ionospheric delay over the ocean, radio occultation and reflectometry (Zavorotny and Voronovich, 2000).

However, GNSS receiver operation in space poses as a unique challenge when compared to its equivalent terrestrial use. The hardware and software components must be modified and tested for various additional conditions that are often not considered when building for land-based applications. One of the challenges is the limited resources available on-board the satellite. Power consumption of the receiver must be low, while having no adverse effect on the receiver performance and reliability. They must also fit the size and weight constraints that are typically low for LEO satellite payloads. Additionally, the developed GNSS receiver must withstand mechanical vibrations and stress during the launch of the satellite. Temperature in the space environment may vary widely leading to thermal stress and failures of receiver components. Heat dissipation in vacuum is another critical design challenge. Also, electronic components might malfunction if the receiver is not resistant to radiation effects in space (Parkinson et al., 2011). Such specifications increase the overall receiver development cost as well.

Some other considerations are that the GNSS receiver will observe significantly higher Doppler rate than what would be observed on the Earth's surface, due to the relative motion of satellite in orbit, hence requiring a wider Doppler frequency search range. Another challenge is that a solution must be calculated faster as the orbital speed of LEO satellite is high, resulting in faster rising and setting of available GNSS satellites.

The above mentioned challenges have successfully been overcome with the use of sophisticated hardware GNSS receivers and hence are preferred for spaceflight. Although there are some software GNSS receivers that have successfully flown in space (discussed

in Chapter 2) there are some technical aspects that still need advancement before they can successfully compete with hardware receivers in space.

Problem Statement

As mentioned earlier, FPGA-based software GNSS receivers built for space applications require customization and represents a design challenge as compared to hardware receivers. GNSS signals require intense signal processing before the navigation messages are processed. Such complex computations increase the software load on the embedded processor and often reduces system efficiency. Even with the use of FPGA with an embedded processor, appropriate task partitioning between the hardware and software components remains a technical challenge.

Thesis Objectives

The main objective of this research is to design and develop a low power, low cost FPGA-based software GNSS receiver for satellite applications. A novel approach to the implementation is demonstrated using a SoC processing platform. The results of this implementation are then examined. Also presented is efficient task partitioning between software and hardware components and its effect on overall processing speed of the receiver. Finally, various parameters and aspects to consider before selecting a processor for receiver implementation has been discussed.

Thesis Outline

Chapter 1 introduces the concept of software GNSS receivers and the challenges with using them. The chapter also describes the FPGA and how its use benefits the software receiver processing. Finally, the research problem is discussed followed by the research objectives.

Chapter 2 presents a general description of software GNSS receiver architecture and signal characteristics. Also discussed are the benefits and drawbacks of using such a receiver for space applications followed by a brief summary of the state of current research and solutions that have been proposed by researchers. The chapter also introduces FPGAs, embedded processors and SoC technology.

Chapter 3 presents detailed description of the receiver implementation. The first section provides an overview of the software development aspect of the receiver. The chapter describes the input data format, processing techniques used, available software libraries and the algorithm flow of each of the different processing blocks implemented. The second section focuses on the hardware components pertaining to the receiver. The hardware requirements of the receiver are examined and the features of the selected processor are explained. Hardware/software task partitioning using the selected processor is also discussed followed by the final hardware design implementation.

Chapter 4 presents the tests and results from the implementation described in Chapter 3. The advantages and limitations of the current implementation are also discussed.

Finally, Chapter 5 concludes the research thesis providing an overview of the research performed and also includes recommendations for potential future work.

Chapter Two: Software GNSS Receiver

Using System On Chip

Software receivers imply that all the digital signal processing is performed using a programmable processor. However, in recent times, due to increasing software processing load, FPGA (hardware) elements are used. Such receivers are still defined as software receivers as they are still reconfigurable, maintaining design flexibility.

Brief History of Software GNSS Receivers

After Akos (1997) first applied the concept of software radio to GNSS receivers successfully, it generated tremendous research interest. He also implemented a PC-based real-time software receiver soon after (Akos et al., 2001). Ledvina et al. (2003) improved the algorithms to include more tracking channels and also significantly accelerated the processing speed.

These developments led to a very active research interest in implementing the receiver as an embedded system. Humphreys et al. (2006) implemented the software receiver using only a Digital Signal Processor (DSP) that manipulates digital data and performs more specific tasks than what a microprocessor or a CPU typically would (Hein et al., 2006). In the DSP implementation, Fast Fourier Transform (FFT) techniques were used to speed up the processing, and based on the results, it was concluded that a high performance DSP can be used to compete against ASIC technology.

In an effort to replace ASIC-based receivers with an FPGA equivalent, Ganguly (2004) implemented correlators using an FPGA, but major software processing still required a PC. Enge et al. (2004) then introduced FPGA-based GPS receivers and successfully verified implementation of different emerging algorithms on a FPGA-based receiver.

Another emerging trend is to use a combination of processors to realize an overall optimum system. As shown in the comparison in Figure 4, different processing platforms are compared in terms of their processing power and flexibility. Processing power refers to the computational capabilities of the processor whereas flexibility refers to the extent of design modification the processor permits. According to Dovic et al. (2005) and Hein et al. (2006), use of a hybrid processing platform is strongly recommended as hybrid FPGA/DSP platforms reduce power consumption by 50 percent and increase the performance by a factor of 10 (Hein et al., 2006). Such a platform can be especially beneficial in the case of GNSS signal processing, where large data rates are required.

Design of a software GNSS receiver for space applications can be a challenge in terms of processing capabilities, power consumption, size, mass and space readiness. Since the processing chip requires a high level of specialisation, one approach is to design a system very specific to the application.

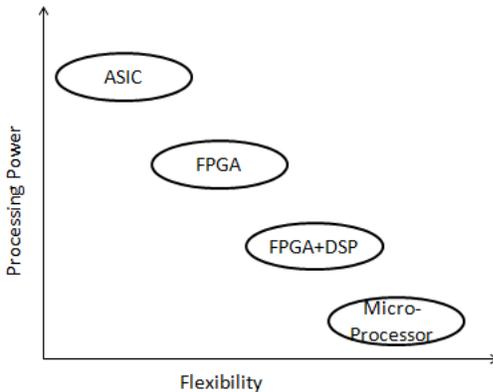


Figure 4: Processing power versus flexibility (Dovis et al., 2005).

BlackJack, a spaceborne GPS receiver developed by NASA’s Jet Propulsion Laboratory (JPL) is one of the first few receivers to have successfully flown on numerous satellites such as Oersted, Champ, SAC-C, ICESat and GRACE. The newer version of BlackJack, Integrated GPS Receiver and Occultation Receiver (IGOR) has been built with improved space hardness (Montenbruck et al., 2006). But such an investment would mean high development costs (in millions) and exceeds most small mission satellite budgets. Therefore, there has been a search for affordable alternatives and a growing interest to use existing commercial-off-the-shelf (COTS) GNSS receivers for space applications (Montenbruck et al., 2006).

Montenbruck (2008) lists commercial GPS receivers developed for space application and provides an insightful comparison. Among the recommendations listed for future receivers, the author explains the need for miniaturization of the receivers. Satellite missions with

limited budget can greatly benefit from reduction in size, mass and power consumption. Considering these factors, a software GNSS receiver can be used as an easy replacement.

In more recent developments, several software GNSS receivers are commercially available. However, their target market is primarily for land-based users and educational purposes (GNSS SDR Front End and Receiver, Nottingham Scientific Ltd.). On the other hand, there is also extensive research and development for spaceborne software receivers. Namuru V3, developed by the University of New South Wales (Parkinson et al., 2011), is a software GNSS receiver that is specifically designed for use on LEO satellites. Namuru V3 uses an FPGA manufactured by Actel with an embedded ARM processor. Additionally, it also uses a second FPGA board (external), increasing the size of the overall receiver. Even though the Namuru hasn't been tested on a spaceflight yet, it has been lab-tested for various space scenarios (Choudhury et al., 2013). It is also part of a project named *Biarri* that forms a part of an international space mission (Glennon et al., 2013; *Biarri* GPS Receiver Project, 2013).

Considering all the design limitations discussed above, an FPGA-based software GNSS receiver can be used as a reasonable replacement. Overall hardware components are reduced when compared to an ASIC, resulting in smaller sized receivers. Since a general chip can be used for the receiver, the development cost is extremely low as compared to the hardware GNSS receivers. This cost-effective solution, however, does not compromise the receiver performance.

GPS Signal Characteristics

The focus of the research is to design and develop a software GNSS receiver specifically for GPS L1 Coarse/Acquisition (C/A-code) signal. It is necessary to understand the signal structure before a receiver is designed. The L1 signal is comprised of three components, carrier signal, navigation data and spreading sequence C/A-code modulation. The carrier signal is simply the carrier wave transmitted at the respective frequency, in this case L1 (1575.42 MHz). Navigation data carries information about satellite orbits and timing information. These data are decoded and read by the receivers. GPS signals consist of two spreading sequences: the C/A-code and encrypted precision code-P(Y). Since the receiver implementation is developed for C/A-code, only this sequence will be discussed in detail.

C/A-Code

The C/A-code is a unique sequence of bits also referred to as pseudo random noise (PRN) codes. They have noise-like properties but at the same time have deterministic sequences. Each satellite has a unique PRN code that is 1023 bits long and is repeated every 1 ms.

Equation 1 represents the signal transmitted by satellite k . Terms P_c , P_{PL1} and P_{PL2} represents the power of the C/A-code or P-code at frequency L1 and L2. As observed in the equation, C/A-code is modulated onto the L1 signal only whereas P-code is modulated onto both L1 and L2 signals. C^k and P^k represent the C/A-code and P-code sequence, respectively. The navigation message is represented by D^k , whereas f_{L1} , f_{L2} represents the carrier frequency L1 and L2, respectively (Borre et al., 2007).

$$\begin{aligned}
s^k(t) = & \sqrt{2P_c} \left(C^k(t) \oplus D^k(t) \right) \cos(2\pi f_{L1} t) \\
& + \sqrt{2P_{PL1}} \left(P^k(t) \oplus D^k(t) \right) \sin(2\pi f_{L1} t) \\
& + \sqrt{2P_{PL2}} \left(P^k(t) \oplus D^k(t) \right) \sin(2\pi f_{L2} t)
\end{aligned}$$

Equation 1

Following are the unique correlation properties that C/A-codes exhibit (Borre et al., 2007).

1. Cross-correlation properties: Cross-correlation is the measure of similarity of two C/A-code sequences, with lag applied to one of them. It is represented by the following equation:

$$r_{ik}(m) = \sum_{l=0}^{1022} C^i(l) C^k(l+m) \approx 0 \text{ for all } m$$

Equation 2

C^i and C^k represents the C/A-codes of satellite i and k , respectively. C/A-codes of two different satellites do not correlate with each other ($r_{ik} = 0$). Such a property ensures the receiver identifies the PRN code accurately during acquisition.

2. Auto-correlation properties: Auto-correlation is a measure of the similarity of the C/A-code sequence with itself but with a lag. It is represented by the equation given below:

$$r_{kk}(m) = \sum_{l=0}^{1022} C^k(l)C^k(l+m) \approx 0 \text{ for } |m| \geq 1$$

Equation 3

C^k represents the C/A-code of satellite k . According to the equation, C/A-code is almost uncorrelated with itself except at zero lag. They have high correlation only when they are exactly aligned at zero delay.

Referring to Equation 1, only the component containing the C/A-code, C^k is used for signal processing in this research. L2 frequency components are filtered out by the front end and the L1 frequency is down-converted to an intermediate frequency (ω_{IF}). Equation 4 represents the filtered signal.

$$s^k(t) = \sqrt{2P_c} C^k(t) D^k(t) \cos(\omega_{IF}t) + \sqrt{2P_{PL1}} P^k(t) D^k(t) \sin(\omega_{IF}t)$$

Equation 4

The P-code is distorted after the narrow band pass filtering around the C/A-code and hence cannot be demodulated. For this reason, the component containing the P-code can be considered as noise and filtered out. In order to read the navigation data D^k , as represented in Equation 4 **Error! Reference source not found.**, the signal s^k is multiplied with replicas of the carrier signal $\cos \omega_{IF}t$ and C/A-code C^k so that the only component remaining is the navigation message, D^k .

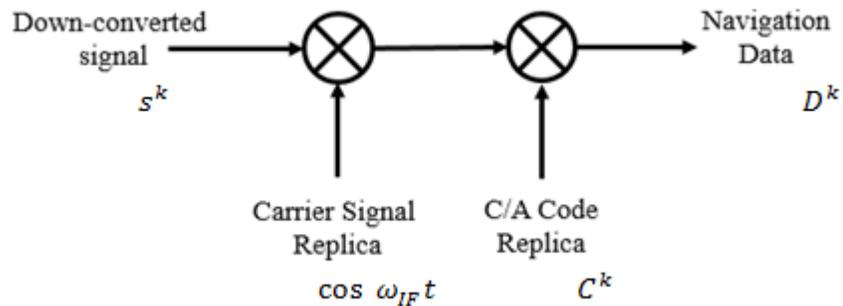


Figure 5: Incoming signal demodulation (Borre et al., 2007).

Navigation Data

Navigation data transmitted by the satellites are received in the form of structured data frames. These frames are further divided into sub-frames. Each frame is repeated every 30 seconds and an entire navigation message lasts 12.5 minutes, after which it is repeated again.

Table 1 lists the contents of one data frame. Every subframe contains 10 words, each word having a length of 30 bits. Telemetry (TLM) and Handover Word (HOW) form part of every subframe. TLM contains an 8-bit preamble that is used by the receiver for frame synchronization, whereas HOW contains a truncated version of the time of the week (TOW). It also contains the subframe ID which is used to identify the subframe that HOW belongs to. The following is a brief description of each subframe and its contents.

1. Subframe 1, *Satellite Clock and Health Data*: This subframe contains all clock information that is required to compute the time the navigation message was transmitted from the satellite.

2. Subframe 2 and 3, *Satellite Ephemeris Data*: These two subframes contain all information related to satellite orbits that is used to compute satellite position.
3. Subframes 4 and 5, *Support Data*: These subframes contain almanac data, i.e., less precise version of ephemeris and clock data. Additionally, they also contain health indicators, ionospheric and UTC parameters.

Table 1: Navigation message frame structure (Kaplan and Hegarty, 2006).

| | | | | | |
|---------------|----|------------|-----|-----|----------------------------------|
| Time (s) ↓ | 0 | Subframe 1 | TLM | HOW | Clock Corrections and SV Health |
| | | Subframe 2 | TLM | HOW | Ephemeris Parameters |
| | | Subframe 3 | TLM | HOW | Ephemeris Parameters |
| | | Subframe 4 | TLM | HOW | Almanac, Ionospheric Model, dUTC |
| | 30 | Subframe 5 | TLM | HOW | Almanac |

Software GNSS Receiver Architecture

Unlike traditional hardware-based receivers, except for the antenna and the radio frequency (RF) front end, all other components are software-based. Figure 6 represents the ideal software GNSS receiver.

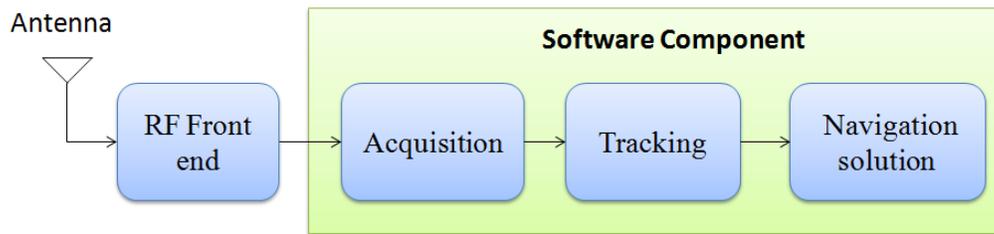


Figure 6: Ideal software GNSS receiver (Gleason and Gebre-Egziabher, 2009).

The RF front end is responsible for the analog to digital signal conversion of the incoming GPS signal. The first processing block of acquisition determines all the satellites visible to the receiver followed by the tracking block that keeps track of these detected satellite signals. The tracking block also demodulates the GPS signal to read the navigation data. The final processing block reads this navigation data and computes the navigation solution. Each processing block is discussed in detail below.

Acquisition

Acquisition is the process of detecting GPS satellites visible to the receiver. It also determines the carrier frequency and phase of each satellite signal. The GPS signal has 3 components, the carrier signal, PRN code and the navigation data. In order to extract the navigation data, the carrier signal and PRN code must be completely cancelled out from the incoming signal. The navigation message can be retained by generating local replicas of carrier signal and PRN code within the receiver and multiplying this with the GPS signal. Due to the velocity of the satellite with respect to the receiver, there is a Doppler shift in the carrier frequency causing it to be higher or lower than the actual nominal carrier frequency. This maximum shift in worst conditions can be within the range of ± 10 KHz

(Akos, 1997). Additionally, the start of the received PRN code sequence can be any of the 1023 bits that constitutes the PRN code sequence. Hence there can be 1023 possible code phases of the PRN sequence that needs to be searched for as well.

Therefore, to correctly cancel out the carrier signal and PRN code from the incoming signal, a range of ± 10 KHz (in steps of 500 Hz) must be searched as well as 1023 possible code phases of the PRN sequence must be checked. Equation 5 (Borre et. al, 2007) estimates the number of search combinations required to detect the correct incoming carrier signal frequency and PRN code phase. An optimal step size of 500 Hz has been chosen as a step size any smaller would lead to a larger number of combinations and a step size much greater than 500 Hz might lead to unsuccessful acquisition.

$$1023 \left(2 \frac{10,000}{500} + 1 \right) = 1023 \times 41 = 41,943 \text{ combinations}$$

Equation 5

As noticed, the number of combinations is very high. There are three different standard approaches usually used to solve this problem which are summarised in Table 2 (Borre et. al, 2007) below.

Table 2: Comparison of acquisition search methods.

| Algorithm | Relative Execution Time | Repetitions | Complexity |
|---------------------------------|-------------------------|-------------|------------|
| Serial Search | 87 | 41,943 | Low |
| Parallel Frequency Space Search | 10 | 1023 | Medium |
| Parallel Code Phase Search | 1 | 41 | High |

In the serial search algorithm, every possible carrier frequency and code phase combination is correlated with the GPS signal. Although it is a simple technique, the number of resulting combinations is very high and hence is an exhaustive search method. Another method is to implement the Parallel Frequency Space Search algorithm. In this method, the frequency search is performed in parallel, therefore speeding up the acquisition process. Parallel frequency search can be implemented using the Fast Fourier Transform (FFT), where time domain signals are converted to their equivalent frequency domain components. This method is more efficient than the first algorithm but has higher complexity.

The final listed method, Parallel Code Phase Search is the most efficient. The number of computations is reduced from 1023 to 41 and is deemed as the most efficient algorithm among them all (Borre et al., 2007). Since this algorithm is used in the receiver implementation in this thesis, only this one will be discussed in detail.

Parallel Code Phase Search Algorithm

The number of search steps is drastically reduced (from 41,943 to 41) when this method is used. Parallel Code Phase Search algorithm uses circular correlation of the PRN code as it is more convenient than multiplying with 1023 code phases of the PRN code. If $x(n)$ and $y(n)$ represent sequences of length N , then the discrete transforms can be described in the following equation (Borre et al., 2007):

$$X(k) = \sum_{n=0}^{N-1} x(n)e^{-j2\pi kn/N} \quad \text{and} \quad Y(k) = \sum_{n=0}^{N-1} y(n)e^{-j2\pi kn/N}$$

Equation 6

The cross correlation of the two sequences $x(n)$ and $y(n)$ with periodic repetition can be represented with $z(n)$ in Equation 7 (Tsui, 2000):

$$z(n) = \frac{1}{N} \sum_{m=0}^{N-1} x(m)y(m+n) = \frac{1}{N} \sum_{m=0}^{N-1} x(-m)y(m-n)$$

Equation 7

If the scaling factor $\frac{1}{N}$ is omitted, the N-point Fourier transform of $z(n)$ can be represented as

$$Z(k) = \sum_{n=0}^{N-1} \sum_{m=0}^{N-1} x(-m)y(m-n)e^{\frac{j2\pi kn}{N}}$$

Equation 8

$$= \sum_{m=0}^{N-1} x(m) e^{j2\pi km/N} \sum_{n=0}^{N-1} y(m+n) e^{\frac{j2\pi k(m+n)}{N}} = X^*(k)Y(k)$$

Equation 9

In Equation 9, $X^*(k)$ represents the complex conjugate of $X(k)$. From the above derivation, it can be concluded that circular correlation of two sequences in the time-domain is equivalent to multiplication of the two sequences in frequency domain, one sequence being a complex conjugate.

Applying this concept for acquisition, referring to the block diagram in Figure 7, the incoming signal is multiplied with a locally generated carrier signal. It is also multiplied with a 90 degree phase shifted version of the same carrier signal. This results in two signals, In-phase (I) signal and Quadrature phase (Q) signal that forms a part of a complex signal and can be expressed as in Equation 10:

$$x(n)=I(n)+jQ(n)$$

Equation 10

The real and imaginary components are then transformed to the frequency domain using the Fourier transform. The PRN code also undergoes Fourier Transformation and is complex conjugated before being multiplied with the Fourier transform of the input signal. This result is then transformed back to time domain using the Inverse Fourier transform. The absolute value of the output is then a measure of correlation of the incoming signal and locally generated PRN code. If there is a peak in the final output signal, it indicates high correlation and the index of this peak marks the code phase of the incoming PRN code (Kaplan and Hegarty, 2006).

This method significantly reduces the search steps when searching for the PRN code phases. However, the 41 frequency combinations must still be serially searched. Therefore for every frequency step, a Fourier transform and Inverse Fourier transform must be performed.

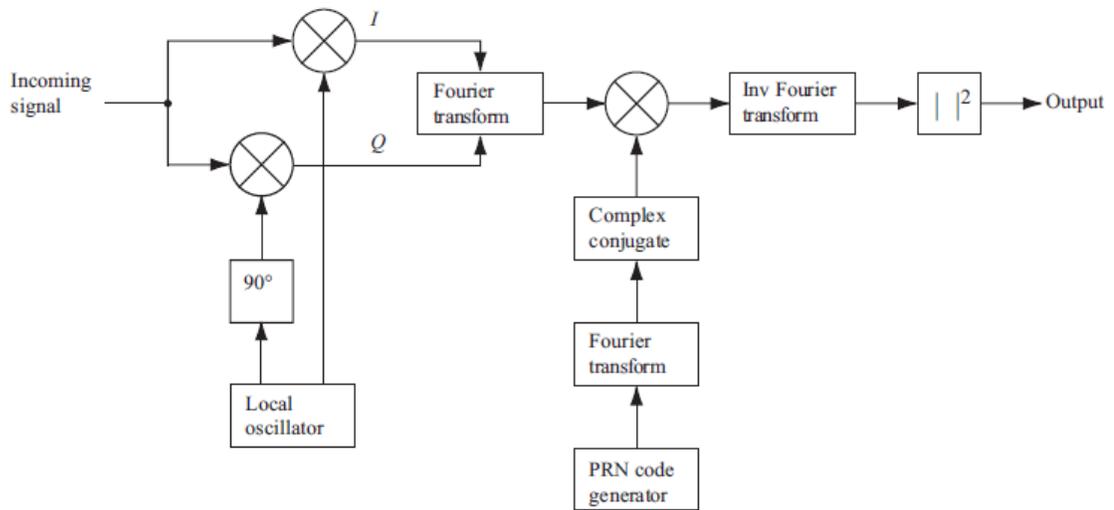


Figure 7: Parallel Code Phase Search Algorithm (Borre et al., 2007).

Tracking

On completion of acquisition, the satellites visible to the receiver are known and so are their coarse frequency and code phase signal values. This information is then transferred to the tracking block where each individual satellite signal is continuously tracked until it is no longer visible. The main purpose of this processing block is to demodulate the navigation data, which can be performed only if the signal frequency and code phase are accurately tracked. Carrier signal replica and PRN code replicas are generated so that these components can be removed from the incoming signal frequency, leaving behind only the navigation data.

Figure 8 (Petovello et al., 2008) demonstrates the functioning of the tracking process for one satellite signal. Each satellite signal is assigned a channel and consequently a receiver

can have multiple tracking channels performing the same process. Since the first step is to remove the carrier signal and the PRN code, the incoming signal is multiplied with the locally generated replicas. If there is a difference between the replica and the incoming signal, it is considered as an error and the replicas are then immediately corrected to again accurately match the incoming signal. This repetitive process ensures correct extraction of navigation message.

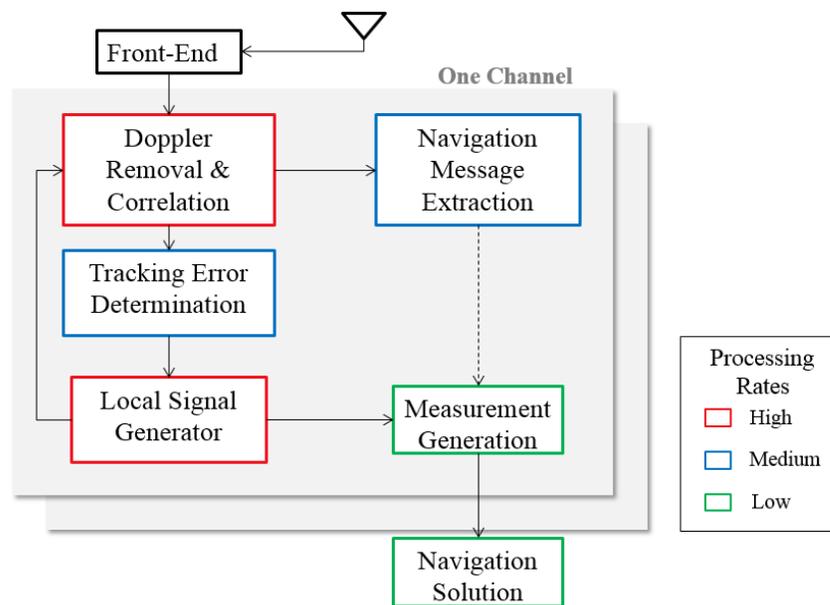


Figure 8: Functional block diagram for tracking (Petovello et al., 2008).

When the navigation messages are correctly read, the subframes are identified and satellite positions are computed. Since the timing information is also obtained, an estimate of the distance between the corresponding satellite and receiver can be made which is also known as the pseudorange. These measurements are then transferred to the final processing block

of navigation solution, where the receiver position is estimated. In the following subsections, tracking of code phase and carrier signal is described in detail.

Carrier Tracking

To generate the exact carrier replicas a Phase Lock Loop (PLL) or Frequency Lock Loop (FLL) is used. These loops track the difference between the incoming signal and the locally generated replica. In Figure 9, a type of FLL is described. It is also called as Costas loop. This particular tracking loop has been used in the developed receiver as it is insensitive to the 180° phase shift in the signal that is typically caused due to navigation bit transitions.

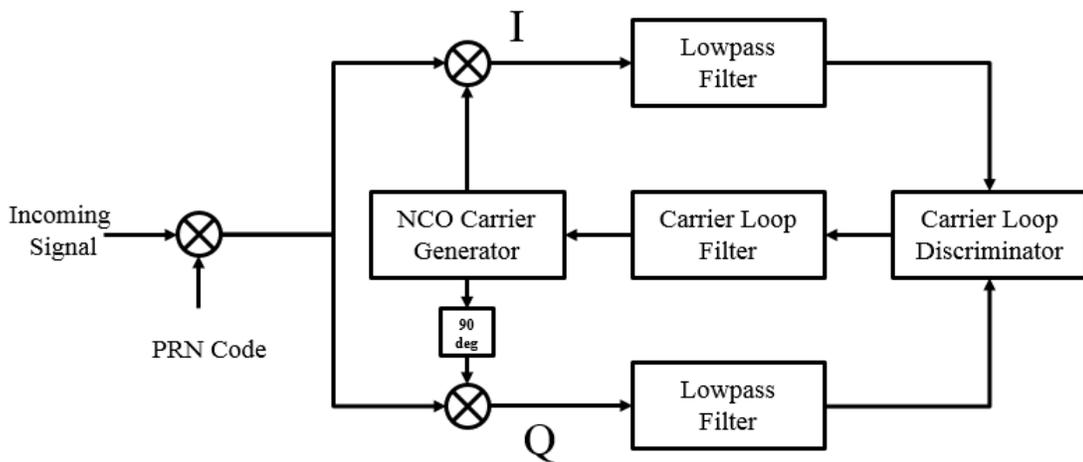


Figure 9: Costas loop (Borre et al., 2007).

In the Costas Loop block diagram, after the first multiplication, assuming the PRN code replica is accurate, the PRN code from the input signal is removed (referring to Equation 4). The resulting signal can be represented in the following form:

$$s(k)=D^k\cos(\omega_{IF}n)$$

Equation 11

This signal is then again multiplied with a carrier signal. It is also multiplied by a 90° phase shifted version of the same carrier signal. This creates the two signal components I and Q expressed in the Equation 12 and Equation 13, respectively. The variable φ represents the phase difference between the incoming signal and the locally generated signal. The goal of the Costas loop is to keep φ to a minimum (ideally zero).

$$D^k(n) \cos(\omega_{IF}n) \cos(\omega_{IF}n+\varphi) = \frac{1}{2}D^k(n) \cos(\varphi) + \frac{1}{2}D^k(n) \cos(2\omega_{IF}n+\varphi)$$

Equation 12

$$D^k(n) \cos(\omega_{IF}n) \sin(\omega_{IF}n+\varphi) = \frac{1}{2}D^k(n) \sin(\varphi) + \frac{1}{2}D^k(n) \sin(2\omega_{IF}n+\varphi)$$

Equation 13

These two signals, one in the in-phase (I) arm and the other in the quadrature (Q) arm, are then passed through a Low Pass Filter (LPF) so that the unwanted signal components are removed. The resulting signal is now as in Equation 14.

$$I^k = \frac{1}{2}D^k(n) \cos(\varphi) \text{ and } Q^k = \frac{1}{2}D^k(n) \sin(\varphi)$$

Equation 14

As seen in Figure 9 above, a feedback system consisting of a carrier loop discriminator and filter is used. The term used as feedback is described in Equation 16 (Kaplan and Hegarty, 2006):

$$\frac{Q^k}{I^k} = \frac{\frac{1}{2}D^k(n) \sin(\varphi)}{\frac{1}{2}D^k(n) \cos(\varphi)} = \tan(\varphi)$$

Equation 15

$$\varphi = \tan^{-1}\left(\frac{Q^k}{I^k}\right)$$

Equation 16

To keep φ to a minimum, the Quadrature component must be minimum and the In-phase component must be maximum. The final block is the Numerically Controlled Oscillator (NCO) that adjusts the carrier replica frequency according to the feedback factor it receives. Accuracy of the carrier tracking loop can be examined by assessing the phase difference (φ) value.

Code Tracking

The code tracking loop is responsible for accurately tracking the code phase of the signal. The loop used in a GPS receiver is the Delay Lock Loop (DLL). In this loop, the incoming signal is correlated with three different locally generated code replicas. The replicas are

named: ‘early’, ‘prompt’ and ‘late’ and have a spacing of $\pm \frac{1}{2}$ chip where a chip corresponds to a bit (Borre et al., 2007). These signals are represented in Figure 10.

As seen in Figure 10, the code phase of the incoming signal and the prompt replica are exactly aligned resulting in a high correlation value which indicates the code phase is being accurately tracked. There may be a possibility that the correlation is high for the late replica instead. A high correlation value for any signal other than the prompt replica would mean that there is a phase error and the code phase replica must be corrected.

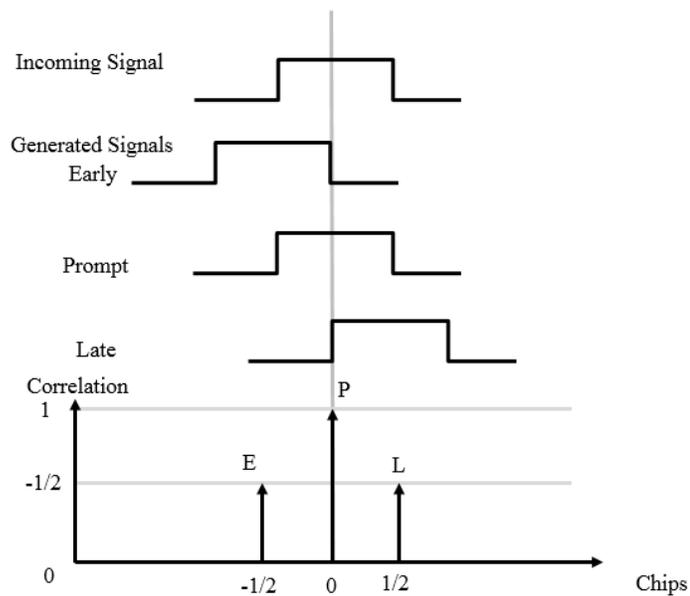


Figure 10: Code tracking correlation with ‘early’, ‘late’ and ‘prompt’ replicas (Kaplan and Hegarty, 2006).

Figure 11 demonstrates the DLL design that has been implemented in the GPS receiver. The incoming signal is first multiplied with a carrier signal replica. I and Q components are then multiplied with early, late and prompt versions of the PRN code replicas. If the locally generated signals are accurate replicas of the input signal, all the energy will be in the in-phase arm of the signal. If the code tracking is not accurate then the energy spreads across both the in-phase and the quadrature arm.

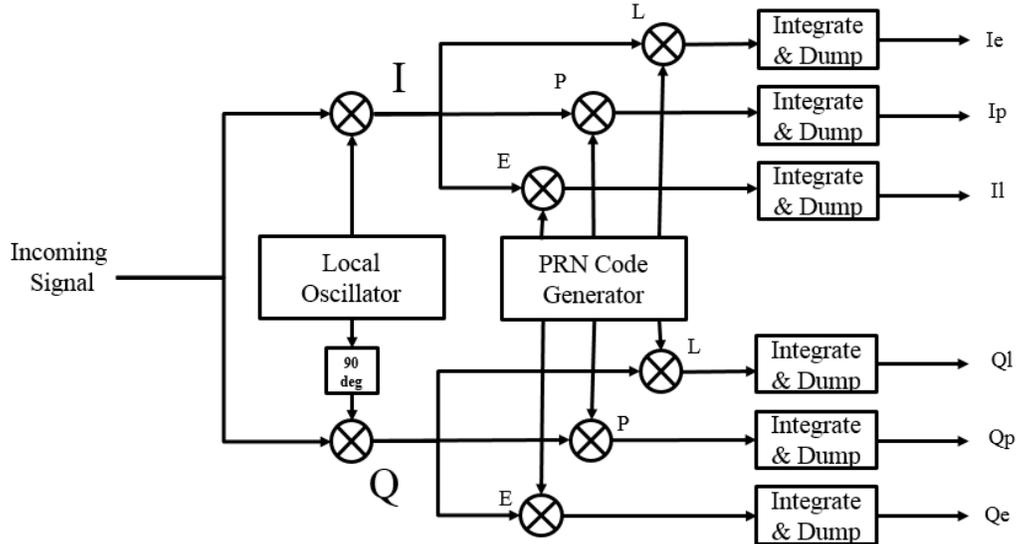


Figure 11: Delay Lock Loop for code tracking (Borre et al., 2007).

Pseudorange Measurements and Navigation Solution

During tracking of the satellite signal, the navigation data are demodulated simultaneously. The decoded subframe parameters hold important satellite orbit information. Using this information, also known as ephemeris data, satellite position can be estimated (GPS Navstar, 1995). Since the navigation message also holds timing information, the time at

which the message was transmitted from the satellite is known. The receiver then estimates accurately when the start of the data frame arrives at the receiver end. The time difference between the satellite signal transmission and arrival of the signal at the receiver end is used to compute pseudoranges.

Receiver Position Estimates

The pseudorange is an estimate of the distance between the satellite and the receiver. With known pseudoranges and satellite position, receiver position can then be estimated.

Mathematically, it is defined in Equation 17:

$$P_i^k = \tau_i^k c$$

Equation 17

Where P_i^k denotes the pseudorange, τ_i^k represents the travelling time between the satellite k and the receiver i and c is the speed of light in vacuum. This equation is however not used practically as there are timing errors to consider as well. For the receiver implementation in this thesis, least squares algorithm is used. Equation 18 defines the observation equation used in the algorithm:

$$P_i^k = \rho_i^k + c(dt_i - dt^k) + T_i^k + I_i^k + e_i^k$$

Equation 18

Where, ρ_i^k is the geometrical range between satellite k and receiver i , c denotes the speed of light, dt_i and dt^k represent receiver and satellite clock offset, respectively. T_i^k is the

tropospheric delay, I_i^k denotes the ionospheric delay and e_i^k is the observational pseudorange error. Troposphere and ionosphere causes signal delay due to various atmospheric conditions and these biases must be accounted for.

ρ_i^k is further represented as:

$$\rho_i^k = \sqrt{(X^k - X_i)^2 + (Y^k - Y_i)^2 + (Z^k - Z_i)^2}$$

Equation 19

Where, satellite position is (X^k, Y^k, Z^k) and the receiver position is denoted as (X_i, Y_i, Z_i) . Equation 18 and Equation 19 are then used find the three unknowns X_i, Y_i, Z_i . Another unknown parameter is the receiver clock bias dt_i . To solve for four unknown variables, a minimum of 4 pseudorange observation equations are necessary. Hence, to obtain a receiver position estimate, at least 4 satellites must be available.

System-on-Chip Technology

GNSS signal processing is computationally intensive and can be a tedious and slow task for the processors. Therefore it is crucial that a processor is selected that will suit the application and final receiver requirements. The processor selection can be based on various parameters such as computational capabilities, design flexibility, power consumption etc. Computational capabilities of the processor is an important selection criterion as GNSS receivers demands heavy signal processing. Also, since the objective of

this research is to develop a software GNSS receiver, the second processor selection is based on the flexibility of the processor.

Figure 12 demonstrates the comparison of different processing platforms that are available. In the comparison, performance of a processor is a measure of the processing speed and computational capabilities whereas flexibility is a measure of design reprogrammability/reconfigurability the processor permits.

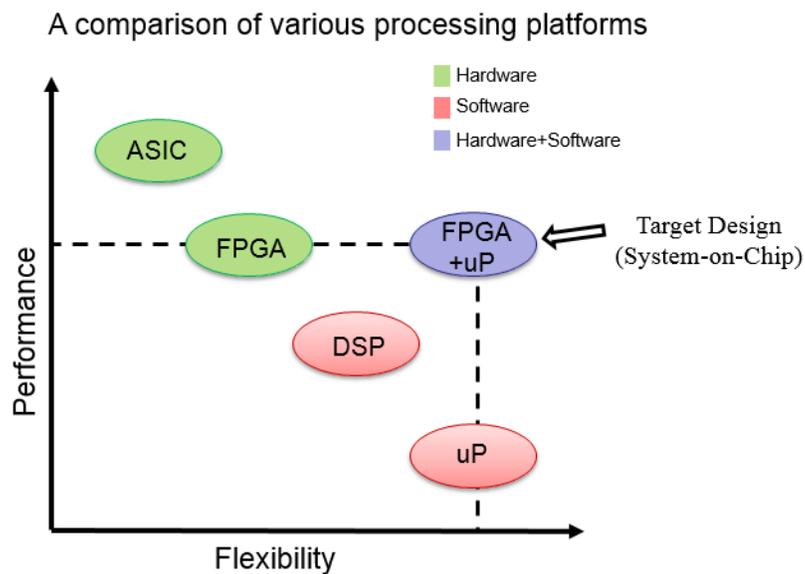


Figure 12: Processor performance vs. flexibility.

As described earlier, an ASIC is a processor chip designed specifically for an application and hence has the most optimum performance. It is also designed such that the power consumption is a minimum. However, once an ASIC is fabricated, hardware design

changes and modifications cannot be made. Hence, an ASIC exhibits minimum design flexibility.

On the other hand, an FPGA has higher flexibility when compared to an ASIC. An FPGA also consists of hardware components but unlike an ASIC, they are reconfigurable. Using the reconfigurability feature, design upgrades to a FPGA-based system can be made even after post-production. This attribute is especially useful for satellite applications. When compared to an ASIC, an FPGA consumes more power and has lower processing speed.

Signal processing can be implemented using components like Digital Signal Processors (DSP) and microprocessors (μ Ps) as well. Since processing is performed using programming language like C/C++, these processors are categorised as software components. A system design change simply requires change in software algorithm without any hardware modifications. For these reasons, DSP and microprocessor offer higher design flexibility when compared an FPGA or ASIC. However, when a comparison is made between DSP and microprocessor with respect to their computational capabilities and processing speed, DSP performs better (Hein et al., 2006). Graphics Processing Unit (GPU) is another platform that has recently become very popular for applications other than video and image processing. Even though GPUs have extremely high computational speed they are not considered for this research as they are not suitable for satellite applications, primarily because they have very high power requirements (Ramesh et. al, 2014). Since every processing platform has a certain trade-off, developers often use a combination of processors resulting in a very robust and highly optimum system.

One very popular combination is to use FPGA and microprocessor together (Hein et al., 2006). FPGA manufacturers realize the benefit and now provide embedded softcore processors (like Nios II, Microblaze). Such embedded processors can be used to control and monitor the heavy signal processing that the FPGA performs. However, developers may want to use more powerful and efficient embedded processors. For such possibilities, FPGA is also available with embedded hardcore processors (like ARM, PowerPC) that are 3-4 times faster than softcore processors (Weber and Chin, 2006). This aspect can be especially beneficial for GNSS signal processing.

SoC is the integration of both software and hardware processing elements on the same chip. In the case of this software GNSS receiver implementation, the SoC processing platform preferred has FPGA and embedded hardcore processor fabricated on the same silicon chip. Using SoC provides huge benefits as they have lower power requirements, are more efficient, occupy lesser board area and facilitates easier integration (Ben Salem et al., 2008). Figure 13 demonstrates the basic architecture of a type of SoC. It consists of FPGA and a hardcore processor, linked together with the interconnect that forms a strong communication bridge between these two entities.

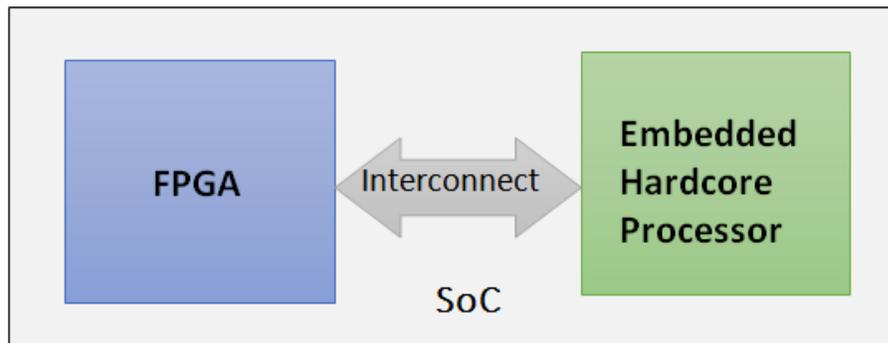


Figure 13: Basic architecture of a System-on-Chip.

Chapter Three: Receiver Implementation

This chapter provides an overview of the techniques and algorithms used to implement the software GNSS receiver. The chapter has been divided into two main sections with the first section focusing on the software components including the algorithmic flow of acquisition, tracking and navigation solution. The second section describes the processor used and also the hardware logic implemented. It also details the software/hardware partitioning of the system components.

Software Implementation

As discussed in the previous chapter, software processing includes the following processing blocks: acquisition, tracking and navigation solution. The complete software has been developed in C/C++. This programming language has been used as it makes the software portable to various processing platforms.

Input Data Format

The complete software receiver development has been divided into various functions that process different datasets. Data files provided in Gleason (2009) have been used for testing and verification purposes. According to Gleason (2009), the 25-second long dataset was collected and processed by SiGe portable L1 data sampler (SiGe GN3S Sampler v3, Sparkfun Electronics). The carrier signal frequency (IF) is 4.1304 MHz whereas the sampling frequency is 16.3676 MHz. The data samples in the file has a 2-bit resolution and are hence represented with four different values as shown in Table 3. The dataset contains

samples in the hexadecimal format and needs to be stored as signed integer prior to their use in different functions.

Table 3: Hexadecimal representation of data and its equivalent integer representation.

| Hexadecimal Representation | Signed Integer |
|----------------------------|----------------|
| 01 | 1 |
| 03 | 3 |
| FF | -1 |
| FD | -3 |

Acquisition Algorithm

The main purpose of the acquisition step is to detect the visible satellites and estimate the coarse values of signal frequency and phase. As discussed in the previous chapter, there are various acquisition methods available. The Parallel Phase Search Method is implemented in this thesis as it is the most efficient (Borre et al., 2007).

Referring to Figure 14, a C/A-code Look-up Table (LUT) is used which is a sequence of the 1023 C/A-code bits for every PRN. The LUT is pre-compiled (as a header file) to save computational time. All the C/A-code sequences for each satellite are transformed to the frequency domain equivalent using the FFT technique and saved for later reference.

The first 2 ms of the dataset (32,735 samples) is initially read, decoded (converted from hex to int) and stored. As shown in Figure 14, each sample from the dataset is multiplied with a carrier signal replica resulting in In-phase (I) and Quadrature (Q) phase components. To convert these into frequency domain components, again the FFT is used. The product

of the output signals from identical blocks FFT 1 and FFT 2 are computed and transformed back to time domain using Inverse Fast Fourier Transform (IFFT).

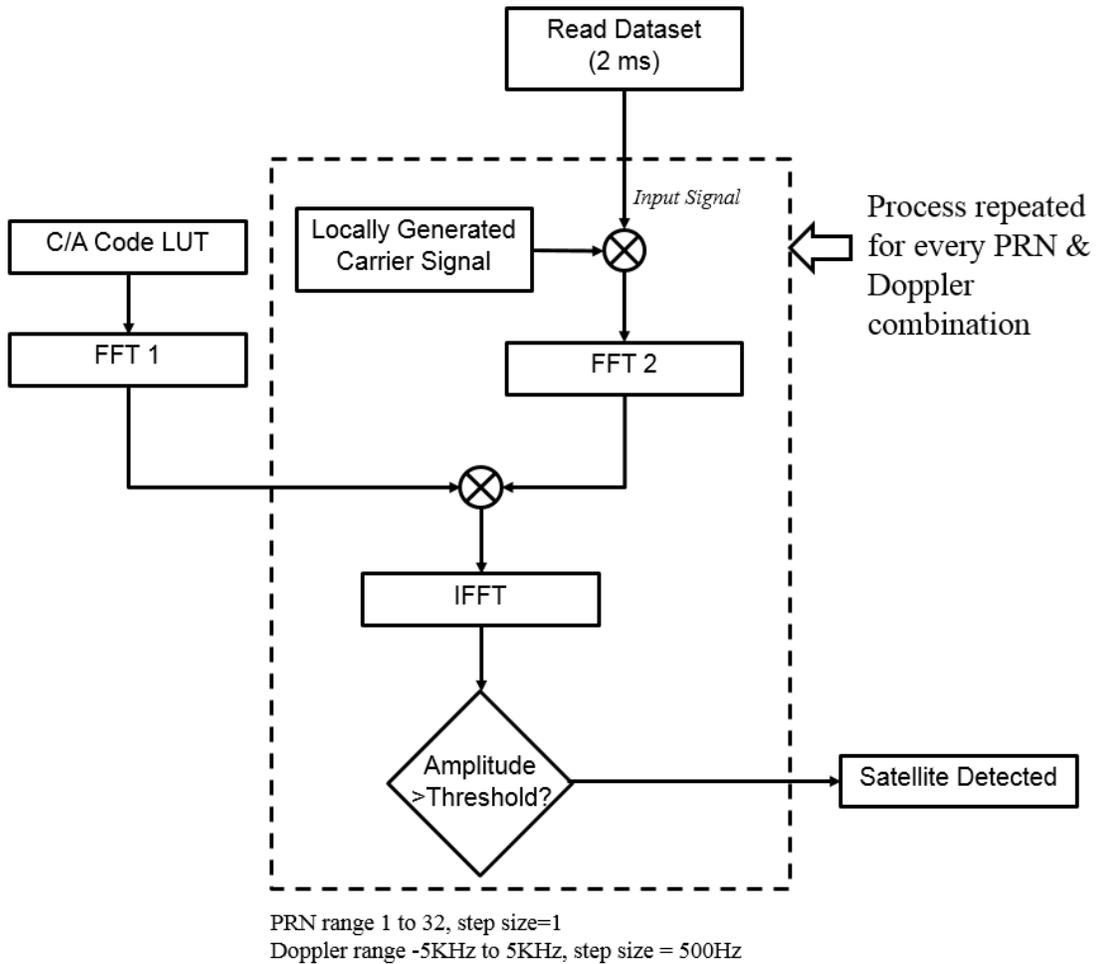


Figure 14: Acquisition algorithm implemented using Parallel Code Phase Search method.

Since the output of the IFFT block is a complex signal and has I and Q components, the absolute value of the signal is considered. This amplitude is then compared to a pre-set threshold value. If the absolute signal value exceeds the threshold mark, the satellite is considered to be detected. If not, then the next combination of PRN and Doppler is

considered and the above described algorithm is repeated until all the satellites have been checked for availability.

For every FFT operation, the transform length (N) is considered as 2^n where n is a positive integer. In this case, $n = 15$, hence $N = 32768$ (value closest to 32735 dataset samples). Since the number of sample bits (32735) is less than N , the remaining transform length sequence is padded with zeros. To implement the FFT function in software (C/C++), the Keep It Simple Stupid FFT (KissFFT) (KissFFT, Source Forge, 2014) library has been used. This library is also used by software receiver developed in Gleason (2009), as it is simple and has a small file size compared to other libraries such as FFTW (Frigo and Johnson, 1998).

Tracking Algorithm

To track the satellite signals, initial carrier frequency and phase value is read from the previous step of acquisition. Tracking continues read

ing data samples (post 2 ms of initial acquisition data) and performs various operations on the samples. The samples are multiplied with accurate carrier and code replicas so that navigation data can be extracted from the signal. Due to the presence of multiple satellite signals, there is a need to track all these signals efficiently.

Humphreys (2009), suggested that task parallelism is best for maximizing execution speed up. Considering this recommendation, the concept of ‘multi-threads’ has been incorporated into the tracking processing block.

As seen in Figure 15, after acquisition is complete, a new processing thread is created for every acquired satellite. If ‘n’ satellites are detected, the same number of tracking threads are created. Each thread calls the same tracking function but is executed independently. For the given dataset used in this research, given the dataset, at least 9 satellites are tracked simultaneously. Without such an implementation, the receiver would be able to track only one satellite at any given time and would fail to read signal information from the rest of the satellites. Additional advantages of multi-threading is also listed in Butenhof (1997).

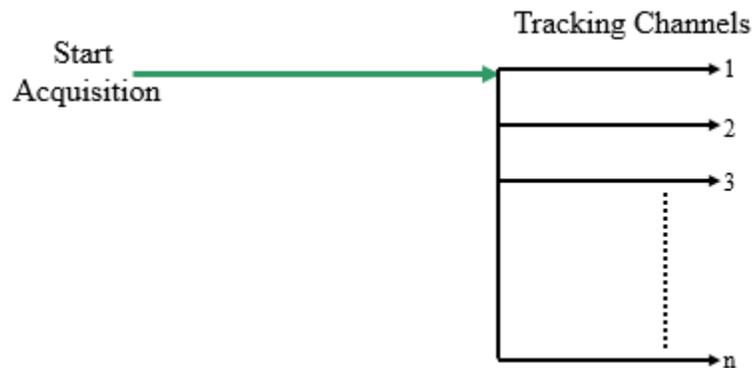


Figure 15: Tracking with multiple n channels.

POSIX Threads

Portable Operating System Interface (POSIX) threads, also known as Pthreads, is an IEEE standard that allows execution of several instructions in parallel. The three aspects that need to be managed with POSIX implementation are the execution context, scheduling and synchronization. This standard allows prioritization of certain threads that are more critical than the others and also allows the user to pause a certain thread process. These features

provide design flexibility that is greatly beneficial in the case of tracking algorithm development (Butenhof, 1997).

Post-acquisition of all the available satellites, threads are created according to the number of satellites detected. As shown in Figure 16, threads when created are initially in the *ready* state and remain in this state until the assigned thread function is called after which it enters the *running* state. Once in the *running* state, it can be *blocked* or *terminated*.

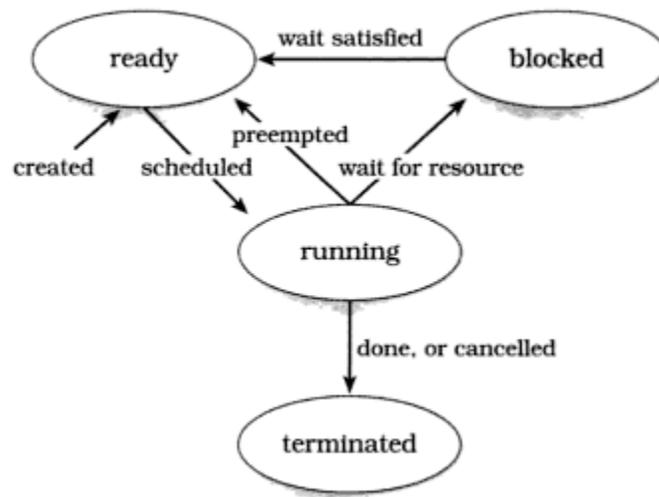


Figure 16: State transitions using POSIX threads (Butenhof, 1997).

When running multiple threads, the processes may try to access and modify the same memory space which may lead to corruption of data. To avoid uncontrolled modification of global variables and shared memory spaces, *mutex* is introduced. Only the thread that possesses the *mutex* has the permission to access a particular memory space or will be permitted to modify a certain global variable. As an example, if Channel 1 and 2 share

common variables and Channel 1 obtains a *mutex*, Channel 2 cannot modify the shared variables unless Channel 1 releases the *mutex* back. Use of a *mutex* during parallel processing, prevents mishandling of the data and corruption of data.

Tracking Loops

To accurately track the carrier signal and code phase of the incoming signal, the frequency and phase need to be continuously monitored and corrected. As explained in Chapter 2, Frequency/Phase Lock Loop (FLL/PLL) and Delay Lock Loop (DLL) are used for carrier signal and code phase tracking, respectively. Coarse frequency and code phase values from the acquisition step is used in tracking as well. The locally generated replicas may have large errors (due to initial coarse estimation) when compared to the incoming signal parameters. To quickly nullify the errors, high-gain filters that form a part of FLL/PLL are initially used.

Referring Figure 17, coarse tracking using FLL is performed in the first 500 ms using a filter gain factor of 1.15. At *FLL switch time*, tracking switches to a finer FLL with a lower gain factor of 0.10 to correct for smaller errors. The fine FLL runs for another 500 ms which is enough for the replica frequency to be almost the same as the real value. At *PLL switch time*, PLL tracking begins. This second order loop, with gain factors 0.10 and 0.93, closely tracks the incoming signal frequency and maintains the error to a minimum. Such combination of gain factors highlight the dependence of the tracking loops on the initial frequency estimated by acquisition. This loop runs for another 500 ms before the filter stops oscillating and settles. After the filter settles, the navigation bit decoding can begin

(Gleason and Gebre-Egziabher, 2009). The performance of these filters will be discussed in Chapter 4.

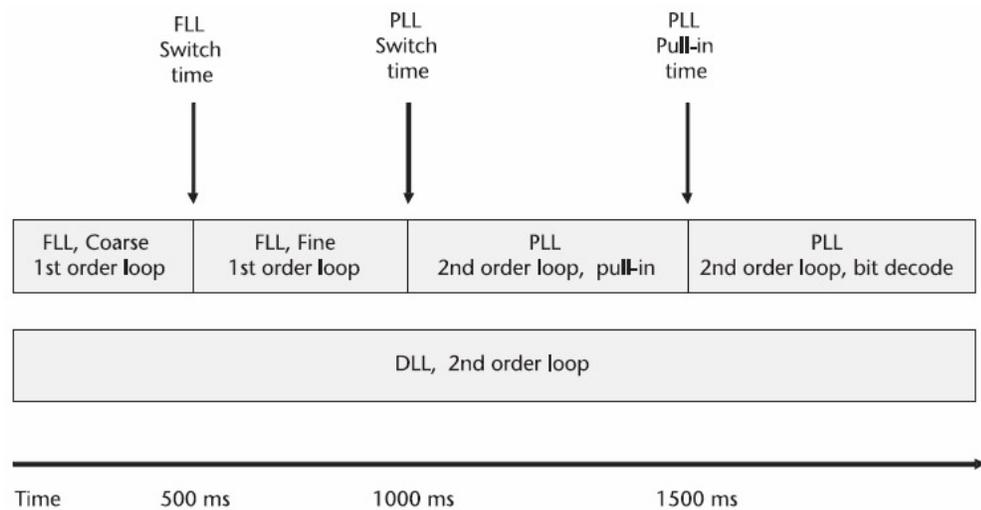


Figure 17: Tracking loop timeline (Gleason and Gebre-Egziabher, 2009)

As shown in Figure 18, the first step in tracking is to remove the carrier signal and PRN code from the incoming signal. Navigation data are then read from the remaining signal components. To detect the start of the navigation subframes, preambles are searched for. A preamble is a sequence of 1 and 0's (in this case *10001011*) and marks the beginning of a subframe (Borre et al., 2007). Once the first preamble is detected, it is expected to be found every 300 bits (as the subframe is repeated). If the preamble is found again in the expected position, it confirms correct reading of navigation data. For additional checks, HOW and TLM words within the subframe also consist of parity words which must match expected values. Once these tests are passed, the information can be trusted and further processing can be performed.

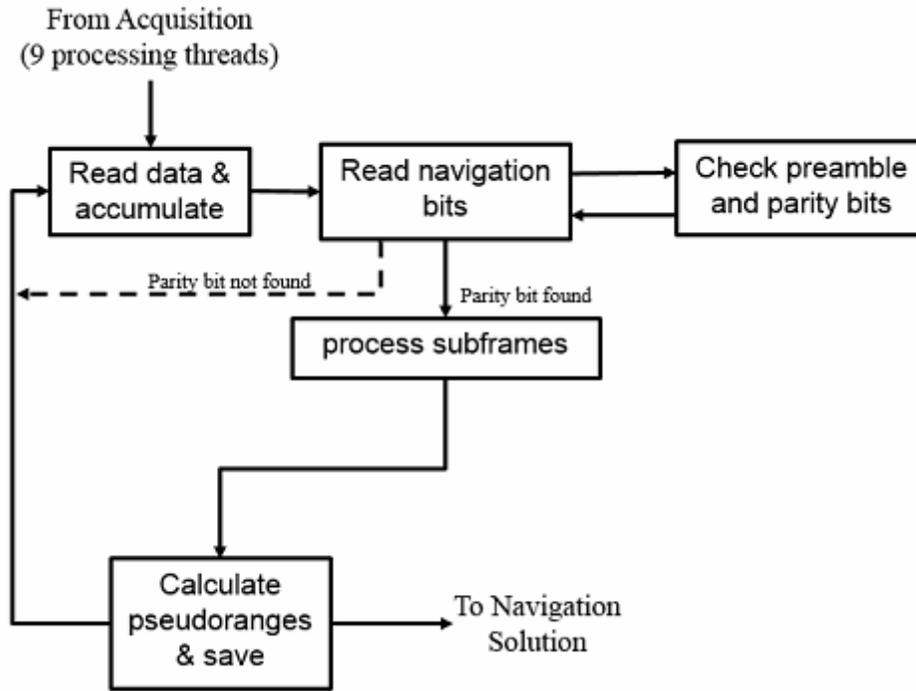


Figure 18: Tracking algorithm implemented with multiple processing threads.

After the presence of correct navigation message is confirmed, the subframes within the message are extracted by recognizing the ‘subframe ID’. Each subframe ID contains unique satellite and timing information. Following the recognition of subframe ID, the key parameter, Time of the Week (TOW) is read which is used to estimate the signal transmit and receive time. When the signal transmit and receive time is estimated, pseudoranges can be computed as shown in Equation 20.

$$\text{Pseudorange} = (\text{signal receive time} - \text{signal transmit time}) \times \text{speed of light}$$

Equation 20

Navigation Solution Algorithm

Each processing thread/channel in tracking computes pseudoranges and these values are stored in a global database so that it can be accessed later by other threads as well. A final processing thread is thus created that runs parallel to the rest of the tracking threads. This new thread reads pseudoranges computed by the 'n' tracking channels and estimates receiver position without interrupting the tracking process.

The processing 'navigation solution' thread is shown in Figure 19. This thread waits until the first set of pseudoranges is generated by each thread/channel and then starts position estimation. This added feature is an advantage over the reference software receiver *fastgps* where position computations do not begin until all the satellites are tracked for a duration of 25 seconds.

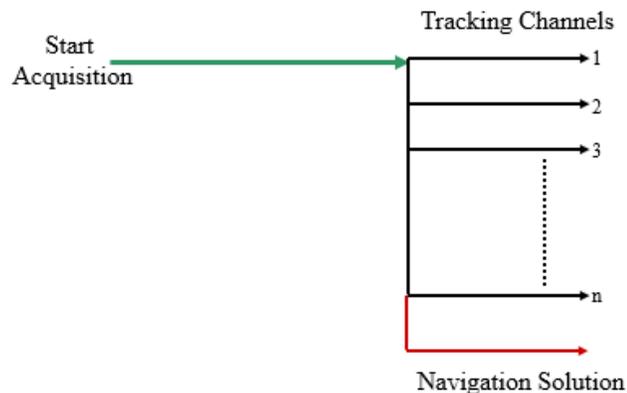


Figure 19: Simultaneous channel tracking and receiver position computation using POSIX threads.

The logic flow of the navigation solution thread is illustrated in Figure 20. Pseudoranges and satellite information (from the subframes) generated by the tracking loops is initially

read. Based on this ephemeris data, position of every satellite detected is determined. These calculated satellite positions and pseudoranges are then used to compute receiver position. The algorithm used for this purpose is least squares algorithm. Referring to Figure 21, the pseudo distance between the receiver and satellite, pseudorange, is already known from previous calculations and so is the satellite position. As described in chapter 2, Equation 18 and Equation 19 are used to find the unknown receiver position coordinates X_i, Y_i, Z_i and the receiver clock error.

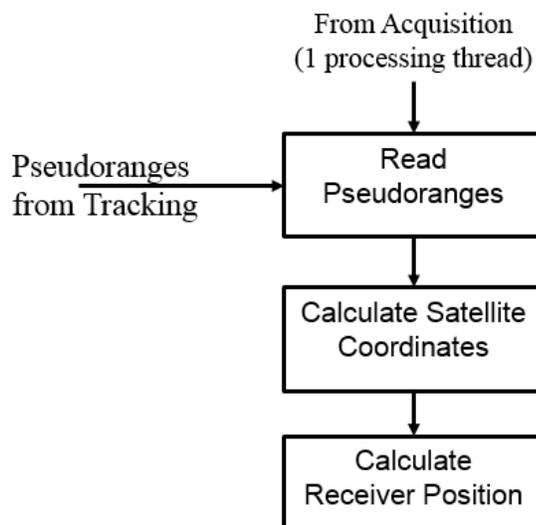


Figure 20: Navigation Solution: Processing steps for receiver position computation.

Some error corrections can also be applied at this stage of calculation. For this receiver implementation, the data file used was 25 seconds in duration, containing only 3 subframes and did not contain essential parameters from subframe 4 and 5 that are required in order

to apply ionospheric corrections. Hence corrections have not been applied during position computation.

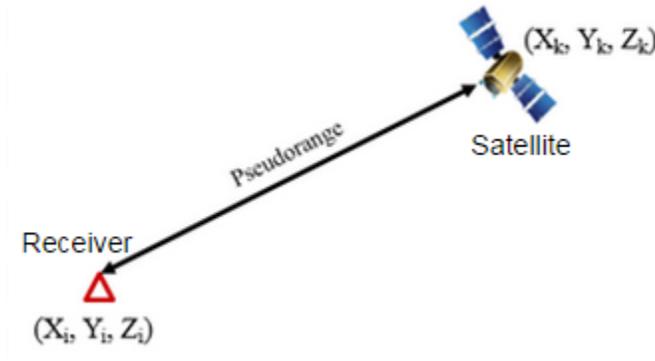


Figure 21: Pseudorange measurement between satellite and receiver.

Hardware Implementation

The software component design described in the previous section has been developed on a desktop computer (PC). Since the targeted application of the receiver is for space use, there is a need to implement the software on a space-ready processing platform. As discussed in Chapter 2, a comparison of various processors has been presented. For the receiver implementation in this research, a combination of a microprocessor and FPGA is used. Such a combination is manufactured by various companies like Altera, Xilinx, Actel, etc. with varying specifications. Hence it becomes critical to examine all the commercially available products and select one that would best suit the application.

Processor Selection

A comparison of software receivers developed is shown in Table 4. These receivers have also been compared with the design target of this thesis. As observed, the trend is to use a

hybrid processing platform of microprocessor and FPGA. Manufacturers such as Xilinx and Altera fabricate such platforms that provide a combination of software and hardware processors in the form of SoC. For this research, Altera's Cyclone V SoC is used as it complies with all the hardware requirements of the target receiver design and also costs lower than other similar products (Xilinx's Zynq SoC). Altera's Cyclone V SoC has an embedded (hardcore) ARM processor that functions at a satisfying frequency when compared with other processing platforms. The FPGA components and peripherals can be easily accessed by the microprocessor via the AXI bridges provided by Altera. FPGA also has the capability to access microprocessor peripherals using the same aforementioned bridge. Such a well-established communication bridge between the FPGA and microprocessor ensures high speed data transfer among the two components and avoids the need for the developer to set custom communication protocols.

The ARM processor that forms a part of Altera's Cyclone V SoC has a higher operating frequency as compared to other receiver platforms listed in Table 4. A higher operating frequency proves to be a major benefit as it indicates high processing speed. Even with this advantage, the SoC does not process in real time due to certain hardware limitations (discussed in Section 3.2.4.) but this capability can be incorporated with additional research work. Also the power consumption of the SoC is within an acceptable range of 2 to 5 Watts.

Table 4: Comparison of target design and recent software receiver design specifications

| | | | | | |
|-------------------------------|---|---|--|----------------------------|---|
| | WARP based GPS Receiver (Hershberger, 2013) | Software Receiver for ALMASat (Avanzi 2010) | Modular GPS Bistatic Radar Receiver (Esterhuizen 2006) | Namuru V3.2 (Glennon 2011) | Target Design |
| Processor type | FPGA + embedded processor (PowerPC 405) | FPGA+ embedded processor (PowerPC 440) | Single Board Computer/ VIA CoreFusion Processor | FPGA + ARM A3 Cortex | FPGA + embedded processor (Dual-core ARM A9) |
| Processor operating frequency | Max 400 MHz | Max 550 MHz | Max 1 GHz | Max 200 MHz | Max 925 MHz |
| Multiple tracking channels | No | Yes | Yes | Yes | Yes |
| Position Estimates | No | Yes | Yes | Yes | Yes |
| Real Time | Yes | Yes | No | Yes | No |
| Programming language | MATLAB/ Simulink | MATLAB/ Simulink | C++ | C | C/C++ |
| Space Applications | No | Yes | Yes | Yes | Yes |
| Power Consumption | - | <10 W | <18W | ~1W | 2-5W |

Embedded ARM

Similar to a computer requiring an operating system (OS) to run various applications, the embedded processor ARM also requires an OS before it can execute applications. Altera's Cyclone V SoC community strongly suggests using Embedded Linux linaro-gcc-arm as the OS (Altera, 2014a). To boot the ARM processor with this OS, a compressed image file

must first be created that can be run on the processor. The image file contains all of the information regarding the Linux drivers and modules. Another important file is the Device Tree Blob (DTB), containing information regarding all the relevant board peripheral attributes and addresses. As an example, if a LED is to be switched on, the OS will refer to the address of that particular LED from the DTB and then communicate to that address space.

Altera provides extensive information regarding generation of these essential files. These files are downloaded to an external SD card and then inserted into the card slot provided on the board. The processor is then powered up with embedded Linux (from the image file).

With Linux running on the processor, various applications can now be executed. Eclipse DS-5, a PC based application provided by Altera, allows for software development in C/C++ environment. Using Eclipse, C/C++ applications can also be run as Linux applications. The major advantage of using Eclipse is that it allows Linux application debugging as well. This way hardware and software parameters can be monitored in real-time and system faults can be narrowed down to certain part of the software/hardware.

Software/Hardware Partitioning

The software developed using the programming language C/C++ can run independently on the ARM processor that forms a part of the SoC. However, as the operating frequency of the microprocessor (900 MHz) is relatively low (when compared to a standard PC's 1.2

GHz oscillator) the application requires longer execution time. Many functions and operations in software require multiple clock cycles to complete and since they must be performed sequentially, additional time is required. Sequential software processing can be avoided by implementing the processes in parallel using FPGA (hardware).

There is no precise method for deciding which operations must be implemented in the FPGA instead of the microprocessor. Generally, the bottlenecks of a software process are ported over to the FPGA. By implementing certain operations in FPGA, processing load is taken off the microprocessor and allows task parallelism in the FPGA, saving several clock cycles and hence execution time.

Considering the software GNSS receiver, parallel phase search method has been implemented for acquisition. This method uses Fast Fourier Transform (FFT) and Inverse Fast Fourier Transform (IFFT) functions. Various FFT libraries are available as open source, such as FFTW (Frigo and Johnson, 1998) but the routine known as KissFFT (KissFFT, Source Forge, 2014) is used for this receiver implementation as it is simple and has a small file size.

After the implementation of the KissFFT library, a profiler tool named Gprof (GNU gprof: the GNU profiler) was used to analyze the complexity of software developed and also the execution time of each function. Such a tool aids in identifying which sections or operations of the software program should be implemented in hardware (FPGA). This way software load can be reduced and task parallelism can be achieved, improving overall execution

time. Results obtained from the Gprof tool indicates that the software FFT function alone requires 70% of the total program execution time. These results are further discussed in Chapter 4.

Due to the complexity of FFT/IFFT functions, along with the high execution time, it was decided to implement these functions in hardware (FPGA), instead of software (microprocessor). Referring to Figure 22, the highlighted area is now implemented in the FPGA.

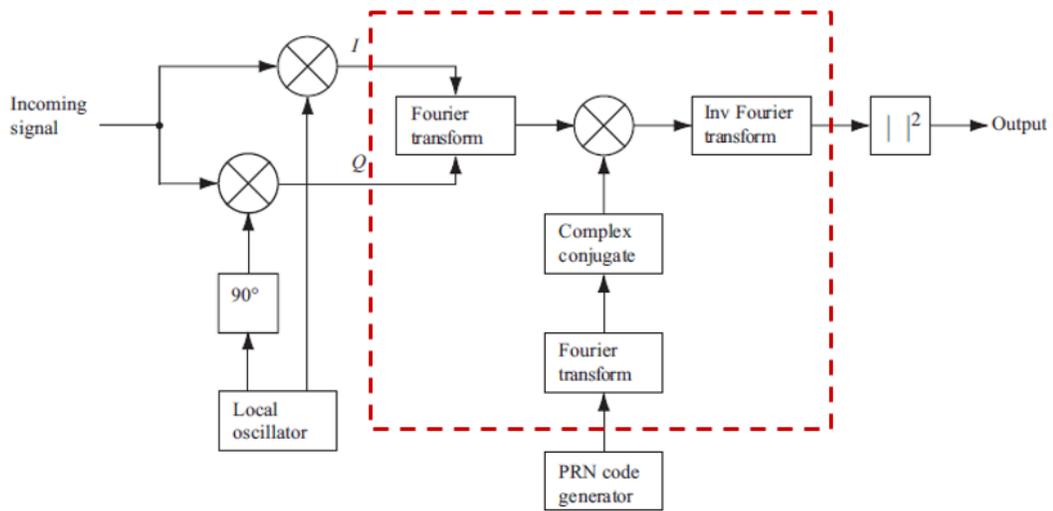


Figure 22: Highlighted region (FFT operations) implemented in the FPGA.

To implement the partial design in hardware, Altera's FFT IP (Intellectual Property) cores were used. FFT IP cores can be customised by the developer according to the required design specifications. For the hardware implementation in Figure 22, two FFT IP cores, one IFFT IP core, a multiplier (for complex signals) is required. The resulting hardware design is illustrated in Figure 23.

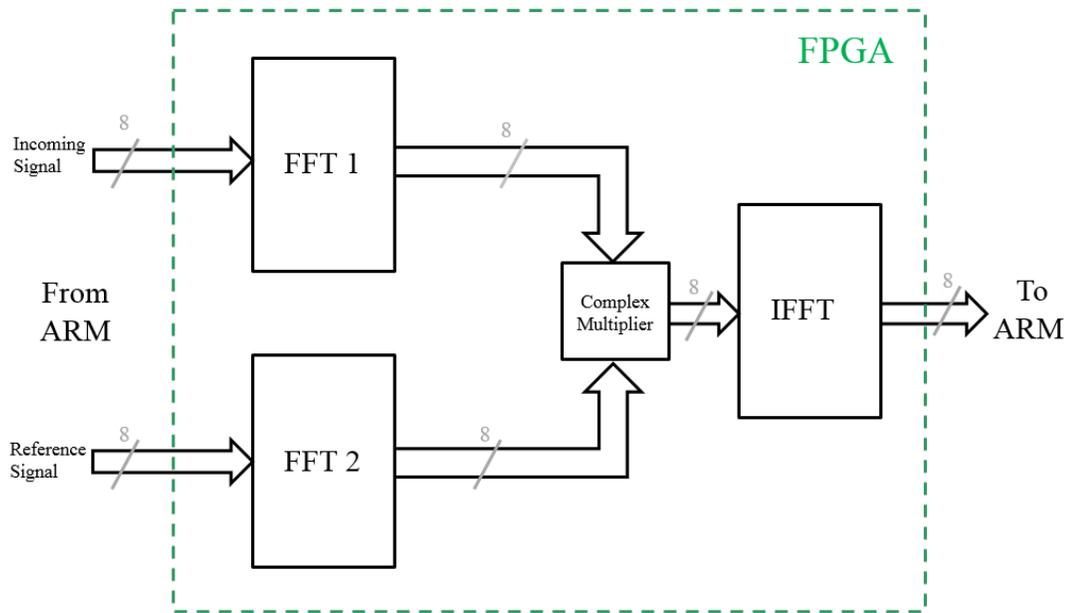


Figure 23: Altera IP Core Implementation in the FPGA.

Although Altera’s Cyclone V FPGA has a total block memory bits of 5662K, it imposes a limitation as the proposed design in Figure 23 does not fit successfully due to insufficient block memory bits. Part of the reason is that for acquisition, 2 ms of data results in a 32,735 bit long data sequence, making the FFT transform length of (2^{15}) 32768. Moreover, the data transfer width between the FPGA and ARM is 8 bit (minimum). This problem is discussed in detail in the following subsection.

FFT Core

Altera’s FFT IP offers different specifications and types of cores that can be implemented in the FPGA as a hardware component. Of the total memory bits available (5662K) the percent occupied by each type is listed in Table 5. Transform Calculation Cycle indicates

the total number of cycles required by the FFT block to transform the given input data sequence. Block Throughput Cycle indicates the total number of cycles required by the FFT to output the complete bit sequence. The memory bits, transform calculation cycle and block throughput cycle in Table 5 have been estimated using Altera’s development tool, Quartus II.

Table 5: Comparison of available FFT IP configurations.

| FFT IP | Memory Bits | Percent of Total Memory Bits | Transform Calculation Cycle | Block Throughput Cycle |
|-----------------------|--------------|------------------------------|-----------------------------|------------------------|
| Streaming | 2752K | 48% | 32768 | 32768 |
| Variable Streaming | 4455K | 78% | 65536 | 32768 |
| Buffered Burst | 2281K | 40% | 28796 | 36864 |
| Burst | 1277K | 23% | 28818 | 94355 |

The first type of FFT IP offered by Altera, Streaming FFT, allows a continuous input data sequence of a fixed transform length. The disadvantage of using Streaming FFT is that it occupies almost half of the total available memory bits although the total throughput and transform cycle is a minimum of 32768. Variable Streaming allows the developer to change the transform length in between FFT sequences. Variable transform length is an unnecessary feature for the current receiver implementation and it also occupies a significant percent of the available FPGA space. Buffered Burst and Burst methods require fewer memory resources but on the other hand requires more number of cycles to throughput the output sequence.

For the receiver implementation in this thesis, FFT of the type Buffered Burst has been used. It occupies a total of 40% of memory bits and has a reasonable Block Throughput. The total memory bits required for the design implementation (3 FFT IP cores) shown in Figure 23 is then estimated as:

$$3 \times 2281\text{K} = 6843\text{K}$$

Equation 21

The estimated required memory bits (6843K) exceeds the available memory bits (5662K) and hence this poses as a major design limitation. As a workaround to this problem, only one FFT core is implemented. The same FFT will be used to transform the reference signal and input signal, sequentially. IFFT can also be performed using the same block by changing certain FFT IP control signals.

Figure 24 shows the FFT block and all involved interface signals. Signals on the left side of the block are inputs and signals on the right are the outputs. If the FFT block is *enabled*, for every clock cycle, input data sequence (for the transform length 32768) is read using the signal ports named as *sink_imag* (imaginary component Q) and *sink_real* (real component I). The user indicates the validity of the inputs by setting the signal *sink_valid*. The start of the valid input data sequence is indicated by setting the signal *sink_sop* (start of packet) and the end is indicated by setting the signal *sink_eop* (end of packet). The input signal *inverse* is used to indicate to the transform block to perform whether FFT (*inverse=0*) or IFFT (*inverse=1*).

Similarly, at the output port, `source_imag` and `source_real` are the transformed data sequence. The signal `source_exp` (exponent) is used for output scaling purposes. The beginning and end of the output data sequence is indicated by `source_sop` and `source_eop`, respectively. The validity of the output signals is marked by `source_valid`, whereas `source_error` signal values can be used to interpret the type of errors being caused during the transformation (FFT MegaCore Function: User Guide, Altera, 2014).

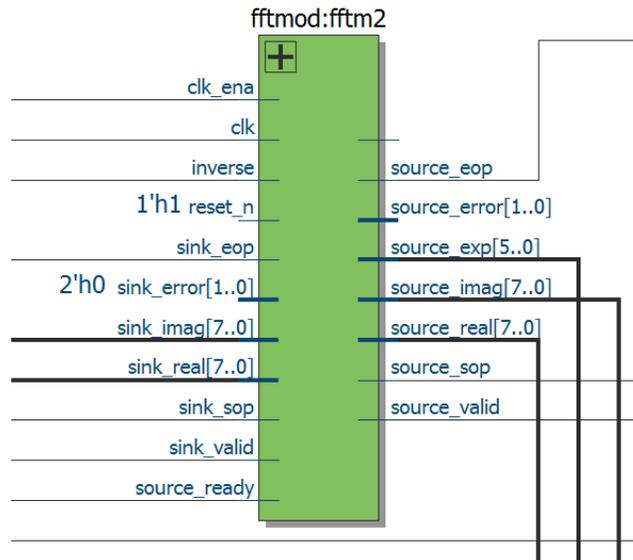


Figure 24: Data and control signal interface for MegaCore FFT IP (using Quartus RTL viewer).

The input control signals `clk`, `clk_ena`, `sink_sop`, `sink_eop`, `sink_valid`, `source_ready` are all set and reset in hardware (FPGA), designed using Verilog Hardware Description Language (HDL). However, signals `sink_imag`, `sink_real`, `inverse` are controlled from microprocessor also referred to as the Hard Processor System (HPS).

HPS-FPGA Interconnect

The digital data transfer between software-hardware components takes place over the communication bridges as shown in Figure 25. This interconnect uses the Advanced Microcontroller Bus Architecture (AMBA) Advanced eXtensible Interface (AXI) protocol for communication (HPS-FPGA bridges, Cyclone V SoC Technical Reference Manual, 2014).

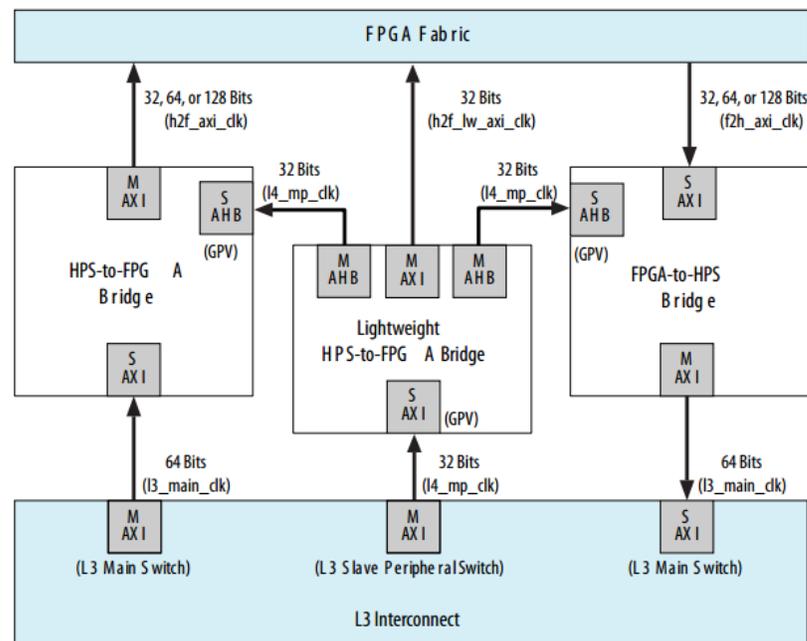


Figure 25: HPS-FPGA Bridges (Altera, 2014a).

Referring Figure 25, the three bridges that form part of the AMBA bridge are HPS-to-FPGA bridge, Lightweight HPS-to-FPGA bridge and the FPGA-to-HPS bridge. The L3 Interconnect forms a part of the ARM architecture. The lightweight (LW) HPS-to-FPGA bridge is 32 bits wide, whereas the HPS-to-FPGA bridge is 64 bits wide. For the receiver

implementation in this thesis, the LW bridge is sufficient for data transfer and hence the other bridges remain unused.

Parallel Input/Output Peripherals

When transferring data from the HPS to the FPGA, the data must be assigned to a particular address in the FPGA fabric. For this purpose, Parallel Input/Output (PIO) pins are implemented in the FPGA. Figure 26 shows the address map of each of the pins and Figure 27 shows the data flow direction between the FPGA and HPS.

The data width of the signals *hps2fftcontrol1* and *fft2hpscontrol* is 8 bits each, whereas *input_data2* and *output_data* are 16 bits wide each.

| | |
|---------------------------|---------------------------|
| <i>hps2fftcontrol1.s1</i> | 0x0001_0070 - 0x0001_007f |
| <i>input_data2.s1</i> | 0x0001_00d0 - 0x0001_00df |
| <i>output_data.s1</i> | 0x0002_00a0 - 0x0002_00af |
| <i>fft2hpscontrol.s1</i> | 0x0002_00c0 - 0x0002_00cf |

Figure 26: Address map of PIO pins

Since the FFT block requires imaginary and real components as input, *input_data2* contains both, 8 bits for each component. Similarly, *output_data* contains an 8-bit imaginary component and an 8-bit real component, concatenated as one signal. It is illustrated in Table 6.

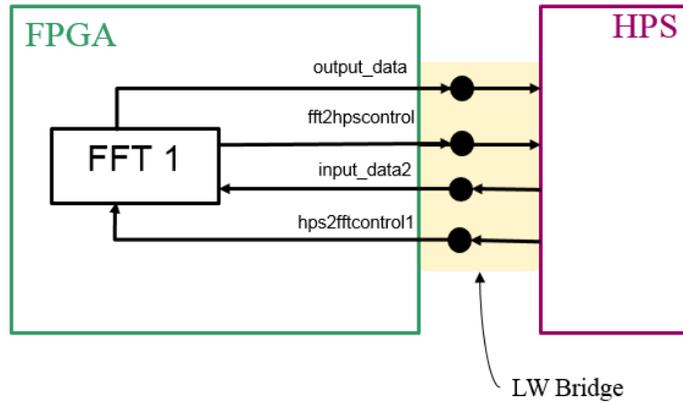


Figure 27: PIO data flow direction between FPGA and HPS.

Table 6: 8-bit real and imaginary bit assignment.

| | |
|-----------|-----------|
| real | 0-7 bits |
| imaginary | 8-15 bits |

The possible values for each signal component (real/imaginary) is -3, -1, 1 or 3 (due to the sample data file format). These values are interpreted differently by the bridge and the FPGA fabric. The representation of these integer values is as shown in Table 7.

Table 7: Binary and hexadecimal representation of integer data samples.

| Integer Representation | 8-bit Hexadecimal Equivalent | 8-bit Binary Equivalent |
|------------------------|------------------------------|-------------------------|
| 1 | 0x01 | 0000 0001 |
| 3 | 0x03 | 0000 0011 |
| -1 | 0xFF | 1111 1111 |
| -3 | 0xFD | 1111 1101 |

When developing the software in C/C++, values can be assigned to the PIO pins defined in the FPGA, by mapping them over the LW bridge. Address mapping is done by using the

predefined LW bridge base address (0xFF200000) and adding this to the base address of the respective PIO pin. As an example, to assign a value to the *input_data2* pin (Figure 26), the address used will be $0x000100D0 + 0xFF20000 = 0xFF2100D0$.

However, data transfer cannot be made directly from the HPS to the FFT block (via LW bridge) as a statement written in C/C++ in software can require more clock cycles for execution whereas operations in FPGA are performed in parallel and hence require fewer clock cycles. Simply put, there is a need to synchronize the data transfer between the FPGA and HPS to avoid data loss or corruption.

FIFO Core

For data synchronization between HPS and FPGA, Altera's SCFIFO IP core is used. SCFIFO stands for 'Single Clock First In, First Out'. FIFO acts like a buffer and is capable of temporarily storing data. Using a FIFO in the design ensures that all the data being transferred over the bridge is not corrupted or lost due to unsynchronized clocks between the FPGA and the HPS.

The FIFO implemented in the design has a depth of 32768 words and a signal width of 16 bits. Referring Figure 28, the signal *aclr* (asynchronous clear) is used to clear the contents of the buffer. When the *wrreq* (write request) signal is set, for every clock cycle, data are read and stored into the FIFO. When processed data are to be read from the FIFO at the q port, *rdreq* (read request) must be set. Signals *full* and *empty* indicate the state of the buffer whereas *almost_full* and *almost_empty* can be used to mark the state of the FIFO before it

is full or empty, respectively. The *usedw* signal indicates the count of words that have been stored in the FIFO. For an empty buffer *usedw* is 0 and when full it indicates 32767. Using these FIFO cores, a final hardware implementation has been illustrated in Figure 29.

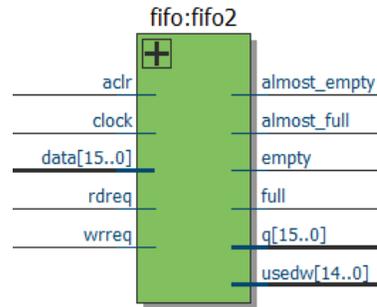


Figure 28: SCFIFO control and data interface signals (using Quartus RTL viewer)

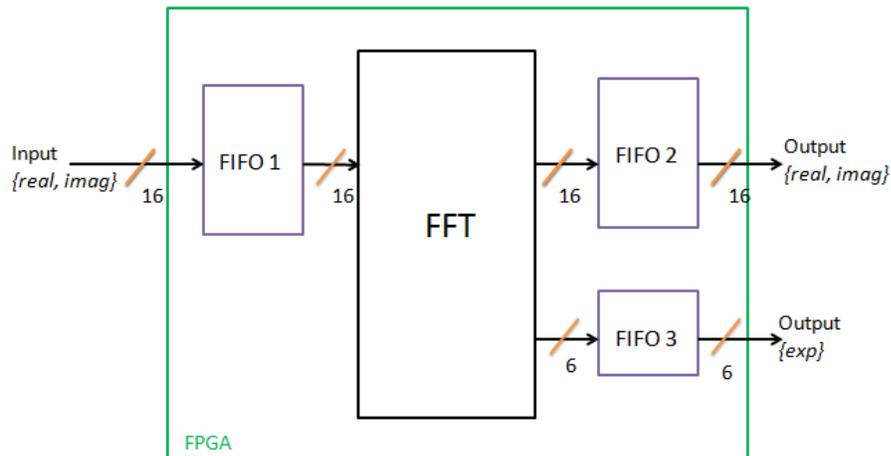


Figure 29: Final hardware design implemented in the FPGA using FFT and FIFO cores.

The two FIFO cores are used at the input and output port of the FFT block. However, FFT output contains an additional scaling signal named *exp* which also requires a FIFO block. Since the *exp* signal is 6 bit wide, a new FIFO is instantiated in the FPGA with a 6-bit

input/output signal width while maintaining the depth as 32768. The final hardware module developed in Figure 29 is used multiple times to perform FFT/IFFT for any data sequence of length 32768. Figure 30 illustrates the final block diagram of the acquisition routine that has been implemented, that contains both software and hardware components. The advantage of using such a system is discussed in Chapter 4.

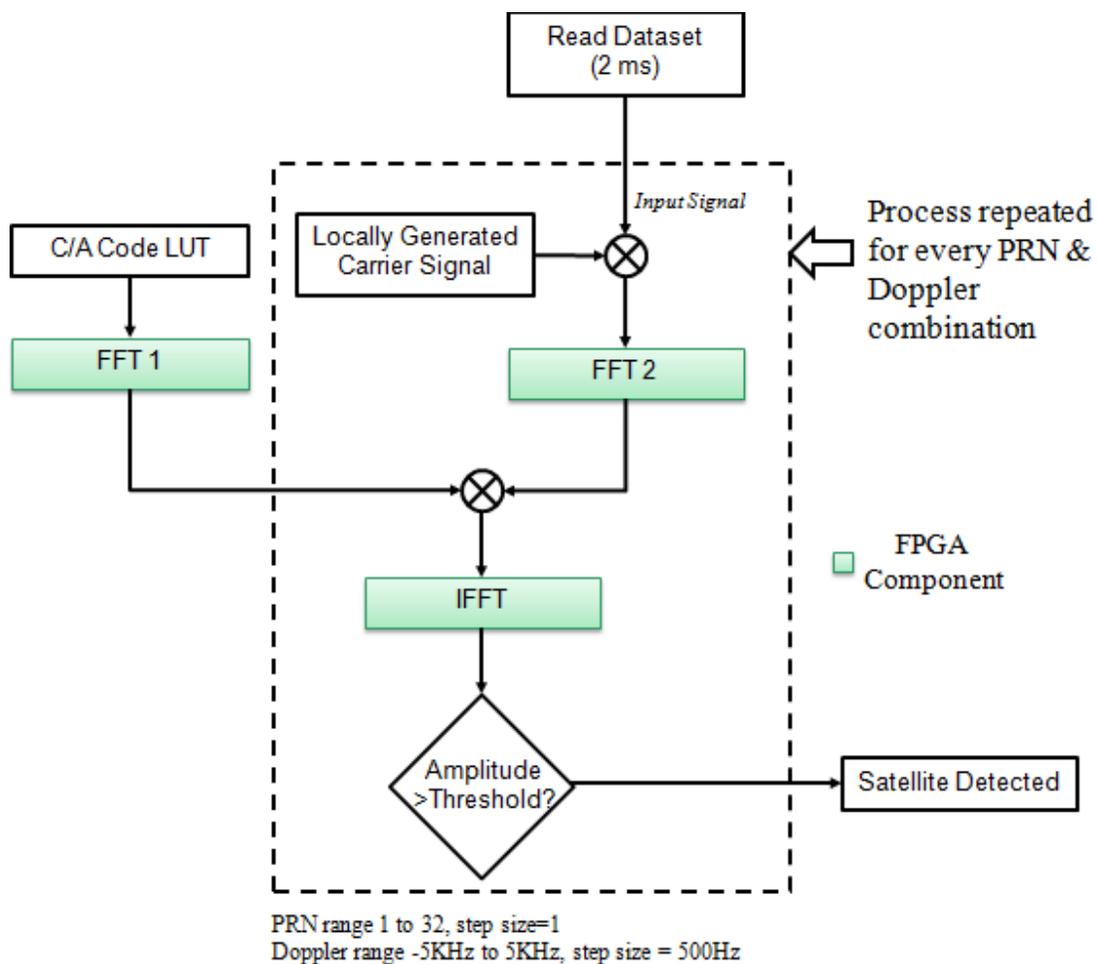


Figure 30: Final acquisition algorithm using FPGA components for FFT and IFFT operations.

Chapter Four: Receiver Performance Tests

And Analysis

To assess the performance of the software GNSS receiver implemented in this research, several tests were performed. Since the complete receiver has several processing blocks, tests were implemented for each block independently. The results obtained are discussed in the following sections.

Test Setup

The developed software GNSS receiver was tested in two phases. In the first phase, receiver algorithm developed on a desktop computer was tested using data files provided by Gleason and Gebre-Egziabher (2009). These results were then compared to the results obtained using the reference software receiver *fastgps* provided by the same source, Gleason and Gebre-Egziabher (2009). Algorithm testing of the software module was the main focus of this testing phase.

In the second phase of testing, competency of the developed receiver algorithm was tested on the selected processing platform (Altera's Cyclone 5) instead of the desktop computer. Since Cyclone 5 has hardware (FPGA) and software (ARM) components, hardware/software synchronisation was also verified and analysed.

Figure 31 shows the test setup used. An Ethernet connection is required for Linux application debugging purposes. USB Blaster and UART connect to the desktop computer and facilitate FPGA programming. The SD card holds important Linux boot information and the Warm Reset button is used to reset the ARM processor.

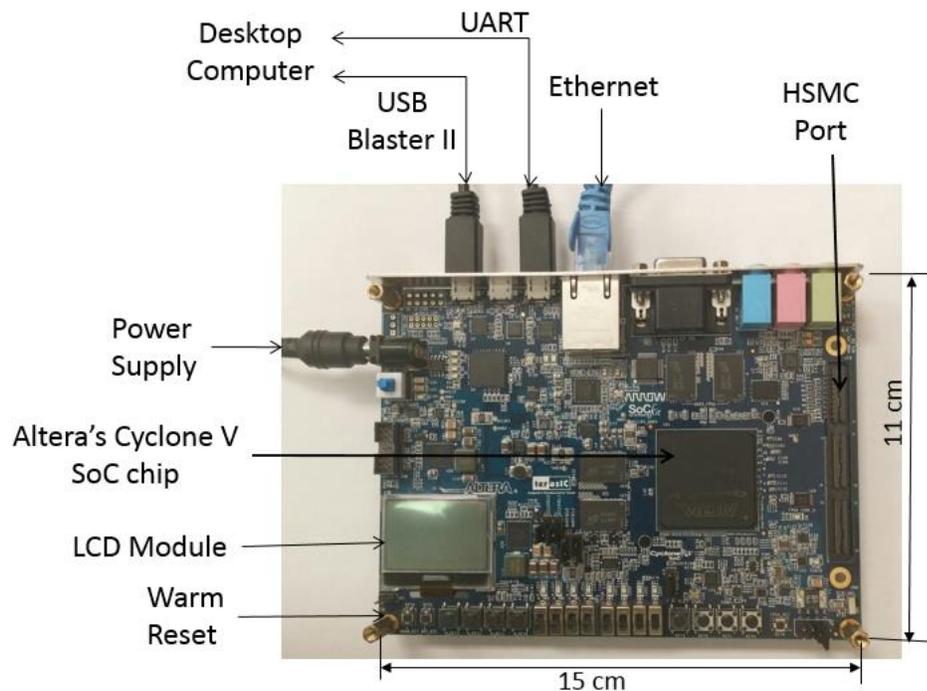


Figure 31: Cyclone V SoC board and desktop computer interface signals.

Signal Acquisition

GNSS signals operate on several frequencies corresponding to the multiple constellations. For the receiver implementation in this thesis, the focus is on the GPS constellation. GPS has a total of 32 satellites (24 satellites in the nominal operating constellation) and each of them have a unique PRN code that is transmitted in the GPS signal. Of the two GPS signals

L1 C/A-code and P(Y)-code (discussed in Chapter 2), L1 C/A-code signal is the focus of this research thesis and hence only this signal is processed and tested.

The input data format used is discussed in Chapter 3. It has four data levels of 1, 3, -1, and -3, as observed in the plot in Figure 32. The pre-calculated PRN code for a satellite is plotted in Figure 33 and as observed, has two data levels, 1 and -1. These two signals act as input to the FFT processing block.

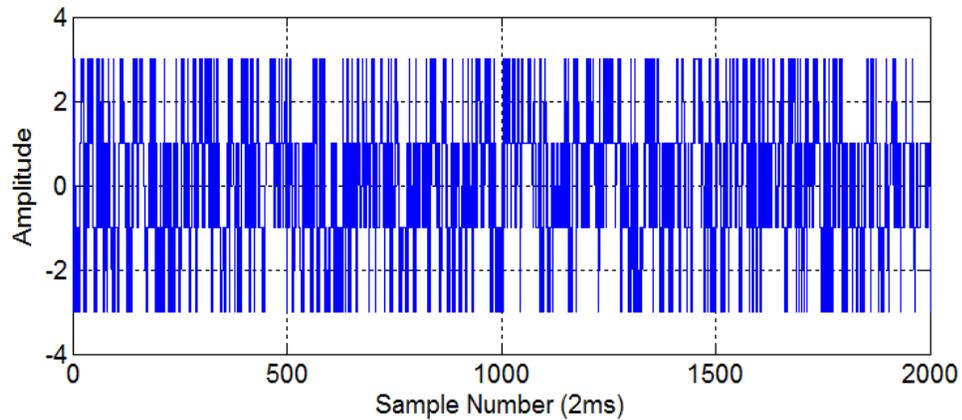


Figure 32: Input sample data with four amplitude levels of -3, -1, 1 and 3.

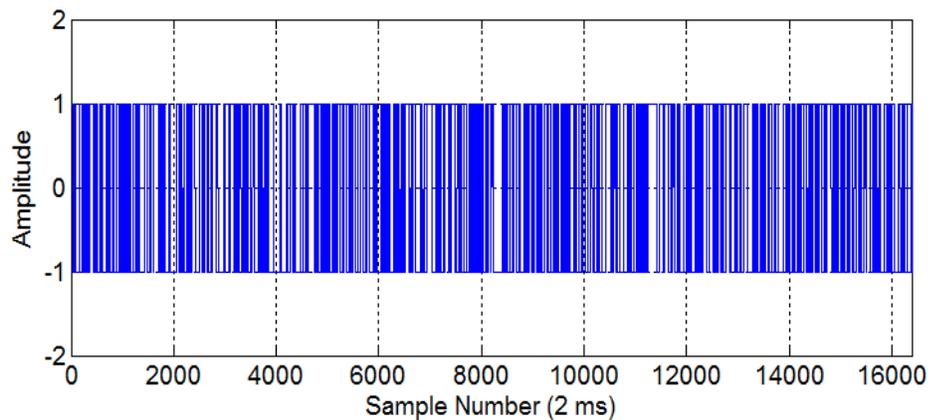


Figure 33: C/A-code for PRN 1 with two amplitude levels of -1 and 1.

Acquisition Result

The purpose of acquisition is to determine the visible satellites and the coarse values of the respective satellite signal frequency and phase. During the data processing of the first 4 milliseconds of data, on detecting a satellite, the calculated coarse and fine Doppler frequencies (in KHz), magnitude and code phase (in chips) of the acquired satellite signal is published in the console (Figure 34).

| PRN | Coarse Acq | Fine Acq | Signal Mag | Max Phase |
|-----|--------------|--------------|------------|-------------|
| 2 | 700.000000 | 659.608000 | 43.471837 | 473.216350 |
| 4 | -2000.000000 | -1967.528000 | 19.081202 | 226.576282 |
| 7 | 2100.000000 | 2196.624000 | 22.083226 | 938.994807 |
| 8 | 3900.000000 | 3964.944000 | 31.074411 | 632.913607 |
| 10 | 3300.000000 | 3389.496000 | 19.103364 | 435.089021 |
| 13 | -1100.000000 | -1005.752000 | 44.631578 | 876.803507 |
| 23 | -2800.000000 | -2711.296000 | 24.946309 | 766.234252 |
| 25 | 400.000000 | 312.088000 | 61.679473 | 1001.873648 |
| 27 | 2000.000000 | 2099.000000 | 25.190308 | 190.199059 |

Figure 34: Acquisition: List of detected satellites along with signal parameters like coarse frequency, fine frequency, magnitude and code phase of the incoming signal (console screenshot).

The information obtained during acquisition is plotted in Figure 35. The plot also provides a comparison of the satellite signal strengths. Only the satellites with signal strength exceeding the acquisition threshold value were considered to be acquired. In this case, the threshold value is 19 and signal strength exceeding this value indicates presence of the corresponding satellite (Kaplan and Hegarty, 2006; Bastide et al., 2002). Available satellites are highlighted in Figure 35. A total of 9 satellites were acquired at the end of

acquisition. These results have been verified with results obtained from the software receiver *fastgps* for the same data set and hence it is verified that the acquisition functions as expected.

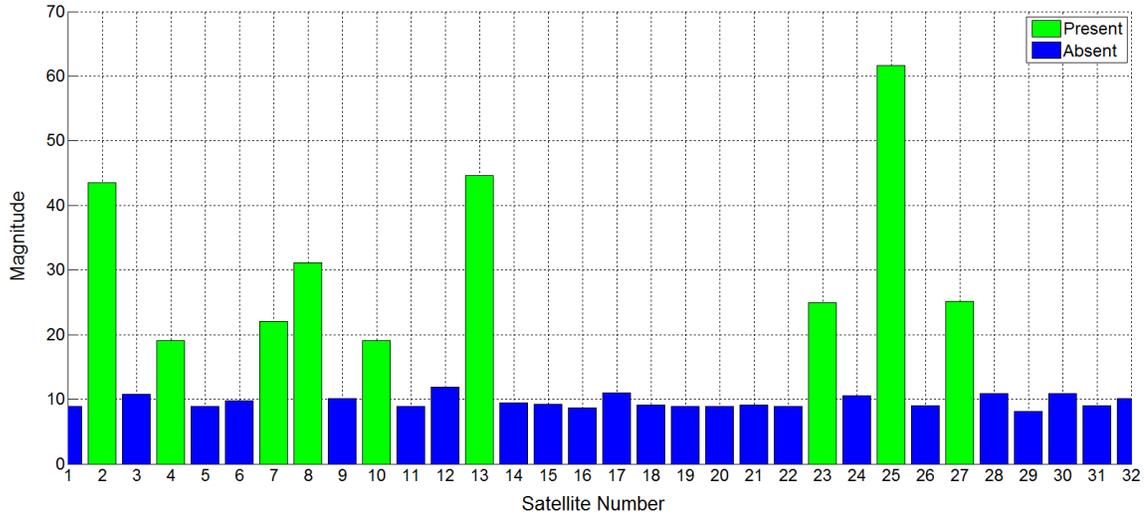


Figure 35: Acquisition plot – signal magnitude versus satellite number.

Doppler frequency is the shift of the incoming signal frequency from its true value due to the relative motion between the satellite and receiver due to Doppler effect. Figure 36 demonstrates the extent of Doppler frequency shift from the true value. Only the frequencies of the acquired satellites is depicted.

It is to be noted that the accuracy of the Doppler frequency depends on the length of the FFT sequence. For acquisition, 4 ms of data were used, resulting in a 65470 bit long FFT. During the second stage of testing using Cyclone 5, the acquisition length was shortened to 2 ms (32734 bit FFT) because of hardware limitations when implementing a 65470 bit

FFT core. Consequently, the Doppler frequency determined when using 2 ms of input data is less accurate than when 4 ms of input data is used.

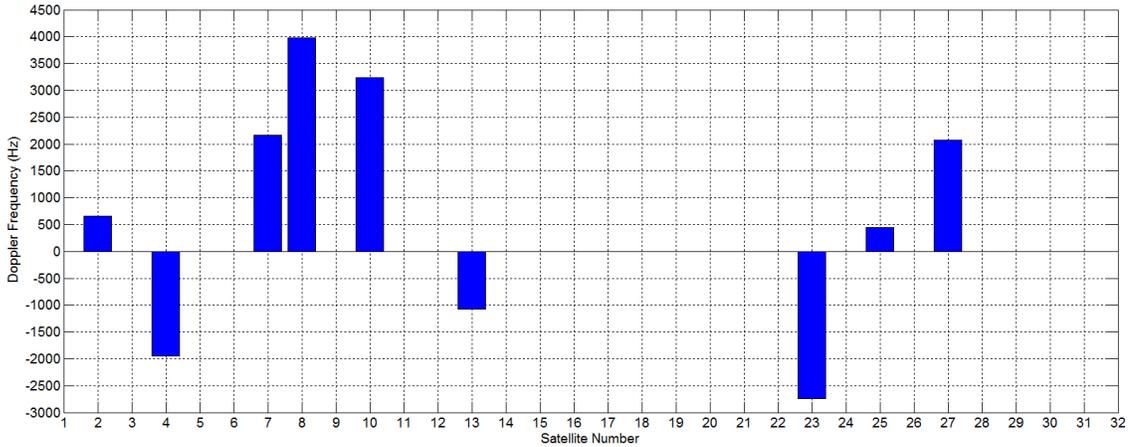


Figure 36: Satellite acquisition - Doppler frequency versus satellite number.

Code Profiling Results

The next step was to optimize the software code (in C/C++) to accelerate the overall processing. Prior to code optimization, it was necessary to determine the amount of time required by the software functions for complete execution. For this purpose, a code profiling tool named gprof was used. Gprof (GNU gprof: the GNU profiler) provides detailed information about the time spent in each function of the software application. Figure 37 provides a summary of the flat profile generated by gprof application.

From the timing information in the results obtained, the compiler required 55% of the total execution time for the function *kf_bfly4*. The function *main* required 19% of the execution time whereas function *kf_work* required 15%. *kf_bfly4* and *kf_work* both are daughter

functions of the primary KissFFT function. Consequently, the total time spent within KissFFT function was approximately 70% of the total processing time.

Each sample counts as 0.01 seconds.

| time | % cumulative seconds | self seconds | calls | self ms/call | total ms/call | name |
|-------|----------------------|--------------|-----------|--------------|---------------|-------------------------|
| 55.18 | 20.78 | 20.78 | | | | kf_bfly4 |
| 19.46 | 28.11 | 7.33 | | | | main |
| 15.60 | 33.98 | 5.88 | | | | kf_work |
| 9.56 | 37.59 | 3.60 | 9 | 400.00 | 400.00 | fine_doppler |
| 0.16 | 37.65 | 0.06 | | | | kf_bfly3 |
| 0.03 | 37.66 | 0.01 | | | | kf_bfly2 |
| 0.01 | 37.66 | 0.01 | | | | kf_factor |
| 0.00 | 37.66 | 0.00 | 248159203 | 0.00 | 0.00 | kiss_fftri |
| 0.00 | 37.66 | 0.00 | 234247 | 0.00 | 0.00 | reset_parameters |
| 0.00 | 37.66 | 0.00 | 11027 | 0.00 | 0.00 | navbit |
| 0.00 | 37.66 | 0.00 | 5680 | 0.00 | 0.00 | kiss_fft |
| 0.00 | 37.66 | 0.00 | 5680 | 0.00 | 0.00 | kiss_fft_stride |
| 0.00 | 37.66 | 0.00 | 32 | 0.00 | 0.00 | kiss_fftr |
| 0.00 | 37.66 | 0.00 | 22 | 0.00 | 0.00 | process_subframe |
| 0.00 | 37.66 | 0.00 | 9 | 0.00 | 0.00 | nav_init |
| 0.00 | 37.66 | 0.00 | 6 | 0.00 | 0.00 | nav_parity |
| 0.00 | 37.66 | 0.00 | 5 | 0.00 | 0.00 | range_calc |
| 0.00 | 37.66 | 0.00 | 4 | 0.00 | 0.00 | kiss_fft_alloc |
| 0.00 | 37.66 | 0.00 | 1 | 0.00 | 0.00 | kiss_fft_next_fast_size |
| 0.00 | 37.66 | 0.00 | 1 | 0.00 | 0.00 | kiss_fftr_alloc |

Figure 37: Gprof results enlisting the percent of execution time required by software functions.

Since the software function KissFFT, due to its computational complexity, required majority of the execution time, it was decided to replace this software FFT module with its equivalent hardware design in the FPGA.

Hardware FFT versus Software FFT

The results from the acquisition stage was obtained by using software components only and processed on a desktop computer. The same software, when executed using the embedded ARM processor provided in Cyclone 5, had a higher execution time. The dissimilarity appears in the execution time due to the differing operating frequencies of the ARM processor and the computer. The desktop computer has an operating frequency of 3.4 GHz, whereas Altera's Cyclone 5 operates up to a maximum of 900 MHz. Due to the lower operating frequency of Cyclone 5, there exists a need to reduce the computational load on the ARM processor and replace them with reconfigurable hardware components implemented in FPGA. As discussed in the previous section, Fourier transformation proves to consume a significant execution time and consequently, the software FFT component was replaced with its hardware equivalent in the FPGA.

According to the acquisition function block diagram in Figure 30, both the signals, input data (Figure 32) and the locally generated C/A-code (Figure 33), was transformed using FFT block. The transformed spectrum is in accordance with the power spectrum of the C/A-code (Kaplan and Hegarty, 2006). The input C/A-code sequence was first transformed using the software FFT and then the same input sequence was transformed using hardware FFT. A comparison of the quality and resolution of the two transformations is illustrated in Figure 38 and Figure 39. As observed from the set of figures, for the same set of input data sequences, there is a noticeable difference in the amplitudes of the two FFT outputs. Also, the pattern (lobes) of the transform is less prominent when hardware FFT is used.

These differences are present due to several limitations when the hardware FFT core is implemented.

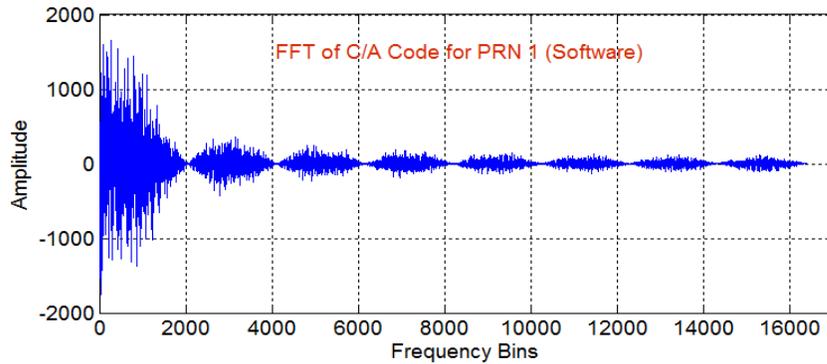


Figure 38: FFT of C/A-code for PRN 1 using software FFT function.

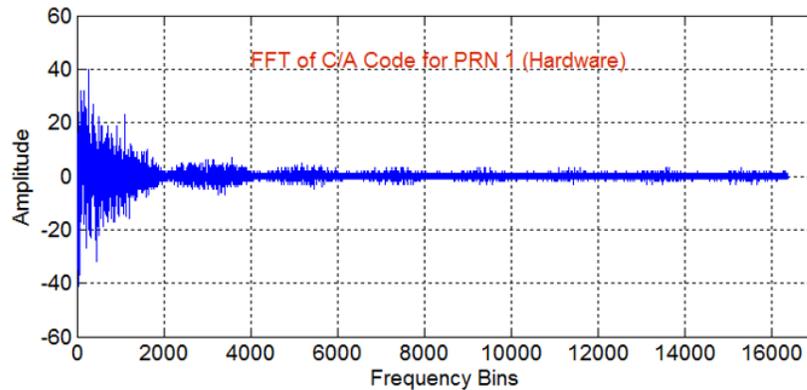


Figure 39: FFT of C/A-code for PRN 1 using Hardware FFT function.

The first limitation is introduced as the output port of the hardware FFT block is represented with a maximum of 8 bits (FFT core configuration) allowing a maximum output signal value within the integer range of -128 to 127. The second limitation of using the hardware FFT is that the output data are represented in integers and hence decimal

values are truncated. These two factors result in truncation of output values which in turn affects the overall FFT performance and accuracy. A similar difference is observed in Figure 40 and Figure 41, when the input sample data was transformed using FFT components in software and hardware, due to the same configuration limitations discussed above.

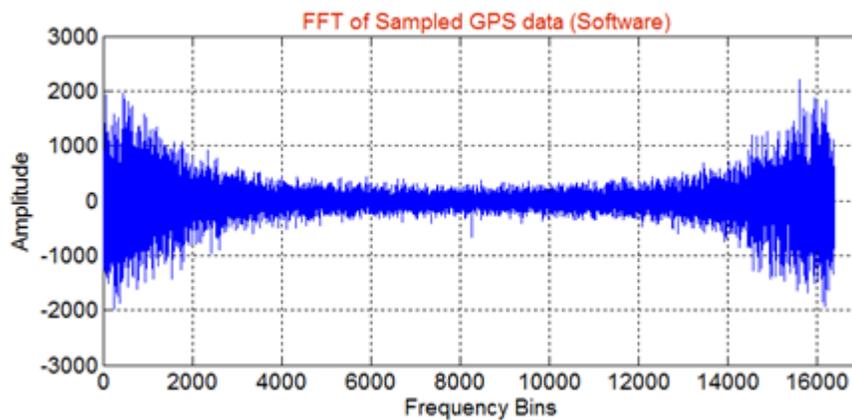


Figure 40: FFT of sampled GPS data using software FFT function.

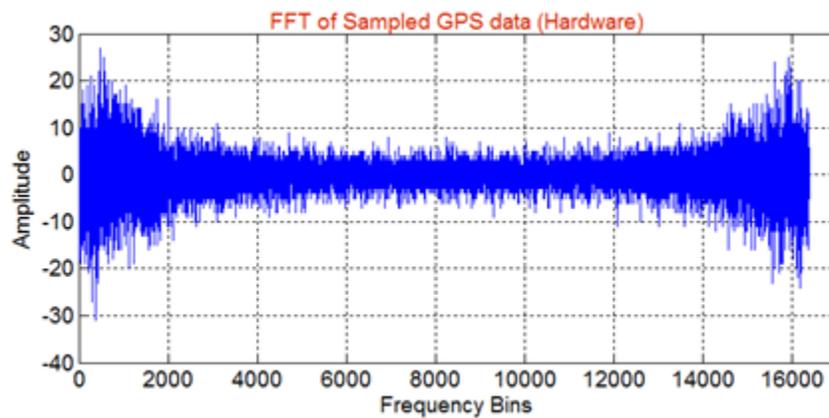


Figure 41: FFT of sampled GPS data using hardware FFT function.

According to the acquisition block diagram illustrated in Figure 30, the next step was to compute the product of the transformed input signal and transformed PRN code. This product is then inverse transformed again using the IFFT block. When using the software component, an inverse FFT function is called, whereas for the hardware component, the same FFT block configured in the FPGA is used to perform inverse FFT as well.

Seen in Figure 42 is the result of the product of the two FFT components, input signal and locally generated signal. This product is inverse transformed back to time domain using IFFT function. When hardware FFT is used, this poses as a limitation. The product of the FFT components, as seen from Figure 42, ranges between the values of -700 to 900. It is impossible to represent all these values with the limited 8 bits (FFT configuration for input data width). As discussed earlier, 8 bit signal width represents values ranging from -128 to 127 only. Hence representing the FFT product is a problem.

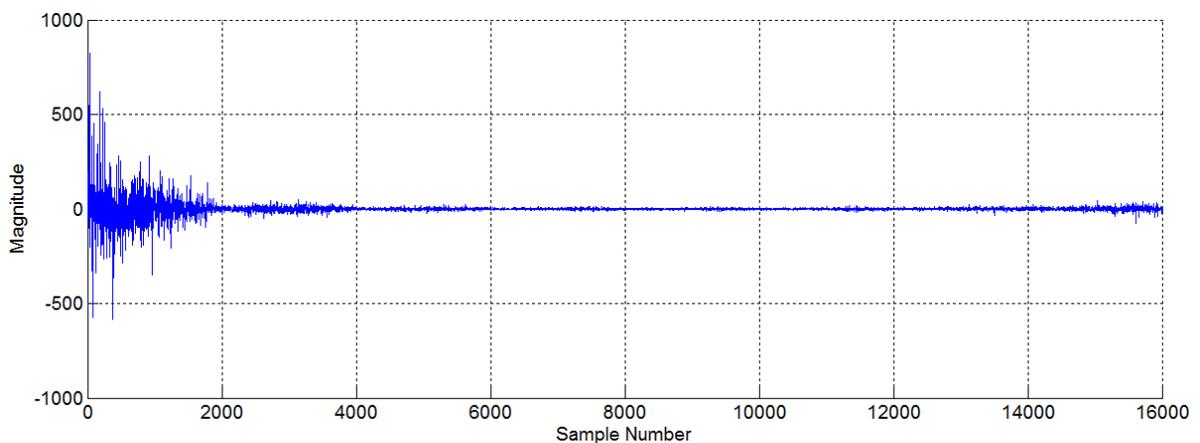


Figure 42: Product of FFT outputs (I component)

As a solution, the amplitude of the FFT product is scaled. Ideally, if the amplitude is scaled by a factor of 0.13 or (1/7.2), the FFT product is within the range of -128 to 127. Due to this scaling factor, FFT product samples below the value of 7.2 are truncated to 0. This amplitude truncation has an adverse effect on the FFT product that leads to an incorrect IFFT computation. Consequently, acquisition results are inaccurate and incorrect satellites are acquired.

After implementing different scaling values and examining the acquisition results, optimal performance was observed for a scaling factor of half (0.5). A comparison of 3 cases of acquisition is illustrated in Figure 43 below.

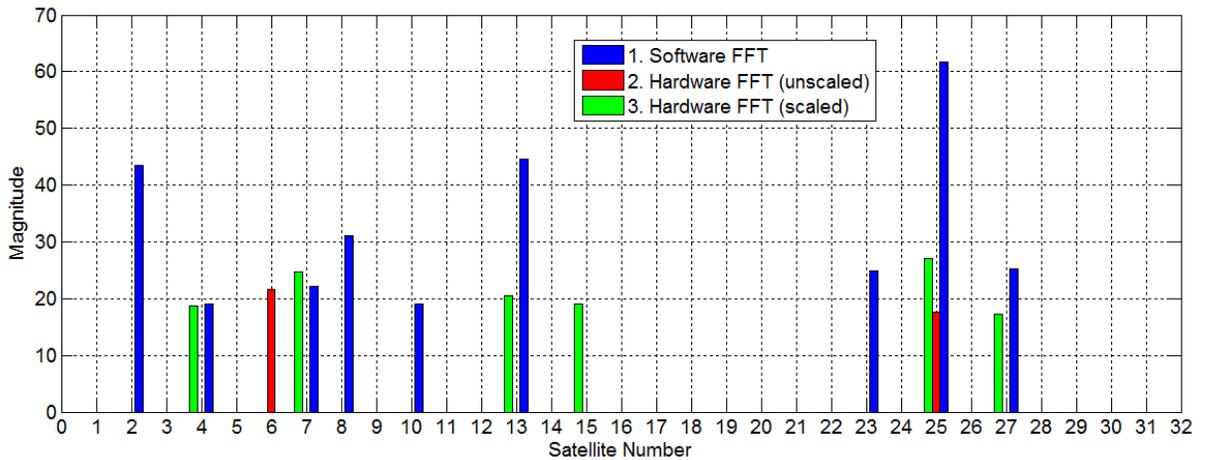


Figure 43: Acquisition – comparison of results obtained when software FFT, scaled and unscaled hardware FFT functions are used.

In case 1, the acquisition results from the software FFT function is plotted. When this software component is replaced with hardware FFT, only 2 satellites are correctly acquired

(case 2). By scaling the FFT samples by half, 6 satellites are detected (case 3). Among these satellites, signal strength of PRN 15 is higher than expected and is acquired incorrectly. A possible reason for the incorrect acquisition of PRN 15 is due to the scaling factor mentioned earlier. Scaling alters FFT outputs decreasing acquisition accuracy. This extra satellite is assigned a channel for tracking, but due to the absence of navigation data, the tracking channel is terminated. Hence, this extra satellite detected costs a channel for a short duration and does not affect the performance of the receiver significantly.

Scaling effect has negative implications on the overall acquisition result quality and can be avoided if the signal bit width to the FFT IP core is increased (up to 16 bits). Such a design implementation requires higher FPGA resources (memory bits). Due to unavailability of sufficient memory bits on the selected FPGA processing platform, the FFT IP core is limited to a signal width of 8 bits.

Processing Time Comparison

The motivation for replacing software components with its hardware equivalent is to reduce the overall processing time. However, synchronizing the software and hardware components within the SoC is a challenge.

During the execution of the receiver application, the C/C++ statements required multiple clock cycles for execution, whereas the hardware operations required less. This observation is made using the SignalTap software provided by Altera that allows the user to examine signals within the FPGA. A comparison of the clock cycles is presented in Figure 44.

The two important waveforms to be considered is highlighted by the arrows. The first waveform is generated by setting and resetting the hardware Parallel Input/Output (PIO) pin using the consecutive statements (in software) written below. The first statement sets ('1') the pin and the second statement resets it back ('0'). The bit transition is noticed in the waveform and according to the time bar, execution of the two statements requires 2000 milliseconds.

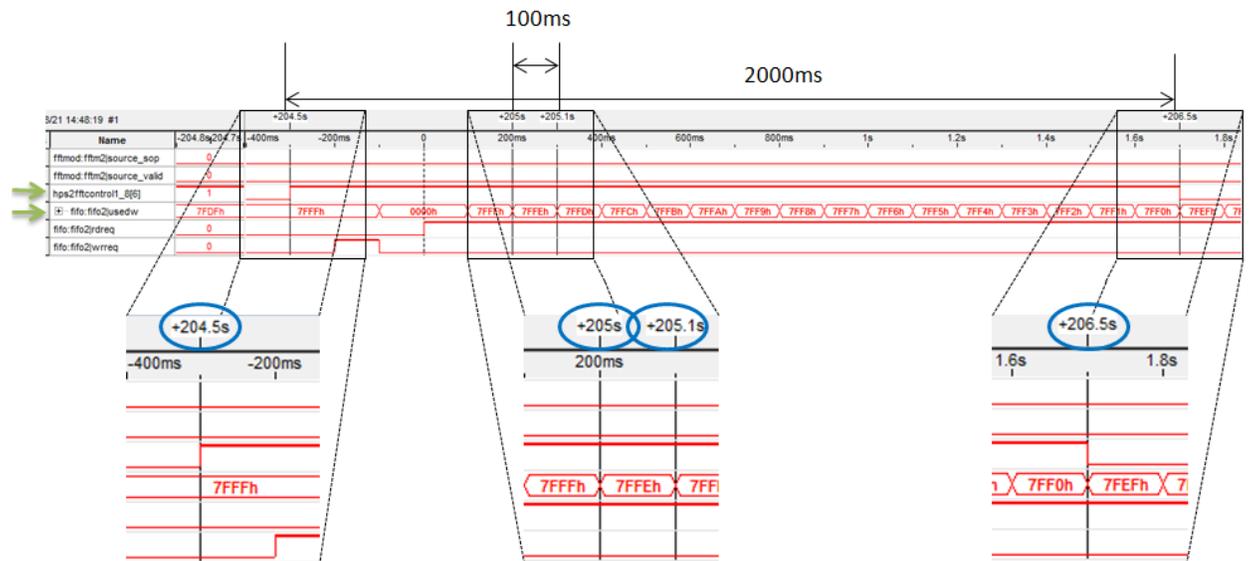


Figure 44: Software/hardware synchronization - SignalTap waveform comparing hardware and software clock cycles

```
alt_write_byte (lw_bridge_map+ HPS2FFTCONTROL1, 0x60); //Write Request Pulse
alt_write_byte (lw_bridge_map+ HPS2FFTCONTROL1, 0x00);
```

The second waveform represents the cycles required for one hardware operation. Since the time duration from one bit transition to the next is considered, referring to Figure 44, the

interval between each transition is 100 milliseconds. This leads to the conclusion that hardware operation in the FPGA is **20** times faster than the software operations.

As described in Chapter 3, the FFT core selected for this receiver design is Buffered Burst and requires a total of 36864 clock cycles for complete FFT transformation in hardware. Whereas the function KissFFT (software FFT) requires 50000 clock cycles (computed using the C function *clock*) to determine the FFT of a data sequence. Hence the use of hardware FFT must reduce the processing time. However, due to poor synchronization during data transfer between the HPS and FPGA, the overall cycles needed for the completion of FFT transformation in hardware is higher than expected.

The transformed data are stored in the output FIFO until they are read by the HPS. For this reason, the number of cycles required by the HPS to read the transformed data sequence is increased to 40000 cycles. This count is still better than the number of cycles required by software FFT. Hence, implementing hardware FFT increases the processing speed by **20%**.

If more FPGA resources are available, multiple FFT cores can be implemented that would allow parallel computations, reducing the processing time even further. FPGA area poses as a limitation when FFT IP core is to be implemented on the selected processing platform.

Signal Tracking

The purpose of tracking is to accurately replicate the incoming signal frequency and phase that would allow accurate navigation data decoding. Following subsections discuss the different tests and results that are performed to verify the signal tracking.

Visual Inspection of In-Phase and Quadrature Signal

According to the signal structure, all the navigation data lies in the In-phase (I) arm of the GPS signal. When the incoming signal is being correctly tracked, all the navigation information and signal power appears in the I arm only and the Quadrature (Q) arm has only noise like properties. This exact trend is observed when I and Q components of PRN 8 are compared during tracking. As seen in the plot in Figure 45, signal magnitude in the Q arm is very low when compared to the I arm. The navigation data bit transitions are observed successfully in the same plot. These transitions are more distinct in Figure 46.

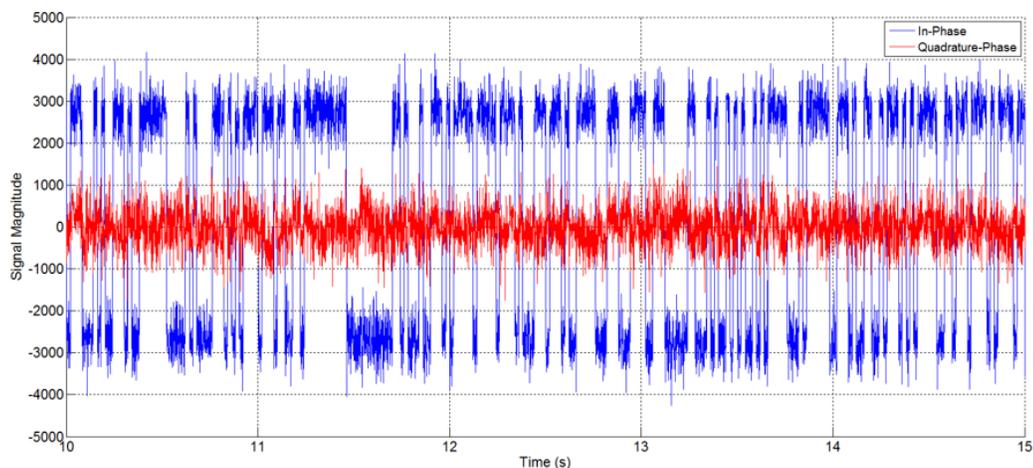


Figure 45: Comparison of I and Q components of the incoming signal

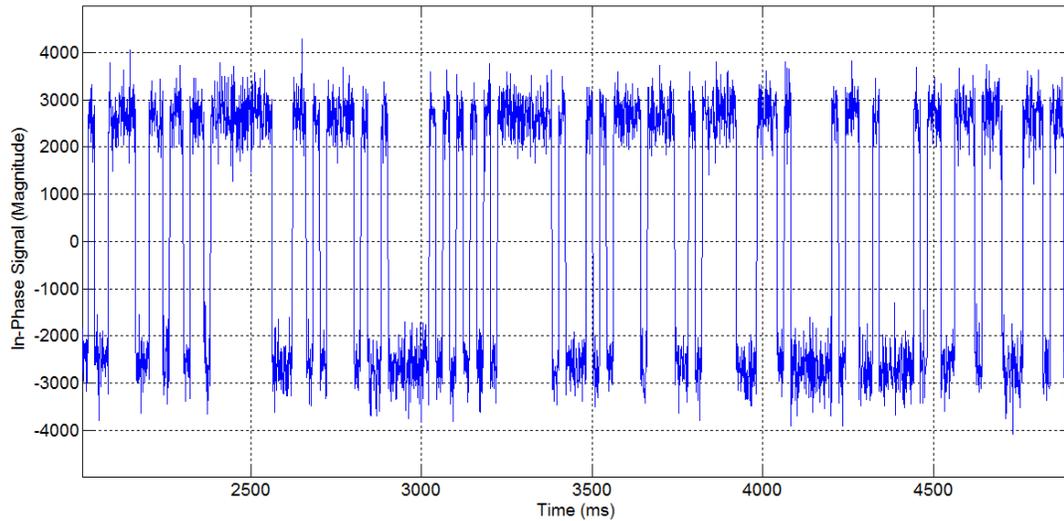


Figure 46: I component containing navigation data with bit transitions.

Code Phase Error

Code phase error is a measure of the phase difference between the incoming signal code and the locally generated code sequence. A Delay Lock Loop (DLL) is used to track the incoming signal and correctly replicate the code phase (as explained in Chapter 2). Within the DLL, the discriminator measures the error during code phase tracking. The discriminator output is plotted in Figure 47. As is seen, the filter requires 3-4 seconds to settle and converge to a minimum error value. On correct tracking, the error is maintained within the range of ± 0.5 chips/s and it remains stable for the rest of the tracking time.

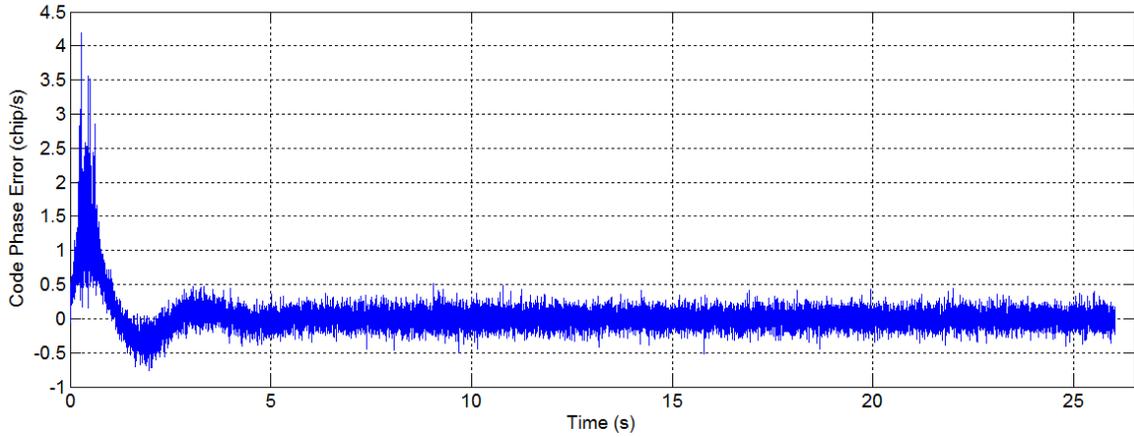


Figure 47: Discriminator output - code phase error varying over time.

Carrier Frequency Error

The second parameter to be tracked is the carrier frequency of the incoming signal. The frequency of the received signal has an offset from the expected value due to the Doppler Effect and is called as Doppler frequency. The Frequency Lock Loop (FLL) is responsible for tracking the incoming signal frequency correctly.

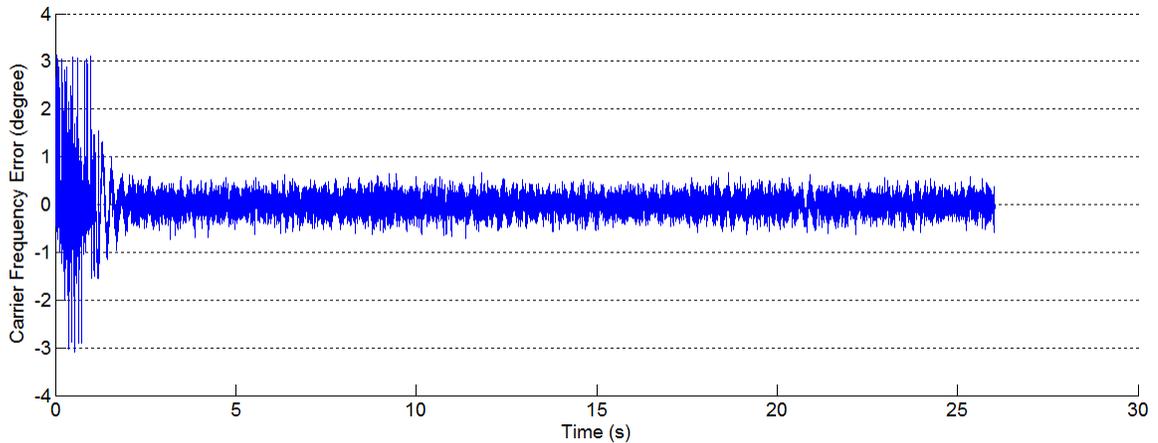


Figure 48: Carrier frequency error varying over time.

As discussed in Chapter 2, FLL also has a carrier discriminator that computes the frequency difference between the incoming signal and the locally generated signal. When the difference is zero, it can be concluded that the carrier frequency is being accurately tracked.

The output of this discriminator loop is plotted in Figure 48. As observed, FLL attempts to reduce the frequency error to zero and is successful at approximately 2 seconds into the sample data set. Errors are initially higher but still within an acceptable range. After the FLL settles down and locks onto the incoming signal frequency, the error is maintained within the range of ± 0.5 degrees and remains stable for the rest of the processing time.

Doppler Frequency

Due to Doppler effect there is also a continuous change in incoming signal frequency. The Doppler frequency of PRN 8 has been plotted in Figure 49. In the first couple of seconds, the estimated Doppler frequency drastically changes as the tracking loops still have not achieved signal lock. When the carrier frequency and code phase are correctly being tracked, the receiver is said to have achieved a successful signal lock.

As the carrier and code tracking loops converge and errors are a minimum, the Doppler frequency tracking gets smoother. Also observed is the gradual change in Doppler frequency corresponding to the satellite's motion in its orbit with respect to the receiver on Earth. As there are no drastic frequency changes after the filters settle down, it can be concluded that the tracking loops have had a successful signal lock for the rest of the processing time.

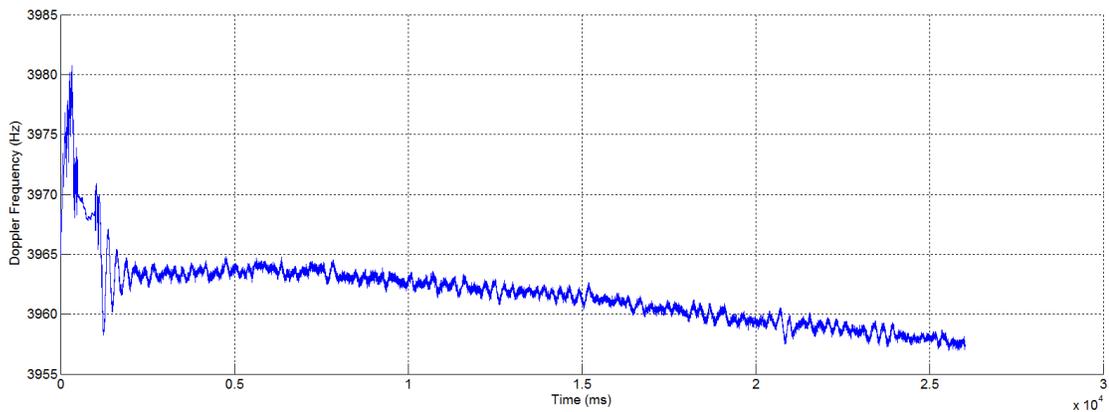


Figure 49: Doppler frequency for PRN 8 varying gradually over time.

Navigation Solution

It is important to note that accurate carrier frequency and code phase tracking is essential for decoding the navigation data. As discussed earlier, tracking loops perform well enough to make it possible to decode the navigation data. To verify the accuracy of the navigation data, preambles in the data frames are detected. Preambles are known bit sequences that appear at certain intervals within the data frame. On detecting preambles at expected intervals it is concluded that the navigation data are decoded correctly. Additionally, parity bits within the data frame are read to confirm the correct decoding of the navigation information.

Navigation solution algorithm has been discussed in Chapter 3 and the results are discussed in this section. The result obtained from the implemented algorithm is presented in Table 8. The position estimates provided by the developed receiver has been compared with the results obtained when the same data set is processed using the software receiver *fastgps*

(Gleason and Gebre-Egziabher, 2009). As observed, the errors are higher than expected and the quality of the estimates are rather poor.

Table 8: Position errors

| | Mean Error (m) | Standard Deviation (m) | RMS Error (m) |
|-------|----------------|------------------------|---------------|
| North | 6 | 7 | 9 |
| East | -121 | 6 | 121 |
| Up | -53 | 3 | 53 |

Unlike *fastgps*, when developing the navigation algorithm, tropospheric corrections, ionospheric corrections and other correction models have been excluded. The data set used for testing purposes is only 25 seconds long, whereas to read a complete GPS data frame at least 30 second long data set is required. Due to the shorter duration of the data set, only subframe number 1, 2, 3 were available and subframe number 4 and 5 were missing. Due to the absence of essential information, corrections are not applied for this data set. Since the corrections are not accounted for during the receiver position computation, it can be one explanation for higher position errors. It is also to be noted that *fastgps* uses ephemeris files for accurate satellite information whereas the developed receiver does not. This could be another reason for the low quality position estimates.

According to Montenbruck (2008), the DORIS tracking system onboard the Jason-1 uses DIODE real-time navigation function to achieve a position accuracy of 0.5 m. In another study (Fridman and Semenov, 2013), FPGA-based software receiver developed for terrestrial applications achieved position accuracy of 3 to 5 m. Considering the accuracies

achieved by these software receivers, navigation algorithm in the receiver developed in this research, requires further advancement.

Chapter Five: Conclusions And Recommendations

The focus of this thesis was to design and implement a software GNSS receiver for satellite application. As discussed in the previous chapters, development of such sophisticated software receivers has always been a technical challenge. However, with advancing technology, receiver designs have also evolved. This thesis provides a novel receiver design and implementation method and this chapter summarises the performance of various receiver processing blocks. Conclusions formed from the test results are followed by recommendations for future developments of FPGA-based software GNSS receivers.

Conclusions

In this research, a FPGA-based software GNSS receiver was implemented that is capable of computing the receiver position by processing the GPS L1 C/A-code signal. The developed receiver successfully determines the available satellites during acquisition and continues to track the incoming satellite signals. Altera's Cyclone V SoC containing both, FPGA and microprocessor, has been used for the receiver implementation. It also provides a low-cost processing platform of approximately \$300, while most receiver kits available in the market costs up to 75% more.

Implementation using SoC results in a unique task distribution between the two distinct hardware (FPGA) and software (microprocessor) components. Algorithm profiling tool

gprof was used to determine the bottleneck of the developed software functions. To relieve the microprocessor's computational load, such functions were then shifted to the FPGA. This reduced the total execution time, as processing in FPGA is highly parallel when compared to the microprocessor. Some of the objectives and performance goals accomplished are discussed in the following sections.

Receiver Performance

The fundamental function of a GNSS receiver is to process the incoming GNSS signals to compute entities like receiver position, velocity and time. The software GNSS receiver developed in this research serves the same purpose, to compute the receiver position. As discussed in Chapter 2, the ideal software GNSS receiver consists of 3 sequential tasks of acquisition, tracking and navigation solution, all performed using a software processor.

Tests were performed at each processing stage to analyse the collective performance of the receiver. The results obtained are compared with the reference software receiver fastgps (Gleason and Gebre-Egziabher, 2009) and the data files processed are provided by the same source. The accomplishments from the implemented receiver are:

- 1) In acquisition, the developed software receiver was capable of detecting all available satellites, i.e., 9 correctly. It also successfully determined the Doppler frequency offset of the corresponding satellite signals.
- 2) In the processing stage of tracking, multiple channels (POSIX threads) successfully tracked the acquired satellite signals in parallel. These threads also decoded

navigation messages from each of the incoming signals and computed pseudoranges simultaneously. During tracking, the carrier frequency error was within the range of -3 to 3 degrees initially and converged to a value within the range of -1 to 1 degrees within 2 seconds of processing. Similarly, during code phase tracking, the errors were in the range of -1 to 4 chips/second. As the tracking loop settled down, the error was again limited to a range of -1 to 1 chips/second. The loops remain stable and maintain lock on the various satellite signals for the complete duration of the dataset.

- 3) In the last processing step, receiver position was estimated using the basic least squares method. The result was then compared to position estimates computed by the reference software receiver *fastgps* (Gleason and Gebre-Egziabher, 2009). The position errors for North, East and Up were estimated as 6, -121, -51 metres, respectively. Software GNSS receivers typically exhibit higher position accuracy (m level) however, significant errors appear in the position estimates of the developed receiver as several atmospheric errors were not accounted for due to insufficient information.

Software/Hardware Partitioning Efficiency

The software receiver developed for this research purpose has been implemented as a C/C++ application using the microprocessor (ARM) provided by Altera's Cyclone 5 SoC. Using gprof profiling tool, it is determined that the mathematical function of FFT in C/C++, required 70% of the total execution time. Due to the higher execution time and increased

software processing load during FFT, it was decided to implement the FFT function in the FPGA instead. Altera's FFT IP core was implemented in the FPGA (hardware), while other functions were still processed in the microprocessor (software). By employing FFT components in the FPGA, the overall data processing speed of the receiver improved by 20%. This leads to the conclusion that task partitioning between hardware and software components improves the processing efficiency. Such an implementation is a novel approach to the developed software GNSS receiver.

The FFT routine in acquisition was completed using Altera's FFT IP core. By replacing software FFT with its hardware equivalent, following points are noted:

- 1) The input and output signal width of the FFT core was limited to 8 bits. Processed GPS signals can however have an amplitude range of -1500 to 1500. Since the FFT IP configuration used in the developed receiver does not support floating point integers, the input/output values can range from -128 to 127 only. All input values outside this range were truncated. This limitation lowered the accuracy and resolution of the FFT results.
- 2) As a workaround to the above stated problem, signal inputs to the FFT core were scaled. After scaling, all the input values ranged between the required values of -128 to 127. Scaling however has an adverse effect on the acquisition results as signal values were no longer accurate. When the hardware FFT was used, a total of 6 satellites were detected out of the actual 9 that are detected using software FFT.

For receiver position estimates, a minimum of 4 satellites are required. Hence this method can still be used to compute receiver position.

Power Requirements

A software GNSS receiver being developed for satellite applications, must have low power requirements pertaining to the limited resources available on board a satellite. The chosen processing platform, Altera's Cyclone V SoC, has relatively low power consumption when compared to processors used by other researchers (Table 4). The SoC platform requires power in the range of 2-5 W. Since the SoC chip is part of a standard development board, the power consumption of the processor can be further reduced by discarding the unnecessary board peripherals.

In summary, a software GNSS receiver for satellite applications has been developed using FPGA and a microprocessor. The microprocessor is a dual-core ARM processor that operates at a frequency of 925 MHz. Power consumption of the processing platform is 2 to 5 W which is relatively low when compared to other similar receivers (see Table 4). The SoC (manufactured by Altera) used in this research is not radiation resistant but for future versions of the receiver a radiation-hardened processor will be considered. The developed software receiver successfully detects and tracks multiple GPS satellites and computes a navigation solution. It has multiple tracking channels and can track up to 12 satellites at any given time. POSIX threads has been implemented to facilitate the parallel tracking feature. The receiver components implemented in the FPGA occupies 40% of the total FPGA area.

Recommendations for Future Work

This research demonstrates that by using a unique combination of hardware and software components on a modern processor such as SoC, an efficient software receiver design can be implemented. As more advanced processors become available, receiver architectures for satellite applications will become more diverse and flexible in terms of hardware/software co-design. Even though the implementation of FFT function in the FPGA has resulted in an increase in the processing speed, it is to be further improved by considering the recommendations provided for future FPGA-based software GNSS receiver implementation.

This research provides the foundation for future functionalities that are to be added. In the long-term, this research aims to implement not only the FFT function but all the remaining software algorithms in the FPGA. Recommendations for future work would also entail implementation of the receiver design using radiation-hardened components. Current receiver also provides a base for features like radio occultation, reflectometry and precise orbit determination that can be added in the future versions.

Sufficient FPGA Resources

As discussed in Chapter 2, System-on-Chip (SoC) is a suitable processing platform for receiver implementation as it accommodates both FPGA and a microprocessor on the same chip. Slow processes in the software can be replaced with its hardware equivalent to increase the processing speed. Ideally, it is desirable to implement the complete receiver in the FPGA component of the SoC however, the memory bits offered by the FPGA is not

sufficient to accommodate the entire receiver design and poses as a technical challenge. In this study, Altera's Cyclone V SoC is capable of accommodating only one FFT core out of the desired three cores. In future implementations, FPGA offering higher memory bits must be considered. For example, Arria SoC, a more recent processor manufactured by Altera has 10 times more memory bits than Cyclone SoC (Altera, 2014c). Using such platform would allow the user to implement more hardware components in the FPGA. Using multiple FPGA (external) chips is also an alternative.

Improve Positioning Algorithm

In the current receiver implementation, the computed position estimates have poor quality. The results can be improved if the basic algorithm can be replaced with advanced positioning algorithm. The future implementation must also include various correction models to nullify errors caused by ionosphere, troposphere, etc. These additions will improve the quality of navigation solution.

Multi-Constellation Receiver

Software receiver developed in this research is capable of processing only GPS L1 C/A-code signal. However, it would be highly beneficial if the receiver can process signals pertaining to different frequencies and constellations including the underlying carrier phase measurements. This feature would allow the same receiver to be used for multiple satellite applications like reflectometry, radio occultation and orbit determination.

Modular Design

GNSS receivers are used for multiple satellite applications; however, since each application requires unique signal processing, several GNSS receivers will be required. Use of multiple GNSS receivers can be avoided if one software GNSS receiver can switch between a set of predetermined applications. Since a software GNSS receiver exhibits design flexibility, the common GNSS receiver can be programmed or reconfigured for multiple applications. Implementation of a software GNSS receiver would hence reduce the number of overall processors required and would also occupy less space on-board the satellite.

References

Akos, D. M., Normark, Enge, Hansson, and Rosenlind. (2001), *Real-Time GPS Software Radio Receiver*, Proceedings of the 2001 National Technical Meeting of the Institute of Navigation. pp. 809-816.

Akos, D. M. (1997) *A Software Radio Approach to Global Navigation Satellite System Receiver Design*. Doctoral dissertation, Ohio University, pp. 85-86.

Altera (2014a), *SoC Embedded Design Suite User Guide*, accessed March, 2014.
http://www.altera.com/literature/ug/ug_soc_edu.pdf.

Altera (2014b) *FFT MegaCore Function: User Guide*, accessed August, 2014.
http://www.altera.com/literature/ug/ug_fft.pdf.

Avanzi, A. and Tortora. (2010), *Design and Implementation of a New Spaceborne FPGA-Based Dual Frequency GPS and Galileo Software Defined Receiver*, Satellite Navigation Technologies and European Workshop on GNSS Signals and Signal Processing (NAVITEC), 2010 5th ESA Workshop. pp. 1-7.

Baracchi-Frei, M. (2010), *Real-Time GNSS Software Receiver Optimized for General Purpose Microprocessors*. Doctoral Dissertation, University of Neuchâtel, pp. 60.

Bastide, F., Julien, Macabiau, and Roturier. (2002), *Study of Acquisition, Tracking and Data Demodulation Thresholds*, ION GPS 2002, pp. 1-11.

Ben Salem, A., Ben Othman, and Ben Saoud. (2008), *Hard and Soft-Core Implementation of Embedded Control Application using RTOS*, Industrial Electronics (ISIE), 2008. IEEE International Symposium, pp. 1896-1901.

Biarrri GPS Receiver Project (2013), Australian Centre for Space Engineering Research, accessed October, 2014. <http://www.acser.unsw.edu.au/biarrri/index.html>.

Borgerding, M. (2008) *Kissfft* Source Forge, accessed January, 2013. <http://sourceforge.net/projects/kissfft>.

Borre, K., Akos, Bertelsen, Rinder, and Jensen. (2007), *A Software-Defined GPS and Galileo Receiver: A Single-Frequency Approach*, Springer, Boston, pp. 75-98.

Butenhof, D. R. (1997) *Programming with POSIX Threads*, Addison-Wesley Professional. pp. 20-30.

Choudhury, M., Cheong, Wu, Shivaramaiah, and Dempster. (2013), *Initial Test Results of Namuru Dual-GNSS Space-Borne Receiver*, International Global Navigation Satellite Systems Society, Outrigger Gold Coast, Qld Australia, 16-18 July, 2013, pp. 1-8.

Dovis, F., Spelat, Mulassano, and Leone. (2005), *On the Tracking Performance of a Galileo/GPS Receiver Based on Hybrid FPGA/DSP Board*, Proceedings of the 18th International Technical Meeting of the Satellite Division of the Institute of Navigation (ION GNSS 2005). pp. 1611-1620.

Enge, F., Mumford, Parkinson, Rizos, and Heiser. (2004), *An Open GNSS Receiver Platform Architecture*, Journal of Global Positioning Systems Scientific Research Publishing. Vol. 3, pp. 63-69.

Esterhuizen, S. (2006), *The Design, Construction, and Testing of a Modular GPS Bistatic Radar Software Receiver for Small Platforms*, Master's Dissertation, Department of Electrical and Computer Engineering: University of Colorado, pp. 25-31.

Fenlason, J. and Stallman. (1997) *GNU Gprof: The GNU Profiler*, Manual, Free Software Foundation Inc., accessed October, 2013.

<http://www.ecoscentric.com/ecospro/doc/html/gnutools/share/doc/gprof.pdf>.

Fridman, A. and Semenov. (2013), *System-on-Chip FPGA-Based GNSS Receiver*, East-West Design & Test Symposium, 2013. IEEE. pp. 1-7.

Friigo, M. and Johnson. (1998), *FFTW: An Adaptive Software Architecture for the FFT*, Acoustics, Speech and Signal Processing. Proceedings of the 1998 IEEE International Conference. Seattle, WA., Vol. 3 pp. 1381-1384.

Ganguly, S. (2004), *Real-Time Dual Frequency Software Receiver*, Position Location and Navigation Symposium, PLANS 2004, IEEE pp. 366-374.

Gleason, S. and Gebre-Egziabher. (2009) *GNSS Applications and Methods*, Artech House. Norwood, Massachusetts, pp. 121-145.

Glennon, É., Parkinson, Mumford, Shivaramaiah, Li and Jiao. (2011), *A GPS Receiver Designed for Cubesat Operations*, Australian Space Science Conference, pp. 26-29.

Glennon, Gauthier, Choudhury, Parkinson, and Dempster. (2013), *Project Biarri and the Namuru V3. 2 Spaceborne GPS Receiver*, International Global Navigation Satellite Systems Society, IGNSS Symposium, Citeseer.

GNSS SDR Front End and Receiver (2010) Nottingham Scientific Ltd., accessed September, 2013. <http://www.nsl.eu.com/primo.html>.

Hein, G., Pany, Wallner, and Won. (2006), *Platforms for a Future GNSS Receiver*, Inside GNSS, Vol. 1, pp. 56-62.

Hershberger, J. (2013), *Real-Time Software Defined GPS Receiver*, Real-time software defined GPS receiver. Doctoral Dissertation, Purdue University.

HPS-FPGA Bridges Cyclone V Hard Processor System Technical Reference Manual, accessed July, 2014. http://www.altera.com/literature/hb/cyclone-v/cv_5v4.pdf. Vol. 2014

Humphreys, T. E., Bhatti, Pany, Ledvina, and O'Hanlon. (2009), *Exploiting Multicore Technology in Software-Defined GNSS Receivers*, Proceedings of the 22nd International Technical Meeting of the Satellite Division of the Institute of Navigation. pp. 326-338.

Humphreys, T. E., Psiaki, Kintner Jr, and Ledvina. (2006), *GNSS Receiver Implementation on a DSP: Status, Challenges, and Prospects*, Proceedings of ION GNSS 2006. Fort Worth, Texas, pp. 2-9.

Kaplan, E. D. and Hegarty. (2006) *Understanding GPS: Principles and Applications*. Second ed. Artech house, Boston.

Ledvina, B., Powell, Kintner, and Psiaki. (2003), *A 12-Channel Real-Time GPS L1 Software Receiver*, Proc. Institute of Navigation National Technical Meeting. pp. 22-24.

Ma, C., Lachapelle, and Cannon. (2004), *Implementation of a Software GPS Receiver*, Proceedings of ION GNSS. Long Beach, California, pp. 21-24.

MacGougan, G., Normark, and Stahlberg. (2005), *Innovation-Satellite Navigation Evolution-the Software GNSS Receiver*, GPS World Eugene, OR: Aster Pub. Corp., 1990. Vol. 16 pp. 48-55.

Montenbruck, O., Garcia-Fernandez, and Williams. (2006), *Performance Comparison of Semicodeless GPS Receivers for LEO Satellites*, GPS Solutions, Vol. 10, pp. 249-261.

Montenbruck, O., Markgraf, Garcia-Fernandez, and Helm. (2008), *GPS for Microsatellites—Status and Perspectives*, Small Satellites for Earth Observation. Springer pp. 165-174.

GPS Navstar (1995) *Global Positioning System Standard Positioning Service Signal Specification*, 2nd Edition, June 2nd 1995, pp. 51.

Parkinson, K., Mumford, Glennon, Shivaramaiah, Dempster, and Rizos. (2011), *A Low Cost Namuru V3 Receiver for Spacecraft Operations*, IGNSS Symposium, pp. 15-17.

Petovello, M. G., O’Driscoll, Lachapelle, Borio, and Murtaza. (2008), *Architecture and Benefits of an Advanced GNSS Software Receiver*, Journal of Global Positioning Systems. Vol. 7, pp. 156-168.

Ramesh, Bharat, Anand and Selvan. (2014), *Analysis of Lossy Hyperspectral Image Compression Techniques*, International Journal of Computer Science and Mobile Computing. Vol. 3, pp. 302.

SiGe GN3S Sampler v3 Sparkfun Electronics, accessed July, 2013. <https://www.sparkfun.com/products/10981>.

Tsui, J. B. (2000), *Fundamentals of Global Positioning System Receivers-A Software Approach*, A John Wiley & Sons, Inc. Publication. Vol. 2, pp. 140.

Weber, J. and Chin. (2006), *Using FPGAs with Embedded Processors for Complete Hardware and Software Systems*, Beam Instrumentation Workshop 2006 (AIP Conference Proceedings Volume 86). pp. 187-192.

Won, J., Pany, and Hein. (2006), *GNSS Software Defined Radio*, Inside GNSS. Vol. 1, pp. 48-56.

Zavorotny, V. U. and Voronovich. (2000), *Scattering of GPS Signals from the Ocean with Wind Remote Sensing Application*, IEEE Transactions on Geoscience and Remote Sensing, Vol. 38, pp. 951-964.