Global Positioning System Signal Acquisition and Tracking

Using Field Programmable Gate Arrays

A Dissertation Presented to the Faculty of the

Fritz J. and Dolores H. Russ

College of Engineering and Technology

Ohio University

In Partial Fulfillment

of the Requirements for the Degree

Doctor of Philosophy

by

Abdulqadir A. Alaqeeli

November, 2002

THIS DISSERTATION ENTITLED

“Global Positioning System Signal Acquisition and Tracking Using Field Programmable Gate Arrays”

by Abdulqadir Alaqeeli

has been approved

for the School of Electrical Engineering and Computer Science
and the Russ College of Engineering and Technology

Janusz Starzyk, Professor
School of Electrical Engineering and Computer Science

Dennis Irwin, Dean
Fritz J. and Dolores H. Russ
College of Engineering and Technology

Acknowledgement

In the Name of Allah, the Most Beneficent, the Most Merciful. All praise and thanks to Allah, Lord of the universe and all that exists. Prayers and peace be upon His prophet Mohammed, the last messenger for all humankind.

First, I thank Allah for His infinite blesses. Allah blessed me with loving parents, brothers, sisters, relatives, and friends. Allah's guidance made the completion of this work possible.

I want to thank my parents for their love, education, support, encouragement, and prayers. Thanks goes to my wife and my daughter. Their support gave me spiritual strength in pursuing my ambition and staying in the United States. Thanks also goes to my brothers and sisters. Life has no taste to me without my parents and my family.

I would like to thank my advisor, Prof. Janusz Starzyk, for his constant patience, great assistance, and guidance during my Masters and Ph.D. degrees. Prof. Starzyk was an unusual advisor who helped me in the academic and the personal matters. His way of teaching and problem solving changed me from an ordinary student to a knowledge seeker.

Thanks goes to my other committee members, Dr. Frank van Graas, Dr. Jeffrey Dill, Dr. Michael Braasch, and Dr. Nicolai Pavel. This committee is a blessing from Allah. Their advice and support helped me in solving many problems during this work. I would like to say again "Thanks" for your precious time, valuable help

and support. Special thanks goes to Prof. Frank van Graas who acted as a co-advisor for me during the last year. Thanks also goes to Dr. Maarten Uijt de Haag. His friendship, advice, and discussions were very helpful to me.

Dr. Ahmad Alsolaim, my dear friend, cannot be forgotten. He helped me in all of my courses. He was my main supporter during the difficult days. He was a real brother who has given me advices since I came to the US in 1995. “Thank you Ahmad” is not enough for such a close friend.

I am very thankful to all of my friends in the VLSI and the Software Radio research groups for their help, discussions, and advice. I especially thank Sanjeev Gunawardena, Jing Pang, and Zhu Zhen for their support. In addition to their helpful discussions, they provided me with useful Matlab codes and synthesizable VHDL codes.

I would like to thank my friends Dr. Saleh Aloteawi and Dr. Mohammed Alsharekh. Their friendship was helpful for me during my study. “Thank You” goes to my friends Hamad Albrethin, Abdulrahman Alsebail, Mazyad Almuhaileb, Saleh Alawirdy, Mohammed Altamimi, Hamed Alsharari and Ahmad Alahmadi.

Finally, I want to thank all friends who assisted me upon request and were very helpful in their suggestions.

Table of Contents

List of Tables

List of Figures

List of Abbreviations

ADC:	Analog to Digital Converter
ASIC:	Application Specific Integrated Circuits
BPSK:	Binary Phase Shift Keying
C/A:	Coarse Acquisition
CDMA:	Code Division Multiple Access
CORDIC:	Coordinate Rotation Digital Computer
DLL:	Delay Lock Loop
DSP:	Digital Signal Processor
DS-SS:	Direct Sequence-Spread Spectrum
EPL:	Early Prompt Late
FFT:	Fast Fourier Transform
FIR:	Finite Impulse Response
FLL:	Frequency Lock Loop
FNT:	Fermat Number Transform
FPGA:	Field Programmable Gate Array
GPS:	Global Positioning System
IFFT:	Inverse Fast Fourier Transform
IP:	Intellectual Propriety
LFSR:	Linear Feedback Shift Register
NCO:	Numerically Controlled Oscillator

NTT:	Number Theoretic Transform
PLL:	Phase Lock Loop
PN:	Pseudo Noise
PPS:	Precise Positioning Service
PRN:	Pseudo Random Noise
PSK:	Phase Shift Keying
RF:	Radio Frequency
SMS:	Single Memory Setup
SNR:	Signal to Noise Ratio
SPS:	Standard Positioning Service
TOA:	Time of Arrival
VHDL:	VHSIC Hardware Description Language
VHSIC:	Very High Speed Integrated Circuit
VLSI:	Very Large Scale Integration

Chapter 1

Introduction

With the global positioning system (GPS) a user can get accurate positioning information at any location on Earth (French, 1996). GPS can accurately guide airplanes by providing navigational information at any stage of takeoff, flight, and landing (Kayton, 1997). GPS receivers perform many signal processing steps to synchronize the received GPS signal with a local code to enhance the positioning accuracy (Braasch, 1999). The time required to acquire satellite signals is the main problem in most GPS receivers; therefore, many researchers have investigated different designs of such receivers.

Many algorithms for acquisition of GPS signals have been developed and evaluated (Lin, 2000). Until now, the developed algorithms were not fast enough to acquire GPS signals in real time (Tsui, 2000 and Molyneux, 2002). The slow acquisition process is due to many reasons, one of them is the large computation cost of the circular correlation. Computing the correlation function in the time-domain is a very time-consuming process (Braasch, 1999). Performing the correlation of two N-point sequences using frequency domain multiplication reduces the

calculations N/logN times, which is a significant reduction of the computation cost and facilitates real-time GPS signal acquisition (Van Nee, 1991).

However, direct implementation of fast Fourier transform (FFT) based circular correlation on digital signal processor (DSP) or microprocessor does not satisfy the requirements of real-time acquisition. The computation time is very long due to the slow calculation of the required FFT. The processing technique in a DSP or a microprocessor uses sequential executions of all the operations. This software-based technique processes all the multiplications of the FFT one at a time, which is not appropriate for real-time applications.

The speed of finding the correlation is affected, not by only the processing technique but by the selected correlation method as well. The FFT based correlation method requires approximately 3NlogN complex multiplications and 3NlogN complex additions (Smith W., 1995). These complex operations are time consuming and need to be simplified. A simpler correlation algorithm is preferable for fast GPS signal acquisition. In addition, parallel operations may reduce the total acquisition time making real-time processing feasible. A parallel processing implementation of signal acquisition is possible using a field programmable gate array (FPGA).

With fast acquisition, the use of software-based GPS receivers can be extended to additional real-time applications. Providing these information in real time will lead to major advancement in many civilian and military applications.

The main challenging problem is to build a GPS receiver that has a simple, fast, and non-computationally extensive algorithm which satisfies real-time requirements.

The purpose of this dissertation is to tackle this problem. Specifically, algorithms and field programmable gate array (FPGA) based architectures are developed for real-time acquisition and tracking of GPS signals. This dissertation uses the parallel processing concept supported in the FPGA to replace the current sequential processing platform. Also, this work investigates many algorithms to reduce the computations required by the correlation function in order to shorten the acquisition time. One algorithm uses an averaging correlation for GPS signal acquisition that simplifies the implementation and reduces the required computations. This method is implemented in FPGA to benefit from the parallel processing of this technology. The implemented architecture solves the presented problem of the GPS receivers by significantly reducing the acquisition time. The performance of this implementation enables real-time acquisition of one satellite in less than one millisecond. Comparing this performance with the performance of the current GPS receivers where the acquisition time is more than 1 second, the developed architecture minimizes the acquisition time 1000 times. The presented solution is very important since it extends the use of GPS to both civilian and military applications. As a result, precise and stable navigational systems are achievable.

Chapter 2 presents a necessary and brief background to provide enough information to understand the problem. First, a brief description of a code division multiple access (CDMA) system is provided. Then the global positioning system is described. The GPS signal structure is also presented. The GPS receiver architec-

ture is presented along with its main processes. The main processes of the GPS receiver, signal tracking and acquisition, are explained. Different acquisition search techniques are also presented along with their advantages and disadvantages.

Chapter 3 presents two algorithms to replace the necessity of the FFT computations to calculate the correlation function. The presented algorithms, use transforms that require simpler operations than the FFT. These transforms are the Fermat number transform (FNT) and the Walsh Hadamard transform (WHT). Descriptions of both algorithms are presented with examples to show how the correlation functions are computed. The possibility of using these algorithms in the GPS signal acquisition is discussed.

Chapter 4 describes an averaging correlation method that re-sizes the 5000-point correlation function to the size of the C/A code, which is 1023-points. The averaging correlation method is inspected and compared against the regular correlation method. The effects of the averaging method on detection and misdetection probabilities are also provided. The limitation of implementing the averaging method is explained. Different algorithms were suggested to ease the implementation of this method in an FPGA. One fast and easy-to-implement algorithm for averaging correlation is developed and explained in detail. The algorithm is called modified-code averaging correlation based GPS block processing. The advantages and the limitations of using the developed algorithm for block processing of the GPS signals are also presented in this chapter.

Chapter 5 provides the implementation of the developed algorithm. Each component implementation is described. Circuits for those components are pro-

vided. The whole system was partitioned and implemented to fit into a small FPGA. The size, speed, and accuracy of the implemented circuits are verified for real-time applications. Full design analysis is also provided in Chapter 5.

Chapter 6 presents a summary, conclusion, description of future work and recommendations. First, the summary of this work is presented. It briefly describes the problem, the developed solutions and the performance. The conclusion is presented next. The accomplished improvements and advantages in the acquisition and the tracking implementations are stated. Possible future work and recommendations are then suggested.

The FPGA platform used during the design development and testing is described in Appendix A. The Matlab, VHDL, and C codes that were developed during this work are presented in Appendices B, C, and D. The FPGA layout (of the mapped designs) are shown in Appendix E.

Chapter 2

Background

2.1 Introduction

Chapter 2 presents an overview of the Global Positioning System (GPS). However, an important related issue, the telecommunication techniques involved with signal processing, needs to be discussed prior to the presentation of the GPS. These techniques include a description of the carrier signal, navigation data, satellite codes, and modulation techniques. Already existing communication systems that are useful for GPS transmission are presented first. The GPS system and its signal structure are discussed next. The final section is an overview of the main GPS signal processes.

2.2 Spread Spectrum and Code Division Multiple Access

A spread spectrum communication system uses larger frequency bandwidth than is needed to transmit information. Therefore, the transmission bandwidth is much larger than the information bandwidth (Peterson, 1995). The transmission bandwidth is found by using a spreading signal, which is independent of the information data. Military applications have used the spread spectrum techniques for

more than fifty years because it offers a system that rejects intentional and unintentional interference (Gibson, 1993). Additionally, a spread spectrum delivers multi-user random access and a high resolution range. These traits were a catalyst for commercial applications switching to spread spectrum communication techniques in the last decade. These applications include mobile radio applications, satellite communications, and positioning systems. The most widely used spread spectrum is the direct sequence spread spectrum (DS-SS) (Dixon, 1994). DS-SS is achieved by multiplying a radio frequency (RF) carrier and a pseudo-random noise code (PRN code). Every user or transmitter uses a different code which is orthogonal to the codes of the other users or transmitters. This method is called the Code Division Multiple Access (CDMA) (Hassan, 1998).

In the transmitter, a PRN code is modulated with the information using phase shift keying (PSK) techniques. The PRN-modulated-information signal is then mixed with the carrier. Therefore, the RF signal is substituted with a wide bandwidth signal with the spectrum equivalent to a noise signal. The demodulation process is then simply carried out by multiplying a local copy of the PRN modulated carrier with the incoming signal. The local copy of the code and the carrier must be synchronized in order to demodulate the signal (Cook, 1983). A peak is achieved when the two signals are aligned. The correlated signal is then filtered and sent to a PSK demodulator.

A simple form of direct sequence spread spectrum (DS-SS) uses binary phase shift keying (BPSK). BPSK modulation changes the carrier phase by 180 degrees if the PRN code chip is -1. Otherwise, the carrier phase is not changed. More details pertaining to these techniques can be found in (Proakis, 1995).

2.3 Global Positioning System

A Global Positioning System (GPS) is a satellite-based system. It uses the concept of one-way time-of-arrival (TOA) ranging. The range is computed by multiplying the speed of light by the amount of delay that a GPS signal needs to travel from the satellite (or the transmitter) to the user (or the receiver) (Braasch, 1999). Calculating a three-dimensional position requires utilizing three satellites. An additional satellite is necessary to solve for the receiver clock bias (Misra, 2001). Since the US military is considered a global force, the GPS system is designed to provide accurate positioning information anywhere in the world 24 hours a day. The GPS system uses 24 satellites that are placed in six orbital planes with four satellites in each plane (Kaplan, 1996 and Parkinson, 1996).

The transmitter in each satellite sends the navigation data along with ranging codes using the CDMA scheme. All GPS satellites use two frequencies for navigation purposes, known as L1 and L2. L1, with a carrier frequency of 1575.42 MHz, is used for both civilian (standard) and military (precise) positioning services, SPS and PPS respectively. Whereas L2, which uses 1227.6 MHz signal, is primarily used for military service (Kaplan, 1996 and Parkinson, 1996). L2 is used in 90% of all surveying receivers (civilians). Civilians have access to the L2 carrier, but not to the encrypted Y-code.

2.4 GPS Signal Structure

Each GPS satellite has a unique PRN code that is orthogonal to the codes of other satellites. This code is called the coarse acquisition code (C/A code). The C/A

code has a rate of 1.023 MChips/s with a code period of 1ms. The navigation data is binary and has a rate of 50bits/s. This data is sent using a direct sequence spread spectrum and CDMA techniques. The navigation data is bi-phase shift keyed onto the carrier signal and the C/A code is also bi-phase shift keyed onto the resulting signal. A simplified model for the transmitted signal is written as:

$$S_i(t) = A_i C_i(t) D(t) sin(2\pi f t + \phi_O) \quad \textbf{2-1.}$$

where, A_i is the amplitude of the signal.

$C_i(t)$ is the C/A code for satellite number i

$D(t)$ is the navigational data

f is the L1 carrier frequency which is 1575.42 MHz

The C/A code is a gold-code type, which has noise-like auto-correlation and cross correlation characteristics. Therefore, identical codes at the exact same chip phase are necessary to produce the maximum correlation peak, otherwise they are not correlated. Each satellite's unique C/A code is used to distinguish the satellite signals from each other. The C/A code phase of the received satellite signal provides important ranging information. To calculate the range from the satellite to the receiver, one can multiply the code phase difference (which is the time a signal spends between satellite and receiver) by the speed of light (Braasch, 1999 and Kaplan, 1996).

At the receiver, the signal changes due to the Doppler effect and noise. A simplified model of a received GPS SPS signal is:

where $R_i(t)$ is the received GPS SPS signal

$$R_i(t) = A_i C_i(t + \Delta t) D(t + \Delta t) sin(2\pi(f + \Delta f)t + \Delta\phi) + n(t) \quad \textbf{2-2.}$$

Δt is the time delay (the time required for the signal to travel to the receiver)

Δf is the carrier frequency offset (due to doppler and LO frequency uncertainties).

$\Delta\phi$ is the carrier phase offset. (due to oscillator, sampler, and hardware)

n(t) is a white noise.

For a more accurate model of the received signal read (Kaplan, 1996 and Parkinson, 1996).

2.5 GPS Receiver Architecture

The design of a digital GPS receiver is divided into two parts. One part is the front-end component which consists of an antenna, filters, amplifiers, and a down conversion step. The signal is then digitized by an analog to a digital converter (ADC). After the signal is digitized, digital signal processing is performed in the second part of the receiver. The second part of a GPS receiver is the base-band processor of the digitized GPS signal (Tsui, 2000). The base-band processor is responsible for many signal processing steps that include, but are not limited to, PRN code synchronization, demodulation, and range calculation. Figure 2-1 shows a block diagram of a GPS receiver. This research will only investigate the base-band processing aspect of the GPS software-radio. Thus, the various designs of the front-end will not be discussed in this dissertation. However, a novel implemented design of a GPS receiver front-end can be found in (Akos, 1997 and Akos, 1996).

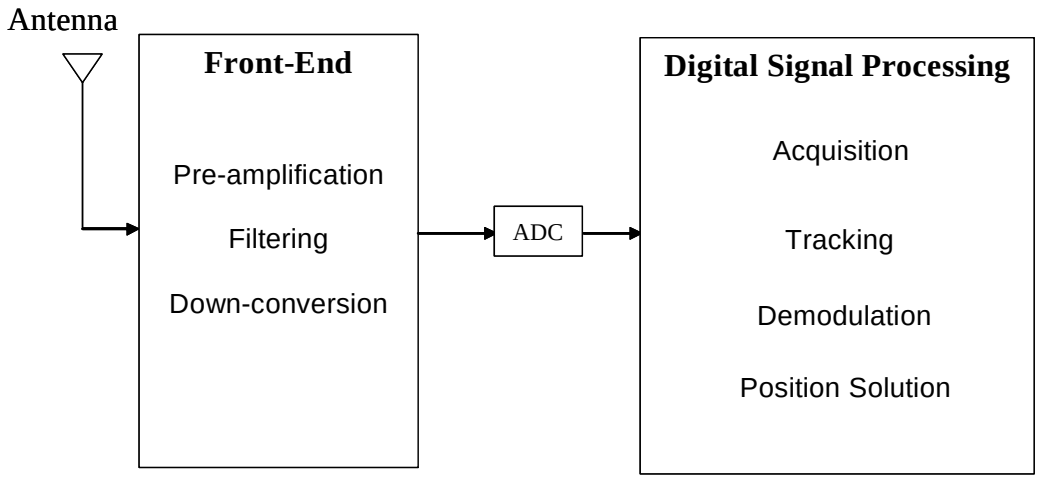

Figure 2-1: Block Diagram of a GPS Receiver

The base-band still needs major advances and new algorithms to speed up the position calculation for real-time applications. Such applications are currently under intense research to determine how much integration with the available GPS services can be accomplished. Navigational systems will benefit from being able to have high speed calculations and quick updates regarding their position and direction.

Therefore, a description of the major and necessary base-band processes of a GPS receiver is presented next. The most important and critical signal processing steps are the acquisition and the tracking processes. Acquisition is sometimes referred to as a search or detection process. It searches for the satellite code and for its code phase and the carrier frequency offset. Whereas, tracking is sometimes referred to as synchronization. Synchronization of the code and the carrier are necessary for a GPS receiver to be able to read the navigation data (Braasch, 1999).

Moreover, acquisition and tracking processes provide information about detected satellites, pseudo-ranges, and precise carrier frequencies. The position computation becomes a straight forward problem in these cases.

2.5.1 Signal Tracking

As was previously stated, tracking needs acquisition information to be able to start functioning. Signal tracking is the process that a receiver does all the time to synchronize or lock-in the GPS signal (Tsui, 2000). Therefore, a description of this process is presented before the description of the acquisition process. If the acquisition estimations are available, tracking processes (or loops) synchronize the local generated code and carrier with the received signal. Tracking loops then remove the carrier and the code to read the navigational data. They also provide critical timing information used in the position solution.

One of the tracking loops is the code tracking loop. The code tracking loop is important due to its role in the range calculation. This loop generates a synchronized copy of the C/A code to remove the spread spectrum modulation. A delay-lock-loop (DLL) can be used for code tracking. Figure 2-2 shows a code tracking loop using a DLL architecture.

The other tracking loop is the carrier tracking loop. This loop tracks either the phase or the frequency of the incoming signal. It is also responsible for decoding the data encoded on the carrier. Frequency and/or phase lock loops are used for carrier tracking (Uijt, 1998). A FLL-based carrier tracking loop is shown in Figure 2-3.

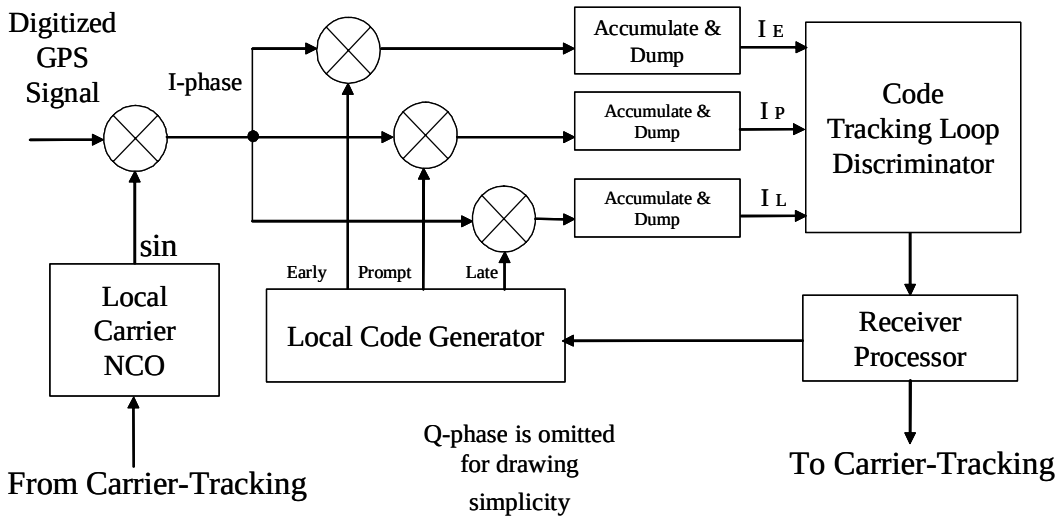

Figure 2-2: Code Tracking Using Delay-Lock-Loop (DLL)

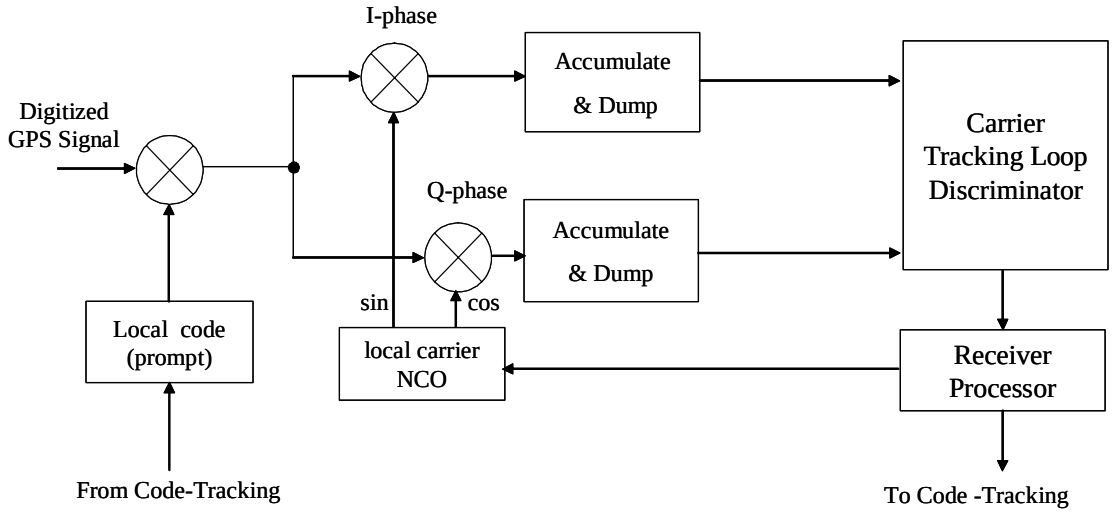

Figure 2-3: Carrier Tracking Using Frequency-Lock-Loop (FLL)

A code-tracking loop requires a precise carrier frequency or phase. Whereas, a carrier-tracking loop requires precise code phase estimation. Therefore, the two tracking loops are coupled in (Akos, 1997 and Uijt, 1998). These techniques are well known and have been studied extensively. These tracking loops were used in GPS receivers. Their characteristics and performances were presented in detail in (Braasch, 1999 and Uijt, 1998).

Tracking loops require initial estimations of the code phase and the carrier frequency. This is usually the task of the acquisition process. Also, when a tracking loop loses lock at any time, a re-acquisition process is required. The next section presents the acquisition process and some of its search methods.

2.5.2 Signal Acquisition

In a CDMA system, when the received signal is multiplied by a synchronized version of the PRN code, the signal is despread. Its power increases over the noise floor. Therefore it appears as a correlation peak. However, in order for a receiver to synchronize to the received signal, an initial estimation of the code and carrier is calculated. The acquisition process is the first process required by the GPS receiver (Kaplan, 1996). The acquisition process conducts a three dimensional search. The three elements (or search bins) are the available C/A code, the code phase, and the carrier frequency offset. If we assume that a GPS receiver knows which satellite code it is searching for, then a 2-D search is required (Ward, 1996).

One dimension is the code phase in range of 1,023 chips. The code phase resolution is half of a code chip. This resolution is required since the correlation peak is considered a true peak only if the code phase is within a half chip. The second

search dimension is for the carrier frequency. Its range is ±10kHz centered at an intermediate frequency (IF) of 1.25MHz. The resolution of the carrier frequency is typically 667Hz for 1-ms integration (Ward, 1996). Searching with 500Hz steps can be used (Uijt, 1998).

Therefore, in a GPS receiver, this search detects the correlation peak and compares it to a certain threshold to determine whether a satellite was detected or not. When a satellite is detected the auto-correlation result provides a rough estimation of the code phase and the carrier frequency. The acquisition process should provide this data. However, if the search process does not locate a peak that passed the detection threshold, a satellite is considered "not-acquired" and the search continues.

These searches are very slow especially when the GPS receiver does not know anything about its last position and the satellites' orbits. When this occurs, the receiver searches all of the satellites in a certain order. This procedure is called a "cold start." Whereas, if the receiver has knowledge regarding a previous position and an approximation of the satellites' orbits, this data aids the search process. This is known as a "warm start." Many different search algorithms were investigated and tested in (Lin, 2000 and Akos, 1997).

A serial search can be done by evaluating each unknown until a correct combination of the parameters is achieved. If there is no correct combination of carrier frequency and code phase, the receiver searches for a different PRN code. Figure 2-4 shows a simplified diagram of the serial search technique. The disadvantage of this search is that it might necessitate exploring all of the combinations of the 2-D search plane serially. Thus, it tests 2,046 half chips on the code phase dimension

for 21 carrier frequency search steps (bin resolution= ±500Hz). This means searching approximately 40,000 combinations, which is time consuming (Akos, 1997 and Uijt, 1998).

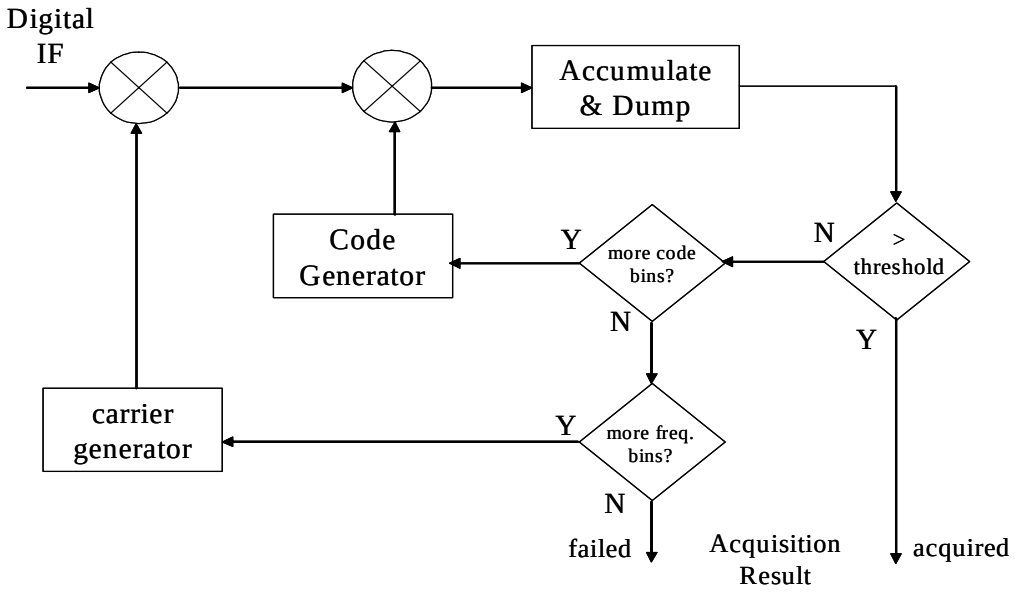

Figure 2-4: Serial Search Technique

The acquisition process time is shortened when the search dimensions can be searched in parallel. Consider that the incoming signal is multiplied by a local copy of the PRN code, assuming that the correct code phase is utilized, a fast fourier transform (FFT) is performed to the signal after wiping off the code (see Figure 2-5). The acquisition is complete if there is a peak in the output of the FFT after its magnitude is squared. This means the search is performed in parallel for all of the frequencies. In the worse case scenario, the FFT is repeated 2,046 times to cover all of the code phase bins (Akos, 1997 and Uijt, 1998). However, the calculation of one FFT requires much more effort than one serial search.

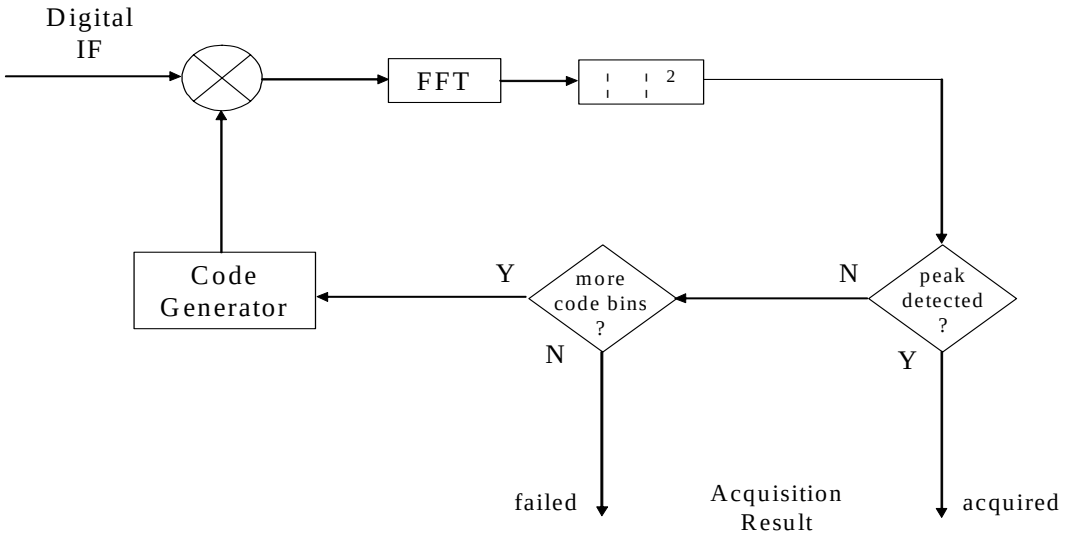

Figure 2-5: Parallel Carrier-Frequency Search Technique

Since the total number of code phase search bins (2,046) is greater than the frequency search bins (21), then searching the code phase dimension in parallel will speed up the acquisition process. This is carried out by using the frequency domain circular correlation. The circular correlation is performed by using the FFT convolution property. FFT-based circular convolution is achieved by multiplying the two sequences (or signals) in the frequency domain. Equation 2-3 shows how the convolution is calculated using the FFT/IFFT functions.

$$y = IFFT(FFT(x1) \times FFT(x2)) \qquad \textbf{2-3.}$$

Conjugate in frequency domain is similar to signal reversal in time domain. Therefore, circular correlation is implemented in the same fashion by conjugating one signal in the frequency domain before the multiplication. Equation 2-4 shows the FFT-based circular correlation formula.

$$y = IFFT(FFT(x1) \times conj(FFT(x2))) \qquad \textbf{2-4.}$$

A block diagram of the FFT-based correlator is shown in Figure 2-6.

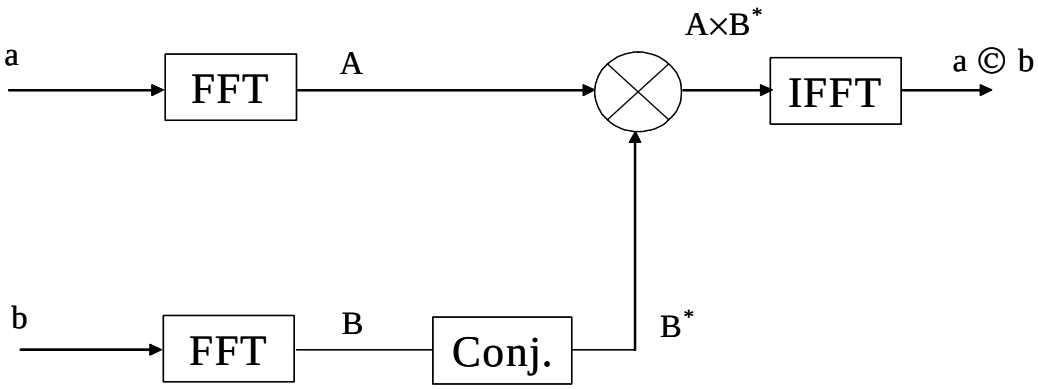

Figure 2-6: FFT-Based Circular Correlator

A parallel code phase search diagram is shown in Figure 2-7. This type of search greatly minimizes the search time since it evaluates all of the code phases in one single search cycle. Therefore, it does the FFT/IFFT correlation a total of 21 times in the worse case scenario (Van Nee, 1991 and Coenen, 1992).

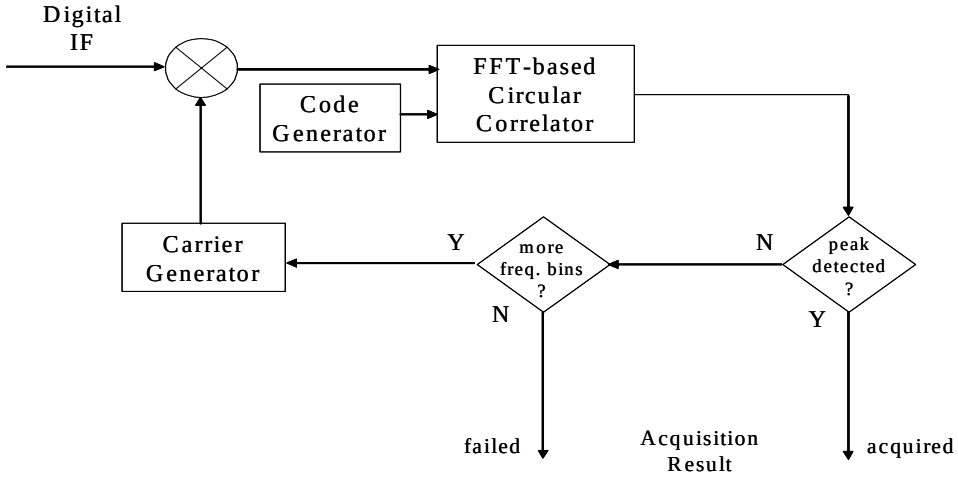

Figure 2-7: Parallel Code-Phase Search Technique

2.6 Block Processing

In the previous section, most of the presented acquisition search methods and the tracking loops are processed sequentially. This means that the samples of the received signal are treated sample by sample. Processing the data sequentially has some disadvantages. One disadvantage is that the receiver may lose the lock of signal due to anomalies (Feng, 1999). Tracking the lost signal needs a re-acquisition step. This problem is considered critical in some conditions such as weak signal tracking or in high-dynamic applications.

One possible solution was introduced in (Feng, 1999). The authors presented a method based on block processing of the data. Block processing has a better performance than the sequential processing. The block size is usually equal to the length of the C/A code. Thus, the block size is 1-ms. If a weak signal detection is desired then a multiple of this size can be used (Lin, 2001). 20-ms block size is used for weak signal processing. Block addition is one of the block processing techniques that helped in weak signals detection (Uijt, 1998). The FFT-based correlator is also employed with block processing to have a fast search method. More information about the GPS block processing can be found in (Feng, 1999 and Tsui, 1997).

2.7 Advantages and Disadvantages of the Current Signal Processing

The block processing method was implemented in software in (Feng, 1999). It is easier to implement algorithms in software because a designer can modify the

algorithms at any stage of the design development. The important issue in the above reference was the validation of the algorithm and the study of the signal quality monitoring system. Therefore, the delay of software processing was not an issue. However, since the acquisition of the GPS signal is the most time consuming process, the hardware implementation is a necessary step to shorten the acquisition time. Application specific integrated chips (ASICs) have been used in GPS receivers for a long time. They complete the sequential search quickly compared to the software based implementation.

There are two primary sources of trouble connected with slow acquisition. One of them is related to the large computation count in the search algorithm. This can be seen in the FFT-based convolution algorithm. While it is a fast search method, it requires the computation of two FFTs and one IFFT. These functions require a large number of multiplication and addition operations for the 1ms data. Thus, the correlation function takes most of the computation time in the GPS receiver (Gunawardena, 2000). Therefore, investigating different methods for convolution (or correlation) that require less computation time is an important research topic. Using parallel processing concepts along with digital design techniques in a reconfigurable platform such as a field programmable gate array (FPGA) would be the optimum and easily available solution. The advantages of using FPGAs are clear since they provide flexibility of software and performance of ASIC that shorten the acquisition time.

This chapter introduced the GPS system and the current GPS receiver implementations. The required processes for a GPS receiver were introduced. They are the acquisition and the tracking processes. Current search techniques were

presented in the chapter. Block processing technique which replaces the ordinary acquisition process and tracking loops was explained. The problems and the challenges of designing a GPS receiver were presented. Two primary and challenging tasks will be covered in this work. One task is the development of a new fast algorithm for acquisition and especially for performing fast circular correlation. The development of such algorithms is discussed next. This is followed by a fast acquisition method and its effect on detection probability in chapter 4. The other challenging task is the hardware implementation of such a method in an FPGA. The implementation of the method in the hardware is presented in Chapter 5.

Chapter 3

Convolution Algorithms Using Real and Binary Transforms

3.1 Introduction

Signal processing applications usually require complex and numerous operations. Therefore, fast and easy-to-implement algorithms have been sought by research groups and industries to cope with the advancement in the very large scale integrated circuits (VLSI) technology. Fast Fourier transforms (FFTs) have gained a tremendous amount of appreciation. They were used in most of the fast algorithms for DSP applications. Despite the fact that FFTs are efficient, they still require a large amount of processing time and a large silicon area when the size of the required FFTs is large. This is due to the complex multiplication computations that the FFTs have to perform.

This chapter investigates two other transforms for their use in calculating circular correlation or convolution. The next section presents a real transform called Fermat number transform (FNT). The FNT-based convolution method is presented along with the method limitations. Another useful transform is pre-

sented later in the chapter. It is a binary transform and is called the Walsh Hadamard transform (WHT).

3.2 Fermat Number Transform and Convolution

3.2.1 Fermat Number Transform

Number theoretic transforms (NTTs) are discrete transforms defined over finite rings. All the arithmetic in this finite ring is modulo the number of elements in the finite ring (Agarwal, 1974). Choosing the modulus as a Fermat number Ft makes the NTT a Fermat number transform (FNT). Fermat number transform (FNT) is defined as

$$X(k) = \langle \sum_{n=0}^{N-1} x(n)\alpha^{kn} \rangle Ft \qquad k = 0, 1, \ldots, N-1 \qquad \textbf{(3-1)}$$

$$Ft = 2^b + 1, b = 2^t$$

where N is the sequence length, α is a root of unity of order N, Ft is the Fermat number or the modulus describing the finite ring (Agarwal, 1974). Similarly the inverse Fermat number transform is defined as

$$x(n) = \langle N^{-1} \sum_{k=0}^{N-1} X(k)\alpha^{-kn} \rangle Ft \qquad k = 0, 1, \ldots, N-1 \qquad \textbf{(3-2)}$$

The FNT can be implemented as a butterfly structure of power-of-two where $\alpha = 2$ as presented in (Li, 1990). In this case, FNT changes to a simpler transform. This transform was defined by Rader and was called Rader transform (Rader, 1972

and Agarwal, 1974). Only addition, subtraction, and shift operations are used to implement this FNT. All these operations are simple operations for digital implementation. When they are compared to the complex operations in FFT butterflies, the advantages of using FNTs are clear.

Fermat number transform (FNT) is one of the candidate transforms for speeding up the correlation calculations since it only requires additions and logic shifts. Logic shift operation is a replacement of a multiplication by a power-of-two number. Therefore, FNTs do not need multiplications. The FNTs can be easily implemented in a butterfly structure. Also, the convolution property is available in the FNTs without round-off errors (Turimella, 1991). FNTs were used to implement linear and circular convolutions, FIR filters, and other important DSP applications. Since the circular correlation is the core of the GPS acquisition process, its implementation using FNTs may reduce the acquisition time. Therefore, descriptions of the convolution algorithm and the architecture of the FNT-based convolver are presented next.

3.2.2 Convolution using Fermat Number Transform

A FNT-based convolution algorithm is shown in Figure 3-1. This architecture is similar to the well-known FFT-based convolution (Arambepola, 1989 and Xu, 1992). The two sequences are changed to transform domain, multiplied, and then changed back to time domain. This way, a convolution is performed as a multiplication in the Fermat transform domain.

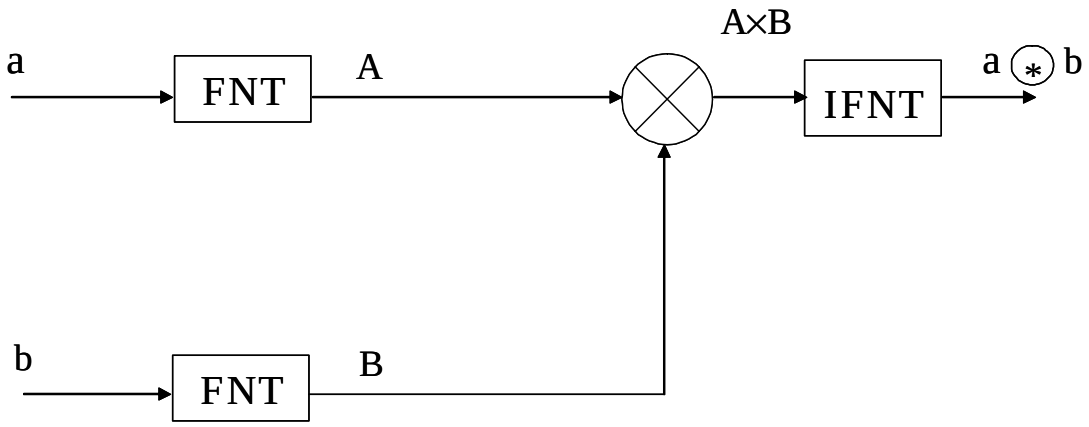

Figure 3-1: FNT-Based Convolver

To convolve two sequences x=(2,-2,1,0) and h=(1,2,0,0), F_2 =17 is used to avoid any overflow for N=4 (Agarwal, 1974). The transform matrix is

$$T = \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & 4 & 4^2 & 4^3 \\ 1 & 4^2 & 4^4 & 4^6 \\ 1 & 4^3 & 4^6 & 4^9 \end{bmatrix} = \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & 4 & -1 & -4 \\ 1 & -1 & 1 & -1 \\ 1 & -4 & -1 & 4 \end{bmatrix} (mod 17) = \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & 4 & 16 & 13 \\ 1 & 16 & 1 & 16 \\ 1 & 13 & 16 & 4 \end{bmatrix} (mod 17)$$

$4^{-1} = -4 \pmod{17}$ and therefore the matrix of the inverse transform is

$$T^{-1} = 4^{-1} \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & 4^{-1} & 4^{-2} & 4^{-3} \\ 1 & 4^{-2} & 4^{-4} & 4^{-6} \\ 1 & 4^{-3} & 4^{-6} & 4^{-9} \end{bmatrix} = -4 \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & -4 & -1 & 4 \\ 1 & -1 & 1 & -1 \\ 1 & 4 & -1 & -4 \end{bmatrix} (mod 17) = 13 \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & 13 & 16 & 4 \\ 1 & 16 & 1 & 16 \\ 1 & 4 & 16 & 13 \end{bmatrix} (mod 17)$$

The transforms of the input sequences are

$$X = Tx = \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & 4 & 16 & 13 \\ 1 & 16 & 1 & 16 \\ 1 & 13 & 16 & 4 \end{bmatrix} \times \begin{bmatrix} 2 \\ 15 \\ 1 \\ 0 \end{bmatrix} = \begin{bmatrix} 18 \\ 78 \\ 243 \\ 213 \end{bmatrix} = \begin{bmatrix} 1 \\ 10 \\ 5 \\ 9 \end{bmatrix} (mod 17)$$

$$H = Th = \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & 4 & 16 & 13 \\ 1 & 16 & 1 & 16 \\ 1 & 13 & 16 & 4 \end{bmatrix} \times \begin{bmatrix} 1 \\ 2 \\ 0 \\ 0 \end{bmatrix} = \begin{bmatrix} 3 \\ 9 \\ 33 \\ 27 \end{bmatrix} = \begin{bmatrix} 3 \\ 9 \\ 16 \\ 10 \end{bmatrix} (mod 17)$$

Thus, $Y = X \times H$. Y = (3, 90, 80, 90) = (3, 5, 12, 5) mod 17

Taking the inverse transform of Y, y = (2, 2, 14, 2) mod 17. Integers are assumed to be within -8 to +8. Therefore, y = (2, 2, (14-17), 2) = (2, 2, -3, 2), which is the correct answer for the convolution.

Similarly longer sequences can be convolved efficiently. The longer the input sequences, the longer the word size required. This means the numbers should be represented by a large number of bits. The addition circuits become bigger and the routing becomes a problem. Therefore, using FNTs to implement correlation of long sequences such as the GPS C/A code is not efficient, if not impossible (Dimitrov, 1994 and Selesnick, 1998 and Proakis, 2002).

Since the FNTs are not useful for the implementation of the C/A code correlator, then another transform must be tested. Each FFT (or IFFT) in the FFT-based correlator requires NlogN complex multiplications and NlogN complex additions. Therefore, this algorithm requires approximately 3N(logN)+N complex multiplications and 3N(logN)+N additions. A binary transformation (such as the

Walsh transform) is suggested since it requires only NlogN additions and subtractions. Finding a method to perform the convolution using only Walsh transforms and then implementing it using parallel processing units such as those in the FPGAs will definitely speed up the acquisition process.

3.3 Convolution and Walsh Transform

3.3.1 Introduction

This section presents a Walsh-based convolution algorithm. First, the Walsh transform is briefly presented. The similarity between the Walsh transform and the PN sequences is then described. The Walsh-based convolution algorithm is provided. This algorithm requires functions that need to be implemented efficiently in hardware to build a very fast convolver. The hardware implementation of the main functions are presented next. A discussion of the design and its results is provided. Evaluations of the design performance, based on clock frequency and system latency, are presented.

3.3.2 Walsh Transform and PN Sequences

The Walsh transform is an orthogonal binary matrix. Its elements contain 1's and 0's, which are (-1) and (+1) in real number representation (Zehavi, 1995). This means that the required operations are addition and subtraction. The Hadamard transform is a Walsh transform and its matrix can be written as

$$Hn = \begin{bmatrix} H_{\frac{n}{2}} & H_{\frac{n}{2}} \\ H_{\frac{n}{2}} & H_{\frac{-n}{2}} \end{bmatrix}$$

For n=8, the Walsh Hadamard matrix is

$$H_8 = \begin{bmatrix} 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & -1 & 1 & -1 & 1 & -1 & 1 & -1 \\ 1 & 1 & -1 & -1 & 1 & 1 & -1 & -1 \\ 1 & -1 & -1 & 1 & 1 & -1 & -1 & 1 \\ 1 & 1 & 1 & 1 & -1 & -1 & -1 & -1 \\ 1 & -1 & 1 & -1 & -1 & 1 & -1 & 1 \\ 1 & 1 & -1 & -1 & -1 & -1 & 1 & 1 \\ 1 & -1 & -1 & 1 & -1 & 1 & 1 & -1 \end{bmatrix}$$

The Walsh function can be transformed into a set of different phase shifts of a single PN sequence using suitable permutations (Cohn, 1977 and Lempel, 1979). This can be accomplished if the first row and the first column of a Walsh matrix are omitted. Reordering the remaining rows and columns using a specific permutation method will produce a different matrix. The produced matrix has a PN code in the first row and all the shifted copies of this code comprise the remaining rows (Budisin, 1989). This is a useful property since all the PN code shifts are available in one matrix. If a received signal contains this PN code, then it is clear that multiplying the signal by the obtained matrix will generate a vector that has a large multiplication value with only one of the PN shift copies (or Walsh rows). This means that the Walsh transform can be used to implement a convolution with a PN sequence (Budisin, 1989).

3.3.3 Convolution Algorithm

The proposed convolution algorithm requires two additional steps in order to use the Walsh transform to implement a convolution. The complete procedure, which includes these two steps, is as follows:

- The input sequence must be permuted to simulate the reordering of the rows.
- The Walsh function is applied.
- The results need to be permuted back to correctly reorder the convolution values to their original locations.

The two steps of permutation processes require two different permutation vectors, S and C (Budisin, 1989). For example, a PN code with a period of 7 is used to illustrate the algorithm. This example is presented in detail in (Budisin, 1989) and is illustrated on Figure 3-2. The PN code is generated using a 3-bit linear feedback shift register (LFSR). An 8-point Walsh transform is used for the 7 bit-length PN code. After omitting the first row and the first column, the Walsh transform becomes a 7-point transform. For the PN code such as (1001110), the input permutation S is generated by reading the contents of the LFSR bits every clock cycle. Initially the LFSR has “001” which means “1”. One clock cycle later, the contents of the LFSR becomes “100” which means “4”. Continue reading the LFSR contents until the code repeats itself. The generated S permutation values therefore become (1,4,6,7,3,5,2) as shown in Figure 3-2. By sorting these values and keeping in mind their original order, the inverse permutation values of S can be computed. For example sorting S requires that the reordering of its values lead to (1,2,3,4,5,6,7). In order to have this reordering, the S values need to be reordered such as the first

value is the same, the second value comes from the last location (which is 7), and continues until the last element of S. The generated inverse permutation of S is then (1,7,5,2,6,3,4). The output permutation Q, or C as in (Budisin, 1989) is (1,2,4,5,7,3,6) as shown in Figure 3-2. The hardware generation of this permutation can be performed using a one-to-many LFSR as described in the next section. The resulting sequence Q can be obtained from LFSR shown in Figure 3-2. To perform the correlation using this method, the input sequence is first permuted by the inverse permutation of S.

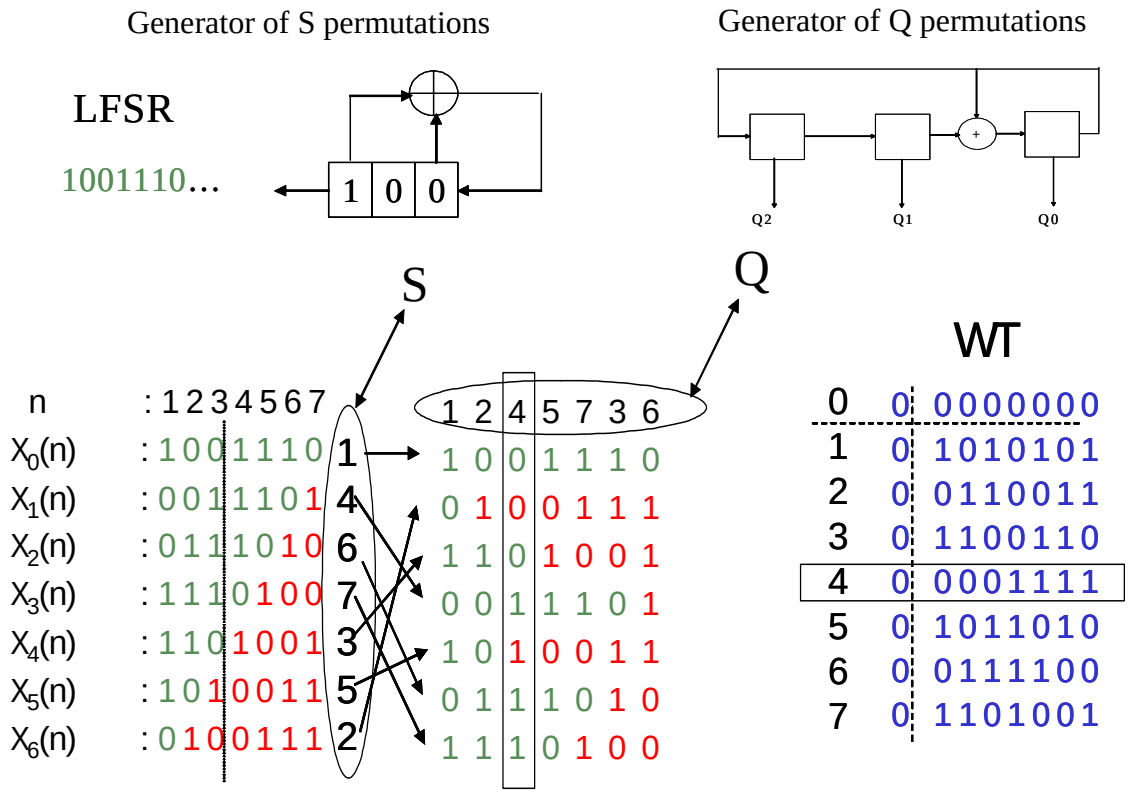

Figure 3-2: Example for the Walsh-Based Convolution

Suppose that the original and the received PN codes are as follows,

PN code: (1 0 0 1 1 1 0)

PND code: (0 1 1 1 0 1 0) (delayed code)

Using inverse permutations of S we will get

PND (S^{-1}) = X= (0 0 0 1 1 1 1)

where S^{-1} = (1 7 5 2 6 3 4),

and Q = (1 2 4 5 7 3 6)

The convolution of PN code with PND is obtained by computing Walsh transform of X. Since the Walsh transform row is larger than the sequence by one, the permuted sequence must be appended with (0) at the beginning of the sequence. After that, the Walsh transform is applied as follows.

Y=X.W= (-1 -1 -1 7 -1 -1 -1)

where W = $\begin{bmatrix} 1 & 0 & 1 & 0 & 1 & 0 & 1 \\ 0 & 1 & 1 & 0 & 0 & 1 & 1 \\ 1 & 1 & 0 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 1 & 0 & 1 & 1 & 0 & 1 & 0 \\ 0 & 1 & 1 & 1 & 1 & 0 & 0 \\ 1 & 1 & 0 & 1 & 0 & 0 & 1 \end{bmatrix}$

The results are not in the order they should be in Qs compared to the convolution results. Thus, the output should be reordered based on the permutations of Q. Therefore we use the output permutation sequence to obtain the final value of convolution as follows:

Y(Q) = (-1 -1 7 -1 -1 -1 -1)

This result gives the correct estimate of shift (2) with respect to the original PN sequence.

This method requires only NlogN real additions and/or subtractions. In comparison, the FFT-based method requires 3 FFTs and N complex multiplications and N complex additions. If the FFT of the PN code was stored in the computer memory, then only 2 FFTs plus N complex multiplications and additions are required with the incoming signal. This is approximately 2NlogN+N complex multiplications and 2NlogN+N complex additions, which is approximately 6NlogN real multiplications and 14NlogN real additions (Smith, 1995). Therefore, the Walsh-based correlation method is preferred especially in real-time applications where the acquisition process needs to be very fast. More details about the Walsh-based convolution algorithm and its in-depth theory can be found in (Budisin, 1989 and Sari, 1995).

3.3.4 Hardware Implementation

The permutation generators and the Walsh transform are the primary steps that need to be implemented carefully. The permutations usually can be stored in lookup tables (LUTs). This type of implementation is not efficient since it will require additional hardware to store and time to retrieve. Therefore, the permutations need to be generated on the fly whenever possible in order to minimize the required silicon area and to speedup the permutation process. However, the implementation of a Walsh transform in an FPGA has two requirements. A Walsh transform needs to use the parallel processing method as much as possible and chose optimum smaller transform block sizes for building very large transforms.

3.3.4.1 Permutation Generators

PN sequences are generated using linear feedback shift registers (LFSRs) (Golomb, 1967). If the permutations are related to the state of the LFSR, then the permutations are generated from the same LFSR. This is true for permutations S, which are the decimal values of the binary bits in the register. When initializing the LFSR with (001) for the previous example, the current state and the next six states (or values) of the register will be (001, 100, 110, 111, 011, 101, 010), or (1, 4, 6, 7, 3, 5, 2) in decimal which are the required permutations sequence of S. Therefore, the permutations S can be generated easily. A hardware implementation of the generator of the permutations S is shown in Figure 3-3. The RAM and the counter are only needed when storing permutations is desired. However, in the correlation algorithm, only the inverse permutation of S is used. Therefore no hardware is used to store S.

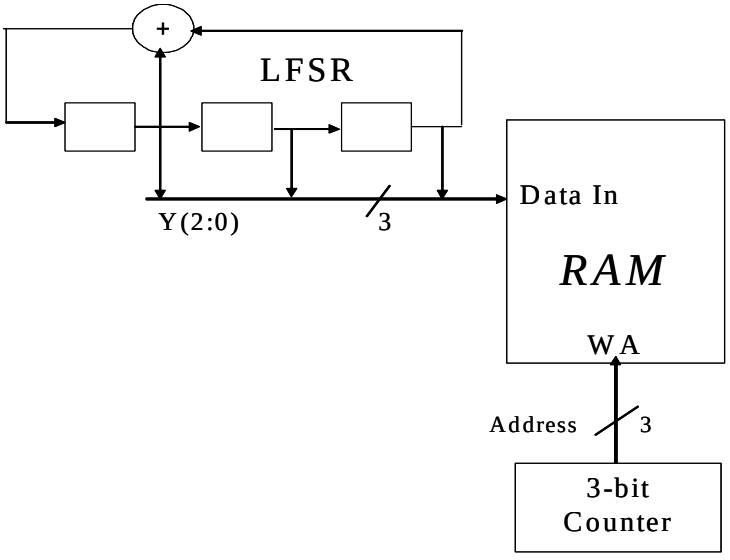

Figure 3-3: Hardware Implementation of the Generator of the Permutations S

The hardware implementation of the generator of the inverse permutations S is shown in Figure 3-4. The only change made to Figure 3-3 is that the LFSR is used to deliver a RAM address while the counter output is fed to RAM as data in. As previously mentioned, storing permutations is not efficient. However, since generating inverse permutations of S is required only once in the beginning of the design, this implementation is accepted. Permutations Q, are related to the content of the LFSR. Each PN code has its unique permutation of LFSR bits to produce Q. Reordering the content of the LFSR (i.e. b2b1b0 becomes b2b0b1) will generate the necessary Q sequence. Whereas, reordering the bits of the content of the LFSR can be implemented using one-to-many type of LFSR as shown in Figure 3-5 (M. Cohn, 1977)

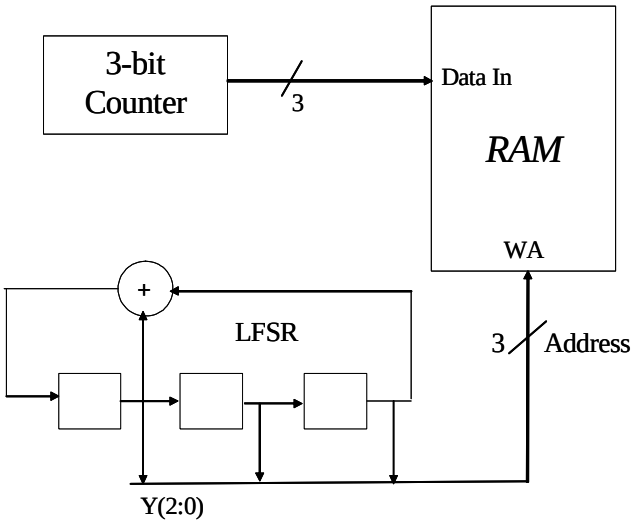

Figure 3-4: Implementation of the Inverse Permutations of S

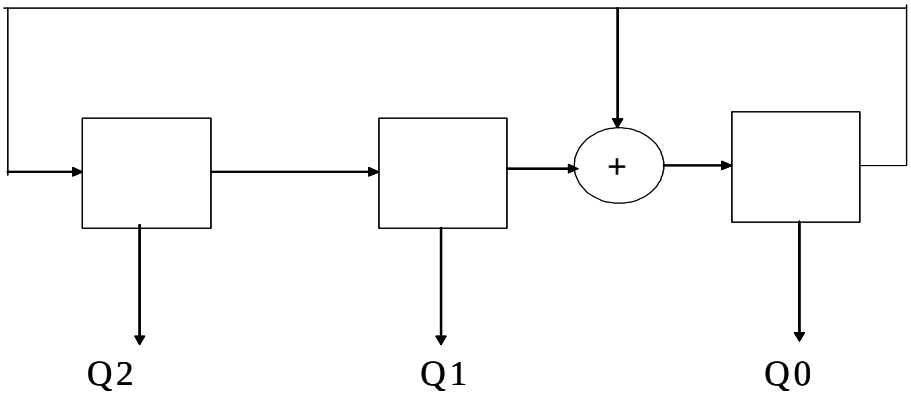

Figure 3-5: Implementation of Generator of Permutations Q

3.3.4.2 Walsh Transform

Designing transforms in a parallel processor platform such as a field programmable gate array (FPGA) is recommended since transforms need a large number of operations that can be easily mapped into an FPGA. The hardware design of a Walsh transform using a butterfly structure is very efficient for numerous reason. The most important reason is that a Walsh butterfly has only one type of operations which is addition, subtraction operations can be performed on the same hardware as addition (see Figure 3-6). Therefore, when a designer wants to partition the Walsh butterfly, the primary concerns are the locations of the inputs, intermediate values, and outputs.

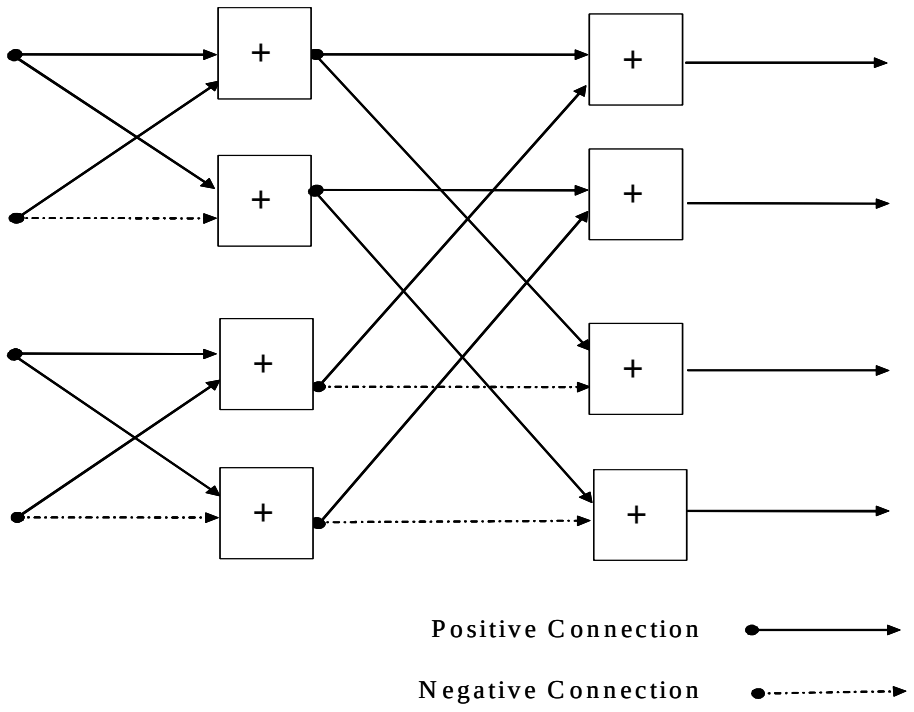

Figure 3-6: 4-Point Walsh-Hadamard Butterfly Structure

The fastest Walsh butterfly should be implemented in parallel fashion in order to reach the maximum speed. Unfortunately, the Walsh transform size can be huge when employing DSP applications. The number of processors required to implement a completely parallel transform is very large and requires a large number of input and output pads. Therefore, a completely parallel design is impossible since at the time of this writing there is no chip that can support these requirements. One possible solution to this problem would be partitioning the butterfly into smaller butterflies. The size and number of the required smaller butter-

flies will affect the design organization and performance of the Walsh transform chip.

For a 1,024-point Walsh butterfly, a designer can use partitions of 128, 64, 32 or 16-point butterflies. A 128-point butterfly may not be a good choice because 8-point butterflies will also be required to build the 1,024-point Walsh transform. Therefore, choosing a different size of butterfly could assist in reducing the number of blocks and will help in designing a well-organized structure. Another factor that controls the design is the available resources in the board or the chip. If an optimum size of smaller butterflies is found, and is still too big for parallel implementation, a designer may need to divide it into smaller butterflies due to the lack of resources. Using large FPGAs, such as the Virtex FPGA series, will provide more resources for designs as large as 1,024-point transforms. For a 1024-point Walsh butterfly, 64 blocks of 32-point Walsh butterflies is the optimum solution if the 32-point Walsh block is designed completely in parallel method.

3.4 Implementation of a Walsh-Based Convolver for a 1,023-bit PN Code

To implement a Walsh-based convolver for a PN code of period 1,023, a 1,024-point Walsh transform is necessary. The optimum size of smaller blocks of butterflies is 32-point. A 32-point Walsh butterfly needs to be processed 64 times and no other smaller size of butterflies are required. This is not the case if a different size was chosen for the implementation.

Since the design platform is the 0.8 million gate Virtex FPGA, the 32-point butterfly is implemented in parallel and used to calculate the first 5 levels of the

1,024-point butterfly. A similar block is also used to calculate the second 5 levels of the 1,024-point butterfly (Figure 3-7). The second block is used to provide a pipe-line structure that reduces the latency of the designed system.

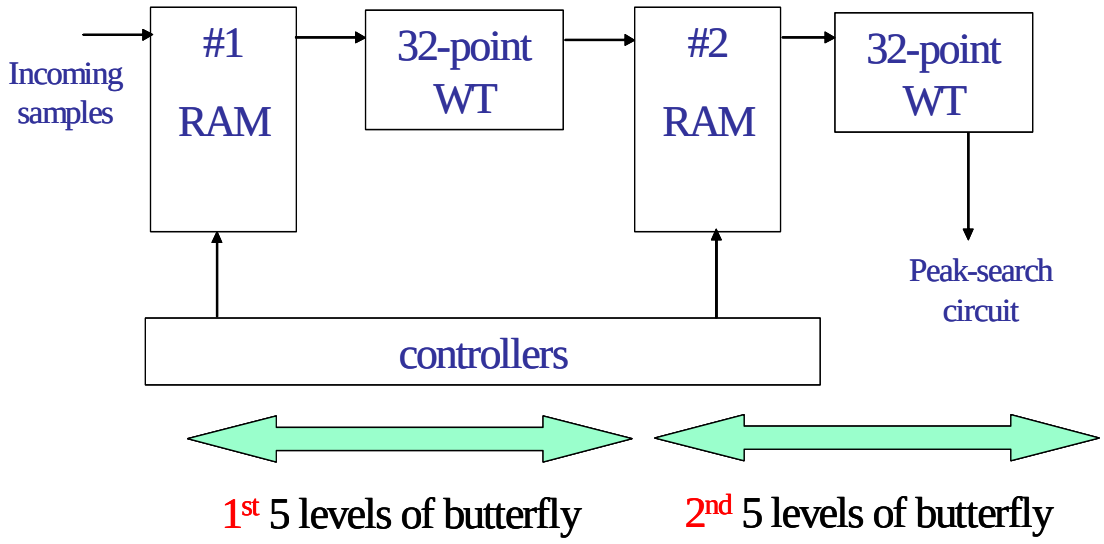

Figure 3-7: Implementation of 1024-Point WHT

The completed design used more components to be able to provide an updated evaluation of the phase shifts every 3 code lengths (3,069 clock cycles). These components are required to perform Walsh transform computations, permutations generation, correlation peak search, RAMs to store intermediate butterfly results, counters for RAM addresses, and state machines to control the whole system to continuously provide phase information in constant time space. The whole design used approximately 264k gates (60% of the Virtex chip). The maximum frequency for this design is 96 MHz (Alaqeeli, 2001).

To evaluate the performance of this design compared to a FFT-based design, an assumption is necessary. If we assume that a multiplication computation takes as much as two additions, then the total number of operations in the FFT-based method is approximately 18 times larger than the Walsh-based algorithm. Furthermore an N-bit multiplier requires N times more area than an N-bit adder. Therefore, if the input signals are 8-bit long, then the time-area efficiency of this algorithm is approximately 144 times better than an FFT-based method (A. Alaqeeli, 2001). This is the case if the whole transforms are computed completely in parallel. However, this is not the case. Therefore, a design analysis within a limited area is more appropriate.

One assumption would be that the design is limited to the area necessary for a 32-point Walsh butterfly. Thus, using the Walsh-based method, the convolver of a 1023-length PN code requires 64 blocks of 32-point Walsh transforms. Therefore, it will take 64*32 (=2048) clock cycles. Whereas, the maximum size of an FFT butterfly that can be mapped to the same limited area is only a 4-point FFT. This means that 4*256*5 (=5,120) clock cycles are necessary to compute a 1,024-point FFT. Furthermore, each clock cycle required in a 32-point Walsh butterfly requires performing 5 consecutive additions, while a 4-point FFT requires 6 multiplications and 10 additions. Calculating a 1024-point FFT is approximately 10 times longer than calculating a 1024-point Walsh transform. Since an FFT-based convolver requires two FFTs, the minimum time required for an FFT-based convolution is approximately 20 times larger than that required for a Walsh-based method (Alaqeeli, 2001). Therefore, a Walsh-based method is at least 20 times faster than

a FFT-based method when the design area is limited (as in a Virtex FPGA implementation).

When comparing this implementation with a software-based implementation of the Walsh transform on Matlab, the FPGA-based implementation was approximately 2,500 times faster than Matlab implementation on a 233MHz processor. This is using a 1MHz clock in the FPGA design. To speed up the processing time, a fast clock with internal processing parts and multiplexers to switch between the two clocks can be used. For example, if loading incoming samples uses 1MHz clock frequency, then the convolution process can be accelerated by allowing the internal parts of the design (such as reading from memories, 32-point Walsh transform, and finite state machines) to use a different clock frequency (such as 16MHz). This insures that the system can be used efficiently and the design can provide a new phase shift every 2 code periods instead of three code periods. However, this may cause synchronization problems. Therefore it will need to be studied carefully. The implemented design has sped up the acquisition of the CDMA signals many times. The design has also opened new research topics for the applications where real-time acquisition is needed.

However, regarding the GPS case, this method cannot be directly used, because C/A codes are gold codes which are generated by combining two PN codes using XOR operations. Therefore, the C/A code does not follow the permutational relations of Walsh transforms and PN codes. However, based on the literature, it was not proven that the Walsh transforms cannot be used for gold code's correlation implementation. In addition, the number of ones and zeros in a gold code are equal to the number of ones and zeros in any row of the Walsh transform after

omitting the first column. The possibility of finding a way to extend the use of the Walsh-based correlation method to the C/A code correlation still exists. Finding such a way would help speeding up the C/A code acquisition significantly.

Chapter 4

Averaging Method for Block Processing

4.1 Introduction

Implementing a correlator using Fermat Number Transform is not applicable for GPS C/A code acquisition due to the limitations on the required sequence length and the large sample word size needed. Additionally, the Walsh Hadamard based convolver was not intended to be used for gold codes, i.e. the C/A code. Therefore, the FFT-based correlator was chosen for the implementation of the acquisition process.

Building power-of-two based FFTs is achieved by using uniform butterfly structure. The GPS front-end system which was used in this research has a sampling rate of 5MHz. This means that every 1-ms of the GPS signal has 5000 samples. Therefore, two 5000-point FFTs and one 5000-point IFFT are required to implement the C/A code correlator. The size of the FFT is not a power of two, so it

will be very difficult to implement since it requires a mixed-radix algorithm that includes non-power-of-two FFTs (Smith W, 1995 and Gunawardena, 2000).

The use of smaller FFTs is needed to map the acquisition process in FPGA. One suggestion method for solving this problem is presented in the next section. This method is called acquisition of C/A code using averaging correlators. It uses FFTs with a similar size of C/A code, 1023 is used instead of 5000-point FFTs (Starzyk, 2001).

4.2 Averaging Correlator

Since the parallel code phase search requires the C/A code to be up sampled to 5000 points to match the received GPS signal, the required FFT size needs to be 5000. Similarly, when an opposite approach is carried out, the FFT size is smaller. Thus, if the incoming sampled GPS signal is down sampled from 5000 to 1023 points then the same FFT-based correlator can be applied, but with a FFT size of only 1023.

The correlation will be carried out by frequency transforming the 1023 averaged (or down sampled) GPS signal using 1023-point FFT. This is performed on both the in-phase and the quad-phase by considering the in-phase values as the real input components, while the quad-phase values are used as the imaginary input components. First, apply 1023-point FFT to the local code and perform the conjugate operation to the FFT output. The next step is to multiply the complex outputs of the FFTs. The results are then changed back to the time domain using 1023-point Inverse Fast Fourier Transform (IFFT). When the magnitude of the

IFFT result is carried out, the 1023 values are inspected for the maximum (or the peak) value. The location of the peak reflects the code phase in chips (or in 1/1023 ms). The next step is to use triangle fitting technique to recalculate the code phase. This process enables one to determine a better estimate for the code phase.

Since the incoming 5000 samples represents 1ms of data, it means that it contains all of the 1023 chips of the C/A code. Averaging 5000 samples to obtain 1023 samples means that most of the C/A code chips are represented by five samples while fewer chips use four samples. Therefore, the averaging of the 5000 samples will not give an approximate representation of the C/A code unless the first value in the 5000 samples is the first sample in a chip. However, it is very difficult to find the starting point of a chip especially when the GPS receiver is in the cold start mode. Therefore, to insure that the averaging method will always find the peak, the down sampled signal needs to be approximately equal to the C/A code or its shifted copy. Therefore, the code phase searcher should try five successive starting points in order to say that one of the down sampled codes is approximately the C/A code or its shifted copy.

The averaging method produces five correlation functions with 1023 values each. Thus, the total number of correlation points is 5115. One of the five tests will have a good approximation of the correlation function, while the other four have more corruption. The best recovered code is responsible for the strongest peak (Starzyk, 2001). Therefore, its peak is selected as the peak of the C/A code correlation.

The averaging method will apply five averaging correlations using five successive starting points. Selecting the location of the strongest peak is not sufficient

for the estimation of the code phase. One must include the time delay caused by using the starting point that is responsible for the peak. The delay is in the unit of samples which are 0.2 microsecond each. Assuming that the fifth starting point is responsible for the strongest peak, which is at location 120, then the peak location is corrected by adding 0.8 microseconds (or 4 samples). Therefore, the rough estimate of the code phase is 120/1023 + 0.8 microseconds. When a refined code phase estimation is required then triangle fitting method can be used for refining the code phase estimation (Zhu, 2002).

In order to use this method for acquisition, one should study its effect on the peak-to-second-peak ratio. This method does not reduce the peak-to-second-peak value compared to the case of a 5000-point correlation method. Moreover, the averaging method may increase the peak-to-second-peak value in some cases while retaining it in the same level in most cases. As a result, the detection probability is not affected by replacing the 5000-point FFT-based method by the averaging method (Starzyk, 2001). Therefore, the acquisition time is reduced by using this method because calculating five 1023-point FFTs and IFFTs requires less time in software and/or hardware than the 5000-point FFTs and IFFTs.

Although this method avoided building the huge 5000-point FFT, it did not simplify the task enough for direct implementation. This is due to the fact that a 1023-point FFT is difficult to implement using a butterfly structure since 1023 is not a power-of-two. The 1023-point FFT is still considered large. Therefore, trying to approximate this averaging method by using a 1024-point FFT block is a good path to investigate. This procedure is desired for the reasons mentioned above and for the availability of an optimized 1024-point FFT/IFFT core from Xilinx. The next

sections present descriptions of two methods of using a 1024-point FFT for acquiring the GPS C/A code.

4.3 Averaging Correlator with Zero-Padding

Since the length of the C/A code is 1023 chips, one extra point is needed to make it a 1024-point code that can be used with a 1024-point FFT-based correlator. One solution is to extend the incoming averaged samples to become 1024 with zero padding. However, this cannot be used directly because it will change the circular correlation properties because the 0-padded C/A code is no longer periodic. Based on the insertion location of the zero in both the incoming averaged samples and local codes, in the worse case scenario the peak will lose 50% of its energy (Zhu, 2002). This is due to the fact that the insertion of the zero may divide the C/A code into two sections. If one section is aligned to the part of the code that is buried in the signal, then the other part is 1-chip shifted and vise versa (see Figure 4-1). Therefore, the correlation function will no longer have one peak, but it will divide its energy into two neighboring peaks. The summation of the values of these two peaks will approximately equal the original peak value. This energy loss affects the signal detection and increases the probability of misdetection (Zhu, 2002).

To resolve the above problem, the insertion of the zero should be chosen to separate the C/A into two parts where one of them is long enough to hold most of the correlation peak energy. When this occurs, the strongest peak is considered a good approximation of the original correlation peak. One possibility for determining a good zero insertion location was presented by Zhu (Zhu, 2002). His method is

based on searching for three or four places for zero insertion. This enables the peak-to-peak ratio to be recovered to approximately the same level of the 1023-point FFT-based correlation. The signal energy loss in the worse case scenario is equal to 1/2H, where H is the number of searched insertion places (Zhu, 2002).

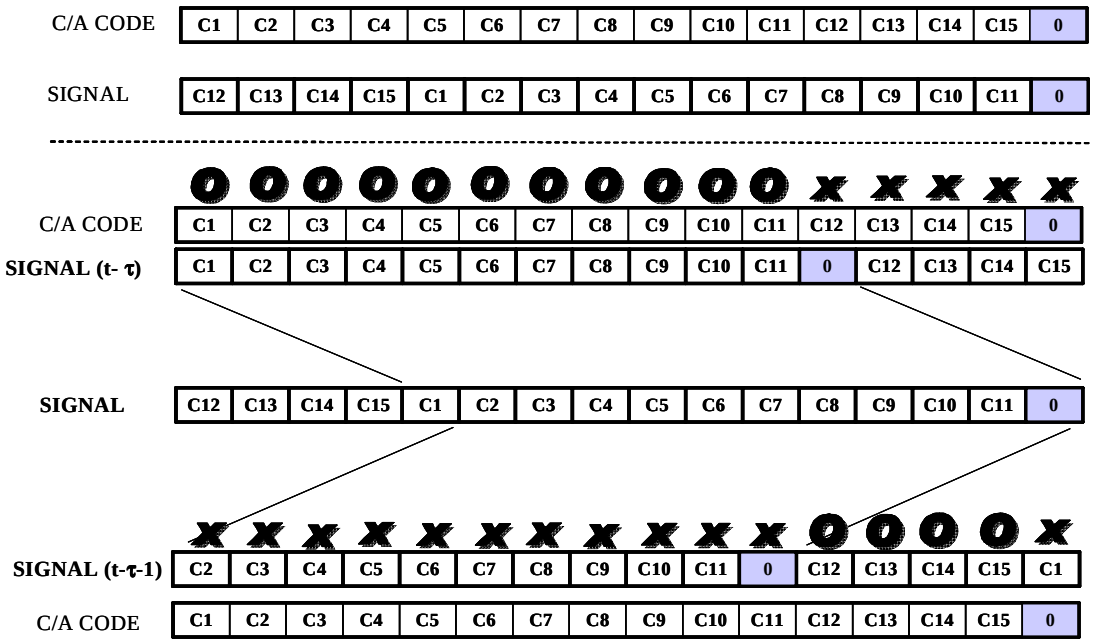

Figure 4-1: Zero-Padding Circular Correlation Problem

Since this method needs another search dimension, which is the zero insertion location, the number of operations required for acquisition increases. This method requires five FFT-based correlation processes, repeated H times to be able to acquire a GPS signal. For H=3, this method requires 15 FFT-based correlators. If the FFT of the local code is precomputed and stored, then this method requires 15 1024-point FFTs and 15 1024-point IFFTs plus 1024*15 complex multiplications, which is approximately 15*2*1024(log1024)+15*1024 = 322,560 multiplications, and 15*2*1024(log1024) = 307,200 additions. The acquisition using 5000-

point FFT-based correlator requires approximately 130,000 multiplications and a similar number of additions and subtractions. Therefore, the averaging method using the zero padding technique will raise the number of operations above direct FFT based method and lengthen the acquisition time. This method should not be used to implement the C/A code acquisition process in a GPS receiver. One possible solution to the problem of size mismatching of the code and the available 1024-point FFT core is presented in the next section. The new approach will show a fast acquisition method that can be implemented much easier than the 5000-point FFT-based method.

4.4 Averaging Correlator with Modified C/A Code

The size incompatibility of the C/A code and the available FFT core can be solved by changing the down sampling rate to produce 1024 averaged samples instead of 1023. So, the 5000 samples will be down sampled (or averaged) to 1024 points. A similar procedure will be done to the local code. Therefore, the local code will be up-sampled to 5000 and then down sampled to 1024 points. The averaging correlator will use 1024 averaged samples and 1024-point averaged C/A code

The important properties of the C/A code as they pertain to the detection probability are the auto and cross correlation functions (Kaplan, 1995). If the correlation function of the above algorithm can show an accepted approximation of the original C/A code auto correlation function, then this can shorten the computation time compared to the 0-padded averaging method and the 5000-point correlation method.

The original C/A code is a gold code. Figure 4-2 shows its auto correlation function. The correlation peak shape is an isosceles triangle and is shown in Figure 4-3. Therefore, the modified C/A code which was up sampled to 5000 and then down sampled to 1024 should provide a good approximation for these figures.

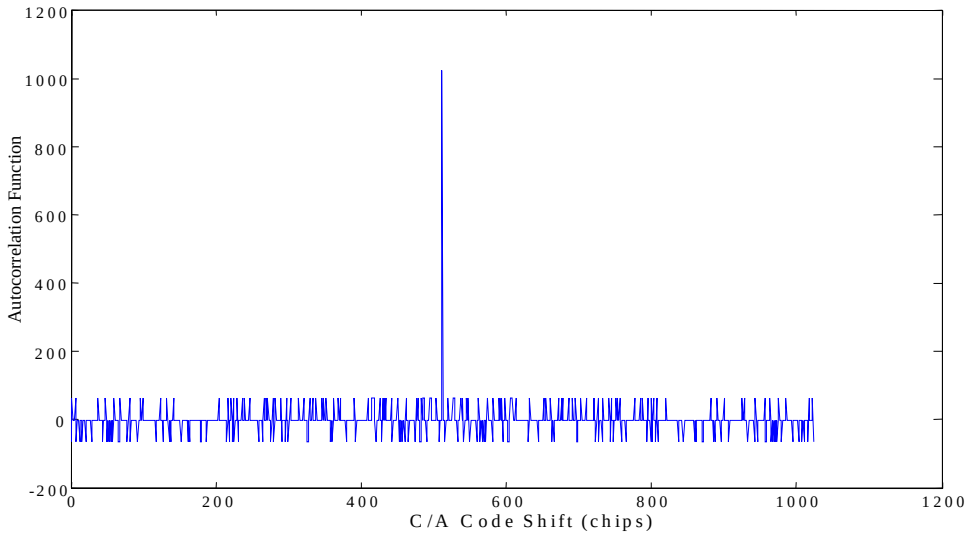

Figure 4-2: C/A Code Auto Correlation Function

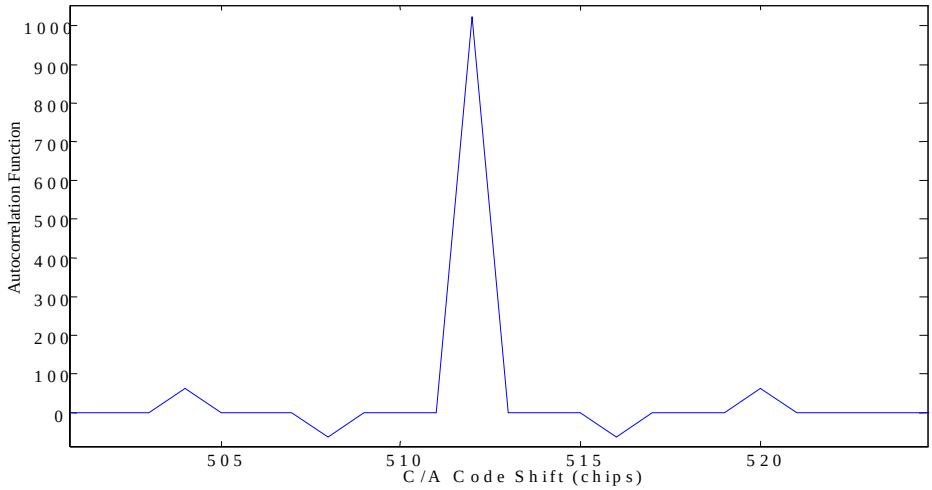

Figure 4-3: C/A Code Auto Correlation Peak Shape

When the C/A code was up sampled to 5000 and then down sampled to 1024, the code was changed from a binary code to a multilevel code which contains the values of ±1, ±.8, ±.6, ±.5, ±.4, ±.2, and 0 (Figure 4-4). However, this modified C/A code is still considered a unique code related to the selected original C/A code. It cannot be generated from a different original C/A code.

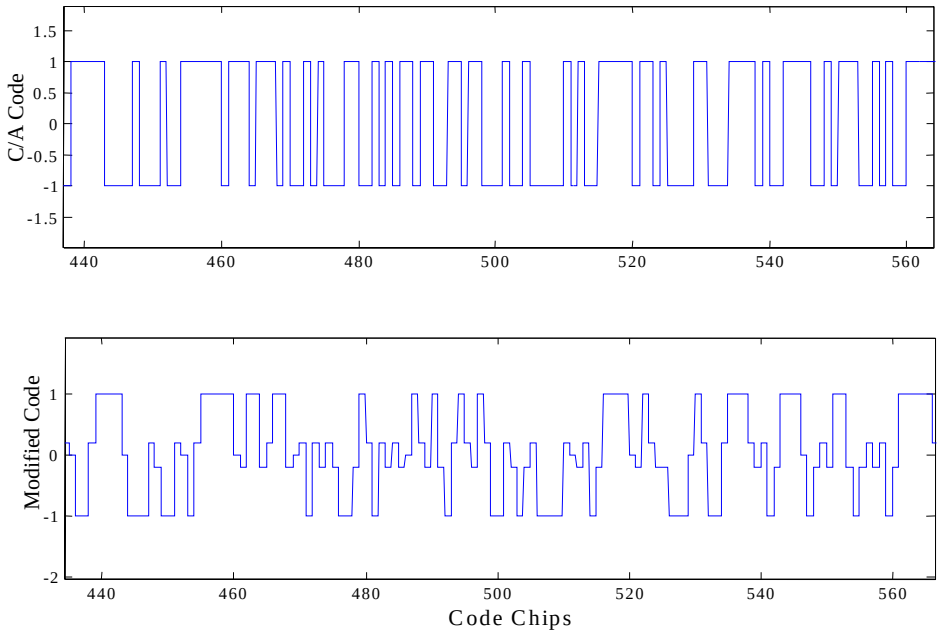

Figure 4-4: C/A Code and the Modified Code

As previously stated, the modified code averaging correlator needs to be applied five times with five successive starting points. When the five correlation functions for this modified C/A code are generated, the correlation function that contains the strongest peak keeps the important information necessary for acquisition. Fig 4-5 shows the winner auto correlation function of the modified C/A code. The peak shape is different than the true peak shape (Figure 4-6). This peak is not

an isosceles triangle, as it should be in a normal C/A code. However, it is sufficient to roughly estimate the code phase for acquisition.

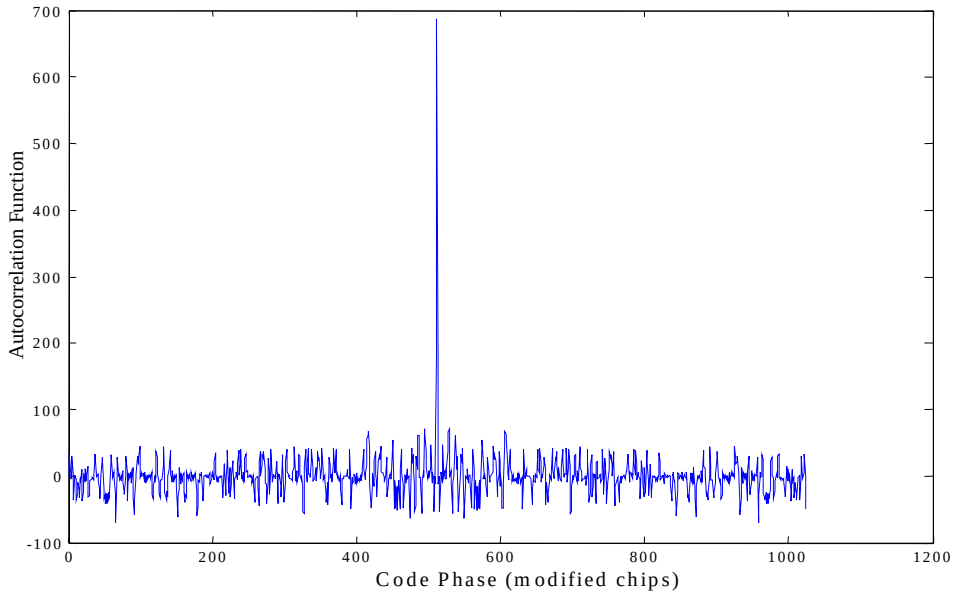

Figure 4-5: Modified-Code Winner Correlation Function

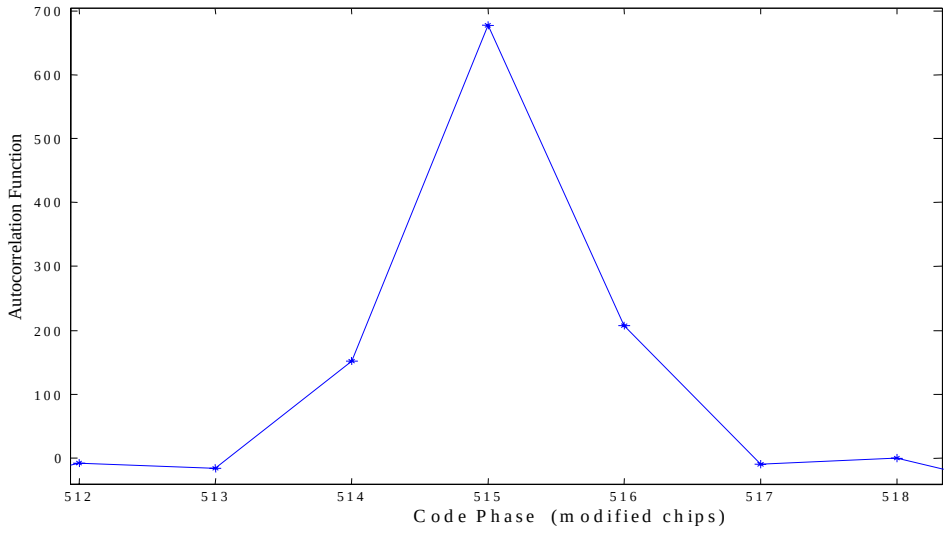

Figure 4-6: Peak Shape of a Winner Correlation Function for the Modified-Code

A close observation will reveal that each point in this correlation function is one out of 1024 modified chips. Each modified chip is 1/1024 (ms) while the original chip width is 1/1023 ms. To estimate the code phase, one needs to consider this change in chip width and should adjust the value of the code phase. For example, if the peak is at location 120 out of 1024, then the estimated code phase equals (120/1024)*1023 original chips, or the code phase is (120/1024)*1ms.

The modified-code averaging correlator needs five times 2*1024*10 plus five times 1024 multiplications and five times 2*1024*10 additions for the GPS signal acquisition. Therefore, the total required number of operations equals 107,520 multiplications and 102,400 additions. Also, the averaging computation requires approximately 20,000 additions and 5,120 divisions. The division operations will not be counted because they can be avoided. Therefore, the total number of operations is 107,520 multiplications and 122,400 additions. This does not significantly reduce the number of computations much compared with the 5000-point FFT-based correlator. However, the implementation of a 1024-point FFT is much simpler than the 5000-point FFT. Utilizing the modified-code averaging method will lead to significant simplification in the hardware implementation (Alaqeeli, 2003).

In order to use this method with block processing, the effects on signal-to-noise ratio and the other characteristics should be studied. The next section presents the characteristics of the modified-code averaging correlation method in terms of signal-to-noise ratio (SNR) loss, code phase accuracy and carrier phase accuracy.

4.5 Characteristics of Using the Modified-Code Averaging Correlation

In the previous section, the modified-code averaging correlation method was described. It is best-suited for the implementation of the block processing of GPS signals. In order to confirm the ability of the modified-code averaging method to replace the acquisition and the tracking loops, the effect the method has on the signal power and on the calculation accuracy of the code and the carrier phases must be checked first. For this reason, a 200 ms of GPS data is used to test the acquisition using the modified-code averaging method. The peak-to-peak ratio is chosen as a measure of signal strength. Matlab simulation showed that the average loss in a peak-to-peak ratio is about 12.7% as shown in Fig 4-7. This loss is approximately 0.5 dB, and is acceptable in most cases, except in the case of a weak signal acquisition.

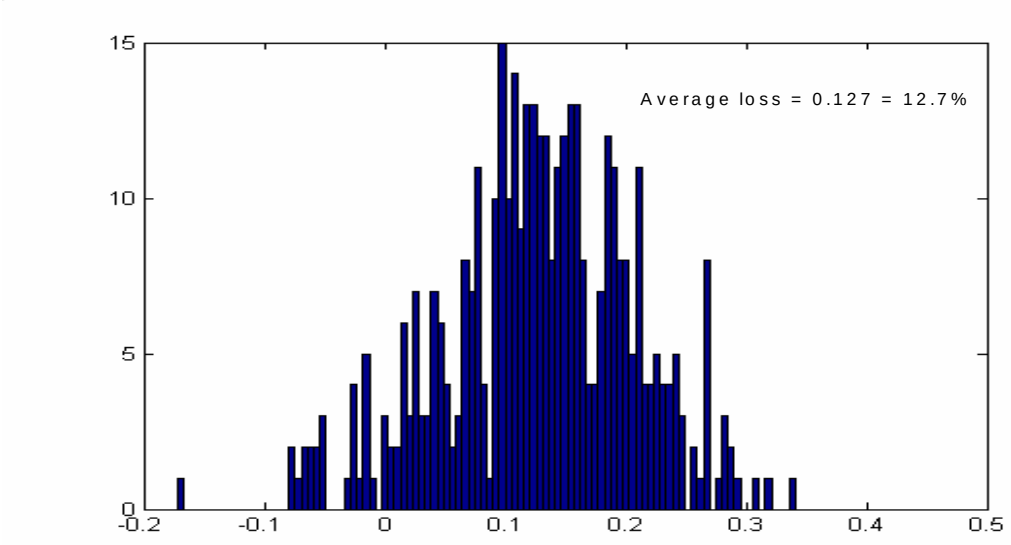

Figure 4-7: SNR Loss in 200-ms of a GPS Signal Using the Modified-Code Averaging Method

For the acquisition process, the code phase estimation accuracy needs to be within ± ½ chips. GPS-type data, which is generated using Matlab with code shift forced to be in the middle of a chip of a C/A code, is used to check for the worse case estimation accuracy of the averaging method. The averaging method was able to produce the right code phase with an accuracy of ± 0.1 chip, or ± 100ns (Alaqeeli, 2003). A histogram of a code phase error based on the modified-code averaging correlation is shown in Fig 4-8. The results show that the direct code phase calculation accuracy is within the acceptable range compared with the 5000-point FFT method (see Figure 4-9). Therefore, the averaging method is acceptable for acquisition.

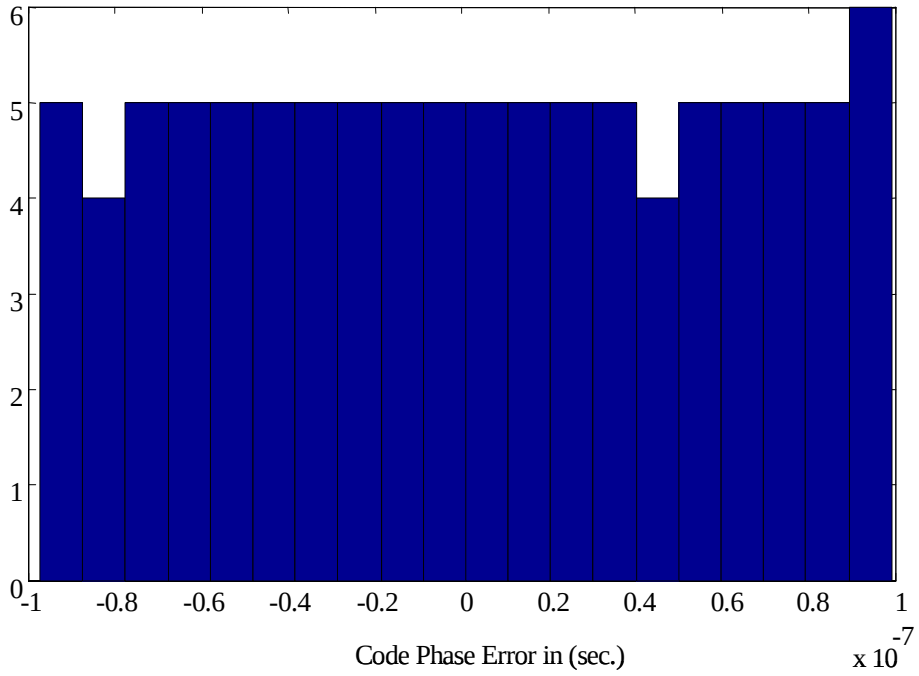

Figure 4-8: Code-Phase Error Using Modified-Code Averaging Method.

A code tracking process produces a refined code phase estimation. Similar to what occurs in block processing, the triangle fitting approach was applied to the

peaks of the averaging method. This approach is able to refine the code phase estimation. The average error in the code phase estimation was 42.7ns. A modified triangle fitting algorithm with an acquisition history feedback was developed. The developed algorithm with the history feedback showed an improvement over triangle fitting with an average error of approximately 10ns in the code phase calculation accuracy. Compared to the GPS block processing, this code error is still considered large. The modified-code averaging method is not accurate enough to replace the tracking loops or the block processing. Therefore, the averaging method will be used to achieve fast acquisition.

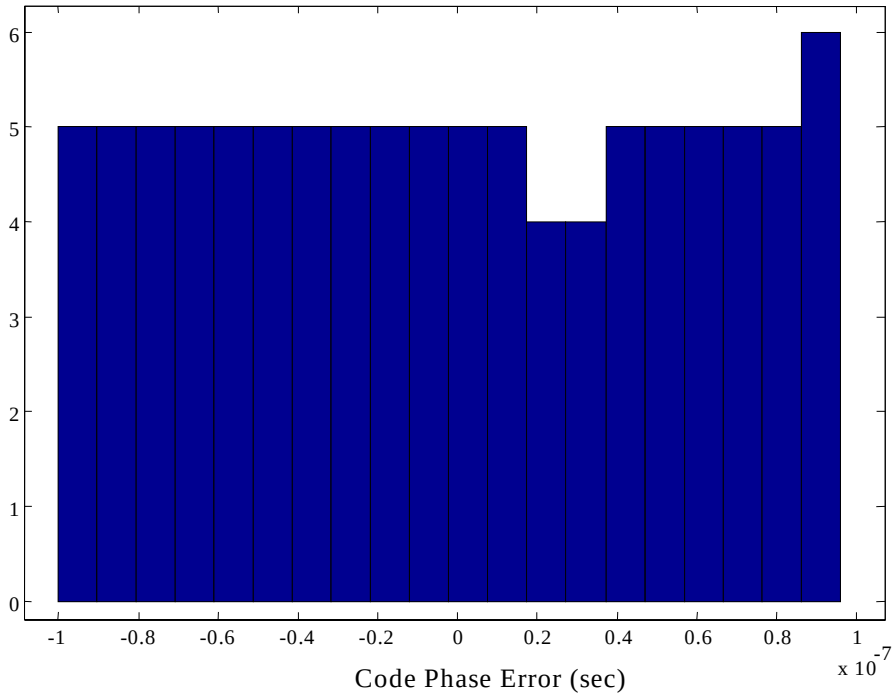

Figure 4-9: Code-Phase Error Using 5000-Point FFT Method.

The carrier phase estimation based on the block processing using the averaging method was also tested. The average accumulated carrier phase error was +0.03 (rad) when computed from Figure 4-10.

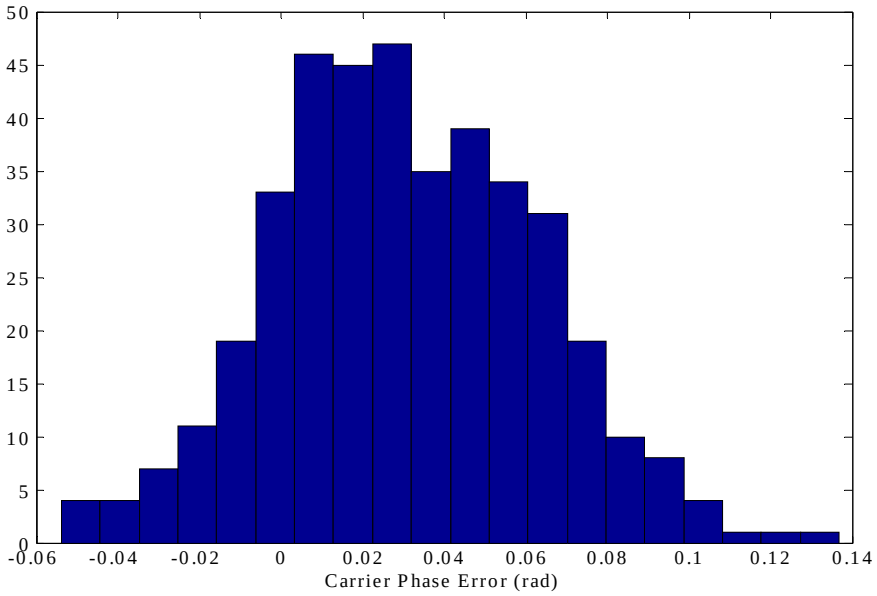

Figure 4-10: Carrier-Phase Error Using Modified-Code Averaging Method

The accumulated phase error was between -4 and +8 degrees. This error was considered acceptable for carrier tracking. However, the limiting factor is the code phase error, which was not acceptable. It needs to be on the level of centimeters. Therefore, the modified-code averaging correlator method was selected for the implementation of the acquisition process only. Another algorithm is needed to replace the tracking loops and the huge block processing system.

4.6 Proposed Architecture for Block Processing Using the Modified-Code Averaging Correlator

When a GPS receiver uses the block processing technique, first it processes blocks of GPS signals by applying the FFT-based correlation. When there is a strong signal, the receiver gets the acquisition estimations based on the 5000-point correlation function for every 1-ms of GPS signal. These estimations are then used to refine the carrier frequency estimation. This refined carrier frequency is then used to remove the carrier from the incoming signal to generate a base-band signal. This base-band signal is applied to the FFT-based correlator. The code phase estimation is refined by applying the triangle fitting for the three strongest correlation values. These values are the nearest points to the real correlation peak. This method (block processing) is a good replacement of the old-fashioned tracking loops.

Each point in the correlation function can be computed in the time domain by accumulating the point-by-point multiplication of the base-band signal (5000 samples) by the 5000 samples of the generated local code with a certain shift in it. This means 5000 multiplications and 4999 additions are required to calculate one correlation point. Calculating all the correlation points requires approximately 5000 times 5000 multiplications and 5000 times 4999 additions. This can be simplified to about 5000 times 4999 additions because the multiplications are by +1 or -1 values only. However, this required number of operations is still very large compared to the FFT-based method. For this reason, the FFT-based correlator is used in the acquisition process since all the correlation points, which represent a full

code-phase search dimension, are covered. However, the FFT-based method is not necessary to refine the code phase and the carrier frequency estimations because only the three correlation points near the peak have to be calculated. Therefore, these points can be calculated using the serial time-domain correlators. Thus, only three times 5000 additions and/or subtractions are required.

While the modified-code averaging correlator is accurate enough to replace the 5000-point FFT method for acquisition, the use of serial correlators to refine these estimations is logical. Therefore, the block processing concept is still valid and the advancement on GPS block processing is applied.

The block processing concept using the averaging correlator and the three serial correlators is used to replace the traditional acquisition and tracking loops. Figure 4.11 shows a block diagram of the proposed architecture of the GPS block processor using the averaging correlation method.

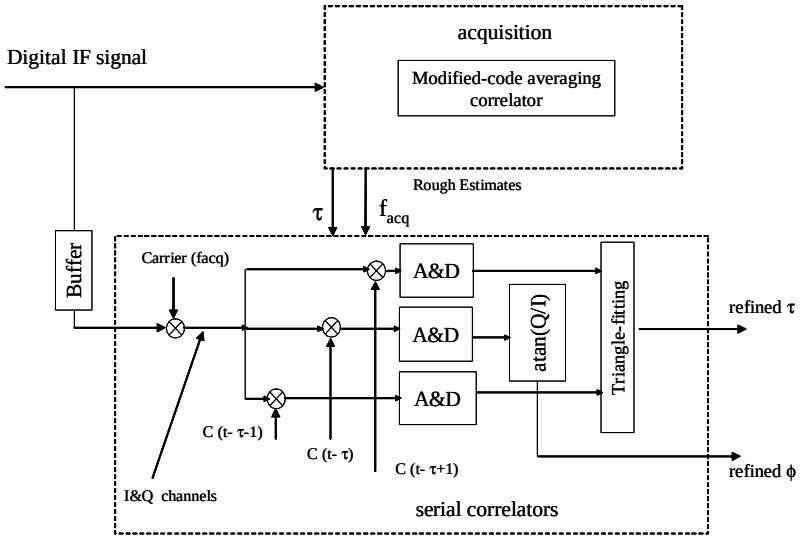

Figure 4-11: Proposed Architecture for Block Processing Using Modified-Code Averaging Method

This new method uses the frequency domain and the time domain correlators. It provides fast acquisition, with the same level of block processing code-phase and carrier frequency estimations losing not more than 0.5 (dBs) of energy on the average. This makes the method a good candidate for hardware implementation in a parallel processing platform such as an FPGA. Additionally, real-time block processing of GPS signals is feasible with this implementation. All the related implementation issues to build such a GPS processor are presented in Chapter 5.

Chapter 5

FPGA Implementation of Acquisition and Tracking Processes

5.1 Introduction

Chapter four showed that the modified-code averaging correlator for the GPS block processing is the proposed algorithm for the implementation of the acquisition process. This was due to the low loss of peak-to-second-peak value with considerable reduction in computation count. In addition, the described method has functions that their implementation on FPGAs can be easily achieved using FPGA-optimized Xilinx™ cores such as the fast Fourier transforms (FFTs) and multipliers. Therefore, the FPGA hardware implementation of the modified-code averaging correlator will be used for acquisition. As was mentioned previously, the averaging method is not good enough to track GPS signals. Therefore, the standard serial early-prompt-late (EPL) correlator will be implemented for block processing to replace the tracking loops. The hardware implementations of both correlators (averaging and serial E-P-L correlators) along with the necessary functions are presented in the next sections.

5.2 GPS Block Processing Algorithm for Hardware Implementation

In the previous chapter, it was shown that the difference in computational effort of the averaging correlator with the modified-code approach and the regular 5,000-point FFT-based correlators is small. However, the implementation of five FFTs that are 1024-point each is much simpler than designing a single 5000-point FFT. Therefore, the averaging correlator is chosen for the implementation of the acquisition process.

A block diagram of the averaging-correlator-based acquisition is shown in Figure 5-1.

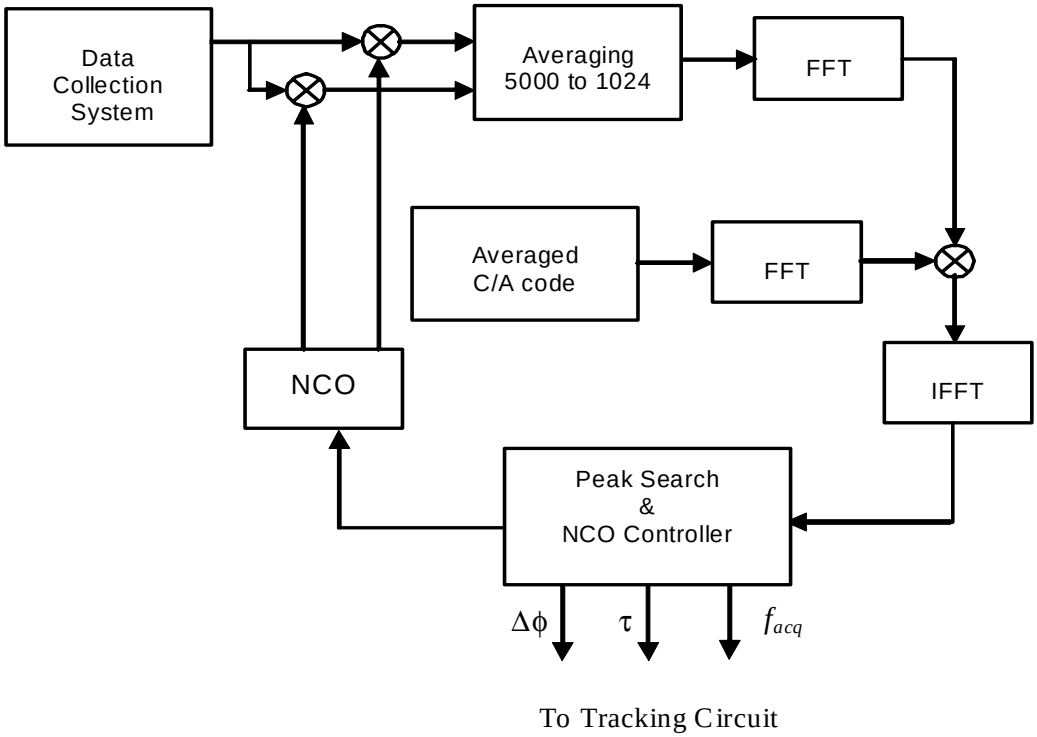

Figure 5-1: Acquisition Using Averaging-Correlator

The samples are first multiplied by in-phase (I) and quad-phase (Q) components of the carrier signal. The I and Q channels are each averaged to 1024 points. Then they are converted to the frequency domain using 1024-point complex FFT and multiplied by the conjugate of the FFT of the local modified-code. A 1024-point complex IFFT is used then to return to the time domain. A peak searcher inspects the 1024 outputs of the IFFT and stores the location of the peak and its value. This process is repeated four more times, each with a different starting point, as described in the previous section. After checking all five loops, the peak searcher compares the peak value to a threshold in order to determine if the peak (or a GPS code) is detected. If the searcher does not detect a peak, then a new search cell with a different frequency bin is inspected using the above described process. This is repeated until a true peak is found or until all frequency search bins are completed. When the GPS signal is acquired, the frequency is selected and the search is conducted only in the code-phase dimension.

When the signal is acquired, the peak searcher sends the real and imaginary values responsible for the peak (I and Q values) to the phase estimator which computes the carrier phase using the “ATAN” function. Estimating the carrier phase for two successive 1-ms blocks and then finding the change in carrier phase will be used to refine the carrier frequency. The rough code phase and the refined frequency are then fed to the serial correlators block to wipe off the code and the carrier. Based on the serial correlators, a refined code phase and a more refined carrier frequency can be achieved. In conclusion, the targeted GPS block processor implementation is simply an averaging-correlator aiding serial correlators and no

ordinary tracking loops are involved. Therefore this type of implementation will be called “averaging-correlator GPS block processor.”

5.3 Required Components for the Implementation of the Averaging-Correlator GPS Block Processing

In this section, the necessary components for implementing the proposed architecture are described along with their implementation methods. A numerically controlled oscillator, a code generator, a complex multiplier, an FFT and its inverse, and a peak searcher are some of the required components. The implementation method for each required component has to be studied carefully because they can affect the overall size, speed, and accuracy of the whole design. The functionality and the implementation of each component is presented in the following sub-sections.

5.3.1 Numerically Controlled Oscillator (NCO)

Numerically controlled oscillators (NCOs) have gained a lot of attention in digital design because of their importance in applications such as communication and especially in designing digital receivers. An NCO can be constructed by three components. They are an accumulator, a register, and a look up table. A typical NCO implementation is shown in Figure 5-2.

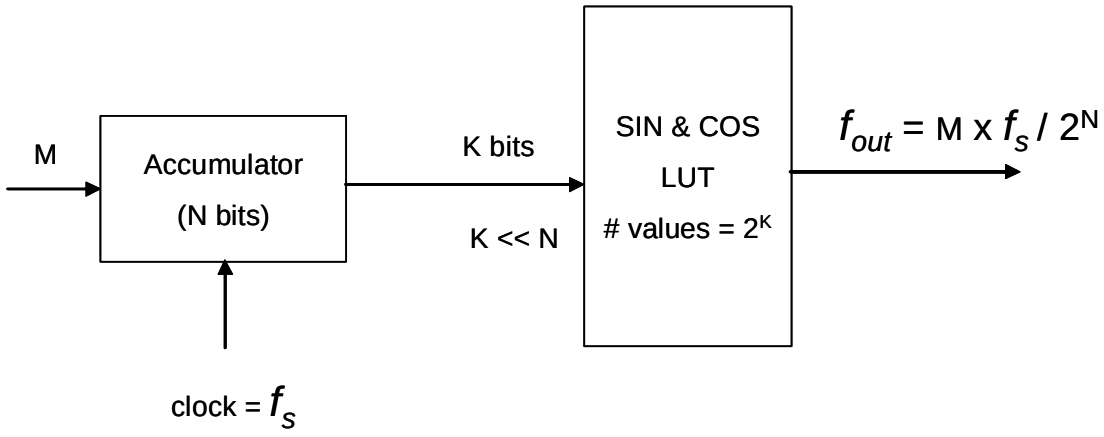

Figure 5-2: Typical NCO Implementation.

The accumulator adds a certain number, which is called a "step", each reference clock cycle and stores the sum in the register. This is a stair-up function that will overflow at certain time to repeat the process and effectively works as a frequency divider. The stair-up period is considered the output frequency period of the NCO. Based on the value of the register, the NCO uses the LUT to select the appropriate SIN or COS value. Since GPS C/A code acquisition and tracking require only three levels of SIN wave and COS wave (Gunawardena, 2000), then this type of implementation is used instead of Xilinx NCO core. When more levels of SIN wave are necessary, then the Xilinx core is the choice because it is optimized for Xilinx FPGAs (Xilinx, 1999). An NCO similar to the one shown in Fig 5-2 was implemented by Zhen (see Appendix C.2.1).

5.3.2 Carrier-Wipe-off

The averaging method is used after removing the carrier. Therefore, a carrier-wipe-off circuit is implemented in front of the averager as shown in Fig 5-3. Generally the carrier wipe-off circuit is implemented as a multiplier that multiplies the incoming sample by the local carrier that comes from the NCO.

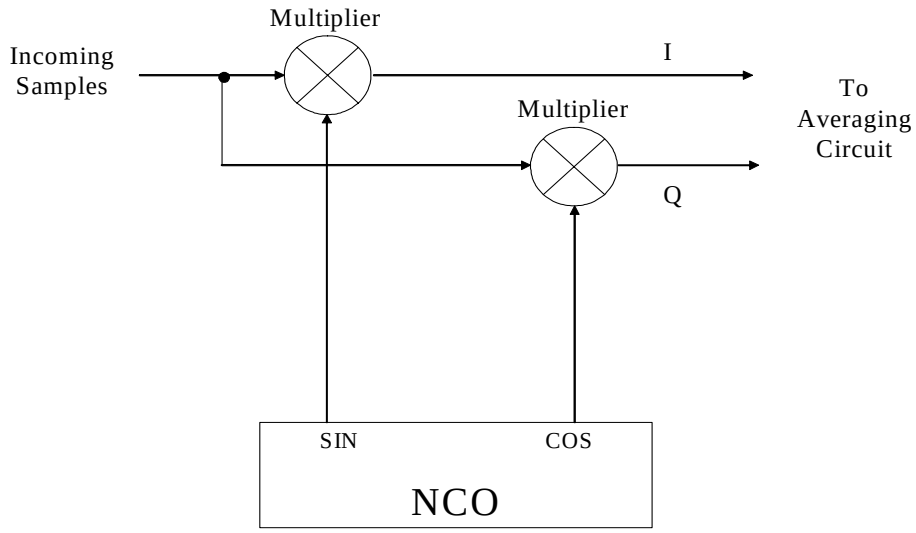

Figure 5-3: Carrier Wipe-off Circuit

Since the NCO output in this dissertation is only one of three values (1,0, or -1), then the multiplier can be replaced by a small circuit. The circuit can be described as a 3x1 multiplexer. It takes the input sample and passes it if the NCO value is one. If the NCO value is -1 then it inverts the sample before passing it out. In the remaining case(s) the NCO value is zero, so the circuit is grounded. The only required operation is the signal inversion whenever there is a negative value of a local carrier. Inversion here means a 2's complement operation that requires one

addition operation plus complementing. Since complementing does not require any extra hardware resources in Xilinx FPGAs (Xilinx, 2002), the carrier wipe off can be constructed using a multiplexer and an adder. This implementation is efficient and small compared to a multiplier.

5.3.3 The Averager

After the carrier is removed the averager circuit performs the operations required to down sample (or average) the incoming 5000 samples to 1024 averaged points. Figure 5-4 shows a simplified circuit of the averager.

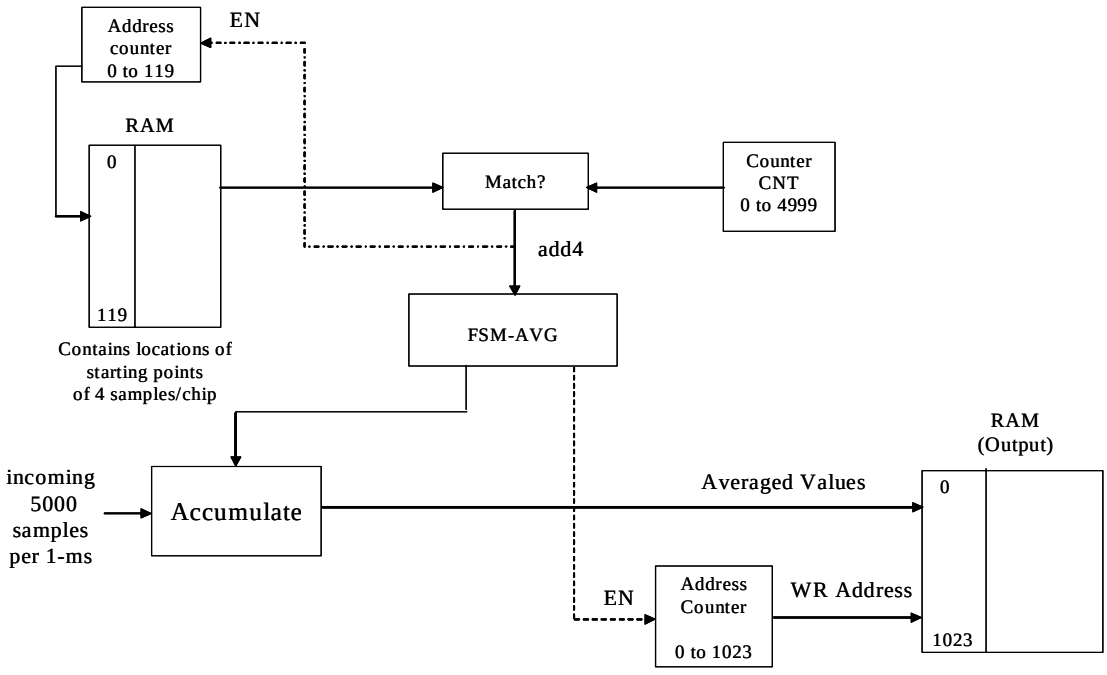

Figure 5-4: Simplified Circuit for the Averager

The averaging algorithm chooses four or five samples per chip to average. Thus, 120 chips will be represented by averaging four samples, while the remain-

ing 904 chips are represented by averaging five samples. Therefore, the averaging algorithm will average five samples most of the time. However, in the 120 cases that averaging four samples operations are required, the algorithm will average four samples. If the circuit normally averages four samples unless average-five occurs, then it is required that the locations of average-five operations are stored. This means a RAM of 904 rows is required. However, if the opposite is the case, then only 120 rows are required. This means it is cheaper to always average five samples unless average-four operation is required.

A 120x13 RAM is used to store the locations where the average-four occurs. A 13-bit counter is used to count all of the 5000 samples. When the counter and the value of the RAM match then the accumulator averages the next four samples. At the same time the address of the RAM is incremented to point to the location of the next average-four operation. When the value of the RAM and the counter does not match, then the accumulator averages five samples.

Averaging requires additions and a division by five or four. Division by four is simple in digital systems, because it is simply a shift by two. However, dividing by five is not a simple operation. Therefore, the averager will not do the division and it will only do the additions. This approximation has a very small effect on the accuracy of the averaging method, but it makes the circuit very simple to implement. It is important to know that this simplification is useful, because each possible simplification of any component of the acquisition architecture would increase the possibility of implementing a complete acquisition process in a small silicon area. This architecture needs two averagers, one for the in-phase and one

for the quad-phase. The averaged values are stored in a RAM. The FFT circuit then takes over and processes them as described in the next section.

5.3.4 Fast Fourier Transform (FFT) and Its Inverse

The fast Fourier transform (or the FFT) is required twice and the inverse FFT (IFFT) is required once in the circular correlator. One FFT is for the incoming samples and the other is for the local code. The FFT of the local code can be replaced by a small RAM that contains the pre-calculated values of the local code in the frequency domain. Therefore, the correlator implementation will require only one FFT and one IFFT. Hardware implementation of FFT has been investigated for about thirty five years (Yubin,1996, Nussbaumer, 1982, and Rader, 1973). Some of the possible hardware architectures were presented in (Smith W., 1995 and Burrus, 1985).

Many algorithms for approximating the FFT function have been developed. One can compute the FFT using the Walsh Hadamard transform (Beauchamp, 1984), Chirp z-transform (Rabiner, 1969 and Davenport, 1991), or CORDIC (Sarmiento, 1998). However, since the implementation is targeting the Xilinx Virtex FPGA, then a careful design process should be applied because the FPGA resources are not always a good choice for implementing a component. Therefore, the optimization of the hardware implementation of a component is necessary. Using the so called "IP cores" whenever possible is the best method for FPGA implementation because they are optimized by Xilinx and other companies targeting Xilinx FPGAs.

One of the available Xilinx cores is the FFT/IFFT core. This core is optimized for Xilinx Virtex and can perform 1024-point complex FFT in about 50 microseconds using an 80MHz clock (Xilinx, 2000). The core can be used as FFT or IFFT by selecting a particular configuration pin. It takes a 32-bit number as 16 bits for real part and 16 bits for imaginary part. The output is also in the same complex number representation. Since the operations are fixed-point, then overflow may occur in the butterfly structure. Therefore, truncation and scaling steps are done inside the core to make sure that there is no overflow. Scaling operations require that the output of the core is scaled by 2^N (or by 2^(N+1)). This does not create a problem for the IFFT since this scaling is included in its equation. However, for the FFT it means that the input values need to be large enough that the core can provide meaningful results. This problem is partially solved in the design since the averaging implementation performs some scaling of the incoming samples. Therefore, this problem is not significant in this work. More information about the FFT core can be found in (Xilinx, 2000).

5.3.5 Local Code Component

Based on the method of code-averaging, the local code generation is not an easy task to implement in an FPGA because of the required processes. The C/A code is normally generated as shown in Figure 5-5. However, this architecture will generate only 1023 chips per C/A code period. Averaging-code method requires the C/A code to be up-sampled to 5000 bits and then averaged to produce 1024 points. A 1024-point FFT is then conducted and the conjugate function is applied. The implementation of such an algorithm is difficult in the hardware and occupies

large design area. Since the RAM resources in the Virtex FPGAs are large enough for the required storage of data during the acquisition process, the averaged-code and all the other described processes are performed externally, i.e. in a PC. The results are then stored in an FPGA RAM. This RAM is then used in the local code component to provide the right code value when it is requested. Using such a RAM will ease the design and save considerable FPGA design space.

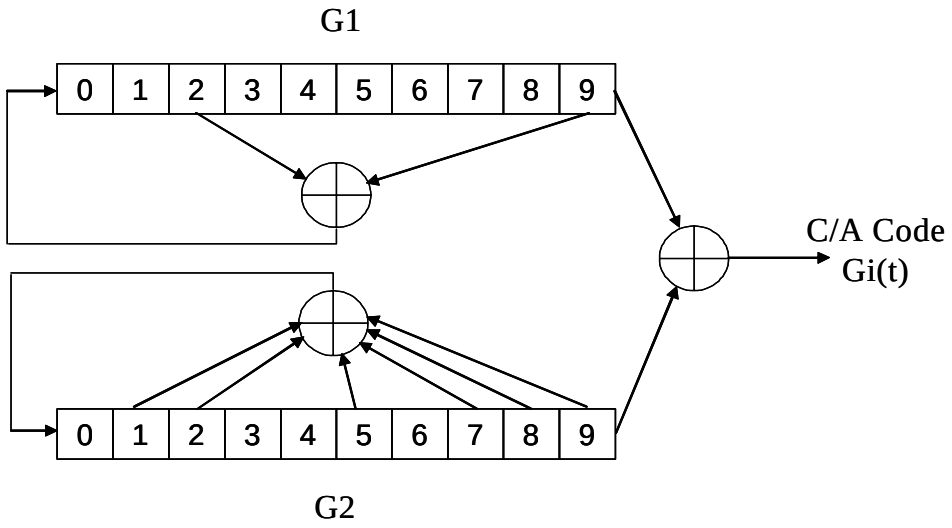

Figure 5-5: Local Code Generator

The size of the required RAM is 1024 in depth, but the word size (or the width of the RAM) is flexible based on the data characteristics. The data that will be stored in this RAM is the frequency domain of the averaged C/A code. The distribution of the values is shown in Figure 5-6. Since the dynamic range of these values is between -63 and 63, seven bits are enough to represent them. Therefore, a total of 14 bits is required since the results are in complex numbers representation. For this reason, a RAM with size of 1024 rows by 14 bits is chosen for the implementation.

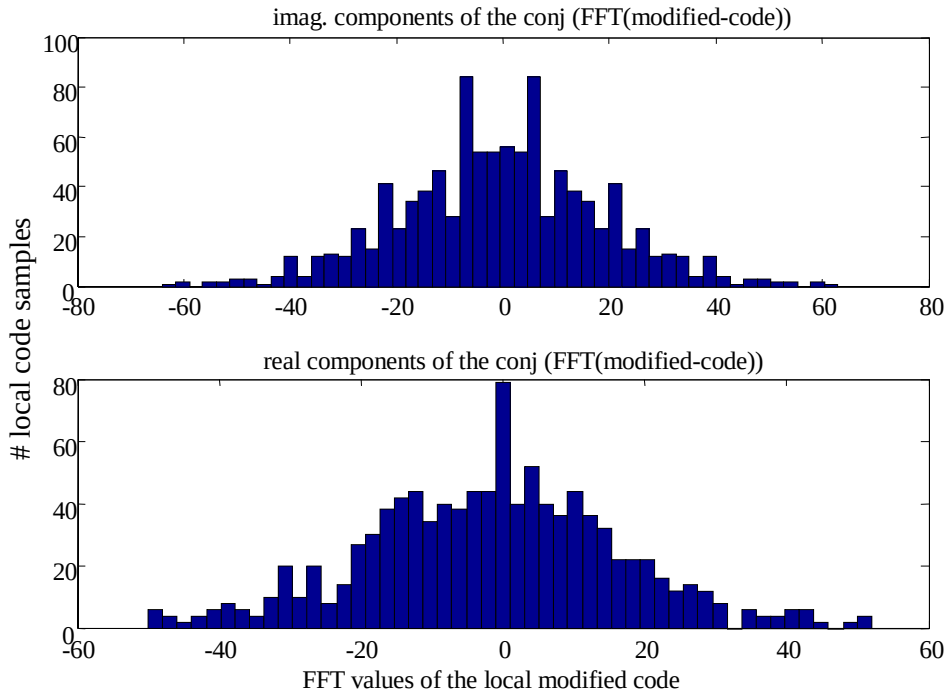

Figure 5-6: Distribution of the Values of the FFT of the Local Modified-Code

Since the Xilinx core generator has the ability to generate optimized cores ready to be used in the designer applications, it was used to implement two 1024-by-7 RAMs (one for real and one for imaginary values). The pre-calculated values are also initiated in the RAM core, so there is no need to load data into RAM during the acquisition process unless this design is time-shared to acquire multiple satellites. For this work, there will be no time-sharing since the goal is to develop and test the design.

5.3.6 Complex Multiplier

Digital multiplication can be processed by shift and adds as described in (Roth, 1992 and Smith, 1996). Different architectures have been presented in the literature (Smith, 1995 and Starzyk, 2000). Each architecture is suitable for a specific application and selected technology. The multiplications in this research are applied to two sequences. These sequences are the frequency domain values of both the incoming data and the local code. Therefore, the multiplier should provide one result every clock cycle since the input values come every clock cycle. However, a latency of few clock cycles would not harm the design. The latency will occur because we are not implementing a real multiplier, which is a relatively easy task, but a complex multiplier, which requires serial multiplications and additions.

Each complex multiplication can be directly represented by four real multiplications and two real additions (or subtractions).

$$(x_r + jx_i) \times (y_i + jy_i) = \quad \textbf{(5-1)}$$

$$((x_r \times y_r) - (x_i \times y_i)) + j((x_r \times y_i) + (x_i \times y_r))$$

Since the real multiplier requires more space and/or time compared with real additions, then another efficient implementation is used. One can perform a complex multiplication by using three real multiplications and five real additions, or subtractions (Smith, 1995, and Myers, 1990). This saves some space compared

with the direct implementation. The operation can be mathematically written as shown in equation 5-2:

$$(x_r + jx_i) \times (y_r + jy_i) = \quad \textbf{(5-2)}$$

$$(y_r \times (x_r + x_i) - (y_r + y_i) \times x_i) + j(y_i \times (x_r - x_i) + (y_r + y_i) \times x_i)$$

Figure 5-7 shows a diagram of the efficient complex multiplication circuit. The implementation of such a circuit is required to use combinational real multipliers and combinational real adders to keep the intermediate results synchronized.

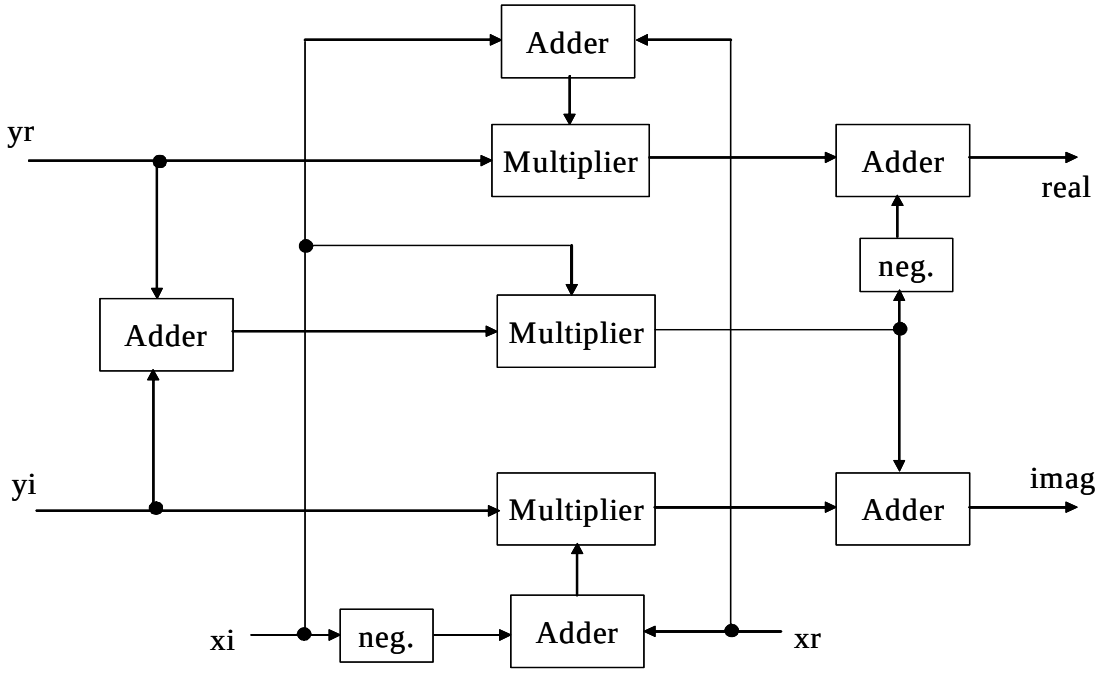

Figure 5-7: Efficient Implementation of a Complex Multiplier

However, the overall efficient architecture of the complex multiplier is considered large. Therefore the routing of the intermediate signals may cause some synchronization problems. Such a problem would put a longer delay on one bit of an intermediate value than the delays on the other bits. Thus, a generated error will accumulate and cause an error in the complex multiplication. This error is critical for the chosen FPGA implementation because routing signals in FPGA are not predictable. One possible solution is to place some registers to hold the intermediate results of these combinational circuits. This technique causes the results of the complex multiplication to be generated after a few clock cycles which in turn increases the latency. Figure 5-8 shows the FPGA-based architecture of the complex multiplier.

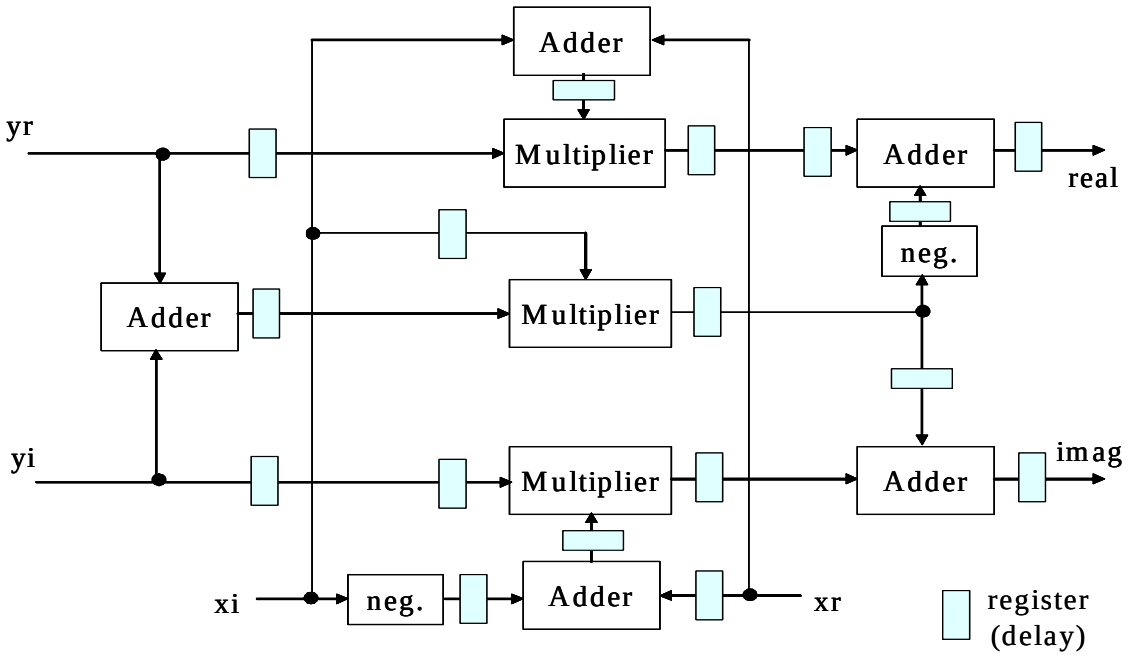

Figure 5-8: FPGA-Based Architecture of the Complex Multiplier

Further reduction of the circuit size can be achieved by investigating the data path sizes. Matlab simulations showed that the outputs of the FFT of the incoming data require only six bits of representation because the FFT output is scaled by 1024. The output of the complex multiplier should be represented in 16 bits for each part of the complex number. Therefore, a Matlab simulation of possible truncation and scaling stages was carried out. The results showed that the frequency components of the incoming GPS data can be scaled by four bits which is similar to multiplication by 16. Therefore, the six bits of each component of the complex number is increased by four and becomes 10 bits each. This means that the complex multiplier takes two complex numbers represented by 20 bits and 14 bits respectively and then produces a complex number with 16-bit real and 16-bit imaginary parts. After truncating and scaling the necessary data paths, modification to the sizes of the adders and the real multipliers were made. Figure 5-9 shows the implemented complex multiplier using these size-modified components and paths. The results of the complex multiplier are then fed to the Inverse FFT component to calculate the correlation function.

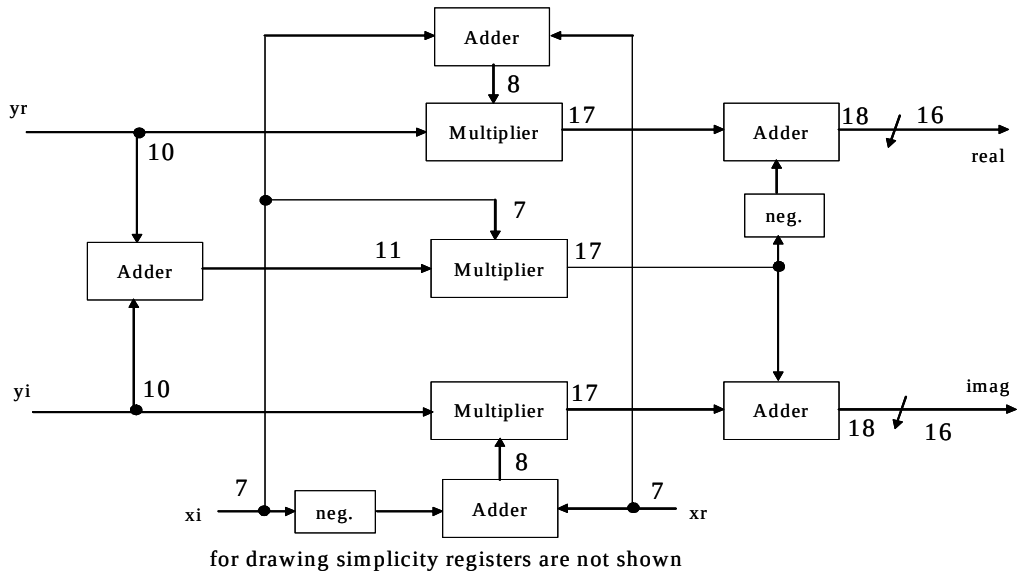

Figure 5-9: The Implemented Complex Multiplier

5.3.7 Peak Searcher

After the Inverse Fourier transform is performed, the 1024 complex results are searched for the strongest energy (correlation peak). This process will be repeated each time the averaging method starts from a different starting point. Thus the peak searcher must search all of the 5x1024 (or 5120) complex numbers to find the location of the peak (code phase), the peak value, and the carrier phase. Since the correlation values come serially from the IFFT block, the serial mechanism is not critical for the processing time. Therefore, a serial peak searcher will be used.

A simplified diagram of the peak searcher is shown in Figure 5-10. First the incoming complex number is held in a register. The most important information of the peak searcher is not the value of the peak itself, but its location. Rather than computing the absolute value of the complex number, which is difficult to do in hardware, a simplification can be used by squaring both real and imaginary values and then summing them to have the magnitude squared. Thus, the circuit can be simplified without affecting the desired information. For this reason, two real multipliers and an adder are used to compute peak values. Initially, the stored peak value is set to zero.

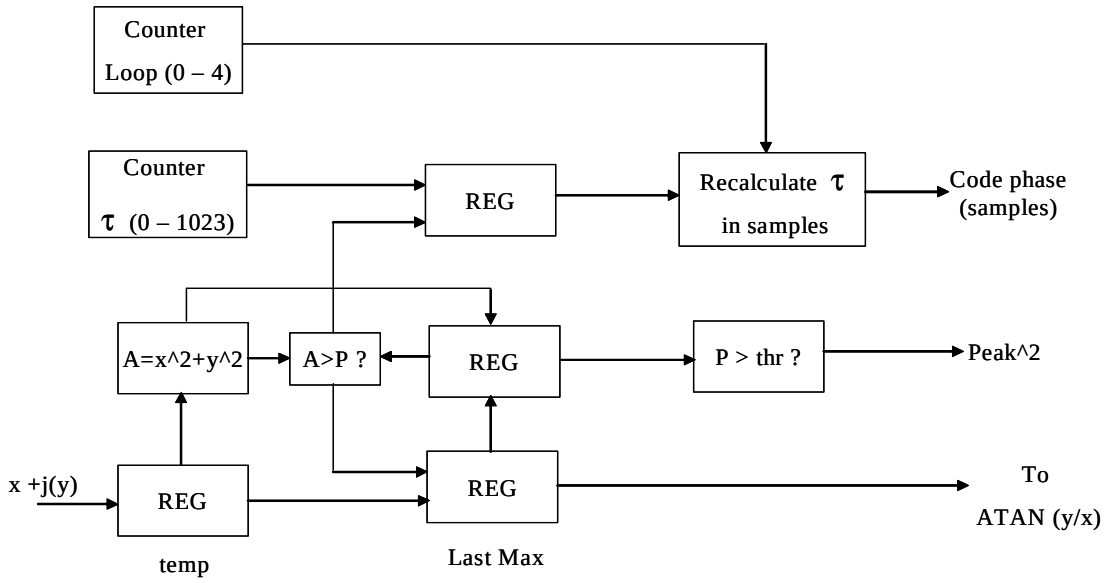

Figure 5-10: Simplified Diagram of the Peak Searcher

Each clock cycle will bring a new squared magnitude value. The new value is compared against the current stored peak. If the new value is larger than the stored peak, it will be stored and its location is also stored. The location value is

computed using a counter that counts the incoming values from the correlator. Two counters are used in the peak searcher because it is required to check the location peak in each averaged sequence and to check which starting point (which sequence) is responsible for the peak. The starting point counter counts from zero to 4, while the other counter counts from zero to 1023 to search the IFFT output of each loop of five. The code phase in samples can be computed by using the following formula

$$\tau = \left(\tau_{1024} \times \frac{5000}{1024} + k\right) \text{ (samples)} \qquad \textbf{(5-3)}$$

where τ is the code phase, τ_{1024} is the peak location in 1024 samples, and k is the starting point count. This can be implemented using an adder and a LUT. The LUT contains all the 1024 possible multiplication results.

Another calculation that is necessary for the peak searcher to do, is the calculation of the carrier phase. The next section covers the implementation options that are investigated to use for the carrier phase estimation in the acquisition process.

5.3.8 Carrier Phase Estimator

The carrier phase is an important measure that the acquisition process provides. It indicates where the GPS information is concentrated. If the GPS information is concentrated in the in-phase components, then the carrier phase is almost zero. Whereas, if the majority of the information is in the quad-phase components, then the carrier phase is almost $90°$. However, in many cases the GPS information

is divided between the in-phase and quad-phase components. Therefore, the acquisition process must provide the carrier phase information to the tracking process which is replaced by serial correlators in this work. These serial correlators use the carrier phase estimation to synchronize the carrier. Therefore, the GPS information is concentrated in the in-phase components. Two implementable carrier phase calculation methods are presented in the next sections.

5.3.8.1 Simple Digital ATAN

A simple calculation of the ATAN function for the acquisition process can be achieved by one addition and one division. In the acquisition process the carrier phase is calculated by computing ATAN(Q / I). However, a simplification is done by approximating the ATAN function. The function ATAN(Q / I) can be approximated by computing 90*(Q/ (Q+I)). Figure 5-11 shows the ATAN function and its approximation. This method requires one addition and one division plus scaling by 90. It calculates the phase when it is bounded by zero and 90. However, it can be extended to cover the whole phase plane with a small circuit that converts the calculated value to its correct form based on the signs of I and Q.

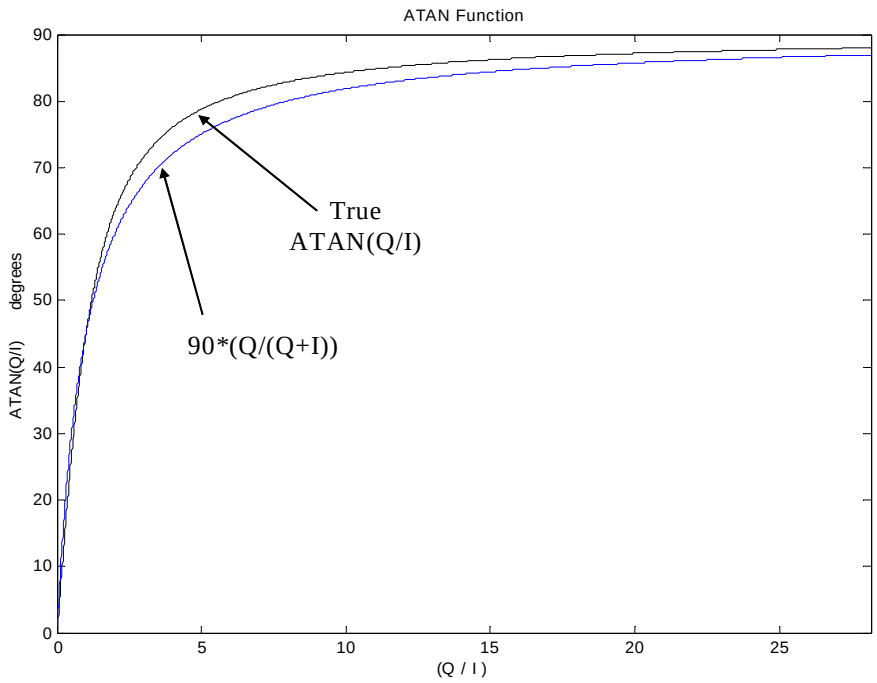

Figure 5-11: ATAN Function and Its Approximation

Using this type of approximation in hardware is fast and requires a very small area compared with the CORDIC method which is presented in the next section. The only disadvantage of this method is the approximation error which can vary from -4 to +4 degrees as can be seen in Figure 5-12. However, this error is acceptable for the acquisition process. Since the averaging method causes additional 4 to 8 degrees error then the total error can vary from -8 to +12 degrees. The approximation error can be reduced by using lookup tables but it is difficult to remove all the error using small lookup tables. So one needs to use either a large look up table or a more accurate algorithm. The latter solution can be implemented

using the CORDIC method, since the calculation of the carrier phase must be very accurate for the tracking process lock-in requirement.

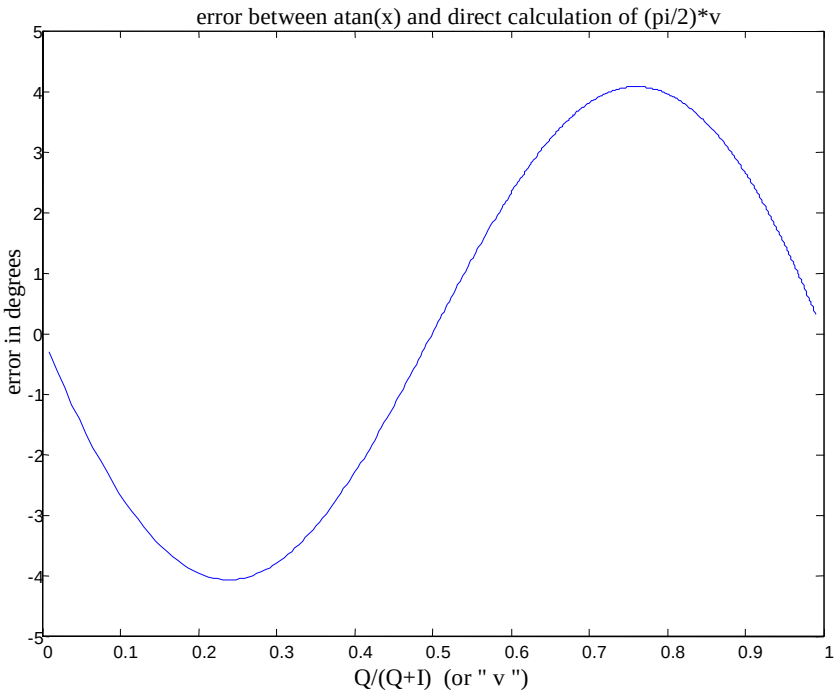

Figure 5-12: ATAN Function's Approximation Error

5.3.8.2 Computing ATAN Function Using CORDIC

Coordinate Rotation Digital Computer or CORDIC is a method for performing elementary operations such as trigonometric functions. It calculates the trigonometric functions by rotating the coordinates through angle steps until the angle goes to zero (Kharrat, 2001). It can be performed using add and shift operations only. CORDIC is a well known method that is usually used in hardware because of its simplicity and compact size (Kantabutra, 1999). Also, it is considered a very accurate method for computation of elementary functions and hence it is widely

used in calculators. No description for CORDIC will be discussed in this work. The CORDIC algorithm is implemented in the FPGA using the available rectangular-to-polar open core from (opencores.com).

5.3.9 Time-Domain Serial Correlators

Since the acquisition process gives a rough estimate of the code phase and carrier phase and frequency every millisecond, then it would be easier to use the serial-correlator based tracking process. The tracking process will re-generate a refined version of the carrier every millisecond based on the acquisition information. It uses the early-prompt-late type of architecture to track the GPS signal. Using the triangle fitting described in (Uijt, 1998), a refined code phase can be computed and used later to demodulate the Navigational data. A simplified block diagram of the serial correlators based process is shown in Figure 5-13.

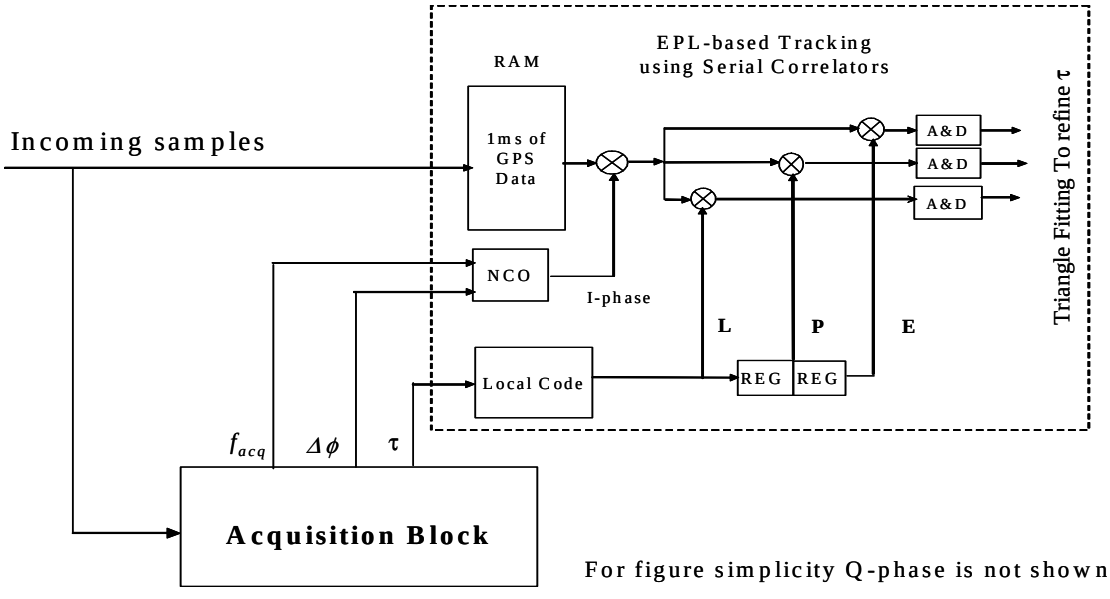

Figure 5-13: Serial Correlators Based Process

Using serial correlators will not affect the real-time performance of the receiver because the processing clock can be as high as 200 MHz. However, a good realistic estimate of the clock speed is about 45-50 MHz since the acquisition and tracking architectures are too large to implement in one FPGA. The large design will fit in the FPGA, but the problem is in the unpredictable routing which leads to clock speed limitations.

Since the correlator needs approximately 5000 clock cycles, the tracking process will take approximately 100 microseconds for 50MHz clock. The advantage of using the serial correlators is that they use a very small area of the FPGA and require only additions (or accumulations). The NCO is similar to the one used in the acquisition process. Feeding back the corrections of the carrier based on the last estimations using the block processing concept produces a refined version of the carrier. The CORDIC or the implemented ATAN architecture is time shared between acquisition and tracking processes. Triangle fitting equation is simpler to be done inside the PC, but it is preferred to have all the computations performed inside the FPGA. In this case, the PC will be responsible for collecting the block processing values every 1 millisecond.

5.4 FPGA Implementation of GPS Block Processor

The FPGA platform used has a Virtex FPGA that provides 800k logic gates. This FPGA cannot implement the whole block processing system. Therefore, the system was partitioned into smaller parts as shown in Figure 5-14. Each block is mapped to the FPGA and then is tested separately.

In the first part, the data are read from the data collection memory to apply to the averaging method. The data are multiplied by the SIN and COS components of the carrier coming from the NCO. The carrier wipe-off multiplication is replaced by the adder and the multiplexer.

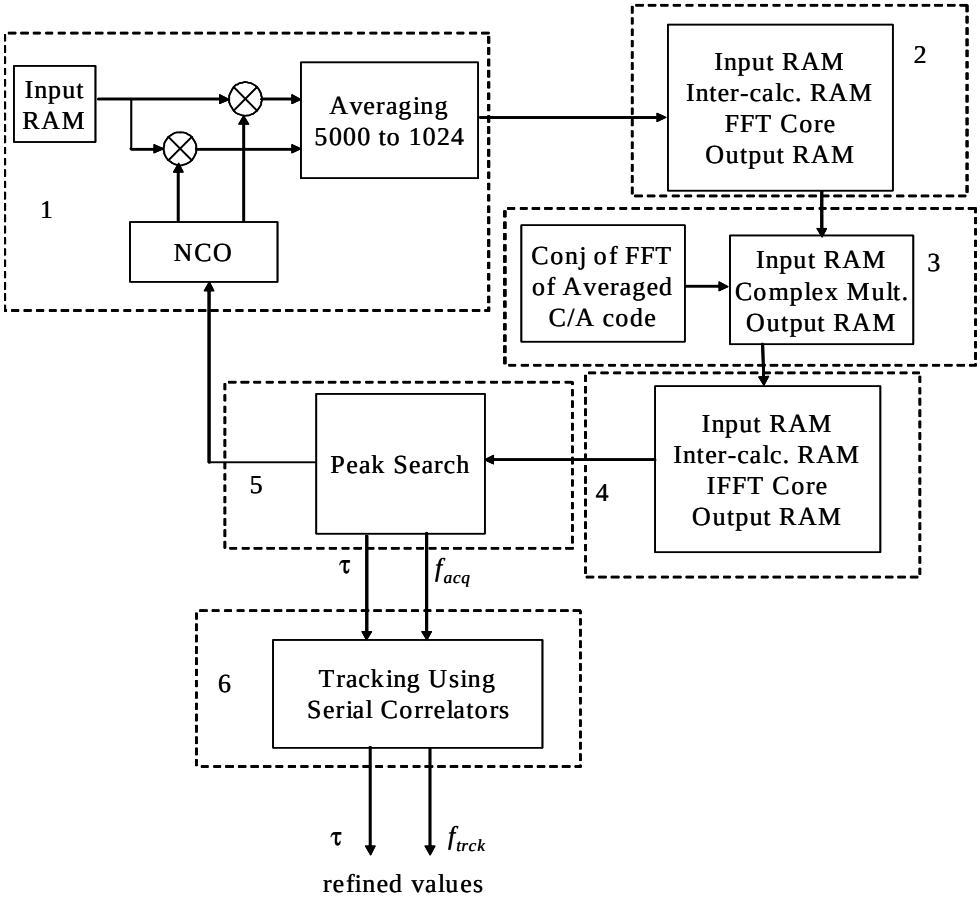

Figure 5-14: System Partitioned into Small Components

Then the averaging circuit takes the 5000 baseband values and averages them to 1024 values. The resulting values are stored in the output RAM. The carrier wipe-off and averager block diagram is shown in Figure 5-15.

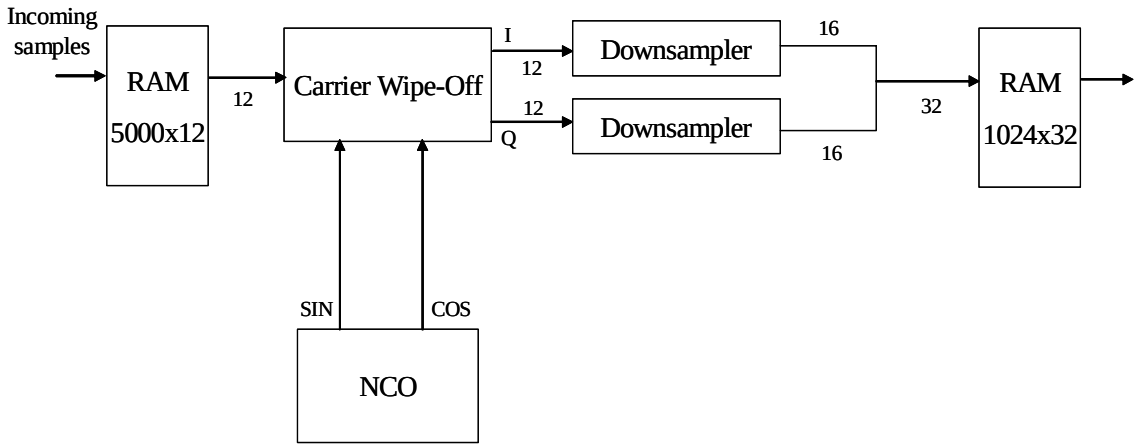

Figure 5-15: Carrier Wipe-off and Averager (Downsampler)

The second section is the FFT component that reads 1024 averaged values and provides their spectrum. The averaged signal data are stored in the memory by the preceding operation. The FFT core is implemented using the SMS-implementation as described in (Xilinx, 2000) then the outputs of the FFT are stored in an output RAM. Figure 5-16 shows the block diagram of the FFT component.

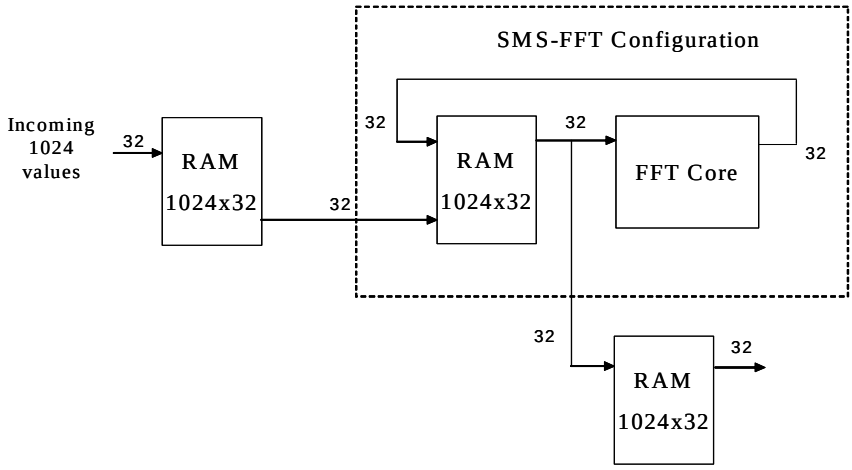

Figure 5-16: FFT Block

The third component is the complex multiplier of both the data and the code. The data are the output of the FFT component and are available in the memory. These values are read and then scaled by 16 for each channel using a shift-by-four operation. Therefore, the scaled data becomes 10 bits per channel, or a 20-bit complex number. In this operation, four ones are padded instead of zeros in the case of negative numbers to keep the 2's complement representation valid. Then these values and the local code values from the other RAM are both fed to the complex multiplier. The output values are stored in the output RAM. Figure 5-17 shows the implemented fequency-domain complex multiplier.

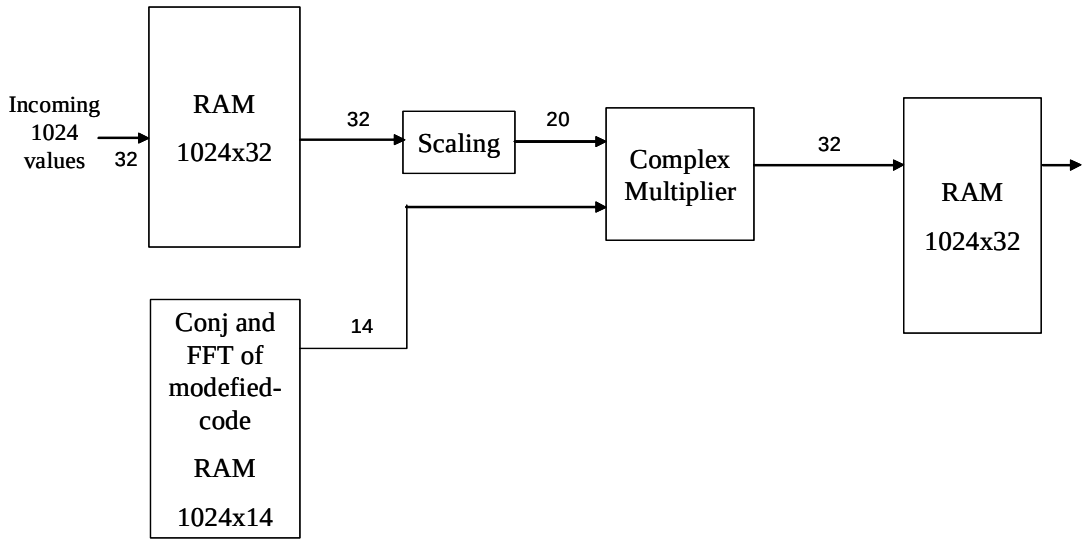

Figure 5-17: Frequency Domain Multiplier Block

The next part, is the IFFT component. Similar to the FFT component, the IFFT is built using the SMS-implementation. Therefore, the IFFT core reads the results of the complex multiplications which are assumed to be in the input RAM

of this part. The output of the IFFT is stored in the output RAM. The block diagram for this component is similar to the FFT component with FFT replaced by IFFT operation.

The fifth component is the peak searcher block. In this section, the results of the IFFT component were stored in the input RAM. The values are read from the RAM sequentially and the peak is searched using the peak searcher architecture described earlier in this chapter (Figure 5-10). The output of the peak searcher are stored in registers since they have two values, the peak location and the real and imaginary values of the IFFT point responsible for the peak (I and Q). The I and Q values are fed to the ATAN component to calculate the carrier phase. The ATAN function was excluded from the peak searcher circuit to make it accessible for both acquisition and tracking circuits. Figure 5-18 shows the implemented peak searcher part.

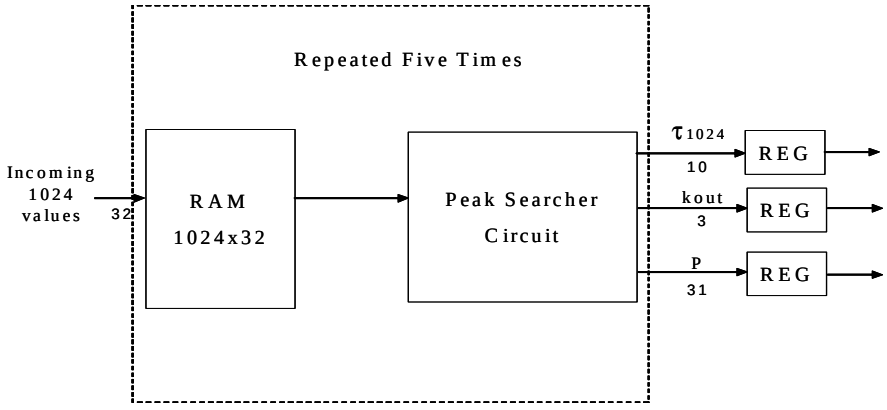

Figure 5-18: Peak Searcher Block

Finally, the serial-correlator section is implemented. This section assumes that the current 1-ms GPS samples are stored in the input RAM. The samples are read sequentially and multiplied by the NCO's sin and cos components. The mul-

tiplication is implemented by an adder and a multiplexer as described earlier. The base-band results then enter the E-P-L serial correlators component where the samples are multiplied by three shifted copies of the local code. These copies are late, prompt, and early copies of the code. This section assumes that the local code is upsampled to 5000 samples in PC and then stored in a RAM during FPGA configuration. The early, prompt, and late code copies are generated with the guidance of the rough code estimation that is already stored in a register. Figure 5-19 shows the implemented architecture for the serial-correlators block.

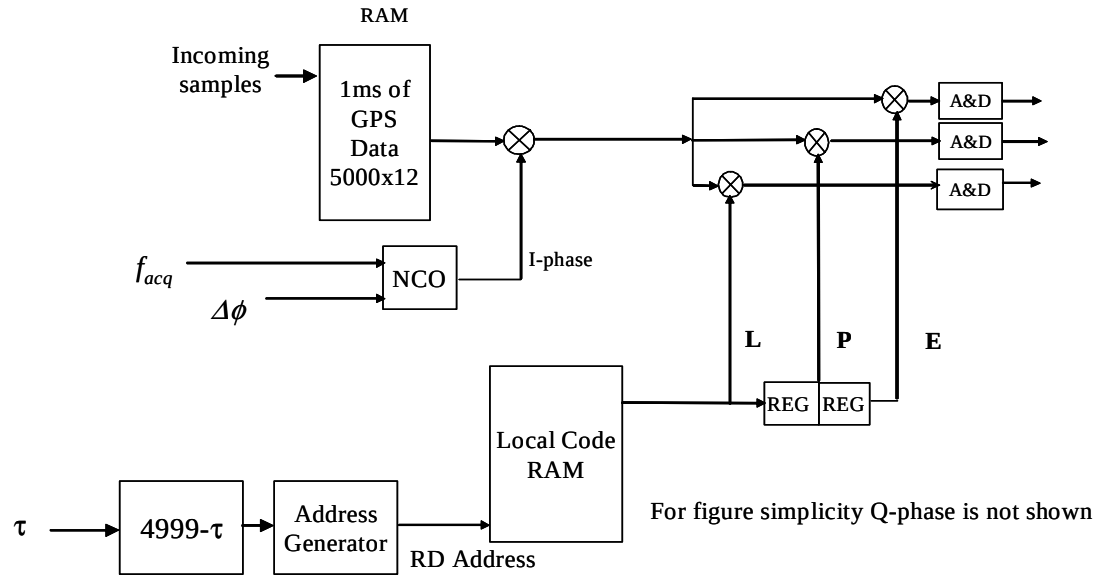

Figure 5-19: Estimator Block (Serial Correlators)

The serial correlators provide the three points around the peak. These values are then used with the triangle fitting and carrier phase estimator to provide accurate estimations of code phase and carrier frequency.

5.5 Overall Performance and Discussion of the Results

The description of how the whole architecture was partitioned for implementation was presented in the previous section. Each block was mapped to the FPGA and tested separately. Figure 5-20 shows the mapped implementation of the data averager and carrier wipe-off partition. One can see how the routing was unpredictable. It uses routing resources from left to right and from top to bottom even though the used logic resources occupy small area.

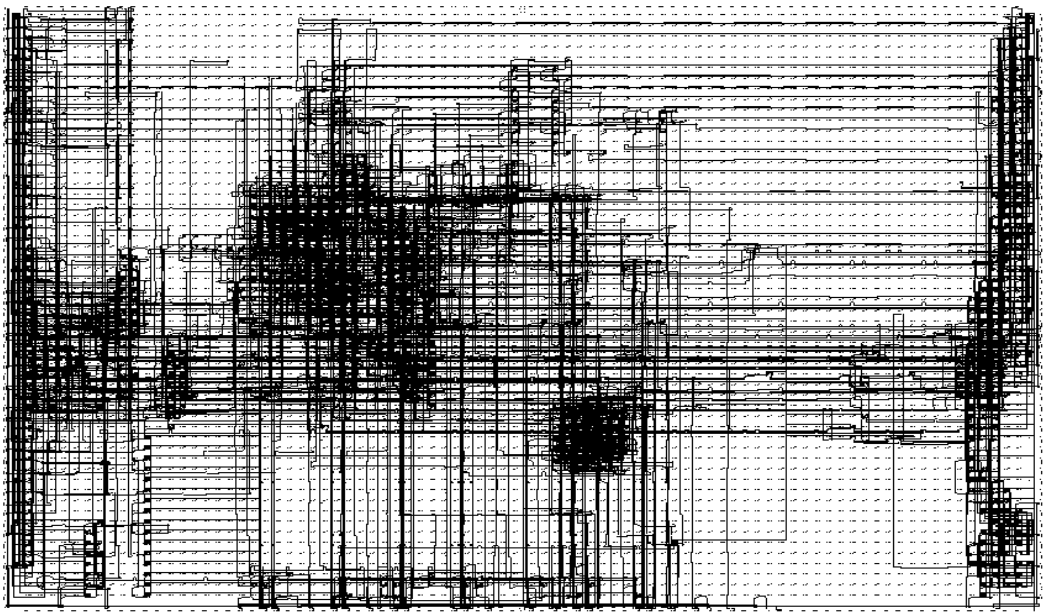

Figure 5-20: FPGA Layout of the Mapped Design of the Averager and the Carrier Wipe-off Components.

The other figures for the mapped implementations of all the parts are presented in Appendix E. The implemented parts were tested in sequence with real GPS data, and acquisition and tracking (serial-correlators zooming function) were

verified. Figure 5-21 shows the hardware simulation for the correlation functions of the averaging-based acquisition. Five correlation peaks are shown. They are approximately similar to the Matlab simulation using fixed-point operations.

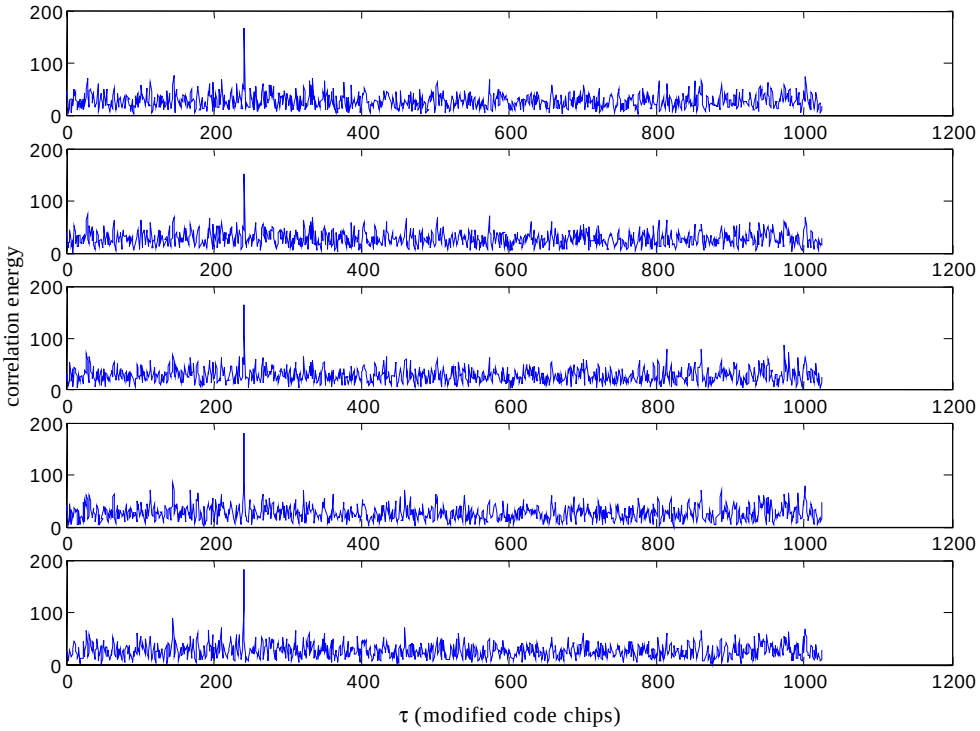

Figure 5-21: Hardware-Based Results of the Averaging Correlation.

The required FPGA resources (or implementation area) for each part of the design are shown in Table 5-1. To estimate the required FPGA resources for the whole system, we cannot simply add the numbers of the logic slices used and the used block RAMs of all the blocks. The reason is each block contains data collection, intermediate calculation handling, and result storage that can be reduced for the complete system implementation to avoid redundancy. Therefore, an estima-

tion of the required FPGA resources of the whole system which implements acquisition and tracking is approximately 7,500 programmable logic slices and 84 Block RAMs. Adding two RAMs for data collection changes the required number of block RAMs to 114. More area is also needed since the whole system requires more controllers plus the carrier phase estimator which has not yet been implemented. Therefore, using 8,000 programmable logic slices and 114 block RAMs is a logical estimation.

Table 5-1. Implementation Cost (Virtex Resources)

	Acquisition					Tracking
	Part.#	1	2	3	4	5	
Configurable Logic Slices (max. 9408)		704	2288	649	2288	862	697
Block RAMs (max. 28)		25	24	18	24	8	17

With the availability of such resources in one FPGA such as the VirtexE1600, the implementation of such a system may be possible. The whole design will take about 50% of the available slices and 80% of the available Block RAMs. This leaves more room for routing that could affect the maximum clock speed. Routing in FPGA is one of the main problems since it is not predictable. However, based on the already implemented blocks and the above estimations, a system clock of 45MHz may be used without having timing problems (Table 5-2).

Table 5-2. Maximum Net Delays in (nsec) for Each Partition

Carrier wipe off and averager	FFT Block	Freq. Domain Multiplier	IFFT Block	Peak Searcher	Tracking E-P-L (I-phase only)
14.69	17.39	15.77	17.39	15.61	17.52

The acquisition process requires about 42,000 clock cycles. The serial correlators block requires about 5,000 clock cycles. If the clock speed is equal to or more than 47MHz, the averaging correlation and serial correlators of 1-ms GPS data is computed in 1-ms or less. Assuming that the carrier frequency is known, the acquisition is conducted in the code phase dimension and therefore processing blocks of 1-ms GPS data can be achieved in real time.

Chapter 6

Summary, Conclusion, and Recommendations

6.1 Summary

The goal of this work was to develop an implementable algorithm for GPS block processing. The most important characteristics of the developed block processing are a short acquisition time, and a high accuracy of code and carrier phase estimations.

The current implementation of the GPS block processing (acquisition and tracking) are too slow for the high accuracy estimations. The main cause of slow processing is the large amount of operations needed to implement the correlator and the FFT functions. Implementing the correlations using a microprocessor-based method is another factor.

This work investigated possible fast algorithms for implementing the correlation function using a real and binary transforms in an FPGA. The correlation algorithm using real transform (the Fermat number transform) is limited to short codes. Therefore, it was not used for the implementation of the GPS receiver. The

other transform investigated in this dissertation was the Walsh Hadamard transform. The provided method used only one Walsh transform, LFSRs, and RAMs. The method was implemented in an FPGA. It showed an improvement in correlation computation speed of 20 times compared to the FFT-based implementation. Comparing the Walsh Hadamard correlation implemented in an FPGA (with a 1MHz processing clock speed) against its implementation with a microprocessor-based architecture (with a 233MHz clock speed), the FPGA-based method was approximately 2500 times faster. The only limiting factor for this algorithm was the type of codes that can be used. The Walsh Hadamard based method was suitable for maximum length pseudo random codes. However, this method cannot be directly applied to gold codes because of the additional XOR operation used to generate the gold codes. Therefore, extending this method to gold codes needs more research since it is not directly applied.

The FFT method was then used for the implementation of acquisition process. The required size of the FFTs are 5000 points each. Implementing 5000-point FFT was a complicated task. Therefore, the averaging correlation method was used. The averaging correlation method requires computing 5 of 1023-point FFTs (or IFFTs) instead of every 5000-point FFT (or IFFT). Direct implementation of the 1023-point FFT was also very difficult. One possible method for extending the averaging correlation concept to a 1024-point correlation function was developed in this dissertation. This method is named modified-code averaging correlation. The modified-code method was used for the acquisition process and connected to time-domain serial correlators to refine the code phase and carrier frequency estimations. The whole design could not fit into a single FPGA. Therefore, it was par-

titioned into smaller blocks and the acquisition and tracking-like estimators were verified.

6.2 Conclusion

In conclusion, various transforms were able to speed up the acquisition process significantly when they were implemented using parallel processing hardware. There were limitations for using some of these transforms in the code length and the code type. The modified-code averaging method, however, shortened the acquisition time significantly using the FFT-based correlation concept. The implementation of this new architecture provided fast acquisition with accurate synchronization without losing much of the signal energy. The implementation was able to process 1-ms of a normal GPS signal in less than 1-ms. Real-time acquisition was achieved when the carrier frequency was determined by a frequency search step. Therefore, the modified-code correlator based block processing architecture is considered a viable solution to the described problems of the slow acquisition and tracking with the current GPS receivers.

6.3 Recommendations

It is recommended that the whole architecture of the block processing using the modified-code averaging correlation is implemented in a single FPGA. Then a real GPS signal will be applied to the architecture to test the mis-detection proba-

bility. Availability of such a measure helps to decide which applications would benefit from such an implementation.

Software block processing requires that the GPS samples are first stored and then processed in a group. Until the time of this writing, this was done using slow software-based processing. Applying the gained techniques of software block processing to the implemented architecture makes use of the high performance design to implement real-time processing of GPS data. Using such techniques with this fast method helps to design a receiver for applications where high doppler occurs. For example, an airplane will benefit from having a high rate updating system that continuously gives values of speed, direction and altitude.

Additionally, an extension of this work to acquire weak signals is very important. In some situations, most of the satellites are in positions that make their signals weak. With the extension of the implemented design to work for weak signals, a receiver with such an architecture can detect the satellites very quickly, thus providing navigational and positioning information most of the time. Since weak signals need longer blocks of data, specialized weak signal block processing techniques are needed.

Developing an architecture using the presented solution to build a GPS receiver that processes the GPS data for multiple satellites in real-time is an important issue. Time-sharing of the implemented functions to acquire or track multiple satellites needs to be studied. Designing a board with larger and multiple FPGAs is recommended since it will facilitate real-time GPS block processing. Attaching this board to a front-end system and a microprocessor based system simplifies the GPS processing for flight testing.

Developing different algorithms for circular correlations without using the FFTs is still a valid future research direction. Even with the limitations found when using the other transforms, a viable solution may exist. Extension of Walsh-based algorithm to C/A codes is an open area for research. Using approximations and other transforms may lead to a break-through in the GPS acquisition. With such a development, a time-sharing concept of multi-satellite acquisition and tracking will become feasible.

References

[1] Agarwal R.and Burrus C., “Fast Convolution Using Fermat Number Transforms with Applications to Digital Filtering,” *IEEE Transactions on Acoustics, Speech, and Signal Processing*, Vol. ASSP-22, No. 2, April 1974, pp. 87 - 97.

[2] Akos D. and Tsui J., “Design and Implementation of a Direct Digitization GPS Receiver Front End,” *IEEE Transactions on Microwave Theory and Techniques*, Vol. 44, No. 12, Dec. 1996, pp. 2334-2339.

[3] Akos D., “A Software Radio Approach to Global Navigation Satellite System Receiver Design”, Ph.D. Dissertation, Ohio University, August 1997.

[4] Alaqeeli A. and Starzyk J., “Hardware Implementation for Fast Convolusion with a PN Code Using Field Programmable Gate Array,” Proc. of 33rd Southeastern Symposium on System Theory, Athens, OH, March 2001, pp. 197 -201.

[5] Alaqeeli A., Starzyk J., and F. van Graas, “Real-Time Acquisition and Tracking for GPS Receivers”, Submitted to ISCAS2003, Bangkok, Thailand, May 2003

[6] Arambepola B., "VLSI Architecture for Convolver Design Using Number Theoretic Transforms," *Electronics Letters*, Vol. 25, No. 23, November 1989, pp. 1604 -1606.

[7] Beauchamp K., **Applications of Walsh and Related Functions**, Academic Press Inc., 1984.

[8] Braasch M. and Van Dierendonck A., "GPS Receiver Architectures and Measurements," Proceedings of the IEEE, Vol. 87, No.1, January 1999, pp. 48 - 64.

[9] Budisin S., "Fast PN Sequence Correlation By Using FWT," Mediterranean Electrotechnical Conf. Proc., Lisbon, Portugal, April 1989, pp. 513 -515.

[10] Burrus C. and Parks T., **DFT/FFT and Convolution Algorithms**, Wiley-Interscience Publication, New York, 1985.

[11] Coenen A. and Van Nee D., "Novel Fast GPS/GLONASS Code-Acquisition," *Electronics Letters*, Vol. 28, No. 9, April 1992, pp. 863 -865.

[12] Cohn M. and Lemple A., "On Fast M-Sequence Transforms," *IEEE Transactions on Information Theory*, January 1977, pp. 135 -137.

[13] Cook C., Ellersick F., Milstein L., and Schilling D., **Spread Spectrum Communications**, IEEE Press, New York, 1983.

[14] Dimitrov V., et al., “Generalized Fermat-Mersenne Number Theoretic Transform,” *IEEE Transactions on Circuits and Systems-II: Analog and Digital Signal Processing*, Vol. 41, No. 2, February 1994, pp. 133 -139.

[15] Dixon R., **Spread Spectrum Systems with Commercial Applications**, 3rd ed., John Wiley & Sons Inc., New York, 1994.

[16] Feng G. and van Graas F., “GPS Receiver Block Processing,” ION GPS-99, Nashville, TN, September 1999, pp. 307 -316.

[17] French G., **Understanding the GPS An Introduction to the Global Positioning System**, 1st ed., Bethesda, MD, GeoResearch, Inc., 1996.

[18] Golomb S., **Shift Register Sequences**, Holden-Day Inc., 1967.

[19] Gunawardena S., “Feasibility Study for the Implementation of Global Positioning System Block Processing Techniques In Field Programmable Gate Arrays,” MS. Thesis, Ohio University, June 2000.

[20] Gibson J., **Principles of Digital and Analog Communications**, 2nd ed., Prentic Hall, 1993.

[21] Hassan A., Hershy J., and Saulnier G., **Perspectives in Spread Spectrum**, Kluwer Academic Pub., 1998.

[22] Herveille R., “CORDIC Core Specification,” OpenCores, www.opencores.org, Rev. 0.4, December 12, 2001.

[23] Kantabutra V, “High-Radix CORDIC for Vector Rotation with Pipelined FPGA Implementation,” IEEE ICECS1999, Pafos, Cyprus, Sept. 1999, pp. 1131 -1134.

[24] Kaplan E., **Understanding GPS Principles and Applications**, Artech House Inc., MA, 1996.

[25] Kayton M. and Fried W., **Avionics Navigation Systems**, 2nd ed., New York, John Wiley & Sons, Inc., 1997.

[26] Kharrat W., et al., “A New Method To Implement CORDIC Algorithm,” IEEE ICECS2001, Malta, Sept. 2001, pp. 715 -718.

[27] Lempel A., “Hadamard and M-Sequence Transforms are Permutationally Similar,” *Applied Optics*, Vol. 18, No. 24, December 1979, pp. 4064 -4065.

[28] Li W., “The Modified Fermat Number Transform and Its Application,” IEEE ISCAS1990, New Orleans, Vol. 3, May 1990, pp. 2365 -2368.

[29] Lin D. and Tsui J., “Comparison of Acquisition Methods for Software GPS Receiver,” ION GPS-2000, Salt Lake City, UT, September 2000, pp. 2385 -2390.

[30] Lin D. and Tsui J., “A Software GPS Receiver for Weak Signals,” IEEE MTT-S Digest, THIF-37, 2001, pp. 2139 -2142.

[31] Misra P. and Enge P., **Global Posistioning System: Signals, Measurements, and Performance**, Ganga-Jamuna Press, MA, 2001.

[32] Molyneux D. and Pratt A., “Post Processing Algorithms for Translator-Type GPS Receivers,” IONGPS2002, Portland, OR, Sept. 2002.

[33] Myers D., **Digital Signal Processing: Efficient Convolution and Fourier Transform Techniques**, Prentic Hall, New York, NY, 1990.

[34] Nallatech Inc, “Ballynuey 2 Virtex PCI Card User Guide,” Ref.: NT107-0045, Dec. 1999.

[35] Nussbaumer H., **Fast Fourier Transform and Convolution Algorithms**, Springer-Verlag, New York, 1982.

[36] Parkinson B. and Spilker J., **Global Positioning System: Theory and Applications**, Volume I, American Institute of Astronautics and Aeronautics, Inc., Washington, DC, 1996.

[37] Peterson R., Ziemer R., and Borth D., **Introduction to Spread Spectrum Communications**, Prentic Hall, 1995.

[38] Proakis J., **Digital Communications**, 3rd ed., McGraw-Hill, New York, 1995.

[39] Proakis J., et al., **Algorithms for Statistical Signal Processing**, Prentice-Hall, Inc., Upper Saddle River, NJ, 2002.

[40] Rader C., "Discrete Convolution via Mersenne Transforms," *IEEE Trans. Comput.*, Vol. C-21, pp. 1269-1273, 1972.

[41] Roth C., **Fundamentals of Logic Design**, 4th ed., West Publishing Company, 1992.

[42] Sari H. and Cochet P., "Transform-Domain Signal Processing in Digital Communications," Tirrenia International Workshop on Digital Communications, Viareggio, Italy, 1995, pp. 364 -384.

[43] Sarmiento R., et al., "A CORDIC Processor for FFT Computation and Its Implementation Using Gallium Arsenide Technology," *IEEE Transactions on VLSI Systems*, Vol. 6, No. 1, March 1998, pp. 18 -30.

[44] Selesnick I., and Burrus C., "Fast Convolution and Filtering", Chapter 8 in **The Digital Signal Processing Handbook**, CRC Press, Boca Raton, FL, 1998.

[45] Smith D., **HDL Chip Design**, Madison, AL, Doone Publications, 1996.

[46] Smith W. and Smith J., **Handbook of Real-Time Fast Fourier Transforms**, IEEE Press, 1995.

[47] Starzyk J., "Matlab Self-Organizing NN Project," v.2, Ohio University, from http:\\www.ent.ohiou.edu\~starzyk, 2000.

[48] Starzyk J. and Zhu Z., "Averaging Correlation for C/A Code Acquisition and Tracking in Frequency Domain," MWSCS Conf., Fairborn, OH, August 2001.

[49] Tsui J., **Fundamentals of Global Positioning System Receivers: A Software Approach**, John Wiley & Sons Inc., 2000.

[50] Turimella S. and Skavantzos A., "Implementation Aspects of Convolvers Using Non-Binary Arithmetic," Asilomar Conference on Circuits, Systems and Computers, Pacific Grove, CA, Nov. 1991, pp. 90 -94.

[51] Uijt de Haag M., "An Investigation Into The Application of Block Processing Techniques for the Global Positioning System," Ph.D. Dissertation, Ohio University, August 1999.

[52] Van Nee D. and Coenen A., "New Fast GPS Code-Acquisition Technique Using FFT," *Electronics Letters*, Vol. 27, No. 2, January 1991, pp 158 -160.

[53] Ward P., "GPS Receiver Search Techniques," IEEE PLANS, 1996, pp. 604 - 611.

[54] Xilinx FFT, “High-Performance 1024-Point Complex FFT/IFFT,” V 2.0, Product Specification, Xilinx Inc., July, 2000.

[55] Xilinx Inc., “Numerically Controlled Oscillator,” V. 1.0.3, Product Specification, Xilinx Inc., December, 1999.

[56] Xilinx Inc, “Virtex 2.5V Field Programmable Gate Arrays,” DS003-2 Product Specification, Xilinx Inc., September, 2002.

[57] Xilinx Inc, “Parallel Multipliers - Performance Optimized,”, Product Specification, Xilinx Inc., July, 1998.

[58] Xu S., Dai L., and Lee S., “Autocorrelation Analysis of Speech Signals Using Fermat Number Transform (FNT),” *IEEE Transactions on Signal Processing*, Vol. 40, No. 8, August 1992, pp. 1910-1914.

[59] Yubin Z. and Turner L., “Report for FFT Processor Design,” University of Calegary, April, 1996.

[60] Zehavi E., “Applications of Walsh Functions and the FHT in CDMA Technology,” Tirrenia International Workshop on Digital Communications, Viareggio, Italy, 1995, pp. 28 -38.

[61] Zhu Z., “Averaging Correlation for Weak GPS Signal Processing”, MS. Thesis, Ohio University, April 2002.

Appendix A

The Ballynuey FPGA Board

Appendix A contains a description of the design platform used in this dissertation. The Board was designed by Nallatech Inc, England. Nallatech produces different DSP and FPGA boards for different applications, video processing for instance. This board is called a Ballynuey PCI card. The Ballynuey card is a general purpose data processing interface card for the PC. It has four DIME module sites to provide the designer with different interfaces that satisfy certain applications. It also has a ZBT SSRAM that is directly attached to the user's FPGA.

Ballynuey has two Xilinx FPGAs. An XLA FPGA is pre-configured with a PCI interface IP core to abstract the interfacing between the board and the PC. Also an on-board Virtex FPGA is available for the user designs. It is connected to the PC through the interfacing XLA FPGA.

The Virtex FPGA is ready for the user's developed designs. An interface design is provided to the user. This interface circuit is responsible for data movement between the Ballynuey card and the PC through the XLA FPGA. The interface design is compact and easy to use. A design developer needs to include this interface circuit in the top level of the implemented design. This permits the devel-

oped design to read the data from the PC or write them back. This simplicity of communicating between PC and Virtex makes Ballynuey a good choice for the problem in hand. More information about the Ballynuey can be found in (Nallatech, 1999).

The Virtex FPGA provides a high performance and a large area of programmable resources. This allows the user to process the data before passing them to or from the PC. The Virtex FPGA is a member of the second generation of the Xilinx FPGAs. It provides the user with the flexibility of a DSP and the performance of an ASIC. It has an array of configurable logic blocks (CLBs) surrounded by programmable input/output blocks (IOBs). This FPGA is a SRAM-based FPGA. It also has a built-in clock management circuitry. For example, it has four delay locked loops (DLLs) for advance clock control. The main difference between Virtex and the first generation of Xilinx FPGA, is the additional user RAMs. Virtex has the old type of RAM which is the distributed RAMs available inside every CLB. The new FPGA RAMs are called Block RAMs. These RAMS are larger, which enables each block RAM to implement up to 4096 bits. More information about the Virtex FPGA can be found in (Xilinx, 2002).

Appendix B

Matlab Codes

B.1 Walsh Hadamard Based Convolution with PN sequences

This Matlab code was written to verify the algorithm of the Walsh Hadamard transform based convolution. This method was described in detail in Section 3.3. This Matlab code uses a Walsh function that is also written for the verification of this method.

```
clc;clear;
PN=[1 0 0 1 1 1 0];
n=3;
p=2^n-1;
PNex=[PN PN(1:2)];
for i=1:p
    S(i)=PNex(i)+2*PNex(i+1)+4*PNex(i+2);
end;
%     the inverse S sequence
for i=1:p
    S_1(S(i))=i;
end;
X=[];
PN1=PN;
for i=1:p
    X=[X;PN1];
    PN1=[PN1(2:p) PN1(1)];
end;
```

```
W=hadamard(8);
W7=(1-W(2:8,2:8))/2;
Xperm=X(1:p,S_1(1:7));
for i=1:7
    for j=1:7
        if Xperm(i,:)==W7(j,:)
            Q(i)=j;
        end;
    end;
end;
%     the inverse Q sequence
for i=1:p
    Q_1(Q(i))=i;
end;
shift=3;
input=[ PN(shift:p) PN(1:shift-1)];
%     permute the input sequence
input_perm=[0 input(S_1)];

%     calculate the Walsh transform of the permuted sequence
Y=walsh(input_perm);
[Z1,I1]=max(abs(Y(2:p+1)));
%II=Q_1(I1-1);
% II is the location of the maximum convolution
Result=Y(Q+1);

% ----------------------------
% Walsh generates Walsh transform of the binary input sequence x
function y=walsh(x);
x1=-2*x+1;
y=x1*(hadamard(length(x));
```

B.2 Approximation of ATAN Function

The following code was developed to find a simple method to calculate the ATAN function needed for the carrier phase estimation. The developed method was described in Section 5.3.8.1. Another Matlab code was written to verify the CORDIC method for calculating the ATAN function as described in Section 5.3.8.2. This code follows the ATAN approximation matlab code.

```
% this matlab code is done to check the error of approximating ATAN(Q/
I).
% from previous simulations, atan(x) can be approximated by (pi/2)*v
% where v=1-z,  and z= 1(1+x),    x is (Q/I) ===> v=Q/(Q+I)
% I assume that Q and I are only positives.
clc;clear;
Q=0.01 :0.01:100;
n=length(Q);
I=ones(1,n);
v=(Q./(Q+I));
x=(Q./I);
y=(180/pi)*atan(x);
v1=v;
y1=(90).*(v1); % y1 is the approximation of atan(Q/I)


figure(1);
plot(x,y-y1);
title('error between atan(x) and direct calculation of (pi/2)*v');
xlabel(' x (or Q/I )');
ylabel(' error in degrees');
% ----------------------------------------------------------------
% the error is based on the PC calculations of (pi/2)*v compared to
atan(x);
```

```
% % testing cordic
xo=2;
yo=1;
zo=0;
% % xnew=x(i+1), ynew=y(i+1);
n=32
% rotation mode:    --------------------
xi=xo;yi=yo;zi=zo;
for i=0:n;
    if zi<0 ,
        di=-1;
    else
        di=1;
    end;
    xnew=xi-yi*di*(2^-i);
    ynew=yi+xi*di*(2^-i);
    znew=zi-di*taninv(i);
    xi=xnew;yi=ynew;zi=znew;
    xr(i+1)=xi;yr(i+1)=yi;zr(i+1)=zi;
end;
[0 xo yo zo ; (1:n+1)' xr' yr' zr'];

% vectoring mode:    --------------------
xi=xo;yi=yo;zi=zo;ai=1;
for i=0:n;
    if yi<0 ,
        di=1;
    else
        di=-1;
    end;
    anew=ai*sqrt(1+2^-(2*i));
    xnew=xi-yi*di*(2^-i);
    ynew=yi+xi*di*(2^-i);
    znew=zi-di*taninv(i);
    ai=anew;xi=xnew;yi=ynew;zi=znew;
    av(i+1)=ai;xv(i+1)=xi;yv(i+1)=yi;zv(i+1)=zi;

end;
disp ('     i       xi       yi       zi        Ai     abs=(xi/ai)');
[0 xo yo zo 1 xo*1; (1:n+1)' xv' yv' zv' av' xv'./av' ]
```

B.3 Averaging Correlation Method

The averaging correlation method was developed by Starzyk and Zhen (Starzyk, 2001). This method averages the 5000 GPS samples to 1023 and perform five correlations. Then the best recovered version of these averaged correlations contains the strongest peak. This method was described briefly in Section 4.2. More information can be found in (Starzyk, 2001, and Zhu, 2002). The following part of the Matlab code was prepared by Starzyk and Zhen

```
% sv_u contains 5000 samples of the GPS signal
uprate=5000/1023;
maxp5=zeros(1,5);
for k=1:5 %shift of the beginning
   if (k-1)==floor(k-1)
   sv_up=[sv_u(k:length(sv_u)) sv_u(1:k-1)];
      i=1;
      for j=1:p%1023
      sum=0;
      cnt=0;
      while (floor(i/uprate)+1==j) & i<pu
         cnt=cnt+1;
         sum=sum+sv_up(i);
         i=i+1;
      end;
      avg=sum/cnt;
      sv_av(j)=avg;
   end;
%finishing average
```

B.4 Modified-Code Averaging Method

This is part of the Matlab code that was used to develop the modified-code averaging method for the GPS acquisition. This method was then verified for its performance and its effect on signal-to-noise-ration (SNR), error in the code phase and error in the carrier phase. These matlab codes were developed based on the block processing Matlab code written by Dr. Frank van Graas and Gang Feng. The following code is the Matlab code used to average the up-sampled C/A code.

```
downrate=1024/5000;
i=1;
for j=1:1024
    sumy=0;
    cnt=0;
    while (floor((i-1)*downrate)==j-1) & i<=pu
        cnt=cnt+1;
        sumy=sumy+ca_code(i);
        i=i+1;
    end; % while
     ca1024(j)=sumy/cnt;
end; % for j
ca_conj =  conj(fft (ca1024));%  conj of fft of 1024-bit ca code
```

Appendix C

VHDL Codes

Appendix C contains parts of the VHDL codes for both the Walsh-based correlator and the modified-code averaging method for GPS block processing. The first section is the design of the Walsh-based correlator.

C.1 Walsh-Based Convolution

The top-level VHDL code for the Walsh-based correlator is presented here. It contains the Walsh butterfly of size 32 that is used to build the whole 1024-point Walsh-Hadamard for the convolution

```
------------------------------------------------------------------
-- top design of the 1024-point Walsh-based convolver
libraryIEEE;
use IEEE.std_logic_1164.all;

entity walshconv is
      port ( CLK, RST, STRT:in std_logic;
                  A_INP: in std_logic_vector (7 downto 0);
                  Q1,Q2,Q3,Q4: out std_logic_vector (9 downto 0));
end entity;
```

```
architecture walshtop_arch of walshconv is

component topcrkt
      port ( CLK, RST, CE, CE1, CE2, CE3,
CE4,CE5,CE6,CE7,CER1,CER2,CER3,CER4:in STD_LOGIC;
      WE1,WE2, WE3, L2, L3,L7, CLR,CLR1,CLR2,CLR3,CLR4,CLR5,CLR6:in
std_logic;
      IN_A: in std_logic_vector (7 downto 0);
    phase1 : out std_logic;
      Q1,Q2,Q3,Q4: out std_logic_vector (9 downto 0);
      count1k,count2,count3,count4,count5: out std_logic_vector (9
downto 0));
end component ;
component main_fsm
      port ( CLK, RST: in std_logic;
                  STRT, DONE, GOTIT, STEPDONE, READYSIG: in std_logic;
            STRT2, STRT4: out std_logic);
end component ;
component s_perm_fsm
      port (CLK, RST,STRT: in std_logic;
                  count1k: in std_logic_vector (9 downto 0);
      CLR1,CLR2,CE1,CE2,WE: out std_logic;
      DONE: out std_logic);
end component ;
component fdsmpls_fsm
      port ( CLK, RST,STRT2: in std_logic;
                  count2: in std_logic_vector (9 downto 0);
                  CLR,CE,WE2,nextstep,gotit: out std_logic);
end component ;
component walsh1_fsm
      port ( CLK, RST,STRT3: in std_logic;
                  count3,count4: in std_logic_vector (9 downto 0);
                  CLR3,CLR4,CE3,CE4,CER1,CER2,L2,WE3,stepdone: out
std_logic);
end component ;
component walsh2_fsm
      port ( CLK, RST,STRT4, phasefound: in std_logic;
                  count5: in std_logic_vector (9 downto 0);
                  CLR5,CLR6,CLR7,CE5,CE6,CE7,CER3,CER4,L3,L7,readysig:
out std_logic );
end component ;

signal
c,c1,c2,c3,c4,c5,c6,c7,cr1,cr2,cr3,cr4,w1,w2,w3,ll2,ll3,ll7,cl,cl1,cl
2,cl3,cl4,cl5,cl6,cl7: std_logic;
```

```
signal  don,gott,stdon,rdy,st,st2,st3,st4,ph: std_logic;
signal  cnt1,cnt2,cnt3,cnt4,cnt5:std_logic_vector (9 downto 0);

begin

      U1: topcrkt port map (CLK=>CLK, RST=>RST, CE=>c, CE1=>c1,
CE2=>c2, CE3=>c3,
CE4=>c4,CE5=>c5,CE6=>c6,CE7=>c7,CER1=>cr1,CER2=>cr2,CER3=>cr3,CER4=>c
r4,
      WE1=>w1,WE2=>w2, WE3=>w3, L2=>ll2, L3=>ll3,L7=>ll7,
CLR=>cl,CLR1=>cl1,CLR2=>cl2,CLR3=>cl3,CLR4=>cl4,CLR5=>cl5,CLR6=>cl6,I
N_A=>A_INP,phase1=>ph,Q1=>Q1,Q2=>Q2,Q3=>Q3,Q4=>Q4,

count1k=>cnt1,count2=>cnt2,count3=>cnt3,count4=>cnt4,count5=>cnt5 );

      U2: main_fsm port map (CLK=>CLK, RST=>RST, STRT=>st, DONE=>don,
GOTIT=>gott, STEPDONE=>stdon, READYSIG=>rdy, STRT2=>st2, STRT4=>st4 );

      U3: s_perm_fsm port map (CLK=>CLK, RST=>RST,STRT=>st,
count1k=>cnt1, CLR1=>cl1,CLR2=>cl2,CE1=>c1,CE2=>c2,WE=>w1,DONE=>don );

      U4: fdsmpls_fsm port map (CLK=>CLK,
RST=>RST,STRT2=>st2,count2=>cnt2,CLR=>cl,CE=>c,WE2=>w2,next-
step=>st3,gotit=>gott );

      U5: walsh1_fsm port map ( CLK=>CLK,
RST=>RST,STRT3=>st3,count3=>cnt3,count4=>cnt4,CLR3=>cl3,CLR4=>cl4,CE3
=>c3,CE4=>c4,CER1=>cr1,CER2=>cr2,L2=>ll2,WE3=>w3,stepdone=>stdon );

      U6: walsh2_fsm port map ( CLK=>CLK, RST=>RST,STRT4=>st4, phase-
found=>ph,count5=>cnt5,CLR5=>cl5,CLR6=>cl6,CLR7=>cl7,CE5=>c5,CE6=>c6,
CE7=>c7,CER3=>cr3,CER4=>cr4,L3=>ll3,L7=>ll7,readysig=>rdy );

end architecture;



-- this is the top level circuit which will be placed in the top level
design
-- this circuit will be the target of all the FSMs in the design later.
libraryIEEE;
use IEEE.std_logic_1164.all;

entity topcrkt is
      port ( CLK, RST, CE, CE1, CE2, CE3,
CE4,CE5,CE6,CE7,CER1,CER2,CER3,CER4:in STD_LOGIC;
```

```
      WE1,WE2, WE3, L2, L3,L7, CLR,CLR1,CLR2,CLR3,CLR4,CLR5,CLR6:in
std_logic;
      IN_A: in std_logic_vector (7 downto 0);
      phase1 : out std_logic;
      Q1,Q2,Q3,Q4: out std_logic_vector (9 downto 0);
      count1k,count2,count3,count4,count5: out std_logic_vector (9
downto 0));
end entity;
architecture archtop of topcrkt is

      component s_block
            port ( CLK,CE,CE1,CE2,CLR,CLR1,CLR2,WE1: in std_logic;
      Q, count1k,count2 : out std_logic_vector (9 downto 0));
      end component;
      component wt32reg
            port ( clock: in std_logic;
               CER1,CER2,L,WE: in std_logic;
               A: in std_logic_vector (7 downto 0);
               WA, RA: in std_logic_vector (9 downto 0);
               B: out std_logic_vector(7 downto 0)
               );
      end component;
      component count_1k
            port (CLK: in STD_LOGIC;
         CE,CLR: in STD_LOGIC;
        Q: inout STD_LOGIC_VECTOR (9 downto 0) );
      end component ;
      component maxconv
            port ( clock: in std_logic;
               COUNT6: in std_logic_vector (9 downto 0);
             A_IN:in std_logic_vector(7 downto 0);
               A_GRT_B: out std_logic
               );
      end component ;
      component q_reordr
            port (clock: in std_logic;
               L,L7,CE6,CE7,CLR6: in std_logic;
               COUNT6: out std_logic_vector (9 downto 0);
               phasefound: out std_logic
               );
      end component ;
      component phase_reg
            port (CLR : in std_logic;
            CE : in std_logic;
```

```
            LOAD : in std_logic;
            CLK : in std_logic;
            DATA : in std_logic_vector (9 downto 0);
            Q1 : out std_logic_vector (9 downto 0);
            Q2 : out std_logic_vector (9 downto 0);
            Q3 : out std_logic_vector (9 downto 0);
            Q4 : out std_logic_vector (9 downto 0));
      end component ;
signal sbar, sig6, wa2,ra2,ra3: std_logic_vector (9 downto 0);
signal wmid, wfin: std_logic_vector (7 downto 0);
signal newphase,agb: std_logic ;

begin
      U1: s_block port map (
CLK=>CLK,CE=>CE,CE1=>CE1,CE2=>CE2,CLR=>CLR,CLR1=>CLR1,CLR2=>CLR2,WE1=
>WE1,Q=>sbar ,count1k=>count1k,count2=>count2);

      U2: wt32reg port map (
clock=>CLK,CER1=>CER1,CER2=>CER2,L=>L2,WE=>WE2,A=>IN_A,WA=>sbar,RA=>r
a2,B=>wmid);

      U3: count_1k port map (CLK=>CLK, CE=>CE3,CLR=>CLR3,Q=>ra2 );

      U4: wt32reg port map (
clock=>CLK,CER1=>CER3,CER2=>CER4,L=>L3,WE=>WE3,A=>wmid,WA=>wa2,RA=>ra
3,B=>wfin);

      U5: count_1k port map ( CLK=>CLK, CE=>CE4,CLR=>CLR4,Q=>wa2  );

      U6: count_1k port map ( CLK=>CLK, CE=>CE5,CLR=>CLR5,Q=>ra3  );

      U7: maxconv port map (clock=>CLK,COUNT6=>sig6,A_IN=>wfin,
A_GRT_B=>agb );

      U8: q_reordr port map
(clock=>CLK,L=>agb,L7=>L7,CE6=>CE6,CE7=>CE7,CLR6=>CLR6,COUNT6=>sig6,p
hasefound=>newphase );

      U9: phase_reg port map ( CLR=>RST,CE=>newphase,LOAD=>new-
phase,CLK=>CLK,DATA=>sig6,Q1=>Q1,Q2=>Q2,Q3=>Q3,Q4=>Q4 );
count3<=ra2;
count4<=wa2;
count5<=ra3;
phase1<=newphase;
end archtop;
```

```
-- higher level that contain WT32 and necessary registers
-- this will simplify building higher levels later.

library IEEE;
use IEEE.STD_LOGIC_1164.all;

entity WT32REG is
      port ( clock: in std_logic;
               CER1,CER2,L,WE: in std_logic;
               A: in std_logic_vector (7 downto 0);
               WA, RA: in std_logic_vector (9 downto 0);
               B: out std_logic_vector(7 downto 0)
               );
end entity;

architecture LOGIC of WT32REG is
     component wt32
         port (X1, X2, X3, X4, X5, X6, X7, X8:  in std_logic_vector(7
downto 0);
               X9, X10, X11, X12, X13, X14, X15, X16:  in
std_logic_vector(7 downto 0);
               X17, X18, X19, X20, X21, X22, X23, X24:  in
std_logic_vector(7 downto 0);
               X25, X26, X27, X28, X29, X30, X31, X32:  in
std_logic_vector(7 downto 0);
               Y1, Y2, Y3, Y4, Y5, Y6, Y7, Y8: out std_logic_vector(7
downto 0);
                 Y9, Y10, Y11, Y12, Y13, Y14, Y15, Y16: out
std_logic_vector(7 downto 0);
                 Y17, Y18, Y19, Y20, Y21, Y22, Y23, Y24: out
std_logic_vector(7 downto 0);
                 Y25, Y26, Y27, Y28, Y29, Y30, Y31, Y32: out
std_logic_vector(7 downto 0) );
     end component;
     component sinpout1
         port (CLK: in STD_LOGIC;
               CE: in STD_LOGIC;
               D: in STD_LOGIC_VECTOR(7 downto 0);
              Q1,Q2,Q3,Q4,Q5,Q6,Q7,Q8,Q9,Q10,Q11,Q12,Q13,Q14,Q15,Q16:
inout STD_LOGIC_VECTOR (7 downto 0);

Q17,Q18,Q19,Q20,Q21,Q22,Q23,Q24,Q25,Q26,Q27,Q28,Q29,Q30,Q31,Q32:
inout STD_LOGIC_VECTOR (7 downto 0));
     end component;
       component pinsout1
```

```
         port (CLK: in STD_LOGIC;
               CE,L: in STD_LOGIC;
              D1,D2,D3,D4,D5,D6,D7,D8,D9,D10,D11,D12,D13,D14,D15,D16:
in STD_LOGIC_VECTOR (7 downto 0);

D17,D18,D19,D20,D21,D22,D23,D24,D25,D26,D27,D28,D29,D30,D31,D32: in
STD_LOGIC_VECTOR (7 downto 0);
         QQ: out STD_LOGIC_VECTOR (7 downto 0) );
     end component;
       component ram1k8d
         port (WE : in  STD_LOGIC;
            CLK : in  STD_LOGIC;
            ADDRrd : in  STD_LOGIC_VECTOR (9 downto 0);
            ADDRwr : in  STD_LOGIC_VECTOR (9 downto 0);
            DATA : in  STD_LOGIC_VECTOR (7 downto 0);
            Q : out STD_LOGIC_VECTOR (7 downto 0));
     end component;
signal x1,x2,x3,x4,x5,x6,x7,x8,x9,x10,x11,x12,x13,x14,x15 :
std_logic_vector (7 downto 0);
signal
x16,x17,x18,x19,x20,x21,x22,x23,x24,x25,x26,x27,x28,x29,x30,x31,x32 :
std_logic_vector (7 downto 0);
signal y1,y2,y3,y4,y5,y6,y7,y8,y9,y10,y11,y12,y13,y14,y15 :
std_logic_vector (7 downto 0);
signal
y16,y17,y18,y19,y20,y21,y22,y23,y24,y25,y26,y27,y28,y29,y30,y31,y32 :
std_logic_vector (7 downto 0);
signal temp_a: std_logic_vector (7 downto 0);
begin
      U1: wt32 port map (X1=>x1, X2=>x2, X3=>x3, X4=>x4, X5=>x5, X6=>x6,
X7=>x7, X8=>x8, X9=>x9, X10=>x10, X11=>x11, X12=>x12,
      X13=>x13, X14=>x14, X15=>x15, X16=>x16, X17=>x17,
X18=>x18,X19=>x19, X20=>x20, X21=>x21, X22=>x22,
                     X23=>x23, X24=>x24, X25=>x25, X26=>x26, X27=>x27,
X28=>x28, X29=>x29, X30=>x30, X31=>x31, X32=>x32,
                               Y1=>y1, Y2=>y2, Y3=>y3, Y4=>y4, Y5=>y5,
Y6=>y6, Y7=>y7, Y8=>y8, Y9=>y9, Y10=>y10, Y11=>y11, Y12=>y12,
                     Y13=>y13, Y14=>y14, Y15=>y15, Y16=>y16, Y17=>y17,
Y18=>y18,Y19=>y19, Y20=>y20, Y21=>y21, Y22=>y22,
                     Y23=>y23, Y24=>y24, Y25=>y25, Y26=>y26, Y27=>y27,
Y28=>y28, Y29=>y29, Y30=>y30, Y31=>y31, Y32=>y32 );
      U2:   sinpout1 port map (CLK=>clock,CE=>CER1,D=>temp_a,
                  Q1=>x1, Q2=>x2, Q3=>x3, Q4=>x4, Q5=>x5, Q6=>x6,
Q7=>x7, Q8=>x8, Q9=>x9, Q10=>x10, Q11=>x11, Q12=>x12,
      Q13=>x13, Q14=>x14, Q15=>x15, Q16=>x16, Q17=>x17,
Q18=>x18,Q19=>x19, Q20=>x20, Q21=>x21, Q22=>x22,
```

```
                    Q23=>x23, Q24=>x24, Q25=>x25, Q26=>x26, Q27=>x27,
Q28=>x28, Q29=>x29, Q30=>x30, Q31=>x31, Q32=>x32);
      U3:   pinsout1 port map ( CLK=>clock,CE=>CER2,L=>L,
                  D1=>y1, D2=>y2, D3=>y3, D4=>y4, D5=>y5, D6=>y6,
D7=>y7, D8=>y8, D9=>y9, D10=>y10, D11=>y11, D12=>y12,
                    D13=>y13, D14=>y14, D15=>y15, D16=>y16, D17=>y17,
D18=>y18,D19=>y19, D20=>y20, D21=>y21, D22=>y22,
                    D23=>y23, D24=>y24, D25=>y25, D26=>y26, D27=>y27,
D28=>y28, D29=>y29, D30=>y30, D31=>y31, D32=>y32,
                  QQ=>B);
      U4:   ram1k8d port map
(WE=>WE,CLK=>clock,ADDRrd=>RA,ADDRwr=>WA,DATA=>A,Q=>temp_a );




end architecture LOGIC;

----------------------------------------

-- this is a higher level that is responsible for
-- the generation of permutations S-1
-- by connecting the necessary components

libraryIEEE;
use IEEE.std_logic_1164.all;
entity s_block is
      port ( CLK,CE,CE1,CE2,CLR,CLR1,CLR2,WE1: in std_logic;
      Q, count1k,count2 : out std_logic_vector (9 downto 0));
end entity;

architecture arch1 of s_block is

component CNT_1K_1
      port (CLK: in STD_LOGIC;
          CE,CLR: in STD_LOGIC;
          Q: inout STD_LOGIC_VECTOR (9 downto 0));
end component;
component lfsr1
      port (CLK: in STD_LOGIC;
         CE,CLR: in STD_LOGIC;
        Q: inout STD_LOGIC_VECTOR (9 downto 0));
end component ;
component ram1k10d
      port (WE : in  STD_LOGIC;
```

```
            CLK : in  STD_LOGIC;
            ADDRrd : in  STD_LOGIC_VECTOR (9 downto 0);
            ADDRwr : in  STD_LOGIC_VECTOR (9 downto 0);
            DATA : in  STD_LOGIC_VECTOR (9 downto 0);
            Q : out STD_LOGIC_VECTOR (9 downto 0));
end component ;
signal ra, wa, a: std_logic_vector (9 downto 0);

begin
      U1: CNT_1K_1 port map (CLK=>CLK,CE=>CE1,CLR=>CLR1,Q=>a );
      U2: CNT_1K_1 port map (CLK=>CLK,CE=>CE,CLR=>CLR,Q=>ra );
      U3: lfsr1 port map (CLK=>CLK,CE=>CE2,CLR=>CLR2,Q=>wa );
      U4: ram1k10d port map (WE=>WE1, CLK=>CLK, ADDRrd=>ra,
ADDRwr=>wa,DATA=>a ,Q=> Q);
count2<=ra;
count1k<=a;
end arch1;


-----------------------------------------

library IEEE;
use IEEE.STD_LOGIC_1164.all;

entity maxconv is
      port ( clock: in std_logic;
               COUNT6: in std_logic_vector (9 downto 0);
             A_IN:in std_logic_vector(7 downto 0);
               A_GRT_B: out std_logic
               );
end entity;

architecture LOGIC of maxconv is
     component absolut8
         port (A : in std_logic_vector (7 downto 0);Q : out
std_logic_vector (7 downto 0));
     end component;

       component nor10
         port (X: in std_logic_vector (9 downto 0);Y: out std_logic);
     end component;
       component max_val
        port ( CLR, LOAD, CLK : in std_logic;DATA : in std_logic_vector
(7 downto 0);
                      Q : out std_logic_vector (7 downto 0)
```

```
                    );
    end component;

      component comp8bit
        port (AGB : out STD_LOGIC;
                    A :in STD_LOGIC_VECTOR(7 downto 0);
                    B : in STD_LOGIC_VECTOR(7 downto 0));
    end component;
signal temp_a, temp_b : std_logic_vector (7 downto 0);
signal temp_c, clr : std_logic;
begin
     U1: max_val port map (CLR=>clr, LOAD=>temp_c, CLK=>clock,
DATA=>temp_a, Q=>temp_b );
     U2:   comp8bit port map (AGB=>temp_c, A=>temp_a, B=>temp_b );
     U3:   absolut8 port map (A=>A_IN, Q=>temp_a );
     U4:   nor10 port map (X=>COUNT6, Y=>clr );
A_GRT_B<=temp_c;
end architecture LOGIC;



------------------------

library IEEE;
use IEEE.STD_LOGIC_1164.all;

entity q_reordr is
     port ( clock: in std_logic;
              L,L7,CE6,CE7,CLR6: in std_logic;
              COUNT6: out std_logic_vector (9 downto 0);
              phasefound: out std_logic
              );
end entity;

architecture LOGIC of q_reordr is
    component q_count
        port (CLK: in STD_LOGIC;
                   CE,CLR: in STD_LOGIC;
           Q: inout STD_LOGIC_VECTOR (9 downto 0));
    end component;
    component qmax_reg
        port (LOAD : in std_logic;
                    CLK : in std_logic;
                    DATA : in std_logic_vector (9 downto 0);
```

```
                    Q : out std_logic_vector (9 downto 0));
    end component;
      component comp10b
        port (AEB : out STD_LOGIC;
                    A : in STD_LOGIC_VECTOR(9 downto 0);
                    B : in STD_LOGIC_VECTOR(9 downto 0));
    end component;
      component qlfsr
        port (CLK: in STD_LOGIC;
                    CE,L: in STD_LOGIC;
                    DATA: in STD_LOGIC_VECTOR (9 downto 0);
                Q: out STD_LOGIC_VECTOR (9 downto 0));
    end component;
      component reg_q
        port (Q : out std_logic_vector (9 downto 0));
    end component;

signal temp_a, temp_b, temp_c, temp_d : std_logic_vector (9 downto 0);
signal qinit: std_logic_vector (9 downto 0);
begin
     U1: q_count port map (CLK=>clock, CE=>CE6, CLR=>CLR6, Q=>temp_c
);
     U2:   qmax_reg port map (LOAD=>L, CLK=>clock, DATA=>temp_d,
Q=>temp_b );
     U3:   comp10b port map ( AEB=>phasefound, A=>temp_a, B=>temp_b);
     U4:   qlfsr port map ( CLK=>clock, CE=>CE7, L=>L7, DATA=>qinit,
Q=>temp_a );
     U5: reg_q port map (Q=>qinit );
COUNT6<=temp_c;
temp_d<=temp_c(4 downto 0)&temp_c(9 downto 5);

end architecture LOGIC;



-----------------------------------------------

-- this is a register that will store the code shifts each loop.
-- the size is 4regs each 10bits.
-- it will be used to store the last 4 computed phase shifts.
-- incoming new result will be pushed in and the result of 4 loops ago
-- will be pushed out from the other side.

libraryIEEE;
use IEEE.std_logic_1164.all;
```

```
entityphase_reg is

      port (
            CLR : in std_logic;
            CE : in std_logic;
            LOAD : in std_logic;
            CLK : in std_logic;
            DATA : in std_logic_vector (9 downto 0);
            Q1 : out std_logic_vector (9 downto 0);
            Q2 : out std_logic_vector (9 downto 0);
            Q3 : out std_logic_vector (9 downto 0);
            Q4 : out std_logic_vector (9 downto 0)
      );
end entity;

architecture struct_arch of phase_reg is
signal qq1,qq2,qq3,qq4: std_logic_vector(9 downto 0);
begin
process (CLK,CLR)
begin
if (CLR='1') then
      qq1<="0000000000";
      qq2<="0000000000";
      qq3<="0000000000";
      qq4<="0000000000";
else
      if (rising_edge(CLK)) then
            if (CE='1')and (LOAD='1') then
      qq1<=DATA;
      qq2<=qq1;
      qq3<=qq2;
      qq4<=qq3;
            end if;
      end if;
end if;
end process;
Q1<=qq1;
Q2<=qq2;
Q3<=qq3;
Q4<=qq4;

end architecture;
```

C.2 Modified-Code Averaging Correlator (Acquisition)

The acquisition design was partitioned into five parts. Each part was implemented independently. A necessary interface VHDL code is included in each part. This interface was provided with the FPGA board and therefore will not be included here.

C.2.1 Carrier Wipe-Off and Downsampling

The following VHDL codes are the top level code that was responsible for the first part of the acquisition algorithm. It contains an input RAM, an NCO, a carrier wipe-off circuit, an averager circuit, and an output RAM. The RAM components were generated using the Xilinx core generator. Therefore, only parts of the VHDL codes are shown here. The NCO VHDL code is named nco_3.vhdl and was written by researchmate Zhu Zhen. All the VHDL codes are available on the network at : http:\\www.ent.ohiou.edu\~webcad\alaqeeli.

```
------------------------------------------------------------------------
------------------------------
--
-- Title       : dwn_smplr
-- Design      : nco_dwnsplr_ballynue
-- Author      : 0
-- Company     : 0
--
------------------------------------------------------------------------
------------------------------
--
-- File        : dwn_smplr.vhd
-- Generated   : Tue May 21 06:01:47 2002
-- From        : interface description file
```

```
-- By          : Itf2Vhdl ver. 1.20
--
-------------------------------------------------------------------------
------------------------------
--
-- Description :


library IEEE;
use IEEE.STD_LOGIC_1164.all;
use IEEE.std_logic_unsigned.all;

entity dwn_smplr is
       port(
             CLK : in STD_LOGIC;
             INI : in STD_LOGIC;
             Din : in STD_LOGIC_VECTOR (11 downto 0);
             RST : in STD_LOGIC;
             AVGD : out STD_LOGIC_VECTOR(15 downto 0);
             RAM_WA : out STD_LOGIC_VECTOR(9 downto 0);
             RAM_WE : out STD_LOGIC
           );
end dwn_smplr;

architecture dwn_smplr_arch of dwn_smplr is
component rm13x120
      port (
      addr: IN std_logic_VECTOR(6 downto 0);
      clk: IN std_logic;
      din: IN std_logic_VECTOR(12 downto 0);
      dout: OUT std_logic_VECTOR(12 downto 0);
      en: IN std_logic;
      we: IN std_logic
      );
end component;

--
signal ZERO,ONE,ADD4,tempWE : std_logic;
signal ZERO16,DBB,SUM, AVGD1 : std_logic_vector (15 downto 0);
signal cnt5k,rm13x120_Q,ZEROS13: std_logic_vector (12 downto 0);
signal TEMP120_Q : std_logic_vector (6 downto 0);
signal TEMP_Q: std_logic_vector(9 downto 0);
signal Din1:std_logic_vector(11 downto 0);
```

```
type statetype is (ST0,ST1,ST2,ST3,ST4,ST5);
signal fsm_stat : statetype;--signal
--signal




begin
ZERO <= '0';
ONE <= '1';
ZERO16 <="0000000000000000";

--Din1<= not(Din);
-------------------------------------------
DBB_gnration:process(CLK)
begin
      if CLK'event and CLK='1' then
            if INI='1' then
                  if Din(11)='1' then
                        DBB <= "1111" & Din;
                  else
                        DBB <= "0000" & Din;
                  end if;
            else
                  DBB <= ZERO16;
            end if;
      end if;
end process;

-------------------------------------------------------------
--          counter 0 to 1023 to generate RAM_WA
      cnt1024:process(CLK, RST)
      begin
            if RST = '1' then
                  TEMP_Q <= "0000000000";
            elsif rising_edge(CLK) then
                  if tempWE = '1' then
                        TEMP_Q <= TEMP_Q + 1;
                  end if;

            end if;
      end process;

registeredproc:process (CLK)
```

```
begin
      if CLK'event and CLK='1' then
            RAM_WA <=TEMP_Q;
            AVGD <= AVGD1;
            RAM_WE <= tempWE;
      end if;
end process;

-----------------------------
FSM: process(CLK,RST)
begin
      if RST='1' then
            fsm_stat <= ST0;
            cnt5k <= (others =>'0');
            tempWE <='0';
            SUM <=ZERO16;
            AVGD1 <=ZERO16;
            ADD4 <='0';
      elsif CLK'event and CLK='1' then
            case fsm_stat is
                  when ST0 =>

                        if INI='1' then
                              fsm_stat <=ST1;
                        else
                              fsm_stat <=ST0;
                        end if;
                        tempWE <='0';
                        SUM <=ZERO16;
                        AVGD1 <=ZERO16;
                        ADD4 <='0';
                        cnt5k <= (others =>'0');
                  when ST1 =>
                        if rm13x120_Q=cnt5k then
                              ADD4 <='1';
                              fsm_stat <= ST3;
                        else
                              fsm_stat <= ST2;
                              ADD4 <='0';
                        end if;
                        cnt5k <= cnt5k+1;
                        SUM <= DBB;
                        tempWE <='0';
                  when ST2 =>
```

```
                        fsm_stat <= ST3;
                        SUM <= SUM+DBB;
                        cnt5k <= cnt5k+1;
                        tempWE <='0';
                        ADD4 <='0';
                  when ST3 =>
                        fsm_stat <= ST4;
                        SUM <= SUM+DBB;
                        cnt5k <= cnt5k+1;
                        tempWE <='0';
                        ADD4 <='0';
                  when ST4 =>
                        fsm_stat <= ST5;
                        SUM <= SUM+DBB;
                        cnt5k <= cnt5k+1;
                        tempWE <='0';
                        tempWE <='0';
                        ADD4 <='0';
                  when ST5 =>
                        AVGD1 <= SUM+DBB;
                        tempWE <='1';
                        ADD4 <='0';
                        if cnt5k = 4999 then
                              fsm_stat <= ST0;
                        else
                              fsm_stat <= ST1;
                        end if;
                        cnt5k <= cnt5k+1;
                  when others => null;
            end case;
      end if;
end process;

-------------------------------------
-- counter 0 to 119 used to address ADD4 locations
cnt120:process(CLK, RST)
      begin
            if RST = '1' then
                  TEMP120_Q <= "0000000";
            elsif rising_edge(CLK) then
                  if ADD4 = '1' then
                        if (TEMP120_Q  ="1110111") then
                              TEMP120_Q <= "0000000";
                        else
```

```
                              TEMP120_Q <= TEMP120_Q + "0000001";
                        end if;
                end if;
            end if;
      end process;

ADD4_RAM: rm13x120port map (
                addr => TEMP120_Q,
                clk => CLK,
                din => ZEROS13,
                dout => rm13x120_Q,
                en => ONE,
                we => ZERO
                );
end dwn_smplr_arch;



-- this component generates I and Q by multipling sin and cos values to
incoming samples.
-- Therefore, it produces the in-phase and quad-phase components.
-- Or in other words, it changes the inoming signal to its baseband.
-- sin and cos are represented using (-1 0 1) values only (2 bits are
enough)
-- "X0" is zero-------    0
-- "01" is one-------   1
-- "11" is minus one --   -1
library IEEE;
use IEEE.STD_Logic_1164.all;
use IEEE.STD_LOGIC_UNSIGNED.all;

entity carr_wipe_off is
      port (
      CLK         : in std_logic;
      Xin   : in std_logic_vector(11 downto 0);
      NCO_COS: in std_logic_vector(1 downto 0);
      NCO_SIN: in std_logic_vector(1 downto 0);
      I           : out std_logic_vector(11 downto 0);
      Q           : out std_logic_vector(11 downto 0)
      );
end entity carr_wipe_off;

architecture ARCH_IQ of carr_wipe_off is
```

```
signal I1,Q1: std_logic_vector(11 downto 0);
begin
--   in-phase
process (CLK)
begin
      if CLK'event and CLK='0' then
      case NCO_SIN is
      when "00" =>I1<=(others =>'0');
      when "10" =>  I1<=(others =>'0');
            when "01" =>I1<=Xin;
      when "11" =>I1<=(NOT(Xin)+'1');
      when others => null;
      end case;
      end if;
end process;
process (CLK)
begin
      if CLK'event and CLK='0' then
      case NCO_COS is
      when "00" =>Q1<=(others =>'0');
      when "10" =>  Q1<=(others =>'0');
            when "01" =>Q1<=Xin;
      when "11" =>Q1<=(NOT(Xin)+'1');
      when others =>  null;
       end case;
       end if;
end process;
registered:process (CLK)
begin
      if CLK'event and CLK='1' then
            I <= I1;
            Q <= Q1;
      end if;
end process;

end ARCH_IQ;




-- This component is an NCO which is responsible for generating the sin
-- and the cos values

library IEEE;
```

```
use IEEE.std_logic_1164.all;
use IEEE.std_logic_unsigned.all;
-- a 32 bit nco
-- written by Zhu Zhen
-- algorithms from sanjeev's thesis and kaplan's book
-- amp=2bit, -1 0 1, acquisition only
-- no absolute phase, relative phase only
-- source=10 Mhz, if=1.25+- Mhz
entitynco_3 is
      port (
            clk: in std_logic; --5 Mhz  --30
            phasein: in std_logic_vector (31 downto 0); --phase, 0-2pi
            m:in std_logic_vector (30 downto 0); --adding step
            srst:in std_logic;                    --reset, input phase
            clo:out std_logic;                   --a clock out at the
freq
            sino: out std_logic_vector(1 downto 0);
            coso: out std_logic_vector(1 downto 0)
            ) ;
end entity;

architecture nco_3 of nco_3 is
      signaladding:std_logic_vector (31 downto 0);
      signal counter: std_logic_vector (31 downto 0);
      signal addingfull, clo1:std_logic;
      signal coso1,sino1: std_logic_vector(1 downto 0);

begin

      addg: process (clk,srst)

      begin

            if (clk'event and clk='1') then
                        if srst='1'then
                              counter<=phasein;
                              clo1<='0';
                              addingfull<='0';
                              sino1<="00";
                              coso1<="01";

                        else
                              counter<=counter+m;
                              if (counter(31)='1') then
```

```
                              addingfull<='1';
                        else
                              addingfull<='0';
                        end   if;
                        clo1<=addingfull;
                        sino1<=counter(31) & counter(30);
                        coso1<=counter(31) & (not counter(30));
--first 2 digits
                  end if;

            end   if;

      end process;
      registerdproc:process (clk)
      begin
            if clk'event and clk='1' then
                  clo <= clo1;
                  sino <= sino1;
                  coso <= coso1;
            end if;
      end process;

end nco_3;
```

C.2.2 The FFT Block

This part is responsible for converting the averaged incoming GPS samples to frequency domain. This section prepares one of the branches of the FFT-based correlator. This section contains the top level design, the input RAM, the output RAM, and the SMS RAM that has to be connected to the FFT core. Only VHDL code of the SMS configuration of the FFT core is shown here. This part was initially prepared by my friend Jing Pang. All the other parts are available on the network at http:\\www.ent.ohiou.edu\~webcad\alaqeeli.

```
--------------------------------------------------------------------
------------------------------
--
```

```
-- Title               : FWDFFT
-- Design              :
-- Original Author      : Pang Jing
-- Cleaned and modified by: Abdulqadir Alaqeeli
--
------------------------------------------------------------------------
------------------------------
--
-- File        : FWDFFT.vhd
-- Generated   : Sun Mar 31 19:48:50 2002
-- From        : interface description file
-- By          : Itf2Vhdl ver. 1.20
--
------------------------------------------------------------------------
------------------------------
--
-- Description :  This file contains the fft core and two RAMs
      (SMS-Configuratin with output RAM)
--    Also it containes a Data-Collection-RAM (CRAM)
------------------------------------------------------------------------
------------------------------



library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_UNSIGNED.all;

entity fwdfft is
      port(
            RST:  in   std_logic;
            CLK:  in   std_logic;
            READ_EN:   in   std_logic;
            WRITE_EN:  instd_logic;
            READ_DONE:  instd_logic;
            D:  in   std_logic_vector(31 downto 0);
            RESULT_RDY:  out  std_logic;
            Q:  out  std_logic_vector(31 downto 0)
            );
end fwdfft;

architecture fwdfft_ARCH of fwdfft is
      component RM1024x32 is
            port (
                  RCLK  :     in  std_logic;
```

```
            WCLK  :     in  std_logic;
            WEA         :     in  std_logic;
            ENA         :     in  std_logic;
            WEB         :     in  std_logic;
            ENB         :     in  std_logic;
            ADDRA :     in  std_logic_vector (9 downto 0);
            ADDRB :     in  std_logic_vector (9 downto 0);
            DIAR  :     in  std_logic_vector(15 downto 0);
            DIAI  :     in  std_logic_vector(15 downto 0);
            DIBR  :     in  std_logic_vector(15 downto 0);
            DIBI  :     in  std_logic_vector(15 downto 0);
            DOAR  :     out std_logic_vector(15 downto 0);
            DOAI  :     out std_logic_vector(15 downto 0);
            DOBR  :     out std_logic_vector(15 downto 0);
            DOBI  :     out std_logic_vector(15 downto 0)
            );
end component;

component vfft1024
      port(
            clk               : in  std_logic;
            rs                : in  std_logic;
            start       : in  std_logic;
            ce                : in  std_logic;
            scale_mode: in  std_logic;
            di_r        : in  std_logic_vector(15 downto 0);
            di_i        : in  std_logic_vector(15 downto 0);
            fwd_inv     : in  std_logic;
            io_mode0: in  std_logic;
            io_mode1: in  std_logic;
            mwr               : in  std_logic;
            mrd               : in  std_logic;
            ovflo       : out std_logic;
            result     : out std_logic;
            mode_ce     : out std_logic;
            done        : out std_logic;
            edone       : out std_logic;
            io                : out std_logic;
            eio               : out std_logic;
            bank        : out std_logic;
            busy        : out std_logic;
            wea               : out std_logic;
            wea_x       : out std_logic;
            wea_y       : out std_logic;
```

```
            web_x       : out std_logic;
            web_y       : out std_logic;
            ena_x       : out std_logic;
            ena_y       : out std_logic;
            index       : out std_logic_vector(9 downto 0);
            addrr_x     : out std_logic_vector(9 downto 0);
            addrr_y     : out std_logic_vector(9 downto 0);
            addrw_x     : out std_logic_vector(9 downto 0);
            addrw_y     : out std_logic_vector(9 downto 0);
            xk_r        : out std_logic_vector(15 downto 0);
            xk_i        : out std_logic_vector(15 downto 0);
            yk_r        : out std_logic_vector(15 downto 0);
            yk_i        : out std_logic_vector(15 downto 0));
end component;


signal      ZERO: std_logic;
signal     ZERO16:      std_logic_vector(15 downto 0);
signal      ONE:        std_logic;
signal      start:            std_logic;
signal      ce:                     std_logic;
signal      scale_mode: std_logic;
signal      di_r:             std_logic_vector(15 downto 0);
signal      di_i:             std_logic_vector(15 downto 0);
signal      fwd_inv:    std_logic;
signal      io_mode0:   std_logic;
signal      io_mode1:   std_logic;
signal      mwr:              std_logic;
signal      mrd:              std_logic;
signal      ovflo:            std_logic;
signal      result:     std_logic;
signal      mode_ce:    std_logic;
signal      done:             std_logic;
signal      edone:            std_logic;
signal      io:                     std_logic;
signal      eio:              std_logic;
signal      bank:             std_logic;
signal      busy:             std_logic;
signal      wea:              std_logic;
signal      wea_x:            std_logic;
signal      wea_y:            std_logic;
signal      web_x:std_logic;
signal      web_y:            std_logic;
signal      ena_x:            std_logic;
```

```
      signal      ena_y:            std_logic;
      signal      index:            std_logic_vector(9 downto 0);
      signal      addrr_x:    std_logic_vector(9 downto 0);
      signal      addrr_y:    std_logic_vector(9 downto 0);
      signal      addrw_x:    std_logic_vector(9 downto 0);
      signal      addrw_y:    std_logic_vector(9 downto 0);
      signal      xk_r:             std_logic_vector(15 downto 0);
      signal      xk_i:             std_logic_vector(15 downto 0);
      signal      yk_r:             std_logic_vector(15 downto 0);
      signal      yk_i:             std_logic_vector(15 downto 0);
      signal      CRAM_REA:     std_logic;
      --signal      CRAM_REAi:     std_logic;
      signal      CRAM_WEB:     std_logic;
      signal      CRAM_ADDRA:     std_logic_vector(9 downto 0);
      signal      CRAM_ADDRB:     std_logic_vector(9 downto 0);
      signal      CRAM_DOA:       std_logic_vector(31 downto 0);
      signal      SMSRM2_REA:     std_logic;
      signal      SMSRM2_WEB:     std_logic;
      signal      SMSRM2_ADDRA:   std_logic_vector(9 downto 0);
      signal     SMSRM2_ADDRBi:  std_logic_vector(9 downto 0);
      signal      SMSRM2_ADDRB:   std_logic_vector(9 downto 0);
      signal      SMSRM2_DOA:     std_logic_vector(31 downto 0);
      signal      DATA_RDY:   std_logic;
      type fsm_type is (STinit, STload, STfft, STdelay, STunload, STre-
sult);
      signal fsm: fsm_type;

begin


      ZERO   <= '0';
      ZERO16 <= (others=>'0');
      ONE         <= '1';
      fwd_inv    <= '1';-- forward fft
      scale_mode <= '1';
      io_mode0   <= '0';
      io_mode1   <= '1';
      CRAM_WEB <= '1' when WRITE_EN ='1' else '0';
      Q           <=  SMSRM2_DOA;
      SMSRM2_REA <= '1' when READ_EN  ='1' else '0';

      -- CRAM is data collection RAM
      CRAM:  RM1024x32 port map (
```

```
            RCLK => CLK,
            WCLK => CLK,
            WEA => ZERO,
            ENA => CRAM_REA,
            WEB => CRAM_WEB,
            ENB => ONE,
            ADDRA => CRAM_ADDRA,
            ADDRB => CRAM_ADDRB,
            DIAR => ZERO16,
            DIAI => ZERO16,
            DIBR => D(15 downto 0),
            DIBI => D(31 downto 16),
            DOAR => CRAM_DOA(15 downto 0),
            DOAI => CRAM_DOA(31 downto 16),
            DOBR => OPEN,
            DOBI => OPEN
--          DOAR => OPEN,
--          DOAI => OPEN,
--          DOBR => CRAM_DOA(15 downto 0),
--        DOBI => CRAM_DOA(31 downto 16)
            );

      SMS1RM:  RM1024x32 port map (

            RCLK => CLK,
            WCLK => CLK,
            WEA => wea,
            ENA => ce,
            WEB => io,
            ENB => ONE,
            ADDRA => addrw_x,
            ADDRB => addrr_x,
            DIAR => xk_r,
            DIAI => xk_i,
            DIBR => CRAM_DOA(15 downto 0),
            DIBI => CRAM_DOA(31 downto 16),
            DOAR => OPEN,
            DOAI => OPEN,
            DOBR => di_r,
            DOBI => di_i
            );

      SMSRM2:  RM1024x32 port map (
```

```
      RCLK => CLK,
      WCLK => CLK,
      WEA => ZERO,
      ENA => SMSRM2_REA,
      WEB => SMSRM2_WEB,
      ENB => ONE,
      ADDRA => SMSRM2_ADDRA,
      ADDRB => SMSRM2_ADDRB,
      DIAR => ZERO16,
      DIAI => ZERO16,
      DIBR => di_r,
      DIBI => di_i,
      DOAR => SMSRM2_DOA(15 downto 0),
      DOAI => SMSRM2_DOA(31 downto 16),
      DOBR => OPEN,
      DOBI => OPEN
      );

FFT: component vfft1024
port map(
      clk         => CLK ,
      rs          => RST ,
      start       => start ,
      ce          => ce ,
      scale_mode => scale_mode ,
      di_r        => di_r ,
      di_i        => di_i ,
      fwd_inv => fwd_inv ,
      io_mode0 => io_mode0 ,
      io_mode1 => io_mode1 ,
      mwr         => mwr ,
      mrd         => mrd ,
      ovflo       => ovflo ,
      result => result ,
      mode_ce => mode_ce ,
      done        => done ,
      edone       => edone ,
      io          => io ,
      eio         => eio ,
      bank        => bank ,
      busy        => busy ,
      wea         => wea ,
      wea_x       => wea_x ,
      wea_y       => wea_y ,
```

```
      web_x        => web_x ,
      web_y        => web_y ,
      ena_x        => ena_x ,
      ena_y        => ena_y ,
      index        => index ,
      addrr_x => addrr_x ,
      addrr_y    => addrr_y ,
      addrw_x => addrw_x ,
      addrw_y => addrw_y ,
      xk_r         => xk_r ,
      xk_i         => xk_i ,
      yk_r         => yk_r ,
      yk_i         => yk_i
      );

-- controlling write and read addresses of CRAM

CRAM_A: process(CLK, RST)
begin
      if RST='1' then
            CRAM_ADDRA  <= (others=>'0');
      elsif CLK'event and CLK='1' then
            if CRAM_REA ='1' then
                  if( CRAM_ADDRA < "1111111111" )then
                        CRAM_ADDRA <= CRAM_ADDRA + 1;
                  elsif ( CRAM_ADDRA = "1111111111" )then
                        CRAM_ADDRA <= (others =>'0');
                  end if;
            else
                  CRAM_ADDRA <= (others =>'0');
            end if;
      end if;
end process CRAM_A;

CRAM_B: process(CLK, RST)
begin
      if RST='1' then
            CRAM_ADDRB  <= (others=>'0');
            DATA_RDY     <= '0';
      elsif CLK'event and CLK='1' then
            if CRAM_WEB ='1' then
                  if( CRAM_ADDRB < "1111111111" )then
                        CRAM_ADDRB <= CRAM_ADDRB + 1;
                  elsif ( CRAM_ADDRB = "1111111111" ) then
```

```
                        DATA_RDY    <= '1';
                        CRAM_ADDRB <= (others=>'0');
                   end if;
             elsif CRAM_WEB='0' and CRAM_ADDRB="0000000000" then
                   DATA_RDY    <= '0';
             end if;
      end if;
end process CRAM_B;

-- controlling   write and read addresses of SMSRM2

SMSRM2_A: process(CLK, RST)
begin
      if RST='1' then
            SMSRM2_ADDRA  <= (others=>'0');
      elsif CLK'event and CLK='1' then
            if SMSRM2_REA ='1' then
                  if( SMSRM2_ADDRA < "1111111111" )then
                        SMSRM2_ADDRA <= SMSRM2_ADDRA + 1;

                  elsif( SMSRM2_ADDRA = "1111111111" )then
                        SMSRM2_ADDRA <= "0000000000";

                  end if;
            end if;
            if READ_DONE='1' then
                  SMSRM2_ADDRA <= "0000000000";
            end if;
      end if;
end process SMSRM2_A;
SMSRM2_Bi: process(CLK, RST)
begin
      if RST='1' then
            SMSRM2_ADDRBi  <= (others=>'0');
      elsif CLK'event and CLK='1' then
            if SMSRM2_WEB ='1' then
                  if( SMSRM2_ADDRBi < "1111111111" )then
                        SMSRM2_ADDRBi <= SMSRM2_ADDRBi + 1;

                  elsif (SMSRM2_ADDRBi = "1111111111") then
                        SMSRM2_ADDRBi <= (others=>'0');

                  end if;
            else
```

```
                        SMSRM2_ADDRBi <= (others=>'0');
                  end if;
            end if;
      end process SMSRM2_Bi;

--    SMSRM2_ADDRB <= SMSRM2_ADDRBi;
      SMSRM2_B: process(CLK, RST)
      begin
            if RST='1' then
                  SMSRM2_ADDRB  <= (others=>'0');
            elsif CLK'event and CLK='1' then
                  SMSRM2_ADDRB <= SMSRM2_ADDRBi;
            end if;
      end process SMSRM2_B;

      -- FSM finite state machine to control loading data/ fft compu-
tation / unloading data

      fsm_machine: process (CLK, RST)
            variable cnt: INTEGER range 0 to 1025;

      begin

            if RST='1' then
                  fsm   <= STinit;
                  cnt   := 0;
                  mwr   <= '0';
                  start <= '0';
                  mrd   <= '0';
                  ce    <= '0';
                  SMSRM2_WEB  <= '0';
                  CRAM_REA <='0';

            elsif CLK'event and CLK = '1' then
                  mwr    <= '0';
                  start  <= '0';
                  mrd    <= '0';
                  case fsm is
                        when STinit =>
                        RESULT_RDY <='0';
                        cnt:= 0;
                        if DATA_RDY ='1' then
                              mwr      <= '1';
                              CRAM_REA <='1';
```

```
      ce       <= '1';
      fsm <= STload;

end if;

when STload =>

if addrr_x="1111111111" then
      start <= '1';
      fsm <= STfft;
      CRAM_REA <='0';
else
      fsm <= STload;
      CRAM_REA <='1';
end if;

when STfft =>
if done='1' then
      fsm <= STdelay;
else
      fsm <= STfft;
end if;

when STdelay =>

mrd   <= '1';
fsm <= STunload;

when STunload =>

if cnt=1025 then
      SMSRM2_WEB  <= '0';
      ce        <= '0';
      fsm     <= STresult;
      cnt       := 0;
elsif cnt<1025 then
      SMSRM2_WEB  <= '1';
      fsm     <= STunload;
      cnt       := cnt+1;
end if;

when STresult =>
RESULT_RDY <='1';
if READ_DONE='1' then
```

```
                        fsm <= STinit;
                    else
                        fsm <= STresult;
                    end if;

                    when others =>
                    fsm <= STinit;
                end case;
            end if;
        end process;
end fwdfft_ARCH;
```

C.2.3 Frequency Domain Multiplication

The frequency domain multiplier reads the output of the FFT block and multiplies it by the conjugate of the FFT of the local code. The multiplication is achieved using the complex multiplier implementation described in Chapter 5. The complete VHDL code that contains the top level of this section including the input RAM, the output RAM, the local code RAM, and the complex multiplier components are available on the network at

http:\\www.ent.ohiou.edu\~webcad\alaqeeli.

The main VHDL code of the complex multiplier component is shown here.

```
--------------------------------------------------
libraryIEEE;
use IEEE.STD_LOGIC_1164.all;
use IEEE.STD_LOGIC_UNSIGNED.all;

entity cmplxmult_registerd is
      port (
            CLK:in std_logic;
            A: in std_logic_vector(19 downto 0);
            B: in std_logic_vector(13 downto 0);
```

```
            R: out std_logic_vector(15 downto 0);
            I: out std_logic_vector(15 downto 0));
end entity cmplxmult_registerd;

architecture cmplxmult_registerd_arch of cmplxmult_registerd is
component mult10x8
      port (A: in std_logic_vector(9 downto 0);
              B: in std_logic_vector(7 downto 0);
              Y: out std_logic_vector(16 downto 0));
end component;
component mult11x7
      port (A: in std_logic_vector(10 downto 0);
              B: in std_logic_vector(6 downto 0);
              Y: out std_logic_vector(16 downto 0));
end component;
component add17s
      port ( A: in std_logic_vector(16 downto 0);
               B: in std_logic_vector(16 downto 0);
               Y: out std_logic_vector(17 downto 0));
end component;
component add10s
      port ( A: in std_logic_vector(9 downto 0);
               B: in std_logic_vector(9 downto 0);
               Y: out std_logic_vector(10 downto 0));
end component;
component add7s
      port ( A: in std_logic_vector(6 downto 0);
               B: in std_logic_vector(6 downto 0);
               Y: out std_logic_vector(7 downto 0));
end component;

signal ar,ar1,arreg: std_logic_vector (9 downto 0);
signal ai,ai1,aireg: std_logic_vector (9 downto 0);
signal br: std_logic_vector (6 downto 0);
signal bi,bi1,bireg: std_logic_vector (6 downto 0);
signal bi_comp: std_logic_vector (6 downto 0);
signal arsumai,arsumaireg: std_logic_vector (10 downto 0);
signal brsumbi,brsumbireg: std_logic_vector (7 downto 0);
signal brsubbi,brsubbireg: std_logic_vector (7 downto 0);
signal tmp1,tmp1reg: std_logic_vector (16 downto 0);
signal tmp2,tmp2reg,tmp2reg_comp: std_logic_vector (16 downto 0);
signal tmp3,tmp3reg: std_logic_vector (16 downto 0);
signal Rtmp,Rtmpreg: std_logic_vector (17 downto 0);
signal Itmp,Itmpreg: std_logic_vector (17 downto 0);
```

```
signal Rx,Ix: std_logic_vector (15 downto 0);

begin
ar<=A(9 downto 0);
ai<=A(19 downto 10);
br<=B(6 downto 0);
bi<=B(13 downto 7);
bi_comp<=NOT(bi)+1;
Rx(15) <=Rtmpreg(17);
Rx(14 downto 0) <=Rtmpreg(15 downto 1);
Ix(15) <=Itmpreg(17);
Ix(14 downto 0)<=Itmpreg(15 downto 1);
ar1<=NOT(ar);
ai1<=NOT(ai);
bi1<=NOT(bi);
-- registering each stage to fix timing problems when this file is
called by top-level design
process (CLK)
begin
      if CLK'event and CLK='1' then
            arsumaireg <= arsumai;
            brsumbireg <= brsumbi;
            brsubbireg <= brsubbi;
            arreg <= NOT(ar1);
            aireg <= NOT(ai1);
            bireg <= NOT(bi1);
            tmp1reg <=tmp1;
            tmp2reg <=tmp2;
            tmp3reg <=tmp3;
            Rtmpreg <=Rtmp;
            Itmpreg <=Itmp;
            R  <= Rx;
            I  <= Ix;
      end if;
end process;
tmp2reg_comp <= NOT(tmp2reg)+1;

U1_ADD10: add10s  port map ( A => ar, B => ai, Y => arsumai);
U2_ADD7: add7s  port map ( A => br, B => bi, Y => brsumbi);
U3_ADD7: add7s  port map ( A => br, B => bi_comp, Y => brsubbi);
H1_MULT: mult10x8 port map ( A => arreg ,B => brsumbireg  , Y => tmp1 );
H2_MULT: mult11x7 port map ( A => arsumaireg ,B => bireg , Y => tmp2 );
H3_MULT: mult10x8 port map ( A => aireg ,B => brsubbireg , Y => tmp3 );
```

```
U1_ADD17: add17s port map ( A => tmp1reg ,B => tmp2reg_comp , Y => Rtmp
);
U2_ADD17: add17s port map ( A => tmp2reg ,B => tmp3reg , Y => Itmp );
end architecture cmplxmult_registerd_arch;
```

C.2.4 The IFFT Block

The IFFT block is responsible for returning the frequency domain multiplication to the time domain as part of the FFT-based correlator. This section has a similar structure as the FFT Block. The only change as compared to the FFT block is the selection of the inverse FFT mode, which requires the following change in the VHDL code:

```
fwd_inv    <= '0';-- inverse fft
```

C.2.5 The Peak Searcher

This is the last part in the acquisition process. The top level VHDL code contains the input RAM, the peak searching circuit including a multiplier used to calculate the square values, and registers to store the acquisition information. All these codes are available at http:\\www.ent.ohiou.edu\~webcad\alaqeeli. Only the peak searching circuit's VHDL code is shown here.

```
-------------------------------------------------------------------
--
-- Title       : pksrchr_no_atan
-- Design      : peak_search_may22
-- Author      : 0
-- Company     : 0
-------------------------------------------------------------------
-- File        : pksrchr_no_atan.vhd
-- Generated   : Wed May 22 11:55:40 2002
-------------------------------------------------------------------

library IEEE;
use IEEE.STD_LOGIC_1164.all;
```

```
use IEEE.STD_LOGIC_UNSIGNED.all;
entity pksrchr_no_atan is
      port(
            RST : in STD_LOGIC;
            CLK : in STD_LOGIC;
            START : in STD_LOGIC;
            DINreal : in STD_LOGIC_VECTOR(15 downto 0);
            DINimag : in STD_LOGIC_VECTOR(15 downto 0);
            new_gps1ms: in STD_LOGIC;
            RDY: out STD_LOGIC;
            KOUT : out STD_LOGIC_VECTOR(2 downto 0);
            TAO1024 : out STD_LOGIC_VECTOR(9 downto 0);
            P : out STD_LOGIC_VECTOR(30 downto 0);
            RESULT_AVAILABLE:out STD_LOGIC
            );
end pksrchr_no_atan;

architecture pksrchr_no_atan_arch of pksrchr_no_atan is

component mult15
      port (
      clk: IN std_logic;
      a: IN std_logic_VECTOR(14 downto 0);
      b: IN std_logic_VECTOR(14 downto 0);
      q: OUT std_logic_VECTOR(29 downto 0));
end component;

signal A16,B16,Atmp,Btmp :std_logic_vector(15 downto 0);
signal A15,B15: std_logic_vector(14 downto 0);
signal AA,BB,AAreg,BBreg: std_logic_vector(29 downto 0);
signal AAplusBB: std_logic_vector(30 downto 0);
signal cntr1024,cntr1024_dly,tmp1: std_logic_vector(9 downto 0);
signal kcount,kcount_dly,tmp2: std_logic_vector(2 downto 0);
signal DONE,DONEdly,EN,ENdly,tmp3 : std_logic;
signal PMAXreg : std_logic_vector(30 downto 0);
signal TAOreg: std_logic_vector (9 downto 0);
signal Kreg: std_logic_vector(2 downto 0);
type statename is (ST0,ST1);
signal fsm : statename;

begin

      A16<=DINreal;
      B16<=DINimag;
```

```
      A15 <=Atmp(14 downto 0); -- abs(A)
      B15 <=Btmp(14 downto 0); -- abs(B)
      process (CLK)
      begin
            if CLK'event and CLK='1' then
                  if A16(15)='1' then
                        if A16="1000000000000000" then
                              Atmp<=not(A16);
                        else
                              Atmp <= not(A16)+1;
                        end if;
                  else
                        Atmp <= A16;
                  end if;
                  if B16(15)='1' then
                        if B16="1000000000000000" then
                              Btmp<=not(B16);
                        else
                              Btmp <= not(B16)+1;
                        end if;
                  else
                        Btmp <= B16;
                  end if;
            end if;
      end process;
      U1: mult15 port map (clk =>CLK, a => A15, b => A15, q => AA);  -
- A^2
      U2: mult15 port map (clk =>CLK, a => B15, b => B15, q => BB);  -
- B^2

      process(CLK)
      begin
            if CLK'event and CLK='1' then
                  AAreg <=AA;               -- A^2  registered
                  BBreg <=BB;               -- B^2  registered
            end if;
      end process;

      AAplusBB <= (('0'&AAreg) + ('0'&BBreg)); -- A^2 + B^2

      fsm_stat:process (CLK,RST)
      begin
            if RST='1' then
                  fsm <= ST0;
```

```
            cntr1024 <=(others =>'0');
            kcount <=(others =>'0');
            EN<='0';
            DONE <='0';
            RDY <='1';
      elsif CLK'event and CLK='1' then
            case fsm is
                  when ST0 =>

                        if START='1' then
                              fsm <= ST1;
                        else
                              fsm <= ST0;
                        end if;
                        cntr1024 <=(others =>'0');
                        DONE <='0';
                        RDY <='1';
                        EN<='0';
                  when ST1 =>

                        if cntr1024="1111111111" then
                              if kcount="100" then
                                    cntr1024 <=(others =>'0');
                                    fsm <= ST0;
                                    kcount <=(others =>'0');
                                    DONE <='1';
                                    RDY<='1';
                              else
                                    cntr1024 <=(others =>'0');
                                    fsm <= ST0;
                                    kcount <=kcount+1;
                                    RDY <='1';
                              end if;
                        else
                              fsm <= ST1;
                              cntr1024 <= cntr1024+1;
                              EN<='1';
                              RDY <='0';
                        end if;

                  when others => null;
            end case;
      end if;
end process;
```

```
process(RST,CLK)
begin
      if RST='1' then
            PMAXreg <= (others =>'0');
            TAOreg <= (others =>'0');
            Kreg <=(others =>'0');
      elsif CLK'event and CLK='1' then
            if new_gps1ms='1' then
                  PMAXreg <= (others =>'0');
                  TAOreg <= (others =>'0');
                  Kreg <=(others =>'0');
            else
                  if ENdly='1' then
                        if AAplusBB > PMAXreg then
                              PMAXreg <= AAplusBB;
                              TAOreg <= cntr1024_dly;
                              Kreg <= kcount_dly;
                        end if;
                  end if;
            end if;
      end if;
end process;

delay_proc:process(CLK)
begin
      if CLK'event and CLK='1' then
            tmp1<= cntr1024;
            cntr1024_dly<=tmp1;
            tmp2<=kcount;
            kcount_dly <= tmp2;
            ENdly <= EN;
            tmp3 <= DONE;
            DONEdly <= tmp3;
      end if;
end process;
      process(RST,CLK)
begin
      if RST='1' then
            P <=(others =>'0');
            TAO1024 <=(others =>'0');
            KOUT <= (others =>'0');
            RESULT_AVAILABLE <='0';
      elsif CLK'event and CLK='1' then
            if DONEdly='1' then
```

```
                    P <=PMAXreg;
                    TAO1024 <=TAOreg;
                    KOUT <= Kreg;
                    RESULT_AVAILABLE <='1';
                else
                 if START='1' then -- new data comes, clear values;
                           P <=(others =>'0');
                           TAO1024 <=(others =>'0');
                           KOUT <= (others=>'0');
                           RESULT_AVAILABLE <='0';
                 end if;
                end if;
           end if;
      end process;
end pksrchr_no_atan_arch;
```

C.3 Serial Correlators (Tracking-Like Estimator)

This section of the design is responsible for implementing the serial correlators that zoom in around the peak for accurate estimations of the code and carrier phases. The top level VHDL code contains input RAM, NCO, RAM for local code, registers for the Early-Prompt-Late design, the accumulators, and registers for the output results. The VHDL code of the NCO is similar to the one used in the acquisition process, therefore it will not be shown here. The local code is generated and upsampled with Matlab and then pre-stored in a RAM that was generated by Xilinx core generator. Only a part of the VHDL code is shown here. The complete VHDL code is available at http:\\www.ent.ohiou.edu\~webcad\alaqeeli.

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_unsigned.all;
entitytrackctrl is
      port (
       datain: in std_logic_vector (11 downto 0);
```

```
       clk: in std_logic; --5 Mhz
       m:in std_logic_vector (30 downto 0); --freq to nco
       phasein:in std_logic_vector (31 downto 0);   --input phase to
nco
       tao:in std_logic_vector (12 downto 0);   --code phase
       readyi: in std_logic;--start track 010 1=start
       ek_o:out std_logic_vector(24 downto 0); --epl acc results I part
       pk_o: out std_logic_vector(24 downto 0);
       lk_o: out std_logic_vector(24 downto 0);
       addresssig: out std_logic_vector(12 downto 0);
       readyo: out std_logic
       ) ;
end entity;

architecture trackctrl of trackctrl is

--componentnco
componentnco_3
       port (
       clk: in std_logic; --10 Mhz
       phasein: in std_logic_vector (31 downto 0); --phase, 0-2pi
       m:in std_logic_vector (30 downto 0); --adding step
       srst:in std_logic;                   --reset, input phase
       clo:out std_logic;                   --a clock out at the freq
       sino: out std_logic_vector(1 downto 0);
       coso: out std_logic_vector(1 downto 0)
       ) ;
end component;

component cacodram
      port (
      addr: IN std_logic_VECTOR(12 downto 0);
      clk: IN std_logic;
      dout: OUT std_logic_VECTOR(0 downto 0);
      en: IN std_logic);
end component;



constant N: integer :=11;--highest bit pos of input signal
      signal ONE,localcode,ramclk,tracking,localcode3, localcode1,
localcode2: std_logic; --1  working
      signal counter: std_logic_vector (15 downto 0);  --counter1 is
for the signal
      signal mreg: std_logic_vector (30 downto 0);
```

```
      signal  phaseinreg: std_logic_vector (31 downto 0);
      signal taoreg: std_logic_vector (12 downto 0); --registered
values for tracking address
            --in the 1st version 11 downto 0 is taken is the input
            --should be changed to mem output
      signal sigcurl, sigcurp, sigcure, sigdemod,
sigcur,sigcurd:std_logic_vector (11 downto 0); --current tao tao-1 e
tao p tao+1 l inputs
      --sigdemod is demodulated
      --sigcur and sigcurd are delayed
      --sigcurl p e are despread
      signal sin_trk,cos_trk: std_logic_vector(1 downto 0);
      signal datain1: std_logic_vector(11 downto 0);
      signal ek, lk, pk, sigcurl1, sigcurp1,
sigcure1:std_logic_vector(24 downto 0);
      signal Qek, Qlk, Qpk, Qsigcurl1, Qsigcurp1,
Qsigcure1:std_logic_vector(24 downto 0);
      signal CA6: std_logic_vector(0 downto 0);
      --temp for acc output

begin
ONE <='1';
nco: nco_3 port map (clk=> clk, srst=>readyi, phasein => phasein, m =>
m,clo => OPEN, sino => sin_trk, coso=>OPEN);
addresssig <=counter(12 downto 0)-1;
ek_o <= ek; pk_o <= pk;  lk_o <= lk;
process (clk)
      begin
            if (clk'event and clk='1') then
                  datain1<=datain;
                  localcode1 <= localcode; localcode2 <= localcode1;
localcode3 <= localcode2;
                  if  readyi='1' then
                        ek <=(ek'range=>'0');
                        pk <=(pk'range=>'0');
                        lk <=(lk'range=>'0');
                        sigcure <=(sigcure'range=>'0');
                        sigcurp <=(sigcurp'range=>'0');
                        sigcurl <=(sigcurl'range=>'0');
                        sigcure1 <=(sigcure1'range=>'0');
                        sigcurp1 <=(sigcurp1'range=>'0');
                        sigcurl1 <=(sigcurl1'range=>'0');
                        sigcur <=(sigcur'range=>'0');
                        sigcurd <=(sigcurd'range=>'0');
                        taoreg <= 4999 - tao;
```

```
                        phaseinreg <=phasein;
                        mreg <=m;
                        counter <=(counter'range=>'0');
                        readyo <='0';
                        tracking <='0';
                  else

                        if taoreg <5000-1 then
                              taoreg <= taoreg+1;
                        else
                              taoreg <= taoreg + 1-5000;
                        end if;  --circular correlation
                        --corr
                        if sin_trk = "01" then
                              sigdemod <= datain;
                        elsif sin_trk="11" then
                              sigdemod <= not(datain)+'1';
                        else
                              sigdemod <="000000000000";
                        end if;

                        sigcur <=sigdemod;
                        sigcurd <=sigcur;
                        if localcode3='0'   then
                              sigcure <=sigdemod;
                              sigcurp <=sigcur;
                              sigcurl <=sigcurd;
                        else
                              sigcure <= not(sigdemod)+'1';
                              sigcurp <= not(sigcur)+'1';
                              sigcurl <= not(sigcurd)+'1';
                        end if;
                        if counter=5 then
                              tracking <='1';
                        end if;
                        if counter>=5005 then-- change this to the
right value (may be 5008)
                              tracking<='0';
                        readyo<='1';
                        else
                              counter<=counter+1;  -- I put this here
instead of puting it later
                        end if;
```

```
                        if sigcure(N)='0' then
                              sigcure1<="0000000000000" & sigcure;
                        else
                              sigcure1<="1111111111111" & sigcure;
                        end if;
                        if sigcurp(N)='0' then
                              sigcurp1<="0000000000000" & sigcurp;
                        else
                              sigcurp1<="1111111111111" & sigcurp;
                        end if;
                        if sigcurl(N)='0' then
                              sigcurl1<="0000000000000" & sigcurl;
                        else
                              sigcurl1<="1111111111111" & sigcurl;
                        end if;

                        if tracking = '1' then
                              ek<=ek+sigcure1;
                              pk<=pk+sigcurp1;
                              lk<=lk+sigcurl1;
                        end if;--tracking
                        -- counter<=counter+1; I put this up a little
bit
                  end if; -- readyi
            end if; -- clk
      end process;


ramclk<=not(clk);
-- 5000 samples of ca code stored in ram
U_CACOD: cacodram
            port map (
                  addr => taoreg,
                  clk => ramclk,
                  dout => CA6,
                  en => ONE);
localcode <= '1' when CA6=1 else '0';
end trackctrl;
```

Appendix D

C Codes

Appendix D contains the C codes that were used for communicating between the PC and the FPGA board. These interface codes were responsible for resetting the board, and downloading the bit-stream implementation files into the FPGA. In addition, these C codes were responsible for sending the GPS data and the intermediate data to the FPGA. Reading the results back to the PC were also called from these C codes.

D.1 C Code for Carrier Wipe-off and Averaging

```
#include <stdio.h>
// Include Header, note that this also includes "dimesdl.h"
#include "vidime.h"
#include <conio.h>
#include <math.h>
#define  m_in 5000
#define  m_out 1024

FILE *stream;

void initialize(DIME_HANDLE hDIME)
{
```

```
    DWORD Cntr;
    DWORD NumModules;

      DIME_SetOscillatorFrequency(hDIME,1,50.000,NULL); // Set SYSCLK
to 50Mhz
      DIME_SetOscillatorFrequency(hDIME,2,40.000,NULL); // Set DSPCLK
to 40Mhz
      DIME_SetOscillatorFrequency(hDIME,3,50.000,NULL); // Set PIXCLK
to 50Mhz

      if( DIME_SmartScan(hDIME) != ssOK )
      {
            printf("Error in Smart Scan!\n");
            exit(2);
      }

      NumModules = DIME_GetNumberOfModules(hDIME);
      printf("Found %d Modules\n",NumModules);
      printf("The Modules are:\n");

      for( Cntr=0 ; Cntr<NumModules ; Cntr++)
            printf("Module %d is a
%s\n",Cntr,DIME_GetModuleDescription(hDIME,Cntr));

      if( DIME_BootDevice(hDIME,"nco_dwnsmplr_ballynue3.bit",NumMod-
ules-1,0,NULL) != cfgOK_STATUS)
      {
            printf("Error configuring on board Virtex!\n");
            exit(3);
      }
}

int main(int argc, char **argv)
{

      DIME_HANDLE   hDIME;
      DWORD         Data_in[5004];
      DWORD         DataFPGA_out[m_out],Data0[m_in];
      DWORD         Data1[m_in],Data2[m_in];
      DWORD         Data3[m_in],Data4[m_in];
      DWORD         done;
      int           error, i, j;
      DWORD         Cntr2,pcaddrs;
      FILE *        stream;
      error = 0;
```

```
pcaddrs = 0;

stream = fopen( "gps1ms7.txt", "r" );
for(Cntr2=0 ; Cntr2<m_in ; Cntr2++)
{
  fscanf( stream, "%d", &Data_in[Cntr2] );
      fscanf( stream, "\n");
}
fclose( stream );

if( (hDIME = OpenDIMEBoard()) == NULL )
{
      printf("No DIME Carrier Card Found!\n");
      exit(1);
}

initialize(hDIME);
DIME_VirtexResetEnable(hDIME);
DIME_PCIReset(hDIME);
DIME_VirtexResetDisable(hDIME);
Data_in[5000]=Data_in[0];
Data_in[5001]=Data_in[1];
Data_in[5002]=Data_in[2];
Data_in[5003]=Data_in[3];

for (j=0; j<5000; j++)
{
      Data0[j]=Data_in[j];
      Data1[j]=Data_in[j+1];
      Data2[j]=Data_in[j+2];
      Data3[j]=Data_in[j+3];
      Data4[j]=Data_in[j+4];
}

// ------------ k=0 ------------
viDIME_DMAWrite(hDIME,Data0,m_in,0,NULL,NULL,5000);
do    {
      done = viDIME_ReadRegister(hDIME,3,5000);
} while(done != 5 );
viDIME_DMARead(hDIME,DataFPGA_out,m_out,0,NULL,NULL,5000);
stream = fopen( "DBB1024k0.txt", "w" );
for(i=0; i<1024; i++)
{
      fprintf(stream,"%d\n",DataFPGA_out[i] );
```

```
}
fclose (stream);
viDIME_WriteRegister(hDIME,2,1,5000);
// ------------ k=1 ------------
viDIME_DMAWrite(hDIME,Data1,m_in,0,NULL,NULL,5000);
do    {
      done = viDIME_ReadRegister(hDIME,3,5000);
} while(done != 5 );
viDIME_DMARead(hDIME,DataFPGA_out,m_out,0,NULL,NULL,5000);
stream = fopen( "DBB1024k1.txt", "w" );
for(i=0; i<1024; i++)
{
      fprintf(stream,"%d\n",DataFPGA_out[i] );
}
fclose (stream);
viDIME_WriteRegister(hDIME,2,1,5000);
// ------------ k=2 ------------
viDIME_DMAWrite(hDIME,Data2,m_in,0,NULL,NULL,5000);
do    {
      done = viDIME_ReadRegister(hDIME,3,5000);
} while(done != 5 );
viDIME_DMARead(hDIME,DataFPGA_out,m_out,0,NULL,NULL,5000);
stream = fopen( "DBB1024k2.txt", "w" );
for(i=0; i<1024; i++)
{
      fprintf(stream,"%d\n",DataFPGA_out[i] );
}
fclose (stream);
viDIME_WriteRegister(hDIME,2,1,5000);
// ------------ k=3 ------------
viDIME_DMAWrite(hDIME,Data3,m_in,0,NULL,NULL,5000);
do    {
      done = viDIME_ReadRegister(hDIME,3,5000);
} while(done != 5 );
viDIME_DMARead(hDIME,DataFPGA_out,m_out,0,NULL,NULL,5000);
stream = fopen( "DBB1024k3.txt", "w" );
for(i=0; i<1024; i++)
{
      fprintf(stream,"%d\n",DataFPGA_out[i] );
}
fclose (stream);
viDIME_WriteRegister(hDIME,2,1,5000);
// ------------ k=4 ------------
viDIME_DMAWrite(hDIME,Data4,m_in,0,NULL,NULL,5000);
```

```
      do    {
            done = viDIME_ReadRegister(hDIME,3,5000);
      } while(done != 5 );
      viDIME_DMARead(hDIME,DataFPGA_out,m_out,0,NULL,NULL,5000);
      stream = fopen( "DBB1024k4.txt", "w" );
      for(i=0; i<1024; i++)
      {
            fprintf(stream,"%d\n",DataFPGA_out[i] );
      }
      fclose (stream);

    printf("GPS signal was downsampled successfully");
      CloseDIMEBoard(hDIME);
      getch();
      exit(0);
}
```

D.2 C Code for FFT Block

```
#include <stdio.h>
// Include Header, note that this also includes "dimesdl.h"
#include "vidime.h"
#include <conio.h>
#include <math.h>
#define  m 1024

FILE *stream;

void initialize(DIME_HANDLE hDIME){

    DWORD Cntr;
    DWORD NumModules;

      DIME_SetOscillatorFrequency(hDIME,1,50.000,NULL); // Set SYSCLK
to 50Mhz
      DIME_SetOscillatorFrequency(hDIME,2,40.000,NULL); // Set DSPCLK
to 40Mhz
```

```
      DIME_SetOscillatorFrequency(hDIME,3,50.000,NULL); // Set PIXCLK
to 50Mhz

      if( DIME_SmartScan(hDIME) != ssOK )
      {
            printf("Error in Smart Scan!\n");
            exit(2);
      }

      NumModules = DIME_GetNumberOfModules(hDIME);
      printf("Found %d Modules\n",NumModules);
      printf("The Modules are:\n");

      for( Cntr=0 ; Cntr<NumModules ; Cntr++)
            printf("Module %d is a
%s\n",Cntr,DIME_GetModuleDescription(hDIME,Cntr));

      if( DIME_BootDevice(hDIME,"fftblock.bit",NumModules-1,0,NULL) !=
cfgOK_STATUS)
      {
            printf("Error configuring on board Virtex!\n");
            exit(3);
      }
}

int main(int argc, char **argv)
{

      DIME_HANDLE   hDIME;
      DWORD         Data_in[m];
      DWORD         DataFPGA_out[m];
      DWORD         fftdone;
      int           error, i, k;
      DWORD         Cntr2;
      FILE *        stream;
      error = 0;

      if( (hDIME = OpenDIMEBoard()) == NULL ){
            printf("No DIME Carrier Card Found!\n");
            exit(1);
      }

      initialize(hDIME);
      DIME_VirtexResetEnable(hDIME);
      viDIME_DMAAbort(hDIME);
```

```
DIME_PCIReset(hDIME);
DIME_VirtexResetDisable(hDIME);

// --------  k=0 -------------//
stream = fopen( "DBB1024k0.txt", "r" );
//stream = fopen( "DBB0.txt", "r" );
for(Cntr2=0 ; Cntr2<m ; Cntr2++)
{
  fscanf( stream, "%d", &Data_in[Cntr2] );
      fscanf( stream, "\n");
}
fclose( stream );
viDIME_DMAWrite(hDIME,Data_in,m,0,NULL,NULL,5000);
do{
    fftdone = viDIME_ReadRegister(hDIME,3,5000);
  } while(fftdone != 5 );
viDIME_DMARead(hDIME,DataFPGA_out,m,0,NULL,NULL,5000);
stream = fopen ("fftout0x.txt","w");
for(i=0; i<1024; i++)
{
      fprintf(stream,"%d\n",DataFPGA_out[i]);
}
fclose (stream);
viDIME_WriteRegister(hDIME,2,1,5000);

// --------  k=1 -------------//

stream = fopen( "DBB1024k1.txt", "r" );
for(Cntr2=0 ; Cntr2<m ; Cntr2++)
{
  fscanf( stream, "%d", &Data_in[Cntr2] );
      fscanf( stream, "\n");
}
fclose( stream );
viDIME_DMAWrite(hDIME,Data_in,m,0,NULL,NULL,5000);
do{
    fftdone = viDIME_ReadRegister(hDIME,3,5000);
  } while(fftdone != 5 );
viDIME_DMARead(hDIME,DataFPGA_out,m,0,NULL,NULL,5000);
stream = fopen ("fftout1x.txt","w");
for(i=0; i<1024; i++)
{
      fprintf(stream,"%d\n",DataFPGA_out[i]);
}
```

```
fclose (stream);
viDIME_WriteRegister(hDIME,2,1,5000);

// --------  k=2 -------------//
stream = fopen( "DBB1024k2.txt", "r" );
for(Cntr2=0 ; Cntr2<m ; Cntr2++)
{
  fscanf( stream, "%d", &Data_in[Cntr2] );
      fscanf( stream, "\n");
}
fclose( stream );
viDIME_DMAWrite(hDIME,Data_in,m,0,NULL,NULL,5000);
do{
    fftdone = viDIME_ReadRegister(hDIME,3,5000);
  } while(fftdone != 5 );
viDIME_DMARead(hDIME,DataFPGA_out,m,0,NULL,NULL,5000);
stream = fopen ("fftout2x.txt","w");
for(i=0; i<1024; i++)
{
      fprintf(stream,"%d\n",DataFPGA_out[i]);
}
fclose (stream);
viDIME_WriteRegister(hDIME,2,1,5000);

// --------  k=3 -------------//
stream = fopen( "DBB1024k3.txt", "r" );
for(Cntr2=0 ; Cntr2<m ; Cntr2++)
{
  fscanf( stream, "%d", &Data_in[Cntr2] );
      fscanf( stream, "\n");
}
fclose( stream );
viDIME_DMAWrite(hDIME,Data_in,m,0,NULL,NULL,5000);
do{
    fftdone = viDIME_ReadRegister(hDIME,3,5000);
  } while(fftdone != 5 );
viDIME_DMARead(hDIME,DataFPGA_out,m,0,NULL,NULL,5000);
stream = fopen ("fftout3x.txt","w");
for(i=0; i<1024; i++)
{
      fprintf(stream,"%d\n",DataFPGA_out[i]);
}
fclose (stream);
viDIME_WriteRegister(hDIME,2,1,5000);
```

```
      // --------  k=4 -------------//
      stream = fopen( "DBB1024k4.txt", "r" );
      for(Cntr2=0 ; Cntr2<m ; Cntr2++)
      {
        fscanf( stream, "%d", &Data_in[Cntr2] );
            fscanf( stream, "\n");
      }
      fclose( stream );
      viDIME_DMAWrite(hDIME,Data_in,m,0,NULL,NULL,5000);
      do{
          fftdone = viDIME_ReadRegister(hDIME,3,5000);
        } while(fftdone != 5 );
      viDIME_DMARead(hDIME,DataFPGA_out,m,0,NULL,NULL,5000);
      stream = fopen ("fftout4x.txt","w");
      for(i=0; i<1024; i++)
      {
            fprintf(stream,"%d\n",DataFPGA_out[i]);
      }
      fclose (stream);
      viDIME_WriteRegister(hDIME,2,1,5000);
      // ----------------------------------

      printf(" --- fft completed ---- ");
      CloseDIMEBoard(hDIME);
    getch();
      exit(0);
}
```

D.3 C Code for Frequency Domain Multiplication

```
#include <stdio.h>
// Include Header, note that this also includes "dimesdl.h"
#include "vidime.h"
#include <conio.h>
#include <math.h>
#define  m_out 1024
```

```
FILE *stream;

void initialize(DIME_HANDLE hDIME){

    DWORD Cntr;
    DWORD NumModules;

      DIME_SetOscillatorFrequency(hDIME,1,50.000,NULL); // Set SYSCLK
to 50Mhz
      DIME_SetOscillatorFrequency(hDIME,2,40.000,NULL); // Set DSPCLK
to 40Mhz
      DIME_SetOscillatorFrequency(hDIME,3,50.000,NULL); // Set PIXCLK
to 50Mhz

      if( DIME_SmartScan(hDIME) != ssOK )
      {
            printf("Error in Smart Scan!\n");
            exit(2);
      }

      NumModules = DIME_GetNumberOfModules(hDIME);
      printf("Found %d Modules\n",NumModules);
      printf("The Modules are:\n");

      for( Cntr=0 ; Cntr<NumModules ; Cntr++)
            printf("Module %d is a
%s\n",Cntr,DIME_GetModuleDescription(hDIME,Cntr));

      if( DIME_BootDevice(hDIME,"freq_domain_multip_reg.bit",NumMod-
ules-1,0,NULL) != cfgOK_STATUS)
      {
            printf("Error configuring on board Virtex!\n");
            exit(3);
      }
}

int main(int argc, char **argv)
{

      DIME_HANDLE   hDIME;
      DWORD         Data_in[m_out];
      DWORD        Data_out[m_out];
      DWORD         DataFPGA_out[m_out];
      DWORD         done;
```

```
int           error, i;
DWORD         Cntr2;
FILE *        stream;
error = 0;

if( (hDIME = OpenDIMEBoard()) == NULL ){
      printf("No DIME Carrier Card Found!\n");
      exit(1);
      }
// ----- k=0 ------- //
initialize(hDIME);
DIME_VirtexResetEnable(hDIME);
DIME_PCIReset(hDIME);
DIME_VirtexResetDisable(hDIME);
stream = fopen( "fftout0x.txt", "r" );
for(Cntr2=0 ; Cntr2<m_out ; Cntr2++)
{
  fscanf( stream, "%d", &Data_in[Cntr2] );
      fscanf( stream, "\n");
}
fclose( stream );
viDIME_DMAWrite(hDIME,Data_in,m_out,0,NULL,NULL,5000);
do{
  done = viDIME_ReadRegister(hDIME,3,5000);
   } while(done != 5 );
viDIME_DMARead(hDIME,DataFPGA_out,m_out,0,NULL,NULL,5000);
stream = fopen("mult0.txt","w");
for(i=0; i<m_out; i++)
{
      fprintf(stream,"%d\n",DataFPGA_out[i]);
}
fclose (stream);

// ----- k=1 ------- //
initialize(hDIME);
DIME_VirtexResetEnable(hDIME);
DIME_PCIReset(hDIME);
DIME_VirtexResetDisable(hDIME);
stream = fopen( "fftout1x.txt", "r" );
for(Cntr2=0 ; Cntr2<m_out ; Cntr2++)
{
  fscanf( stream, "%d", &Data_in[Cntr2] );
      fscanf( stream, "\n");
}
```

```
fclose( stream );
viDIME_DMAWrite(hDIME,Data_in,m_out,0,NULL,NULL,5000);
do{
  done = viDIME_ReadRegister(hDIME,3,5000);
   } while(done != 5 );
viDIME_DMARead(hDIME,DataFPGA_out,m_out,0,NULL,NULL,5000);
stream = fopen("mult1.txt","w");
for(i=0; i<m_out; i++)
{
      fprintf(stream,"%d\n",DataFPGA_out[i]);
}
fclose (stream);

// ----- k=2 ------- //
initialize(hDIME);
DIME_VirtexResetEnable(hDIME);
DIME_PCIReset(hDIME);
DIME_VirtexResetDisable(hDIME);
stream = fopen( "fftout2x.txt", "r" );
for(Cntr2=0 ; Cntr2<m_out ; Cntr2++)
{
  fscanf( stream, "%d", &Data_in[Cntr2] );
      fscanf( stream, "\n");
}
fclose( stream );
viDIME_DMAWrite(hDIME,Data_in,m_out,0,NULL,NULL,5000);
do{
  done = viDIME_ReadRegister(hDIME,3,5000);
   } while(done != 5 );
viDIME_DMARead(hDIME,DataFPGA_out,m_out,0,NULL,NULL,5000);
stream = fopen("mult2.txt","w");
for(i=0; i<m_out; i++)
{
      fprintf(stream,"%d\n",DataFPGA_out[i]);
}
fclose (stream);

// ----- k=3 ------- //
initialize(hDIME);
DIME_VirtexResetEnable(hDIME);
DIME_PCIReset(hDIME);
DIME_VirtexResetDisable(hDIME);
stream = fopen( "fftout3x.txt", "r" );
for(Cntr2=0 ; Cntr2<m_out ; Cntr2++)
```

```
{
  fscanf( stream, "%d", &Data_in[Cntr2] );
      fscanf( stream, "\n");
}
fclose( stream );
viDIME_DMAWrite(hDIME,Data_in,m_out,0,NULL,NULL,5000);
do{
  done = viDIME_ReadRegister(hDIME,3,5000);
   } while(done != 5 );
viDIME_DMARead(hDIME,DataFPGA_out,m_out,0,NULL,NULL,5000);
stream = fopen("mult3.txt","w");
for(i=0; i<m_out; i++)
{
      fprintf(stream,"%d\n",DataFPGA_out[i]);
}
fclose (stream);

// ----- k=4 ------- //
initialize(hDIME);
DIME_VirtexResetEnable(hDIME);
DIME_PCIReset(hDIME);
DIME_VirtexResetDisable(hDIME);
stream = fopen( "fftout4x.txt", "r" );
for(Cntr2=0 ; Cntr2<m_out ; Cntr2++)
{
  fscanf( stream, "%d", &Data_in[Cntr2] );
      fscanf( stream, "\n");
}
fclose( stream );
viDIME_DMAWrite(hDIME,Data_in,m_out,0,NULL,NULL,5000);
do{
  done = viDIME_ReadRegister(hDIME,3,5000);
   } while(done != 5 );
viDIME_DMARead(hDIME,DataFPGA_out,m_out,0,NULL,NULL,5000);
stream = fopen("mult4.txt","w");
for(i=0; i<m_out; i++)
{
      fprintf(stream,"%d\n",DataFPGA_out[i]);
}
fclose (stream);
      printf(" I am done from testing");
CloseDIMEBoard(hDIME);
getch();
exit(0);
```

```
}
```

D.4 C Code for FFT Block

```
#include <stdio.h>
// Include Header, note that this also includes "dimesdl.h"
#include "vidime.h"
#include <conio.h>
#include <math.h>
#define  m 1024

FILE *stream;

void initialize(DIME_HANDLE hDIME){

    DWORD Cntr;
    DWORD NumModules;

      DIME_SetOscillatorFrequency(hDIME,1,50.000,NULL); // Set SYSCLK
to 50Mhz
      DIME_SetOscillatorFrequency(hDIME,2,40.000,NULL); // Set DSPCLK
to 40Mhz
      DIME_SetOscillatorFrequency(hDIME,3,50.000,NULL); // Set PIXCLK
to 50Mhz

      if( DIME_SmartScan(hDIME) != ssOK )
      {
            printf("Error in Smart Scan!\n");
            exit(2);
      }

      NumModules = DIME_GetNumberOfModules(hDIME);
      printf("Found %d Modules\n",NumModules);
      printf("The Modules are:\n");

      for( Cntr=0 ; Cntr<NumModules ; Cntr++)
            printf("Module %d is a
%s\n",Cntr,DIME_GetModuleDescription(hDIME,Cntr));

      if( DIME_BootDevice(hDIME,"ifftblock.bit",NumModules-1,0,NULL)
!= cfgOK_STATUS)
      {
            printf("Error configuring on board Virtex!\n");
```

```
            exit(3);
      }
}

int main(int argc, char **argv)
{

      DIME_HANDLE   hDIME;
      DWORD         Data_in[m];
      DWORD         DataFPGA_out[m];
      DWORD         fftdone;
      int           error, i, k;
      DWORD         Cntr2;
      FILE *        stream;
      error = 0;

      if( (hDIME = OpenDIMEBoard()) == NULL ){
            printf("No DIME Carrier Card Found!\n");
            exit(1);
      }

      initialize(hDIME);
      DIME_VirtexResetEnable(hDIME);
      viDIME_DMAAbort(hDIME);
      DIME_PCIReset(hDIME);
      DIME_VirtexResetDisable(hDIME);

      // --------  k=0 -------------//
      stream = fopen( "mult0.txt", "r" );
      for(Cntr2=0 ; Cntr2<m ; Cntr2++)
      {
        fscanf( stream, "%d", &Data_in[Cntr2] );
            fscanf( stream, "\n");
      }
      fclose( stream );
      viDIME_DMAWrite(hDIME,Data_in,m,0,NULL,NULL,5000);
      do{
          fftdone = viDIME_ReadRegister(hDIME,3,5000);
        } while(fftdone != 5 );
      viDIME_DMARead(hDIME,DataFPGA_out,m,0,NULL,NULL,5000);
      stream = fopen ("ifftout0.txt","w");
      for(i=0; i<1024; i++)
      {
            fprintf(stream,"%d\n",DataFPGA_out[i]);
```

```
}
fclose (stream);
viDIME_WriteRegister(hDIME,2,1,5000);

// --------  k=1 -------------//
stream = fopen( "mult1.txt", "r" );
for(Cntr2=0 ; Cntr2<m ; Cntr2++)
{
  fscanf( stream, "%d", &Data_in[Cntr2] );
      fscanf( stream, "\n");
}
fclose( stream );
viDIME_DMAWrite(hDIME,Data_in,m,0,NULL,NULL,5000);
do{
    fftdone = viDIME_ReadRegister(hDIME,3,5000);
  } while(fftdone != 5 );
viDIME_DMARead(hDIME,DataFPGA_out,m,0,NULL,NULL,5000);
stream = fopen ("ifftout1.txt","w");
for(i=0; i<1024; i++)
{
      fprintf(stream,"%d\n",DataFPGA_out[i]);
}
fclose (stream);
viDIME_WriteRegister(hDIME,2,1,5000);

// --------  k=2 -------------//
stream = fopen( "mult2.txt", "r" );
for(Cntr2=0 ; Cntr2<m ; Cntr2++)
{
  fscanf( stream, "%d", &Data_in[Cntr2] );
      fscanf( stream, "\n");
}
fclose( stream );
viDIME_DMAWrite(hDIME,Data_in,m,0,NULL,NULL,5000);
do{
    fftdone = viDIME_ReadRegister(hDIME,3,5000);
  } while(fftdone != 5 );
viDIME_DMARead(hDIME,DataFPGA_out,m,0,NULL,NULL,5000);
stream = fopen ("ifftout2.txt","w");
for(i=0; i<1024; i++)
{
      fprintf(stream,"%d\n",DataFPGA_out[i]);
}
fclose (stream);
```

```
viDIME_WriteRegister(hDIME,2,1,5000);
// --------  k=3 -------------//
stream = fopen( "mult3.txt", "r" );
for(Cntr2=0 ; Cntr2<m ; Cntr2++)
{
  fscanf( stream, "%d", &Data_in[Cntr2] );
      fscanf( stream, "\n");
}
fclose( stream );
viDIME_DMAWrite(hDIME,Data_in,m,0,NULL,NULL,5000);
do{
    fftdone = viDIME_ReadRegister(hDIME,3,5000);
  } while(fftdone != 5 );
viDIME_DMARead(hDIME,DataFPGA_out,m,0,NULL,NULL,5000);
stream = fopen ("ifftout3.txt","w");
for(i=0; i<1024; i++)
{
      fprintf(stream,"%d\n",DataFPGA_out[i]);
}
fclose (stream);
viDIME_WriteRegister(hDIME,2,1,5000);
// --------  k=4 -------------//
stream = fopen( "mult4.txt", "r" );
for(Cntr2=0 ; Cntr2<m ; Cntr2++)
{
  fscanf( stream, "%d", &Data_in[Cntr2] );
      fscanf( stream, "\n");
}
fclose( stream );
viDIME_DMAWrite(hDIME,Data_in,m,0,NULL,NULL,5000);
do{
    fftdone = viDIME_ReadRegister(hDIME,3,5000);
  } while(fftdone != 5 );
viDIME_DMARead(hDIME,DataFPGA_out,m,0,NULL,NULL,5000);
stream = fopen ("ifftout4.txt","w");
for(i=0; i<1024; i++)
{
      fprintf(stream,"%d\n",DataFPGA_out[i]);
}
fclose (stream);
viDIME_WriteRegister(hDIME,2,1,5000);
// ----------------------------------
printf(" --- ifft completed ---- ");
CloseDIMEBoard(hDIME);
```

```
    getch();
      exit(0);
}
```

D.5 C Code for Peak Search

```
#include <stdio.h>
// Include Header, note that this also includes "dimesdl.h"
#include "vidime.h"
#include <conio.h>
#include <math.h>
#define  totcnt 5120
#define  part1 1024

FILE *stream;

void initialize(DIME_HANDLE hDIME){

    DWORD Cntr;
    DWORD NumModules;

      DIME_SetOscillatorFrequency(hDIME,1,50.000,NULL); // Set SYSCLK
to 50Mhz
      DIME_SetOscillatorFrequency(hDIME,2,40.000,NULL); // Set DSPCLK
to 40Mhz
      DIME_SetOscillatorFrequency(hDIME,3,50.000,NULL); // Set PIXCLK
to 50Mhz

      if( DIME_SmartScan(hDIME) != ssOK )
      {
            printf("Error in Smart Scan!\n");
            exit(2);
      }

      NumModules = DIME_GetNumberOfModules(hDIME);
      printf("Found %d Modules\n",NumModules);
      printf("The Modules are:\n");

      for( Cntr=0 ; Cntr<NumModules ; Cntr++)
            printf("Module %d is a
%s\n",Cntr,DIME_GetModuleDescription(hDIME,Cntr));
```

```
      if( DIME_BootDevice(hDIME,"pksrch_top.bit",NumModules-1,0,NULL)
!= cfgOK_STATUS)
      {
            printf("Error configuring on board Virtex!\n");
            exit(3);
      }
}

int main(int argc, char **argv)
{

      DIME_HANDLE   hDIME;
      DWORD         Data_in[totcnt];
      DWORD         Data_in1[part1];
      DWORD         RDY,RESULT_AVAILABLE;
      int           error, i,ii,k;
      DWORD         Cntr2,KOUT,TAO,PK;
      FILE *        stream;
      error = 0;


      stream = fopen( "ifftouts7.txt", "r" );

      for(Cntr2=0 ; Cntr2<totcnt ; Cntr2++){
        fscanf( stream, "%d", &Data_in[Cntr2] );
            fscanf( stream, "\n");       }
      fclose( stream );


      if( (hDIME = OpenDIMEBoard()) == NULL ){
            printf("No DIME Carrier Card Found!\n");
            exit(1);                          }

      initialize(hDIME);
      DIME_VirtexResetEnable(hDIME);
      DIME_PCIReset(hDIME);
      DIME_VirtexResetDisable(hDIME);
      for (k=0 ; k<5 ; k++) {
            for (ii=0 ; ii<1024; ii++){
                  Data_in1[ii]=Data_in[(k*1024)+ii]; }

            do{
                        getch();
                        RDY = viDIME_ReadRegister(hDIME,2,5000);
```

```
                        printf(" K= %d, REG2= %d\n", k, RDY);
            } while(RDY != 1 );

            viDIME_DMAWrite(hDIME,Data_in1,1024,0,NULL,NULL,5000);

      }
      getch();
            do{
                        RESULT_AVAILABLE =
viDIME_ReadRegister(hDIME,3,5000);
            } while(RESULT_AVAILABLE != 1 );

      KOUT=viDIME_ReadRegister(hDIME,4,5000);
      TAO=viDIME_ReadRegister(hDIME,5,5000);
      PK=viDIME_ReadRegister(hDIME,6,5000);

      printf("PK= %d, TAO = %d KOUT= %d \n",PK, TAO,KOUT );
      printf(" I am done from testing");
      CloseDIMEBoard(hDIME);
        getch();
      exit(0);
}
```

D.6 C Code for Tracking

```
#include <stdio.h>
// Include Header, note that this also includes "dimesdl.h"
#include "vidime.h"
#include <conio.h>
#include <math.h>
#define  m_in 5000
#define  m_out 1024

FILE *stream;

void initialize(DIME_HANDLE hDIME)
{
```

```
    DWORD Cntr;
    DWORD NumModules;

      DIME_SetOscillatorFrequency(hDIME,1,50.000,NULL); // Set SYSCLK
to 50Mhz
      DIME_SetOscillatorFrequency(hDIME,2,40.000,NULL); // Set DSPCLK
to 40Mhz
      DIME_SetOscillatorFrequency(hDIME,3,50.000,NULL); // Set PIXCLK
to 50Mhz

      if( DIME_SmartScan(hDIME) != ssOK )
      {
            printf("Error in Smart Scan!\n");
            exit(2);
      }

      NumModules = DIME_GetNumberOfModules(hDIME);
      printf("Found %d Modules\n",NumModules);
      printf("The Modules are:\n");

      for( Cntr=0 ; Cntr<NumModules ; Cntr++)
            printf("Module %d is a
%s\n",Cntr,DIME_GetModuleDescription(hDIME,Cntr));

      if( DIME_BootDevice(hDIME,"track_top6.bit",NumModules-1,0,NULL)
!= cfgOK_STATUS)
      {
            printf("Error configuring on board Virtex!\n");
            exit(3);
      }
}

int main(int argc, char **argv)
{

      DIME_HANDLE   hDIME;
      DWORD         Data_in[5000];
      DWORD         done,ce,cp,cl;
      DWORD         Cntr2;
      DWORD        RAMdata;
      FILE *        stream;

      stream = fopen( "samples5000.txt", "r" );
     for(Cntr2=0 ; Cntr2<5000 ; Cntr2++)
      {
```

```
            //Data_in[Cntr2]=Cntr2;
          fscanf( stream, "%d", &Data_in[Cntr2] );
              fscanf( stream, "\n");
        }

        fclose( stream );

        if( (hDIME = OpenDIMEBoard()) == NULL )
        {
              printf("No DIME Carrier Card Found!\n");
              exit(1);
        }

        initialize(hDIME);
        DIME_VirtexResetEnable(hDIME);
        DIME_PCIReset(hDIME);
        DIME_VirtexResetDisable(hDIME);
        viDIME_DMAWrite(hDIME,Data_in,m_in,0,NULL,NULL,5000);
        do    {
              done = viDIME_ReadRegister(hDIME,2,5000);
        } while(done != 1 );

        ce=viDIME_ReadRegister(hDIME,3,5000);
        cp=viDIME_ReadRegister(hDIME,4,5000);
        cl=viDIME_ReadRegister(hDIME,5,5000);

        printf(" %d, %d, %d \n",ce,cp,cl);
      printf("GPS signal was tracked successfully");
        //viDIME_WriteRegister(hDIME,2,1,5000);
        //viDIME_WriteRegister(hDIME,7,1,5000);
        //getch();
        //RAMdata=viDIME_ReadRegister(hDIME,8,5000);
        //printf(" %d \n",RAMdata);
        CloseDIMEBoard(hDIME);
        getch();
        exit(0);
}
```

Appendix E

FPGA Layout of the Mapped Designs

Appendix E contains the layout or the chip view of the mapped designs. This appendix is divided into two sections. Thr first section shows the layout figures of the mapped design blocks of the modified-code averaging algorithm. The second section presents the layout design figure for the serial correlators based tracking block.

E.1 Modified-Code Averaging Correlator (Acquisition)

The implemented circuit has five subcircuits. First subcircuit is responsible for averaging the data from 5000 points to 1024. In addition to the averager circuit, this part includes the data collection RAM, the NCO, and the carrier wipe-off circuit.

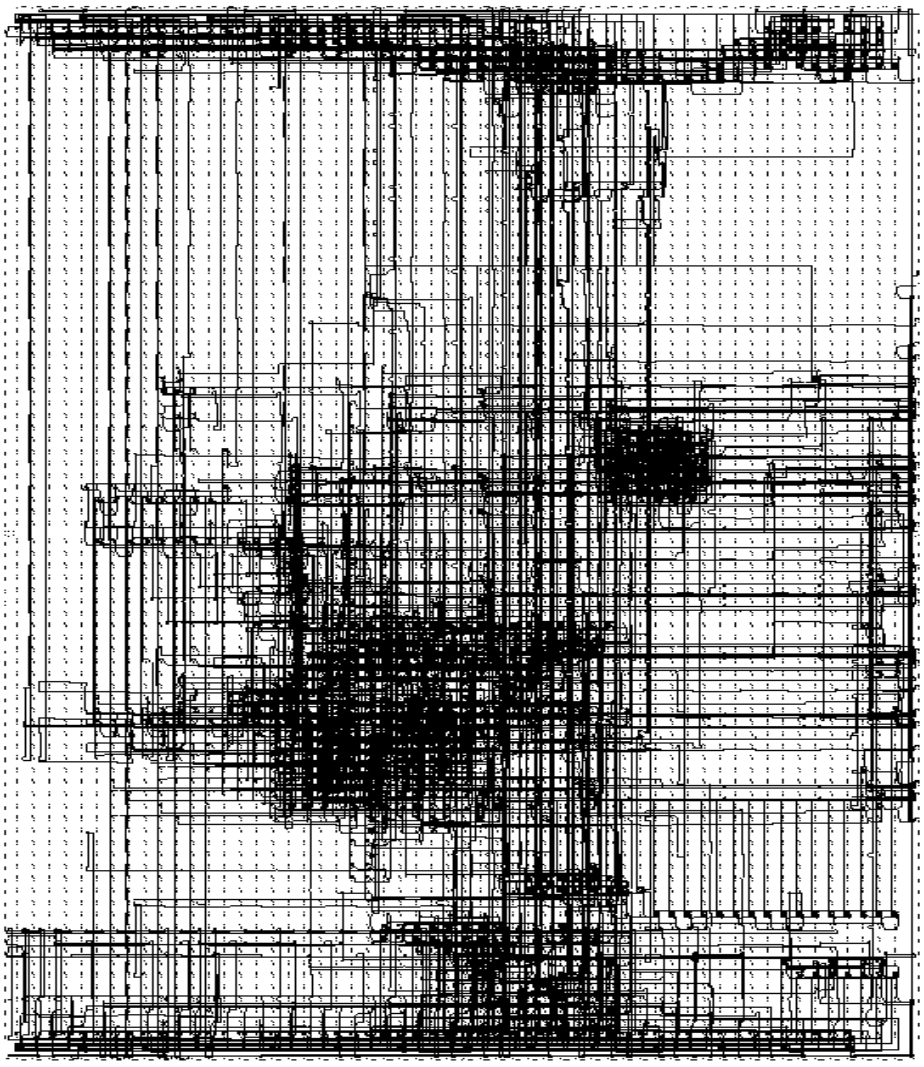

Figure E-1: FPGA Layout of the Averaging and Carrier Wipe-off Circuit

The layout of the mapped implementation of the FFT-block circuit is presented. This part contains the Xilinx FFT core configured as Single-Memory-System, Input RAM, and Output RAM.

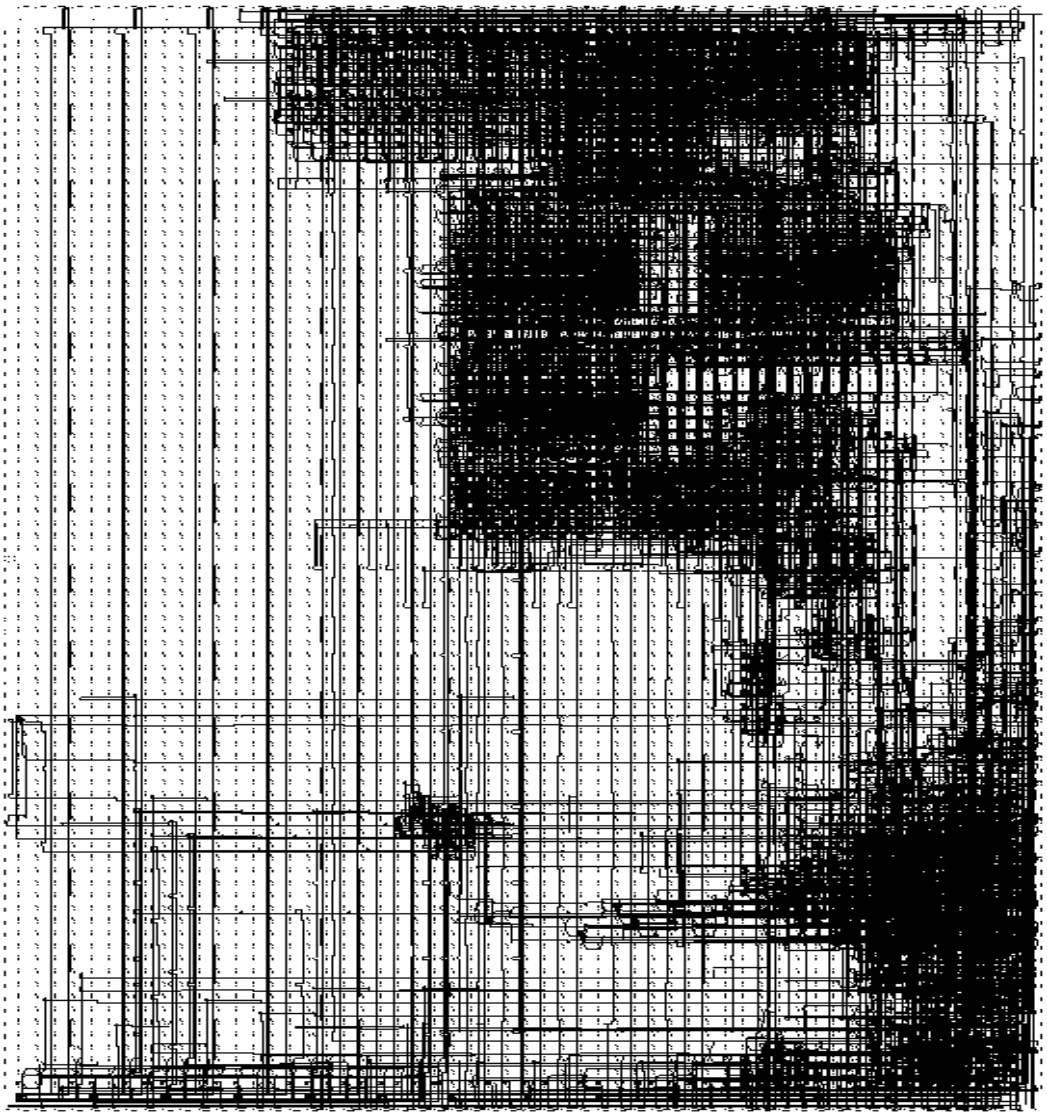

Figure E-2: FPGA Layout of the FFT Block Used in the Acquisition Circuit.

The frequency domain multiplication subcircuit is implemented. This part includes a complex multiplier, a RAM for the FFT values of the local code, a data collection RAM for the results of the previous circuit (FFT-block), and an output RAM.

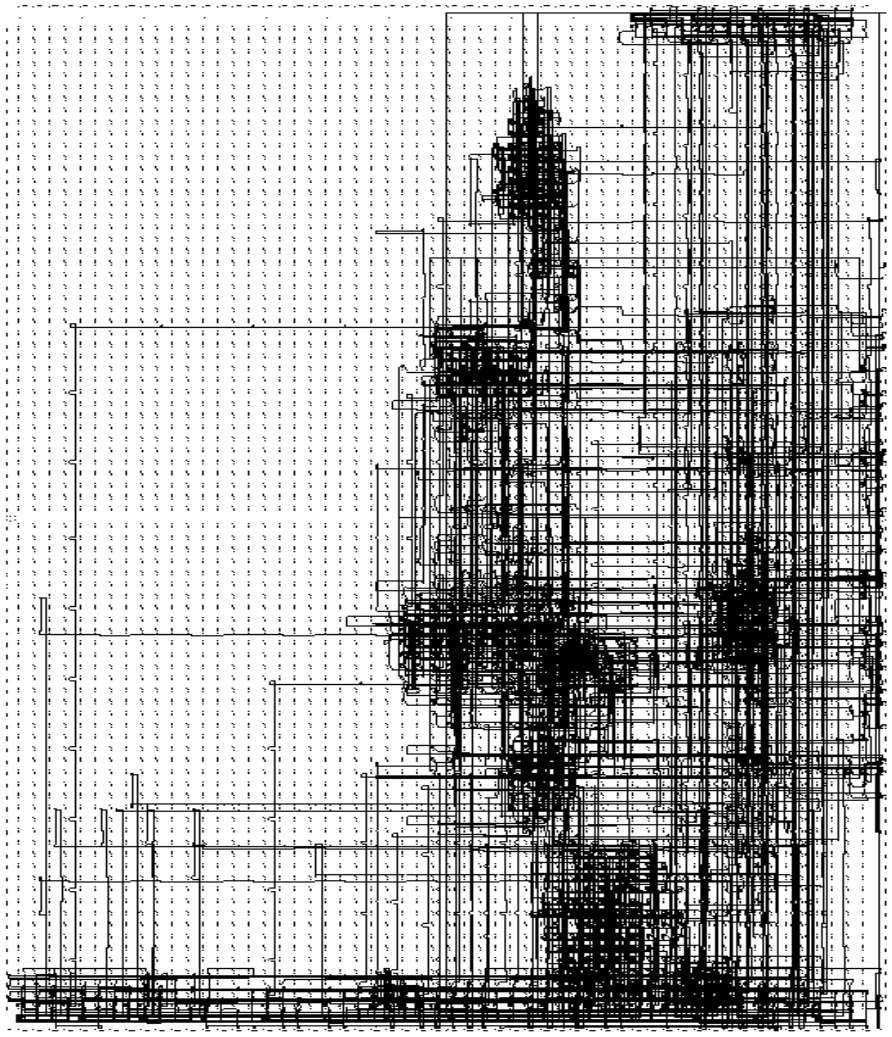

Figure E-3: FPGA Layout of the Frequency Domain Multiplication Circuit.

The mapping layout of the implemented IFFT-block is shown here. This part is identical to the FFT-block circuit in the top level design, but the only differences are in the mapping and routing since they are based on the implementation tools.

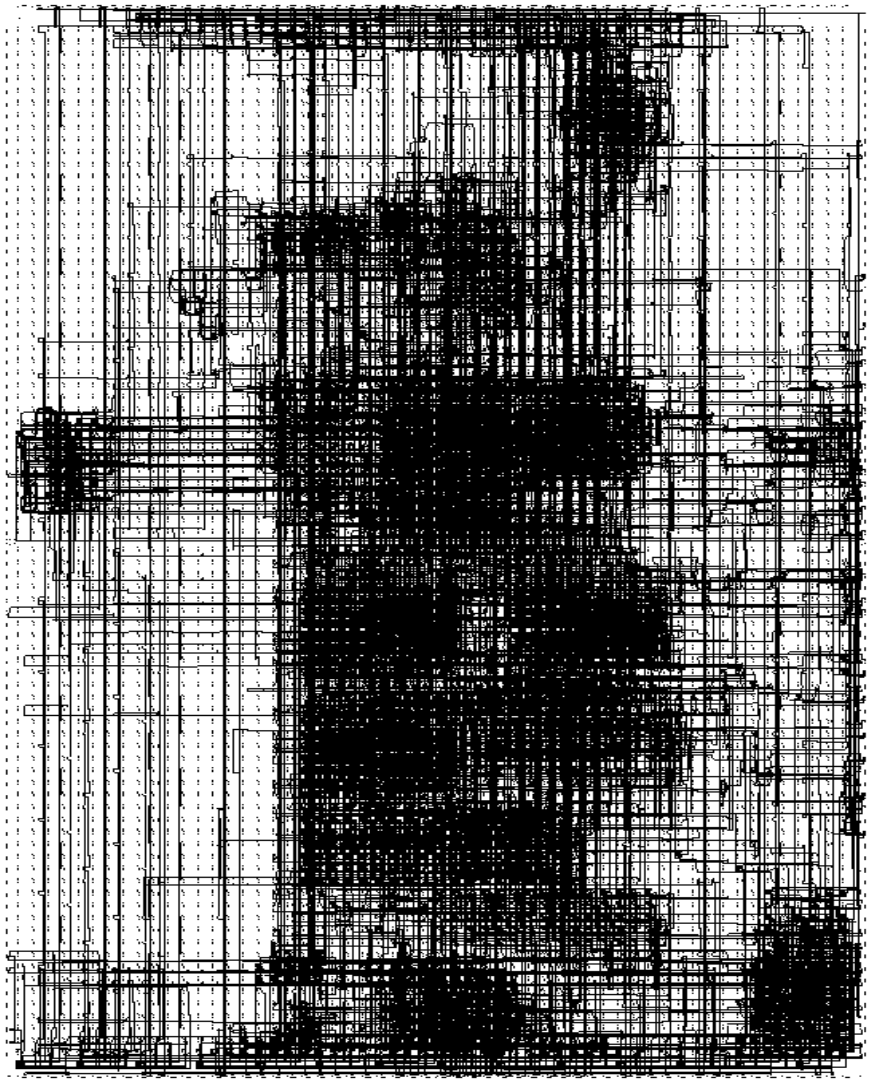

Figure E-4: FPGA Layout of the IFFT Block Used in the Acquisition Circuit

The last subcircuit in the acquisition block is the peak-searcher. The FPGA layout of the peak searcher is presented here. It contains all the counters and registers that were used to count and store the peak's location and its magnitude. The ATAN function which computes the carrier phase was not included in this implmentation.

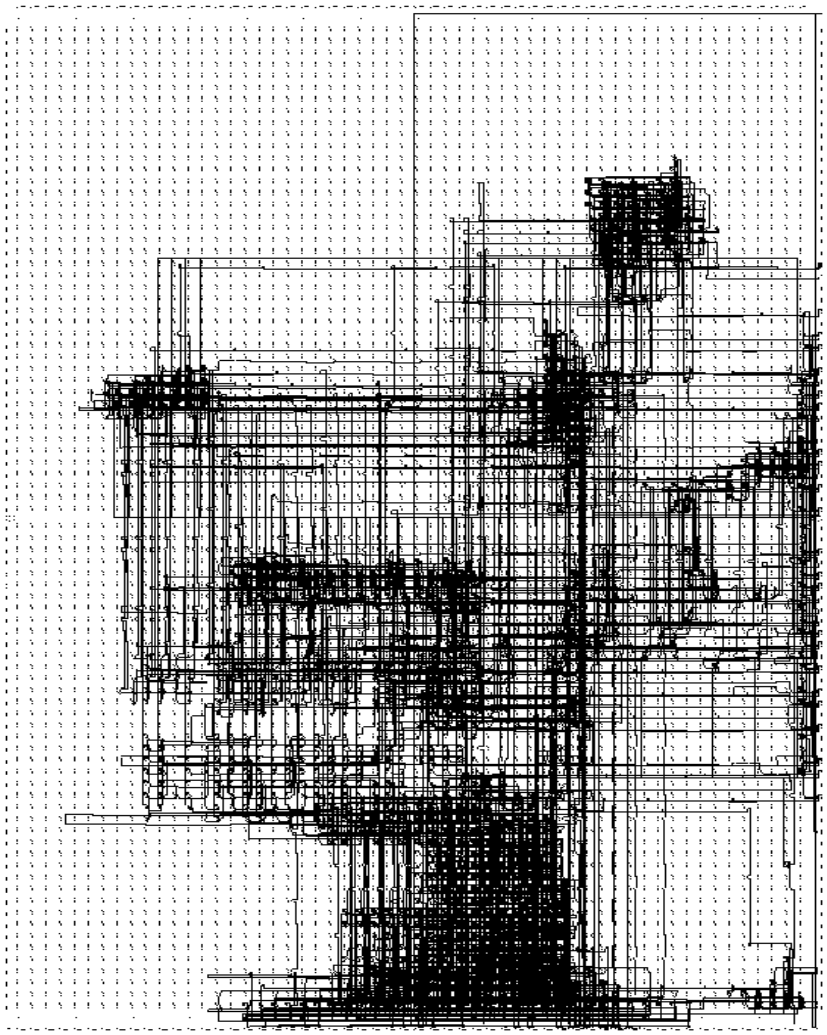

Figure E-5: FPGA Layout of the Peak Searcher.

E.2 Serial Correlators of the GPS Block Processing

This section shows the FPGA layout of the serial correlators which replace the tracking with the block processing concept. This part contains the input RAM, the three serial accumulators, the NCO, and the carrier wipe-off circuit.

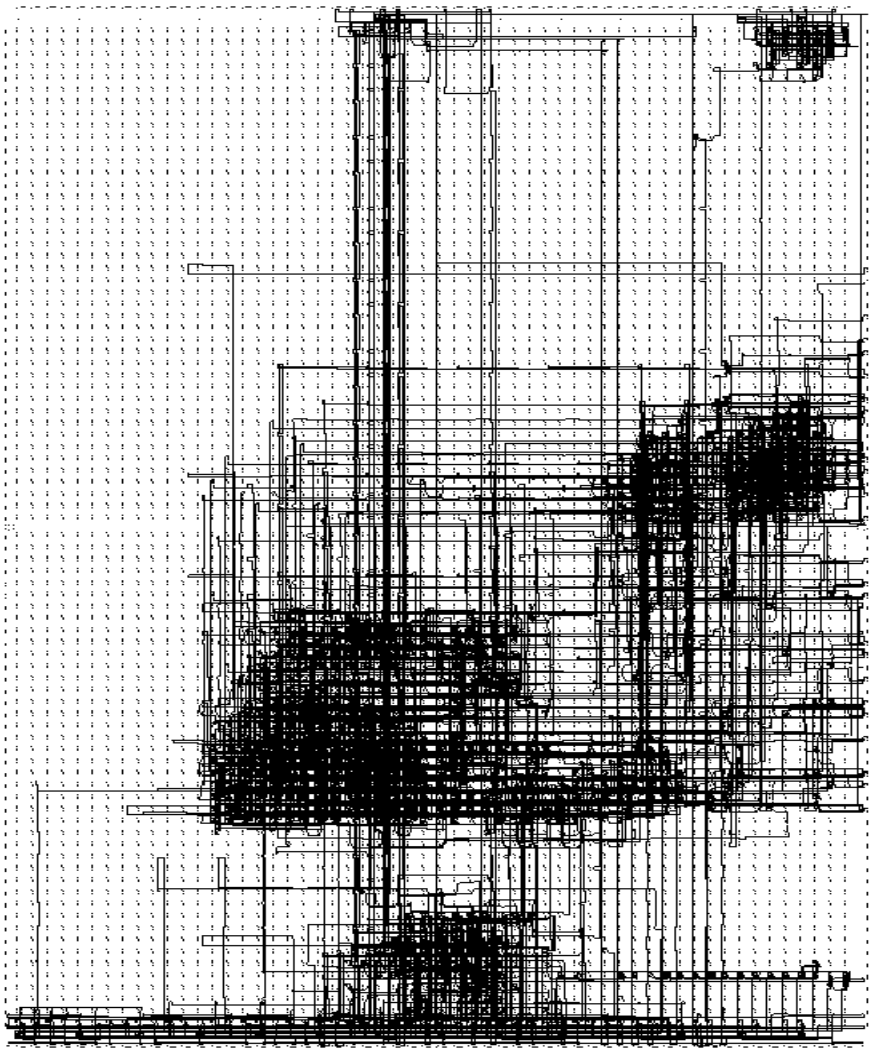

Figure E-6: FPGA Layout of the Estimator (Serial Correlators)

Abstract

Alaqeeli, Abdulqadir Abdulaziz. Ph.D. Nov. 2002

Electrical Engineering and Computer Science

Global Positioning System Signal Acquisition and Tracking Using Field Programmable Gate Arrays (188 pp.)

Director of Dissertation: Professor Janusz Starzyk

GPS receivers spend much of their time on acquisition and tracking. Slow acquisition is due to the large computation time of the correlation function. The correlation function searches for the code phase between the GPS receiver signal arrival time and the GPS satellite's signal transmission time. The computation of the correlation function in frequency domain is speed up N/logN times compared with the time domain implementation.

The long computation time for the correlation function is due to the computation of the FFT functions. One possible solution to speed up the calculations of the correlation function is by replacing the FFTs with simpler transforms. Two transforms were studied in this work. A real transform called the Fermat number transform (FNT) was presented. The FNT-based convolution algorithm was shown. However, the FNT-based convolution has a sequence length restriction that makes it not applicable to the GPS case. A binary transform called the Walsh Hadamard transform (WHT) was also investigated. The WHT-based correlation algorithm was presented and verified. This method shows a significant reduction in the com-

puting time of the correlation function by approximately 20 times compared to the FFT method. The Walsh Hadamard method is not directly applicable to the GPS C/A code, so was not used for GPS signal acquisition.

This dissertation also illustrates a realistic solution to the slow acquisition of the GPS receivers. It uses the FPGA technology along with an averaging method to speed up the calculations of the FFT-based correlation function and to reduce the hardware requirements. The developed method approximated the correlation function by using a modified version of the C/A code. This approximation was accurate enough to use in the acquisition process while maintaining an acceptable level of signal power. This algorithm was used to guide three serial correlators to zoom-in around the correlation peak and provide refined versions of the acquisition estimates. This unique algorithm was implemented and then successfully acquired a GPS signal in less than 1-ms.

Approved:_____