2006

FPGA Frequency Domain Based Gps Coarse Acquisition Processor using FFT

Cyprian D. Sajabi
Wright State University

FPGA FREQUENCY DOMAIN BASED GPS COARSE ACQUISITION PROCESSOR USING FFT

A thesis submitted in partial fulfillment of the requirements for the degree of

Master of Science in Engineering

By

CYPRIAN D. SAJABI

B.A. BIOLOGY, Earlham College, 1995

2006

Wright State University

WRIGHT STATE UNIVERSITY

SCHOOL OF GRADUATE STUDIES

June 8, 2006

I HEREBY RECOMMEND THAT THE THESIS PREPARED UNDER MY SUPERVISION BY Cyprian Sajabi ENTITLED Design and Implementation Of an FPGA Frequency Domain Based GPS Coarse Acquisition Processor Using FFT BE ACCEPTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF Master of Science in Engineering

_____
Chien-In Henry Chen, Ph.D.
Thesis Director

_____
Fred Garber, Ph.D.
Department Chair

Committee on
Final Examination

_____
Chien-In Henry Chen, Ph.D.

_____
Raymond Siferd, Ph.D.

_____
Marty Emmert, Ph.D.

_____
Dr. Joseph F. Thomas, Jr., Ph.D.
Dean, School of Graduate Studies

Abstract

Sajabi, David, Cyprian, M.S.E.E., Department of Electrical Engineering, Wright State University, 2006.
FPGA Frequency Domain Based GPS Coarse Acquisition Processor Using FFT.

The Global Positioning System or GPS is a satellite based technology that has gained widespread use worldwide in civilian and military applications. Direct Sequence Spread spectrum (DSSS) is the method whereby the data transmitted by the satellite and received by user is kept secure, low power and relatively noise-immune. The first step required in the GPS operation is to perform a lock on the incoming signal, both with respect to time synchronization and frequency resolution. Because of the need for reduced time to lock and also reduced hardware, algorithms based in the frequency domain have been developed. These algorithms take advantage of the time to frequency matrix operation known as the fast Fourier transform or FFT. For this thesis, a Direct Sequence Spread Spectrum Coarse Acquisition code processor based on the FFT was implemented in VHDL and targeted to a Xilinx Virtex –II Pro Field Programmable Gate Array (FPGA). The use of the FFT allows simultaneous lock on coarse acquisition (C/A) code and carrier frequency. Because of hardware limitations, a novel technique of sub-sampling is used in this system to obtain data block sizes that match hardware limitations. In addition, design challenges related to scheduling and timing were addressed, allowing a system with 19 pipeline stages to be built. The system, which fits on a Xilinx Virtex-II pro XC2VP70 FPGA, uses 10 ms of data to perform the lock with 5.5 ms of processing time at 100 MHz and theoretically can operate on signals 20 db below the noise floor.

TABLE OF CONTENTS

List of Figures

Figure **Page**

List of Tables

Table **Page**

Acknowledgements

This work was supported in part by the program of Receiver and Processing Concepts Evaluation (RAPCEval), Department of Defense, Air Force Research Lab, USA.

I especially thank my wife, Nacim Sajabi and my whole family both locally and overseas for their patience, support and strong encouragement throughout this thesis. I know it has not been easy, thanks for hanging in there and helping keep up my spirits.

I would like to heartily thank my advisor, Dr. Henry Chen for his invaluable insights unflagging energy, and facilitation of my thesis work over the last year and a half. It has been a great learning and growth journey for me. Thank you ever so much Dr. Chen.

The members of my thesis defence committee have my heartfelt thanks for taking the time out of their busy schedules to read my thesis and advise me in big and small ways throughout the circuit design process.

To all the folks in the VLSI lab, it has been great spending time with you, debating EE and life topics and sharing all our various experiences on our respective thesis paths. It is always easier to go through these experiences together rather than singly.

I would also like to thank numerous individuals from the Wright Patterson Air force Base who gave me pointers on tips on how to make the design simpler and practical. The list includes, but is not limited to David Lin, Dr. James Tsui, James Stephens, Ted Vandewerker, George Gonczy, Ed Huling, and Cliff Bullmaster.

To Vicky Slone, and Jenny and Barry Woods and Marie Donohue, thank you ever so much for helping me fill in all the big and small details that would otherwise go unfilled in all my haste and myopic focus on the all consuming “circuit”.

DEDICATION

I dedicate this thesis to the memory of my beloved parents, Lorna Hope Forbes Sajabi and Samuel Sembuze Sajabi, who always encouraged their children, natural or adopted, to excel in all that they do.

1 INTRODUCTION

1.1 GPS Background

The Global Positioning System, usually called **GPS** is referred to by the United States Military as **NAVSTAR GPS - Navigation Signal Timing and Ranging Global Positioning System.** GPS has a number of applications such as ranging and targeting of ammunition, civilian navigation, and tracking of goods, personnel, and vehicles, just to name a few [1].

The rapid development in Very Large Scale Integration (VLSI) means that GPS units can now be purchased for less than $100.00 or integrated into Cell phones, PDAs and vehicle navigation systems. There is almost no limit the range of applications for GPS and it promises to become as fundamental as the telephone in modern society. GPS is based on 24 orbiting satellites in **Intermediate circular orbit** (**ICO**) orbiting at around 11,000 nautical miles in such a manner that there will always be at least four satellites visible from anywhere on earth [2]. The precise position and velocity of each satellite is known and is used as a reference point for the GPS calculations. Each satellite is generating and continuously transmitting a **Pseudorandom Noise (PN)** sequence of ones and zeroes that can be used to identify it uniquely. This sequence is combined with a very low frequency signal that is used in further data processing for the GPS process. The GPS receiver unit can generate the same PN sequences as the satellites and uses the similarity or **correlation** between these and the received sequence to identify which satellite or satellites are visible at a given time. [2] This step of identifying the satellites is part of an overall process known as **acquisition**. Acquisition is only the first step in the GPS process and is required for the next phases of GPS to proceed. This thesis will

focus on initial acquisition, as the other phases of the GPS are beyond the scope of this project.

1.2 Motivation

This thesis is based on a proven model of a GPS acquisition system on MATLAB. The use of powerful mathematical software such as MATLAB to model systems is very important because it allows design and prototyping to take place at a high level of abstraction. In this way proof-of-concept can be demonstrated efficiently and if needed, rapid modifications can be made to a model under study. The acquisition system model for this thesis is based in a frequency-domain approach, which, although using more hardware than a time-domain approach, is considerably faster [3]. The need for rapid algorithms to carry out GPS acquisition is critical, and in most cases is worth the hardware cost. A rapidly moving system such as an airplane or rocket cannot afford large delays when determining its relative position and velocity.

1.3 Spread Spectrum Basics

Now, a few words about Spread Spectrum technology are in order. A spread-spectrum communication system is one that uses much more bandwidth than would ordinarily be needed simply for information transmission. Sometimes the transmitted bandwidth is as much as 10^5 times the information bandwidth. Spread Spectrum technology was first used developed by the US Navy in the 1950 and was conceived by Hedy Lamar, a Hollywood actress, during WWII [4]. DSSS techniques are ubiquitous in wireless devices such as cell phones, Wireless Ethernet standard 802.11, and of course

GPS units. DSSS technology allows multiple users access to the same frequency band at the same time, leading an efficient use of spectrum. DSSS is designed to operate in low signal to noise ratio environments, and this allows transmitters to use low power signals, major factor to consider when designing a GPS system. Another benefit of DSSS is that it is resistant to jamming (intentional or otherwise) by narrowband signals, making it ideal for military applications and critical civilian applications. In a Spread Spectrum system, the bandwidth of the transmitted signal is much greater than minimum bandwidth needed to transmit it. For example a typical GPS satellite data signal bandwidth is about 50 Hz, so 100 Hz is the minimum bandwidth needed to transmit this signal. The spread spectrum bandwidth is over 2 MHz - a 20,000 fold increase in the needed bandwidth to transmit the signal if double sideband transmission is used. This spreading of the spectrum is accomplished by modulating or multiplying the information with a wideband encoding signal. There are three types of techniques that are generally thought of as spread spectrum. They are direct sequence, frequency hopping, and "chirp" modulation. This thesis is focused on the direct sequence method, so the details will only be given on this method. In direct sequence, there is modulation of a carrier by a digital code sequence whose "chip" rate is much higher than the information signal bandwidth.

The basis of spread spectrum technology is expressed by Claude Shannon in the form of channel capacity:

$$C = W \log_2\left(1 + \frac{S}{N}\right) \qquad \textbf{(1.1)}$$

C = capacity in bits per second, W = bandwidth in Hz

N = noise power, S = signal power

This equation shows that the higher the signal to noise ratio and the higher the bandwidth, the more information we can transmit though a channel.

If we change to natural logarithms and rearrange the equation, we get the following expression:

$$\frac{C}{W} = 1.44 \log_e\left(1 + \frac{S}{N}\right), \qquad \textbf{(1.2)}$$

and for small S/N of less than 0.1 (which is the case in a GPS system),

$$\frac{C}{W} \approx 1.44\left(\frac{S}{N}\right) \qquad \textbf{(1.3)}$$

Further rearranging to make W the subject of the equation gives

$$W \approx \left(\frac{NC}{S * 1.44}\right) \qquad \textbf{(1.4)}$$

From this equation we can see that if we have a fixed channel capacity and fixed signal strength, then in the presence of a large amount of noise, we need to increase the transmitting bandwidth to counteract the effects of the noise. This is the price that needs to be paid to ensure secure and error free transmission in a noisy environment. There are various methods of embedding the information onto the spread spectrum signal. One common method is to add the information to the spectrum spreading code before a carrier wave is added. Alternately, one may modulate the information and then apply the spreading code. Each approach has pros and cons which need to be weighed before proceeding. There are numerous reasons for using spread spectrum technology. Below are just a few:

1. Selective addressing capability
2. Code division multiplexing is possible for multiple access (CDMA).

3. Low density spectra for signal hiding
4. Message screening from eavesdroppers
5. High resolution ranging
6. Interference rejection. (High Jamming Margin)

The major figure of merit for a spread spectrum system is the **jamming margin.** It is closely related to another property of the system known as **process gain.** The process gain is simply the ratio of the spread bandwidth to the minimum bandwidth needed to transmit a signal. In a spread spectrum signal process gain is given by:

$$G_P = \left(\frac{BW_{RF}}{R_{\inf o}} \right), \qquad \textbf{(1.5)}$$

where RF bandwidth (BW_{RF}) is the bandwidth of the transmitted spread spectrum signal and the information rate (R_{info}) is the data rate in the information base-band channel.

Processing Gain is inherent in the transmission of the system and is an upper bound on the quality of the system. No real system achieves its potential process gain. A more realistic figure of merit is the jamming margin. To understand jamming margin, we need to discuss how the spectrum is "de-spread" at the receiver end. Basically, what happens at the receiver is as follows: First carrier needs to be wiped off, base banding the incoming signal. Then and the incoming signal is correlated with a local reference code identical to the code that was used to spread the spectrum. If there is a match between the local code and the signal, the spectrum collapses back to its original bandwidth before spreading. Any uncorrelated signal, such as a jamming signal or noise, is spread by the local code to the local reference bandwidth. A filter is then used to reject all but the desired narrowband signal of interest. One thing to note is that the spreading codes tend to be periodic. They are generated by a device known as a Linear Feedback Shift

Register or LFSR. This periodicity is important in the synchronization of the incoming signal with the locally generated version of the code. Figures 1.1 and 1.2 give an overview of the spectrum changes occurring in a spread spectrum system.[a]

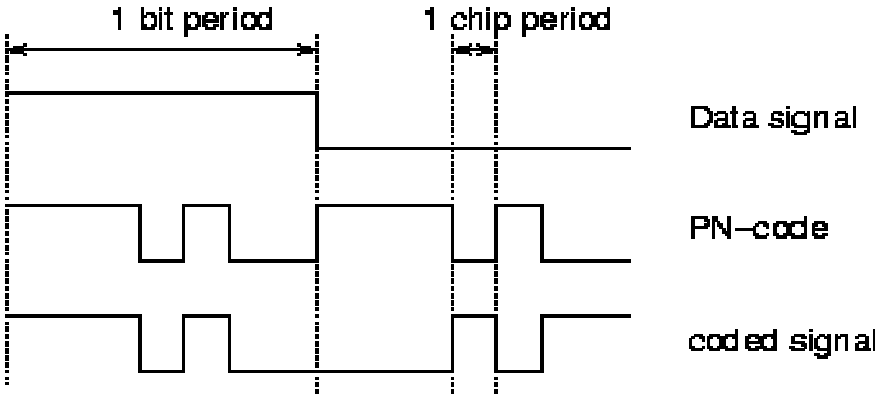

Figure 1.1 Direct Sequence code spreading of data

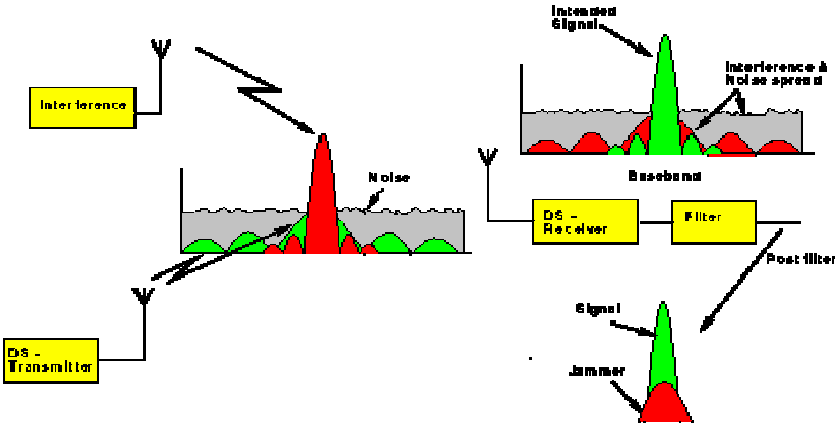

Figure 1.2 DS- Concept, before and after despreading[a]

This process results in enhancement of the desired signal and attenuation of the spurious signals. The difference in output and input signal to noise ratios is defined as the process gain. Ideally this should also be equal to the jamming margin. In reality the jamming margin is less than the process gain. Jamming margin takes into account internal

[a] Source : http://cas.et.tudelft.nl/~glas/ssc/techn/techniques.html

losses and the minimum SNR needed to decode the received information. Jamming margin in dB is expressed as:

$$M_j = G_P - \left[L_{sys} + \left(\frac{S}{N} \right)_{out} \right], \qquad \textbf{(1.6)}$$

Where L_{sys} = system implementation losses, and $(S/N)_{out}$ = SNR at the information output.

A sample calculation with a system that has a 30-dB process gain, minimum$(S/N)_{out}$ of 10 dB and L_{sys} of 2dB would have an 18-dB jamming margin (Mj). Therefore, if the jamming signal is more than 18-dB above the signal of interest, the system would not operate properly [5].

1.4 The Fast Fourier Transform

The fast Fourier transform (FFT) is based on the discrete Fourier transform (DFT), an algorithm that performs a frequency analysis on a discrete sampled signal. The DFT transforms a sequence of N complex numbers $\{xn_1, xn_2 \ldots xn_{N-1}\}$ in the time domain to a sequence of N complex numbers $\{XK_1, XK_2 \ldots XK_{N-1}\}$ in the frequency domain. The formula is as follows:

$$X_k = \sum_{n=0}^{N-1} x_n e^{-\frac{2\pi i}{N} kn} \qquad k = 0, \ldots, N-1 \qquad (1.7)$$

The DFT gives a representation of the signal in the frequency domain and is useful for understanding where in the frequency domain most of the energy or information in a signal is concentrated.

The inverse DFT or IDFT translates data from the frequency domain to the time domain and the formula is as follows

$$x_n = \frac{1}{N} \sum_{k=0}^{N-1} X_k e^{\frac{2\pi i}{N} kn} \qquad n = 0, \ldots, N-1. \tag{1.8}$$

The DFT and IDFT are used extensively in digital signal processing algorithms such as linear filtering, correlation analysis and spectrum analysis. [6]

1.4.1 Direct Computation of the DFT

"For a complex valued sequence *x(n)* of N points, the DFT may be expressed as

$$X_R(k) = \sum_{n=0}^{N-1} \left[x_R(n) \cos \frac{2\pi kn}{N} + x_I(n) \sin \frac{2\pi kn}{N} \right] \tag{1.9}$$

$$X_I(k) = -\sum_{n=0}^{N-1} \left[x_R(n) \sin \frac{2\pi kn}{N} - x_I(n) \cos \frac{2\pi kn}{N} \right] \tag{1.10}$$

The direct computation of **(1.9)** and **(1.10)** requires:

1. $2N^2$ evaluations of trigonometric functions.
2. $4N^2$ real multiplications.
3. 4N (N-1) real additions.
4. A number of indexing and addressing operations.

These operations are typical of DFT computational algorithms. The operations in items 2 and 3 result in the DFT values $X_R(k)$ and $X_I(k)$. The indexing and addressing operations are necessary to fetch the data x(n), $0 \le n \le N-1$, and the phase factors (W_N) and to store the results."[7]

The DFT and IDFT are very computation-intensive, and for large blocks of data are not very practical in real-time systems. Efficient computation of the DFT is carried using what are termed Fast Fourier Transforms or FFTs. The two kinds of FFTs used in this thesis are the divide-and-conquer approach and the decimation-in-time approach.

1.4.2 Divide-and-Conquer Approach

If we adopt a divide-and-conquer approach, the DFT can be carried out in a fairly efficient manner. By prime factoring, the N point DFT is broken into successively smaller DFTs, whose results are then used to compute the final DFT.

The algorithm below is one of many that can be used to accelerate the calculation of the DFT:

Algorithm to Directly Calculate the DFT of any sequence using the Divide and Conquer Approach.

1. Factor N into the product of two prime integers; N = LM;
2. We can zero pad if needed to ensure factorization
3. Store the signal column-wise in L rows and M columns.
4. Compute the M-point DFT of each row.
5. Multiply the resulting array by the phase factors
6. Compute the L-point DFT of each column
7. Read the resulting array row-wise

Here is an example to illustrate the algorithm. Consider the computation of an N = 10 DFT. The following summary information is given.

- a = [a0 a1 a2 a3 a4 a5 a6 a7 a8 a9]; a is assumed to be complex.
- b = fft(a); b = [A0 A1 A2 A3 A4 A5 A6 A7 A8 A9]; b is complex.
- The number 10 can be factored into 2 * 5. We select L = 5 and M = 2.

Step 1: Store the 10-point sequence column-wise in 2 columns. (M = 2)

row1	a(0,0) = a0	a(0,1) = a5
row2	a(1,0) = a1	a(1,1) = a6
row3	a(2,0) = a2	a(2,1) = a7
row 4	a(3,0) = a3	a(3,1) = a8
row5	a(4,0)= a4	a(4,1) = a9

Step 2: Compute the 2-point DFT on each row

f(0,0) = a0 + a5= f0	f(0,1) = a0 - a5 = f5
f(1,0) = a1 + a6 = f1	f(1,1) = a1 - a6 = f6
f(2,0) = a2 + a7 = f2	f(2,1) = a2 - a7 = f7
f(3,0) = a3 + a8 = f3	f(3,1) = a3 - a8 = f8
f(4,0)= a4 + a9 = f4	f(4,1) = a4 - a9 = f9

Step 3: Multiply each of the terms f(l,q) by the phase factors $W_{10}^{l.q}$

f(0,0)* $W_{10}^{0.0}$ = f0*1 =g(0,0) = g0	f(0,1)* $W_{10}^{0.1}$ = f5*1 =g(0,1) = g5
f(1,0)* $W_{10}^{1.0}$ = f1*1 =g(1,0) = g1	f(1,1)* $W_{10}^{1.1}$ = f6* (0.8090 - 0.5878i) =g(0,2) = g6
f(2,0)* $W_{10}^{2.0}$ =f2*1 =g(2,0) = g2	f(2,1)* $W_{10}^{2.1}$ = f7* (0.3090 - 0.9511i) =g(0,3) = g7
f(3,0)* $W_{10}^{3.0}$ = f3*1 =g(3,0) = g3	f(3,1)* $W_{10}^{3.1}$ = f8* (-0.3090 - 0.9511i) =g(0,4) = g8
f(4,0)* $W_{10}^{4.0}$ = f4*1 =g(4,0) = g4	f(4,1)* $W_{10}^{4.1}$ = f9* (-0.8090 - 0.5878i) =g(0,5) = g9

Step 4: Compute the 5-point DFT of each column. A(k) = DFT(g(n))

Step 5: Read the resulting array row-wise

A(0,0) = A0	A(0,1) = A1
A(1,0) = A2	A(0,2) = A3
A(2,0) = A4	A(0,3) = A5
A(3,0) = A6	A(0,4) = A7
A(4,0) = A8	A(0,5) = A9

1.4.3 DECIMATION-IN-TIME APPROACH TO COMPUTING THE DFT

One of the most efficient forms of the FFT is the decimation in time FFT. It is derived as follows:

If we perform the substitution: $e^{-j2\pi/N} = W_N$., where W_N is known as the phase factor, then the DFT then becomes:

$$X(k) = \sum_{n=0}^{N-1} x(n) W_N^{kn} \quad . \tag{1.11}$$

Similarly, the IDFT becomes:

$$x(n) = \frac{1}{N} \sum_{n=0}^{N-1} X(k) W_N^{-kn} \quad . \tag{1.12}$$

Using the properties of complex exponentials, we discover two interesting characteristics in the phase factors that allow us to simplify the DFT and IDFT computations. These two characteristics are:

Symmetry Property: $-W_N^k = W_N^{k+N/2}$ (1.13)

Periodicity Property: $W_N^k = W_N^{k+N}$ (1.14)

All the computationally efficient FFT algorithms exploit these two basic characteristics of the phase factor.

1.5 Radix-2 FFT Algorithms

By far the most widely used FFT algorithm is the radix-2 FFT algorithm. If N is a power of 2, then the dataset lends itself to a radix-2 FFT algorithm. The mathematical details of

implementation of the radix-2 FFT algorithm are beyond the scope of this thesis, but a brief description here will suffice.

1 The data set is successively divided (or decimated by 2) into odd and even sequences until resulting sequences are reduced to one-point sequences.

2 These one point sequences are then operated on and successively combined to produce the final results.

The basic operation at each stage, as illustrated in figure 1.3 is know as a butterfly. In this operation, take two complex numbers (a,b) multiply b by the appropriate phase factor and then add and subtract the product from a to form two new complex numbers (A,B). The combination of the results of each butterfly is shown in figure 1.4. [8]

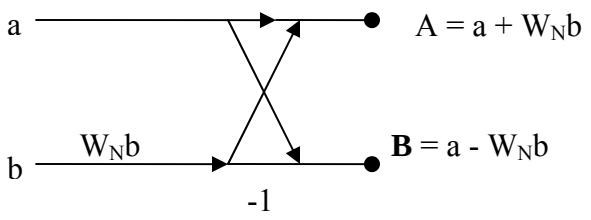

Figure 1.3 Basic Butterfly Computation in the FFT Algorithm

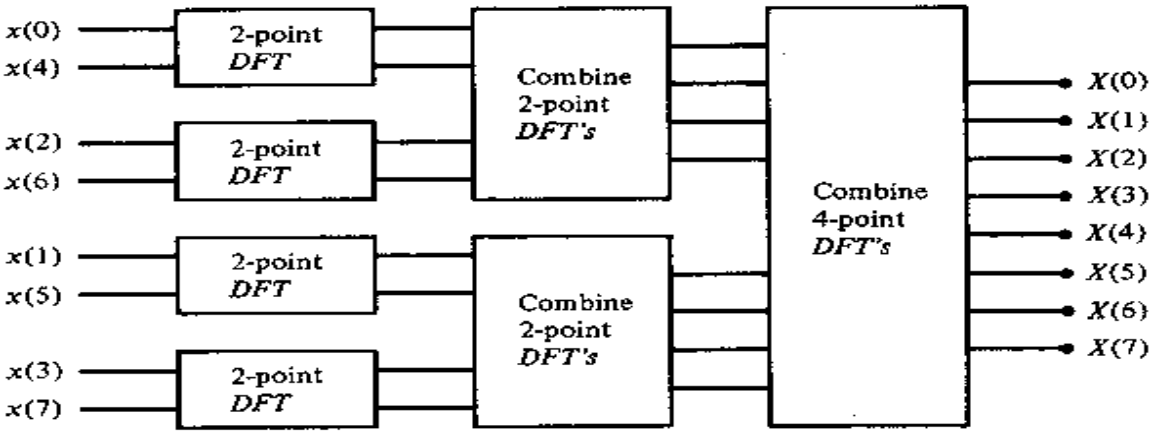

Figure 1.4 Three stages in the computation of an N = 8-point DFT (Proakis P. 459)

An analysis of the reduction in computations in a radix-2 FFT versus a straight DFT is shown below in table 1.1

Table 1.1 Complexity of Direct Computation of the DFT vs FFT Algorithm

Number of Points N	Complex Multiplications in Direct Computation N^2	Complex Multiplications in FFT algorithm $(N/2) \log_2 N$	Speed Improvement Factor
4	16	4	4.0
8	64	12	5.3
16	256	32	8.0
32	1,024	80	12.8
64	4,096	192	21.3
128	16,384	448	36.6
256	65,536	1,024	64.0
512	262,144	2,304	113.8
1,024	1,048,576	5,120	204.8

There are a number of other kinds of FFTs such as Radix-4 and split Radix, which are also commonly used.

1.5 Hardware Implications of FFT

There are numerous multiplications at each butterfly stage of the FFT, which potentially leads to bit growth. The following excerpt from the Xilinx fast Fourier

transform Datasheet gives a summary of the considerations surrounding this bit growth: "For a radix-4 Decimation in Time FFT (4,096 is a radix-4 number), the values computed in a butterfly stage (except the second) can experience a growth to $4\sqrt{2} \approx 5.657$. For radix-2 (2,048 is a radix-2 number), the growth can be up to $1+\sqrt{2} \approx 2.414$.

Various approaches are used to deal with this dynamic range expansion. There are three main approaches:

- Performing the calculations with no scaling and carrying all significant integer bits to the end of the computation.
- Scaling at each stage using a fixed-scaling schedule
- Scaling automatically using block-floating point

All significant integer bits are retained when doing full-precision unscaled arithmetic. The width of the data path increases to accommodate the bit growth through the butterfly. The growth of the fractional bits created from the multiplication are truncated (or rounded) after the multiplication. The width of the output will be the (input width + number of stages + 1). This will accommodate the worst case scenario for bit growth. For example, a 1024-pt transform with an input of 16 bits consisting of 1 integer bit and 15 fractional bits, will have an output of 27 bits with 12 integer bits and 15 fractional bits.

When using scaling, a scaling schedule is used to scale by a factor of 1, 2, 4, or 8 in each stage. If scaling is insufficient, a butterfly output may grow beyond the dynamic range and cause an overflow. As a result of the scaling applied in the FFT implementation, the transform computed is a scaled transform.... If a radix-4 algorithm uses a scaling schedule of all 2's, the factor of *1/s* will be equal to the factor of *1/N* in the

inverse FFT equation. For radix-2, a scaling schedule of all 1's provides the factor of *1/N*. Otherwise, additional scaling is necessary."

With block floating point, each data point in a frame is scaled by the same amount, and the scaling is tracked by a block exponent. Scaling is performed only when necessary (to prevent data overflow), which is detected by the core. As with unscaled arithmetic, for scaled and block floating point arithmetic, the core does not have a specific location for the binary point. The location of the binary point in the output data is inherited from the input data and then shifted by the scaling applied. [9]

2 STATEMENT OF NEED

2.1 The Field Programmable Gate Array (FPGA)

The Field Programmable Gate Array or FPGA is an attractive hardware platform for implementation of signal processing algorithms. It provides a compromise between flexibility of a general purpose processor (GPP) and the speed of a dedicated Application Specific Integrated Circuit (ASIC). The development of an algorithm on an FPGA is not as rapid as a software approach for a GPP, but is considerably more rapid than a full custom design. In addition, the FPGA is easily reprogrammable and carries out many more effective operations per clock cycle than a GPP [10].

2.2 GPS Locking in on GPS Coarse Acquisition Code

The "lock" on a GPS signal is accomplished initially via the coarse acquisition or "C/A" code. Subsequent to this, the system will proceed to lock onto another code known as the P code using information gathered from the data embedded in the C/A code.

Under normal conditions GPS signal strength is about 130 dBm in a bandwidth of about 2 MHz where the thermal noise at R^0 is about -111 dBm [11]. In this case, 1 ms of data is adequate to acquire the signal. However, in many cases, such as indoors or in cloudy or forested areas, the GPS signal is 20 dB below the noise floor (-131 dBm), so instead of the usual 1 ms of data usually required, a receiver may have to process 10 ms of data to acquire the signal [12]. This provides special challenges for extracting the phase and carrier information from the received signal. The computational and time requirements are quite large, and if the acquisition is attempted in the time domain,

something on the order of $4*N^2$ multiplications and additions are required [13], where N represents the number of data samples. These computational demands discourage the performing acquisition in the time domain, especially in a weak signal environment where repeated locks may be required. A number of approaches, based in the frequency domain, have been developed to deal with this challenge. One way to speed up the acquisition process is to move the intermediate operations from the time domain to the frequency domain. This results in reduced time and power requirements for the complete operation. The FFT-and-multiply implementation of circular correlation is a very popular and standard method of transferring the correlation operation to the frequency domain. This is followed by an IFFT, which transfers the finished results back to the time domain [13, 14]. If a fast Fourier Transform (FFT) is used, the number of calculations is reduced to $(2*\log_{2*}2N+l) * 2N$ additions and half of the number of multiplications as before. Compared with the time domain approach the overall efficiency of the system improves exponentially as the number of data points increases.

There have been numerous software approaches to GPS acquisition because of their relative flexibility, but if real-time processing is wanted for long blocks of data, a hardware approach remains the most attractive because of speed of operation [15, 16, 17]. The following sections will cover the GPS C/A/ lock Algorithm in more detail.

2.2.1 Spreading Data with C/A Code

Each of the 24 GPS satellite generates a number of unique PN sequences know as coarse acquisition code (C/A code), and precision code (P-code). This analysis will focus on the C/A code acquisition because this is the fundamental step in the GPS process.

Within each satellite, the C/A code is generated by a specialized class of PN code generators known as Gold Code generators, which are beyond the scope of this thesis. The C/A code is generated at a rate of 1.023 MHz and sampled at 5 MHz. The 50 Hz GPS navigation data that is needed for performing lock on the P code is also sampled at 5 MHz and multiplied by the C/A code samples. The product of this multiplication is now termed the C/A GPS signal. We now have a 2.046 MHz spread spectrum signal that is ready for transmission. The following step is the mixing of the spread spectrum signal up to 1575.42 MHz using Binary Phase shift keying or BPSK. In BPSK, the phase ϕ of the carrier is $\pm\pi$, depending on whether the signal is a '1' or a '0'. The satellite then transmits the GPS C/A signal at a center frequency of 1575.42 MHz (L1). The transmitted signal can be written as:

$$S_{L1} \equiv A_c C(t) D(t) \cos(2\pi f_1 t \phi) \qquad \textbf{(2.1)}$$

Where S_{L1} is the signal at L1 frequency, A_c is the amplitude of the C/A code, C (t) = ± 1 represents the C/A code, D (t) = ± 1 is the navigation data code, f1 is the L1 frequency in Hz, and ϕ is the initial phase of the carrier. [18]

2.2.2 Despreading at the Receiver

At the receiver side, in order to obtain the navigation data, S_{L1} has to be down converted back to base band and C(t) has to then be stripped from the signal yield D(t). Base-banding involves multiplying S_{L1} by $\cos(2\pi f_1 t \phi)$. This is a very simplified approach to basebanding, and in reality a multi-stage approach is used. C(t) then has to be stripped from the resulting signal. This is achieved by a point multiplication with a

locally generated copy of C (t). The process of stripping the C (t) from a spread spectrum signal is often called "despreading" because the resulting spectrum is collapsed back to its original bandwidth-in our case 100 MHz. If the despreading is successful, the navigation data, D (t) is available and can be used to perform lock on the P code. If the despreading is not successful, the process of point multiplying the two signals needs to be repeated for different delays of the locally generated code. This process basically amounts to an autocorrelation function, which generally has its maximum when the two sequences are aligned. The mathematical representation of the correlation function, a(n) between two discrete time signals, x(n) and h(n) and is given as:

$$a(n) = \sum_{m=0}^{N} x(n)h(n+m), \qquad \textbf{(2.2)}$$

where N is the number of sample points in either discrete time signal.

The figures below show the autocorrelation properties of 5,000 samples of C/A code and 1,023 samples of C/A code. It is apparent that the incoming signal and the locally generated code need to be very closely aligned to give a strong correlation peak needed for lock. If the two are not properly aligned, the spectrum remains spread and acquisition cannot follow. Another point to note is that the magnitude of the correlation peak is equal to the number of samples of data. Therefore if there is a very weak signal, a longer data record is needed for acquisition.

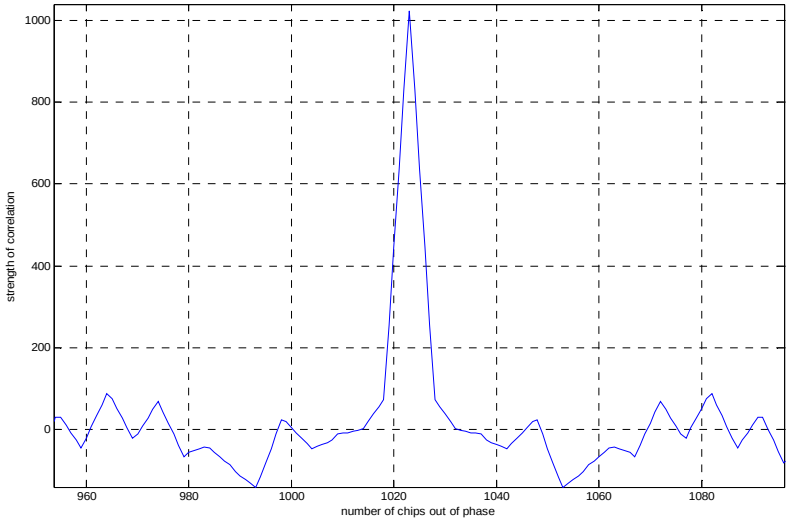

Figure 2.1 Autocorrelation of 1,023 C/A code chips

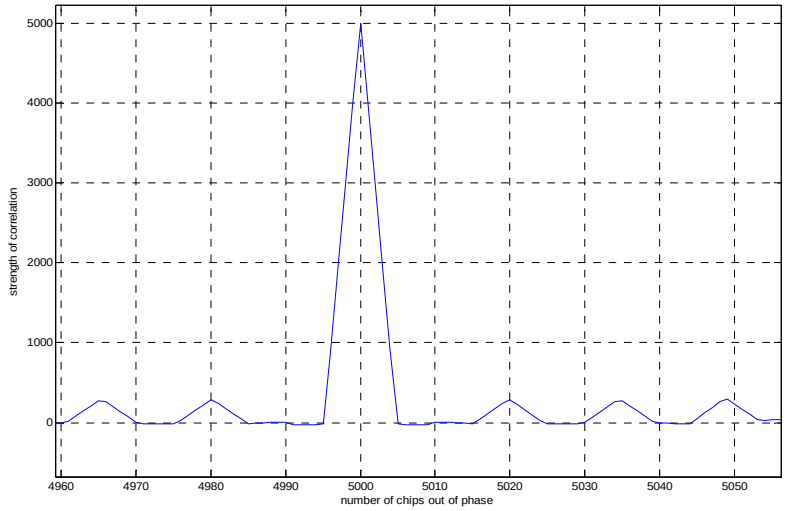

Figure 2.2 AutoCorrelation 5,000 C/A code chips.

The acquisition process in complicated by the fact that the receiver is moving relative to the satellite. This relative movement introduces a Doppler shift in frequency into the received signal. This Doppler shift must be accounted for otherwise acquisition is bound to fail. For a low speed vehicle, the Doppler shift is in the range of ± 5 KHz. For a high speed aircraft, the shift is in the range of ± 10 KHz [19]. Because of the unknown Doppler shift, the receiver must attempt a number of down conversion multiplications and select the one that gives an output that is closest to base band.

A successful acquisition therefore gives two important parameters about the received C/A code, namely the code phase and the Doppler shift. It is clear then that C/A code acquisition is a two-dimensional search among a number of C/A code phases and Doppler shifts. These two parameters are continually used to keep a lock on the incoming navigation data, which are needed to lock onto the P code.

One can perform C/A "acquisition" on two consecutive 10 ms of data. Between two consecutive sets of 10 ms of data there is at most one navigation data bit phase transition because the navigation data frequency is 50 Hz. Therefore, one set of these data will have no data bit transition and can result in successful acquisition. [20]

2.2.3 Frequency Steps In Acquisition

It is necessary to determine the down conversion frequency steps needed in acquisition. Let us assume a $\pm 10 KHz$ Doppler range. The frequency step is closely related to the length of data used. One chip difference between locally generated C/A code and the input signal will result in almost zero correlation. If the signals are out of step by half a chip, then there is partial correlation between them. Therefore the

maximum allowable frequency separation between the two signals is ± 0.5 cycles. If the data length is 1 ms, then a1 KHz signal will toggle once in 1 ms. The maximum frequency offset allowable from baseband is $\pm$ 0.5 of a cycle or ± 0.5 Hz/ ms. For a 1 ms dataset, this maximum frequency step is 1KHz in order to allow for partial correlation. This arrangement will center the input signal in between baseband and 1 KHz in the worst case. If the 10 ms of data are taken, then a 100 Hz frequency step is needed. This fits with conventional FFT results in which the frequency resolution is inversely proportional to the number of points in the FFT [21]. The number of FFT bins increases by a factor of 10 and the memory needed to hold the data record increases by the factor of 10. The above discussion brings home the point of using as short a data record as possible for acquisition. The hardware requirements and operations increase by a factor of about 100 for a 10 fold increase in the data record.

2.2.4 C/A Code Multiplication and The FFT

Acquisition has as its objective the dispreading of the input signal and obtaining the carrier frequency. Assuming the reference C/A code has the same phase as the C/A code in the transmitted GPS signal, then the input signal will be despread and become a narrowband signal continuous wave (cw) signal as shown in the figure 2.3. The top plot is an RF signal carrying the C/A code. The second plot is the locally generated reference C/A code, and the bottom plot is the result of multiplying the two top signals, assuming they are phase aligned with respect to the C/A code. The FFT of the bottom signal would reveal the Doppler shift and then the carrier could be stripped off the cw signal to reveal

the navigation data. The order of operations is matter of preference and in some implementations the carrier is stripped off before the spectrum is despread.

If 1 ms of data sampled at 5 MHz were used, then the record would consist of 5,000 samples. The DFT of this signal would have a frequency resolution of $\frac{5\times10^6}{5\times10^3}$ or 1 KHz. A 5,000 point DFT generates 5,000 frequency components, but the first 2,500 contain the majority of the energy relative to last 2,500 points. If the frequency range of interest is $\pm 10KHz$,

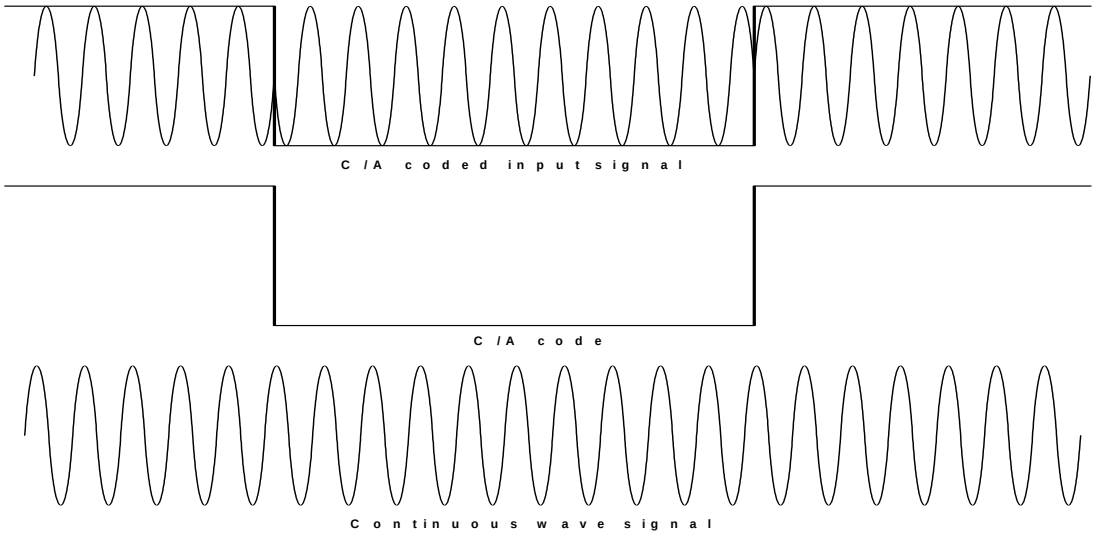

Figure 2.3 C/A coded input signal multiplied by C/A code
Source : Tsui, Fundamentals of GPS Receivers P 137

then in some implementations, only 21 frequency components need to be calculated if speed is an issue.

3 FREQUENCY DOMAIN BASED APPROACHES TO ACQUISITION.

3.1 Time-Domain Circular Correlation

In order to complete the necessary acquisition operations in real-time, a number of approaches have been used. They range from the combination of time and frequency domain discussed above to more rapid frequency domain based approaches.
Since this thesis is focusing in a frequency domain approach, the discussion will be limited to this area.

Before going into details about frequency domain acquisition, a mathematical exploration of the circular correlation is in order. If a signal passes through a linear time invariant (LTI) system, the output of that system can be found by either convolution in the time domain or through the Fourier Transform in the frequency domain. We will limit our discussion to discrete LTI systems because of the digital nature of today's computers. If the impulse response of a discrete LTI system is given as h(n), then an input signal x(n) will produce an output y(n) through convolution as follows:

$$y(n) = \sum_{m=0}^{N-1} x(m)h(n-m) \qquad \textbf{(3.1)}$$

Note the similarity to the correlation function expressed earlier. The exception is the negative sign in the second term. Convolution in the time domain is equivalent to multiplication in the frequency domain, hence the above expression transforms to:

$$Y(k) = H(k)\sum_{m=0}^{N-1} x(m)e^{(-j2\pi mk)/N} = X(k)H(k) \qquad \textbf{(3.2)}$$

where Y(k), X(k) and H(k) are the DFTs of y(n), x(n) and h(n). These convolution expressions are often termed **circular convolution** because the results are periodic due the periodic nature of the DFT.

The acquisition algorithm uses correlation and not convolution, but the two processes are very similar, with a simple flip of sign on one of the operands. In the time domain circular correlation is given by:

$$a(n) = \sum_{m=0}^{N-1} x(m)h(n+m)\,; \qquad \textbf{(3.3)}$$

in the frequency domain the function is given by the formula:

$$\textbf{|A(k)| = |X*(k)H(k)| = |X(k)H*(k)|,} \qquad \textbf{(3.4)}$$

where **X*(k)** and **H*(k)** are the complex conjugates of **X(k)** and **H(k).**

As mentioned earlier, this formula represents the circular correlation of the signal x(n) and h(n), which is the required operation for acquisition.

3.2 Resolving range and Doppler uncertainties

The GPS signals that are collected have two uncertainties associated with them. These are in the realm of velocity (which gives rise to a Doppler ambiguity) and distance (which gives rise to a range ambiguity). This combination of uncertainties means that there are a large number of range-Doppler estimations that must be carried out in a fairly short period of time. For example, if there are 10 possible Doppler shifts and 5,000 possible time delays, then there are a total of 50,000 combinations of range-Doppler

uncertainties to consider. This concept is illustrated in Figure 3.1. This algorithm operates on a10-ms block of data and generates Doppler shifted copies of the input signal and correlates them with the baseband reference code. This result is sometimes referred to as **the ambiguity function** of a signal and is often used in RADAR processing. [22] The algorithm will give a global maximum that corresponds to the best Doppler and delay match for the input data. In discrete-time notation, the ambiguity function is written as:

$$\left|\Psi(\tau,\nu)\right| = \left|\sum_{n=0}^{N-1} x(n)h^*(n-\tau)e^{j2\pi\nu n/N}\right| \qquad \textbf{(3.5)}$$

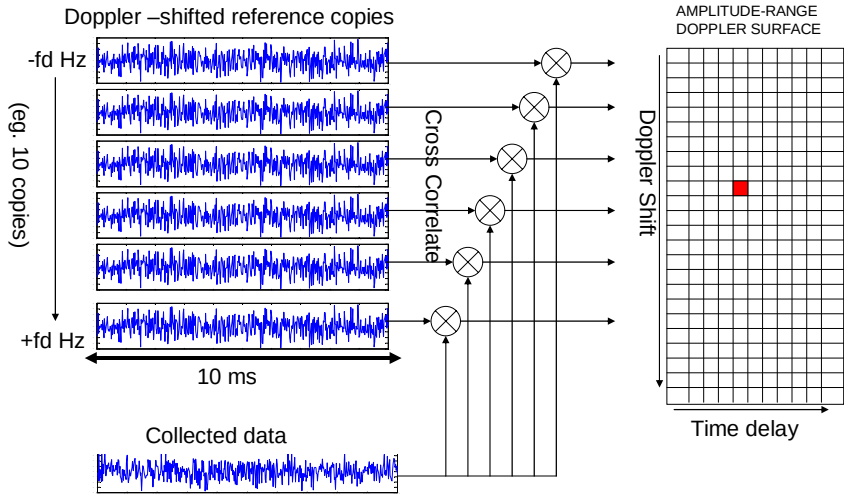

Figure 3.1 The Ambiguity Function Concept

If each block of 10-ms input data were used for correlation with 10-ms of the reference C/A code, the number of calculations would be prohibitive. A modified algorithm is used in which the input signal is broken up into 10 blocks of 1-ms and processed in parallel.

3.3 Noise and Amplification Considerations

The GPS signal is typically about -130 dBm at the antenna [23]. The maximum processing gain of a typical GPS system is given by:

10 log (chip rate /data rate) = 10 log (1.023 MHz/50 Hz) = 43 dB for this system.

The formula for available noise power N_i, in watts, at the input of the receiver is:

$$N_i = kTB \text{ watts,} \tag{3.5}$$

where *k* is Boltzman's constant (1.38 X 10^{-23} J/K), T is the absolute temperature of a 50 Ω resistor R, and B is the bandwidth of the receiver in Hz. Over a sampled null-to-null bandwidth of 2.046 MHz, the integrated thermal noise power (T = 290 K) expressed in dBm (dB relative to a milliwatt) is given as follows:

Ni = 1.38 X 10^{-23} x 290 x 2.046 x 10^{6} = **8.1881 x 10^{-15} W** over a 2.046 MHz bandwidth.

If we take this power relative to a milliwatt and find the power per Hz, then we get:

$$\text{Ni} = \frac{8.1881 \times 10^{-15}}{10^{-3} \times 2.046 \times 10^{6}} = \mathbf{4 \times 10^{-18}} \tag{3.6}$$

We can then take log to base 10 and multiply by 10.

So now N_i (dBm**) = 10*log10(4 x 10^{-18}) = -174 dBm/Hz.**

This works out to approximately -111 dBm, for a 2.046 MHz bandwidth [24]. In this arrangement, the GPS signal is 19 dB below the noise floor at the input to the receiver. One common approach is to amplify the signal so that the noise floor at right at the maximum range of the ADC. A typical ADC will have a 100 mV dynamic range. The corresponding power is:

$$P = \frac{V^2}{2R} = \frac{(0.1)^2}{2 \times 50} = 0.1 \text{ mW} = -10 \text{ dBm.} \tag{3.7}$$

Since the noise floor is -111 dBm, then 101 dB of amplification would be adequate to get it to this range. In the RF chain there are insertion losses caused by mixers, filters and cables and these must also be compensated for when calculating the required net gain of the system. We want to avoid saturation of the ADC and also wastage of the quantization levels.

Assuming a base band processor requires a Bit Error Rate (BER) of 10^{-5} , the corresponding post correlator E_b/N_0 for the BPSK modulation used is no less than 9.5 dB (assuming additive white Gaussian noise). E_b/N_0 is defined as the ratio of energy per bit to the spectral noise density [25]. Subtracting 43 dB of theoretical processing gain from the required 9.5 dB post correlator E_b/N_0, the minimum SNR at the correlator input is -33.5 dB. Assuming implementation losses of 3.3 dB for software GPS, the SNR required at the quantizer input is – 30 dB. Assuming insertion losses of about 10 dB for amplifiers and filters, the GPS signal can theoretically be 23.5 dB below the noise floor and still be acquired. Using results of experimental work by Tsui et.al, very low energy signals of -20 dB relative to the noise floor can be acquired. A block size of 10 ms input data is found to be adequate for C/A code acquisition at these low signal levels. The longer the data record used, the weaker the signals that can be acquired.

This value for block size yields a searching frequency resolution of 100 Hz. This operation is suitable for a block of data. The data are sampled at 5 MHz and stored in memory for use as needed by the C/A system.

3.3 Acquisition By FFT Based Circular Correlation

Van Nee and Coenen have devised a technique to perform the two dimensional search for code phase and Doppler shift in a parallel manner [26]. The algorithm used in this paper is based largely on this technique with some modifications for handing a longer data record. Assuming we know the satellite were are trying to acquire, following steps, as shown in Figure 3.1, are taken to perform acquisition on the input data:

1. Downconvert the input data y (n) to 10 slightly different frequencies close to baseband by multiplication with a 10 different complex RF signals f_i separated by 1 KHz. The resulting complex signals are now called d_c(n)[i] where i = 1 to 10 .
2. For each i, perform the DFT on the 10 ms of d_c (n) in 1ms blocks, and convert the input into the frequency domain as D_C (k) where n = k = 0 to 4999 for each 1 ms block. This will result in 10 different frames of D_C (k) each of length 5,000.
3. Take the complex conjugate of each frame of D_C (k) and the outputs become D_C (k)*.
4. Discard the second half of the D_C(k)* sequence, we only lose a small amount of energy from this step. There are now 10 frames of D_C(k) of length 2,500.
5. Generate one local reference C/A code c_a (n) of length 1 ms for the given satellite. The local code must be sampled at 5 MHz to generate 5,000 samples.
6. Perform DFT on c_a(n) to transform it to the frequency domain as C_A(k).
7. Discard the second half of the C_A(k) sequence. 2,500 samples remain.
8. Multiply C_A(k) and each frame of D_C(k) point by point and call the result D_S(k). There are 10 frames of D_S(k), each of length 2,500.

9. Take the inverse DFT of each frame of D_S(k) to transform the result into the time domain as d_s(n). There are now 10 frames of d_s(n).
10. Store the frames in parallel in a 10 * 2,500 matrix.
11. Access the matrix one column at a time and perform a 10 point DFT on each column and repeat until all columns have been transformed. The result is a 10*2,500 matrix in called D_S_P(k).
12. Sort through the matrix to locate the maximum of the | D_S_P(k)|. Store the row and column index of this maximum.
13. Repeat steps 2 through 12 for all i to generate 10 maxima and their associated row and column indices.
14. Sort through these maxima to generate a global maximum with an associated row, column and i index.
15. Via some simple arithmetic shown in the following equations , decode these three indices to infer the C/A code phase and carrier Doppler shift.

The complex RF signal is represented by the equation:

$$e^{j\omega t} = \cos(\omega t) + \sin(\omega t) \qquad \textbf{(3.8)}$$

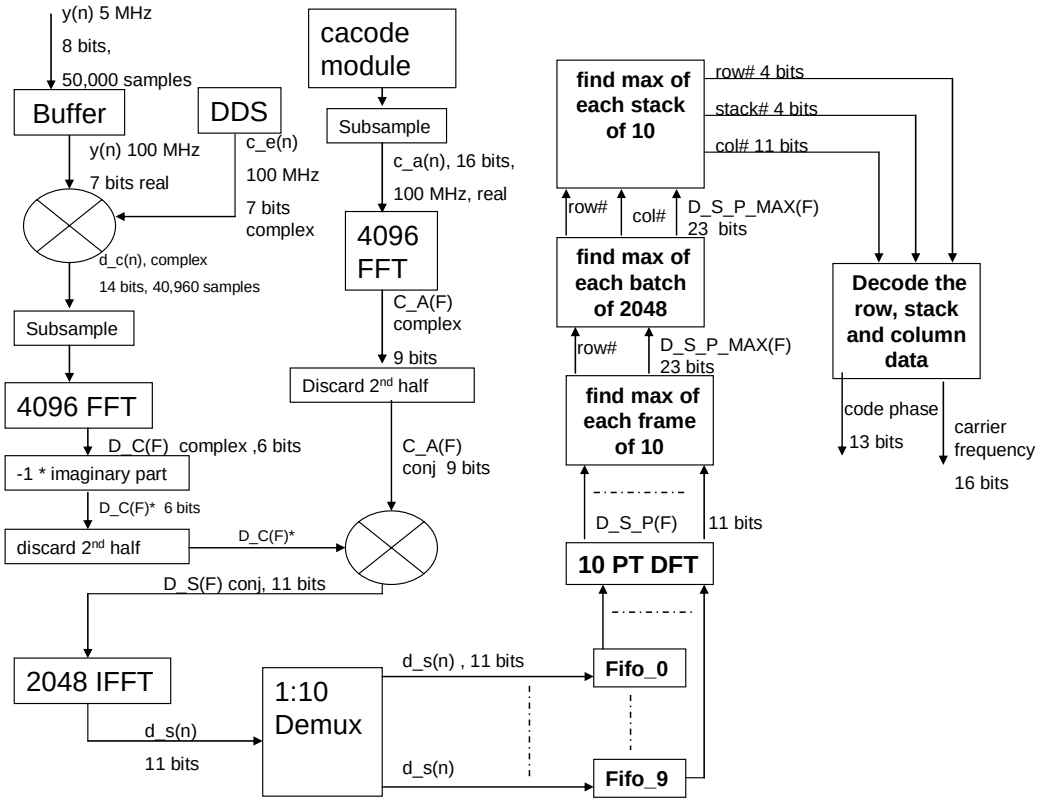

Figure 3.1 A block diagram of the proposed C/A code acquisition process.

The equations for decoding the code phase are shown below. Let us define the inputs as follows:

i_c is the column index, *m_c* is the stack index, and *m_f* is the row index.

The code phase in samples, *c_p* is therefore calculated as:

$$c\_p = (2500 - i\_c) * 2 \qquad \textbf{(3.9)}$$

Multiplication by 2 is as a result of the dropping of the second half of the DFT results in steps 4 and 7. The result needs to be mapped back from a data set of 2,500 to a data set of 5,000. The code phase time resolution is accurate to 200 ns in this approach. Since the code chips are about 977 ns long, this is well within the half-chip resolution needed by the system.

The calculation for Doppler frequency, *dopp* is a little more involved and depends on the value of the row index *m*_f. If *m_f* is less than or equal to 5, then the following calculation is used:

$$dopp = (m\_c - 10/2) * 1000 + (m\_f - 1) * 100 \qquad \textbf{(3.10)}$$

If *m_f* is greater than 5, then the following calculation is used:

$$dopp = (m\_c - 10/2) * 1000 + (m\_f - 11) * 100 \qquad \textbf{(3.11)}$$

This acquisition process repeats every 10 ms, with a fresh block of data in order to keep a lock on the C/A code for a given satellite.

3.4 Threshold Values

In the above discussion, no mention of threshold has been made, but this is a very important consideration. Depending on the environment where this GPS receiver is operating, a threshold is set to determine if the maximum is a valid or spurious maximum. So not only does the maximum have to be determined, it then has to be compared to a threshold to determine its validity. The discussion and calculation of threshold is beyond the scope of this paper, but it is important to be aware that generating a maximum is not sufficient for determining if acquisition is successful.

4 ARCHITECTURE AND DESIGN CONSIDERATIONS

The FFT based C/A code lock algorithm chapter was written in first in MATLAB to demonstrate proof of concept and then was implemented in VHDL code targeted to a Xilinx Virtex II-Pro FPGA. There are numerous choices an engineer must make when translating from high level code such as MATLAB, to more hardware-faithful code such as VHDL. This section highlights some of the design choices that were made to implement the design efficiently in digital FPGA hardware.

4.1 Design Choices

4.1.1 Buffering

An effective architecture will require that while the first 10 ms of data are being processed; the second 10 ms worth of data is being buffered. The 10 ms worth of data is equal to 50,000 samples. This leads to a decision of designing a 2-port RAM of depth 100,000 to read from and write to the buffer at the same time. The designed 2-port RAM is more effective than the FIFO generated by the Xilinx core generator for two reasons:

1. Xilinx Intellectual Property (IP) cores only generate FIFOs with memory size of radix-2, so the closest size to 100,000 is around 131,072 leading to a waste of resources.
2. The FIFO has a “quirk” called first-word-fall-through after reset, which would cause errors and requires extra hardware for the control logic.

4.1.2 Sample Rate Conversion

Another design challenge was the generation of the C/A code samples. The code samples correspond to the code that is generated at 1.023 MHz and sampled at 5 MHz. One could generate a 1.023 MHz clock for a LFSR that generates the code, and then use a separate 5 MHz clock for the sampling circuit. This however would lead to a synchronization issue between the two clock domains. Both clocks would have to be rising at exactly the same time to ensure alignment of the code samples with the originally generated code. Even if this issue could be overcome, there is yet another challenge to design FPGA circuits that can produce a 1.023 MHz clock from the existing crystal-generated on-chip clock. The ratio of 5 to 1.023 is not a rational number and hence cannot be used as an input to the delay locked loop (DLL) that is used to generate the clocks of desired frequency.

The solution that was arrived at (with assistance from some Air Force Engineers) is as shown in figure 4.1: A counter is set to increment by multiples of 1,023, and 5,000 is subtracted from the count every time the count exceeds 5,000. So basically we are dividing 5,000 by 1,023 and only keeping the remainder. This cycle repeats indefinitely. An output is enabled for one clock every time the count exceeds 5,000. This signal enables the shift logic of the code generator, advancing it by one state. To the outside world it then appears that the 1.023 MHz code is being sampled at 5 MHz.
A check with the MATLAB generated and sampled C/A code reveals identical sequences.

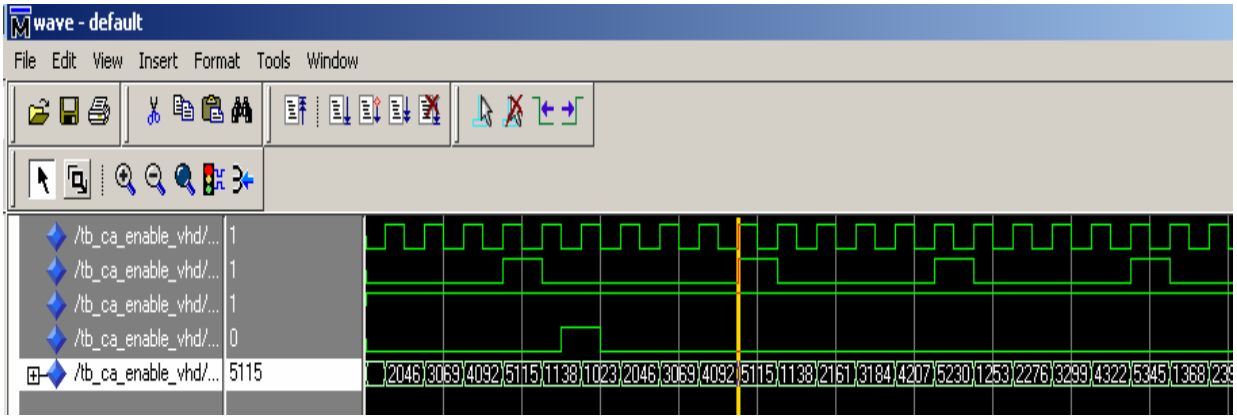

Figure 4.1. Count sequence of LFSR_enable module showing enable signal

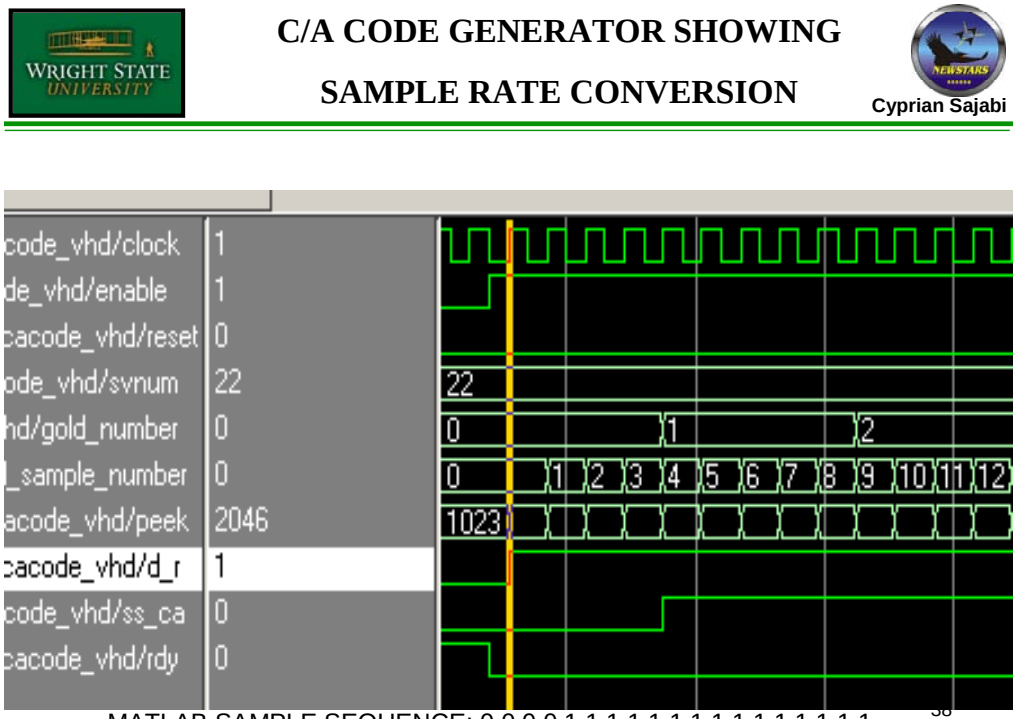

Figure 4.2. C/A code generator showing sample rate conversion

4.1.3 Mapping 5,000 Point DFT To Hardware

The major obstacle to be overcome with this design was the mapping of a 5,000 point DFT from the MATLAB onto the hardware. In the MATLAB model, the input

data is processed in frames of 5,000. A 5,000 point DFT is straightforward in MATLAB because of the use of floating point calculations. In fixed-point hardware, this is a challenge. All hardware implemented DFTs are radix-2 or radix-4 FFTs because they need to be completed in real-time. The closest hardware implementation of a 5,000-point DFT is a 4,096 point FFT. Therefore, there needs to be a way to convert the 5,000 samples to 4,096 samples. One choice explored was the averaging correlation technique [27]. In this technique, the 5,000 samples would be averaged to 4,096 samples prior to application of the FFT. Although there has been success using the averaging correlation method [27, 28], it is relatively hardware and time-intensive. It seems to work well for a 1 ms acquisition, but for a 10 ms acquisition, it would prove too cumbersome. In the interest of hardware simplicity a pseudo-random sampling technique is proposed instead. If a pseudo-random scheme is used, then the data set retains most of its autocorrelation properties as shown in Figure 4.2. These autocorrelation properties are important for direct sequence spread spectrum acquisition [29].

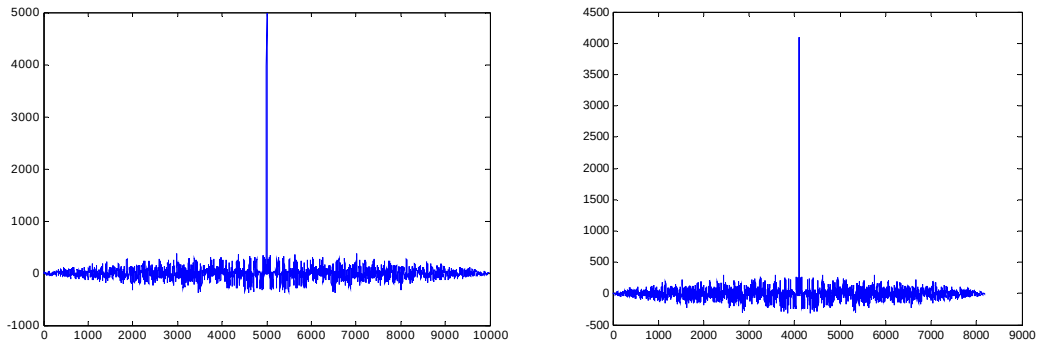

Figure 4.3 Autocorrelation properties of C/A code before and after "subsampling."

After experimenting with a few "sub-sampling" algorithms, an effective algorithm was proposed by D. Lin of WPAB, which proves to give the desired results. The algorithm is described below:

1) Out of every 11 samples, the 5th and 11th samples are dropped. This procedure is repeated 90 times. We would have now processed 900 input samples and stored 810 samples and dropped 90 samples.

2) For the 91st set of 11, only the 5th sample is dropped. We have now processed 1,000 samples and having stored 819 samples and discarded 181 samples.

3) Steps 1 and 2 are repeated 5 times for each set of 1,000 input samples to yield 4,095 stored samples. The 4,096th stored sample is always a zero. Now there are a total of 4,096 samples for further possessing. This "sub-sampling" technique is applied to both the C/A code and the downconverted data preserve the relative sample position of the code to the incoming data.

4.1.4 Direct Digital Synthesizer (DDS) Challenges

Another challenge was the generation of the complex RF sinusoids used for downconversion. In MATLAB, one can generate any arbitrary frequency and sample it at any other arbitrary frequency as long as the Nyquist criterion is met, i.e. the sampling frequency must be at least twice the bandwidth of the signal being sampled. In hardware, the RF signal is generated by a Direct Digital Synthesizer or DDS. The frequencies generated by the DDS are constrained by the input clock frequency. The DDS is required to generate the equivalent of complex sinusoids ranging in frequency from (1.25MHz - 4 KHz) to (1.25 MHz + 5 KHz), and sampled at 5 MHz. However, the rest of the system is configured to run at 100 MHz, so the DDS must be configured to produce frequencies 20 times (100/5) the actual value stated. For example a 25 MHz RF signal sampled at 100 MHz would generate the same sample set as 1.25 MHz sampled at 5 MHz.

5 GLOBAL VIEW OF DATAFLOW THROUGH THE SYSTEM

The dataflow is partitioned into 4 functional domains: 1) Data Capture Domain; 2) 4,096 FFT domain; 3) 2,048 IFFT domain; and 4) 10 point DFT / Sorting Domain. Considering these breakdowns, each functional domain will discussed as follows.

5.1 Data Capture Domain.

This domain functions to capture the data that is streaming in at 5 MHz, downconvert it to baseband, "subsample" it, and stream it to the 4,096 FFT at 100 MHz. It also regulates the generation and storage of the C/A code. The Data Capture Domain consists of two dual-port RAMs, a Direct Digital Synthesizer, the C/A code generator, a complex multiplier, counters, and some multiplexing ability to allow reuse of hardware. The block diagram is shown in figure 5.1.

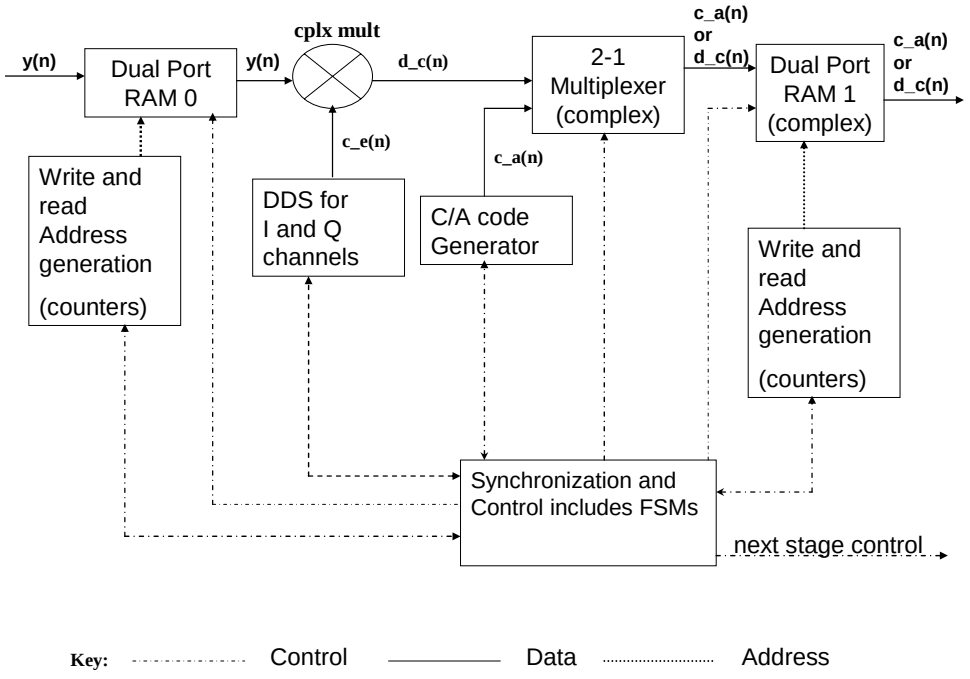

Figure 5.1. The Data Capture Domain

5.1.1 Details of Datapaths In The Data Capture Domain

5.1.1.1 The Dual Port RAMs

Each dual-port RAM has two address ports, one read port and one write port, and allows data to be written to it from one port, while data is being read from it via a different port. The ports can operate on different clock domains.

As shown in Figure 5.1 the **Dual Port RAM 0** is of width 8 bits and depth of 100,000, and is configured as a circular buffer that continuously reads in the data at 5 MHz. **Dual Port RAM 0** buffers the incoming 5 MHz data streams**.** When this RAM is about 39,000 full, or 89,000 full, we start to read from it at 100 MHz for further processing. There is **control** so that when we finish reading the first 50,000 samples, we wait until the buffer is 89,000 samples full again before we read from it again. This is done continuously as long as the system is running. Once **Dual Port RAM 0** is almost full (39,000 or 89,000), the buffered data is then complex multiplied at 100 MHz. The 50,000 block of data is read from Dual Port RAM using two counters; a 0 to 50,000 and a 50,000 to 100,000 counter each running at 100MHz. The counters are enabled as needed. The **Dual Port RAM 1** is actually two dual port RAMs, each of depth 40,960 and width 16 bits. They store the real and imaginary "subsampled" downconverted data for streaming to the FFT module. When the system is initialized for given satellite, **the Dual Port RAM 1** stores the "subsampled" C/A code (which is all real) hence the need for a multiplexer. This C/A code is streamed to the 4,096 FFT for further processing before the primary data is written to **Dual Port RAM 1**. This initialization process occurs very early on in the cycle and is complete within about 95 μs of asserting global start.

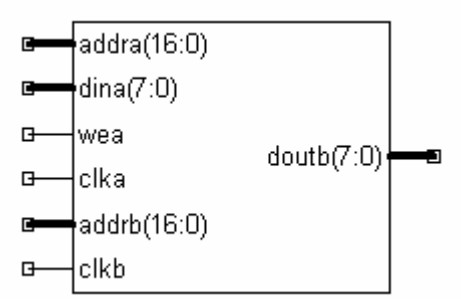

Figure 5.2 Dual Port RAM used in Data Capture Domain

5.1.1.2 Counters

1. **Counter_99999** is configured to count to 99,999 and asserts two terminal count signals, *tc_0* when its count reaches 48,000 and *tc_1* when the count reaches 98,000. When the count reaches 99,999, it rolls over to 0 and restarts its count. This counter resets to 0 and needs an enable signal to increment. This counter generates the write address to the Dual Port RAM 0 to collect the streaming 5 MHz input.
2. C**ounter_50000** asserts its terminal count, tc_0 signal when it reaches 50,000. This counter generates the read address for read operations from the bottom half of the Dual Port RAM 0.
3. **Counter 50000_100000** counts from 50,000 to 100,000 and asserts its terminal count signal when it reaches 100,000. This counter generates the read address for read operations from the top half of Dual Port RAM 0.
4. There are two instances of **Counter_40960,** which has a terminal count of 40,960. It asserts *tc_0* when the count reaches 31,000 and *tc_1* when the count reaches 40,960. These counters are used to control the write and read address respectively of the Dual -port RAM 1. The one that controls the write address is designated as **Counter_40960_w**, and the one that controls the read as **Counter_40960_r**.

5. **Counter_4096** asserts its terminal count signal, *tc_0* at its terminal count of 4,096. This count is used to control the asserting of the start signal for the FFT module every 4,096 clocks.

6. There are also two counters with a terminal count of 4 and 5 respectively. These counters are used to control which data points are "dropped" during "subsampling" via the manipulation of the write signal to Dual Port RAM 1.

5.1.1.3 The Direct Digital Synthesizer (DDS)

The **DDS** is responsible for generation of the complex RF signal. It is generated by the Xilinx Logicore IP generator. The DDS has a large number of active high inputs. The relevant ones are as follows: 28 bit *data* for programming new frequencies, *write_enable* for clocking in the new frequency, and *clock_enable* for enabling the generation of outputs. When programming in a new frequency, the *clock_enable* is deasserted while the *write_enable* is asserted for one clock.

The relevant outputs from the **DDS** are the 9-bit *sine* and *cosine* outputs, which are out of phase by $\pi/2$, and the active high *data_ready* output, which indicates when the outputs samples are available and valid. Figure 5.3 shows the DDS schematic symbol.

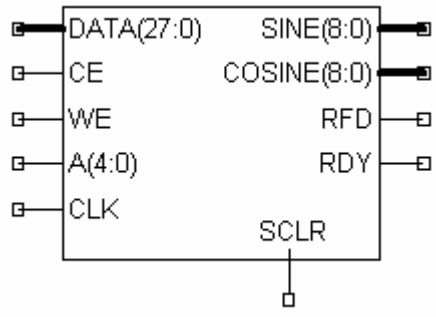

Figure 5.3 Schematic symbol of the Direct Digital Synthesizer

5.1.1.4 The C/A Code Generator (CCG)

The C/A code generator, **cacode** is a Gold code generation module, which consists of a pair of Linear Feedback Shift Registers, a controller that schedules loading and shifting the code generator, a ROM for the initial fills, and a clock division circuit for controlling the generation of a sampled version of the code. This module, although clocked at 100 MHz, generates the equivalent of 5 MHz C/A code samples. The inputs to this, which are self-explanatory are: *enable, reset, and svnum*, the satellite number. The *enable* input needs to remain asserted for the code to be continuously generated. The relevant outputs are the *ss_ca*, the code samples and *data_ready*, indicating the data output data are valid. This module has to be reset before programming in a new satellite.

There are also a number of other output busses named *peek, gold_number*,and *gold_sample_number*,which are used for synchronization and debugging.

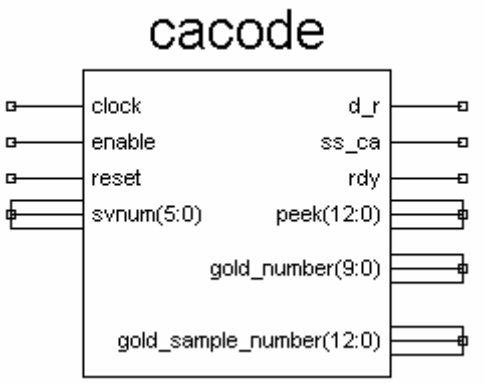

Figure 5.4 Schematic Symbol of C/A code Generator

5.1.1.4 The Complex Multiplier

The Complex Multiplier **(cplx_mult)** is a combinational complex multiplier that completes its multiply operation within the 10 ns window provided by the 100 MHz clock. It serves to point multiply y (n) with the complex output c_e(n) of the **DDS**. Because

there is no imaginary component to y (n), there is no collection and addition of terms at its output. Because of the simplicity of the operation, **cplx_mult** is implemented simply as a pair of multipliers, one for the real (I) part of y (n) and one for the imaginary (Q) part. The output of the multiplier is therefore complex. This multiplication is used to down-convert the input sequence to a base band signal. The complex data from the multiplier are subsequently "subsampled" and stored in the Dual Port RAM 1.

5.1.1.5 Complex Multiplexor (CM)

This multiplexer selects between c_a(n) and d_c(n). Once the C/A code has been generated and stored in Dual Port RAM 1, it selects d_c(n) as its input source and remains in this state until a new satellite is programmed in.

5.1.2 Controllers in the Data Capture Domain.

This is an overview of the controllers in the data capture domain and their role in controlling the components in this domain. The control in this domain serves to: 1) prevent "collisions" i.e. trying to read from and write to the same memory location and 2) synchronize the timing of the write and read operations to minimize time wastage.

5.1.2.1 Front_dual_port_control (FDPC)

FDPC controls the write to the **Dual Port RAM 0**. FDPC is in the 5 MHz clock domain and has a *start* and *reset* input. Its outputs are an *write_enable* for **Dual Port RAM 0**, and an *enable* and *reset* for **counter_99999.** When *start* is asserted, on the next

rising clock edge, counter_99999 is enabled and it begins counting. This counter remains enabled until the system is reset.

5.1.2.2 DDS / Data Controller (DDC)

DDC serves two purposes:

1. To ensure that the proper down conversion frequencies are generated from the DDS at the correct time
2. To synchronize the input data samples, y (n) with the sinusoid samples c_e(n) during point multiplication.

This controller has control and feedback loops with the DDS and all the counters described previously. For the sake of simplicity the names of the inputs and outputs will not be mentioned here. When started, DDC activates the DDS using the *clock_enable* input in its default setting and when the DDS begins to generate valid c_e (n) samples. The DDS needs to be enabled for all the data streaming in from **Dual Port RAM 0**, and is idle for one clock while the **DDS** frequency is being changed. There are no data being read from **Dual Port RAM 0** when the DDS frequency is being changed. DDC also initiates the reading of y(n) samples from Dual RAM input 0. The read address for Dual RAM 0 is generated via counter **Counter_50000** or C**ounter_50000_10000,** which are both under the control of DDC**.** The DDS output c_e(n) and the primary data samples, y(n) are piped into **cpx_mult** where a point multiplication occurs to generate the downconverted signal d_c(n) When the read address counters reach their terminal counts, they signal DDC, which then controls the loading of a new down conversion frequency into the DDS via the *write_enable* pin and restarts the multiplication of the

new c_e(n) with the same y(n). This process is repeated 10 times until all downconversion frequencies have been covered. This is a total of 500,000 multiplications. 10 batches of d_c(n) are generated in this fashion. Each batch is 50,000 samples long. The batches are then processed in frames of 5,000 samples. After all 10 downconversion frequencies have been covered DDC goes back to the init state and waits until either of the terminal count signals from the 5 MHz **counter_99999**.

5.1.2.3 Supplementary Control for Subsampling (SCS)

For each of the of d_c(n) frames of 5,000, **SCS** initiates and controls the "subsampling" and storage of the 4,096 samples in **Dual RAM 1**. The subsampling is mediated by a series of nested loops as described in the following sequence of events. The subsampling process is mediated via **Counter_4, Counter_5** and internal counters **Counter_91, Counter_5 and Counter_10.** The data is processed in batches of 11. For each set of 11 input samples, **SCS** causes the write signal to be asserted for 4 clocks, low for 1 clock, high for 5, and low for 1 and repeats this sequence 90 times. For the 91st iteration, the write signal remains high for all the final 6 of the 11 samples. This process is repeated 5 times (mediated by **counter_5**) for a total of 5,000 input samples. At this point 4,095 samples have been written to the RAM, so a zero is padded onto the data. There is a further layer of this loop, mediated by counter_10 that allows 50,000 samples to be processed. This cycle repeats 10 times (i.e. once for each down conversion frequency.) After the 10th frequency has been completed, **SCS** goes back to an init state and waits for the next flag from the *dds_rdy* signal.

5.1.2.4 Pre FFT Control (PFC)

PFC has two inputs: *count_4096_tc, count_40960_pre_done,* which are connected to *tc_0* of **counter_4096** and *tc_0* output of **counter_40960_w** respectively. The PFC outputs are: *count_reset_4096, count_enable_4096, count_reset_40960, count_enable_40960*, and *fft_start*. These outputs are connected to **counter_4096, counter_40960_r**, and **FFT_ 4096** respectively. The names are self explanatory.

Using the counters mentioned above, this controller serves to synchronize the data samples coming out of **Dual Port RAM 1** with the 4,096 point FFT module as well as to initiate the processing of each frame of 4,096 data samples. The read address for streaming the data out of **Dual Port RAM 1** is supplied by **counter_40960_r. PFC** is configured to start streaming data to the FFT from **Dual Port RAM 1** when about 31,000 d_c(n) samples (when *tc_0* from **counter_40960_w** goes high) have been written to **Dual Port RAM 1**. This number was arrived at after careful analysis of preliminary simulation results. The purpose is to allow a minimum of time wastage in between downconversion frequencies, but still keep collisions from occurring, i.e. trying to write to and read from the same memory location. Because the 4,096 point FFT module can only process data in frames of 4,096, the *fft_start* signal must be pulsed once every 4,096 clocks a total of ten times to process 40,960 samples. The synchronization of these pulses is achieved via a feedback loop with **counter_4096**.

5.1.2.5 C/A Code Control (CCC)

This controller is designed to enable the generation and storage of the PN code by the C/A code generator as well as initiate the FFT of the C/A code to yield the C_A (F)

samples. It is necessary to subsample the C/A code in order to preserve autocorrelation relationships between c_a(n) and d_c(n).

This controller works only once for each satellite programmed into the system, and it completes its task early on in the acquisition process-within about 90 μs of the initiation of the start signal. The relevant inputs *are enable,* and *tc_4096* from **counter_4096.** The relevant outputs are *start_pre_fft_cont, reset_pre_fft_cont*, *mux_sel_cacode, start_cacode, cacode_reset, start_supp*, and *reset_supp*. These outputs connect to **PFC, CM, CCG,** and **DDC** respectively. At the start of an acquisition cycle for a given satellite, when system enable is pulsed, the CCC module sends a start signal to the CCG to start generating C/A code. At the same time it sends a start signal to the DDC to start "subsampling" the code and storing it in Dual Port RAM 1. When the tc_4096 input goes high, this means that 5,000 c_a(n) samples have been generated and 4,096 of these have been subsampled and stored in Dual Port RAM 1. CCC now switches CM to the d_c(n) input once this initialization is complete. To complete its function, CCC triggers **PFC** to carry out one 4,096 FFT of the C/A code samples and resets **PFC** once *tc_4096* goes high. **CCC** resets **CCG** and **DDC** once it has initiated the **FFT**. These modules all sit idle until the first 31,000 data samples are written into **Dual Port RAM 0. CCC** allows reuse of the **Dual Port RAM 1**, the "subsampling" circuitry, and the 4,096 point FFT.

The system described in this chapter functions continuously on each unit of 50,000 samples until a new satellite is selected.

5.1.2.6 Some Observations and Notes on Synchronization:

The DDS only needs one clock to write in a new frequency and at the next clock it starts showing the new samples on its output. Because of this we are able to incorporate an extra count cycle into the counter_50000 and counter_50000_100000 to take care of this extra clock cycle. This saves coding an extra state in DDC. This is only true of the DDS when zero cycle latency is selected for the DDS at the Xilinx Logicore GUI. The *dds_rdy* signal is always high in this setting unless the core is reset.

In the MATLAB code the C/A code samples were represented by -1 and 1 instead of 1 and 0 as in hardware. This format largely eliminates DC bias and keeps the dynamic range of the C_A (F) signal much lower than in the latter case. In order to exploit these benefits in hardware, a special combinational module was added to the output of the C/A code generator. This module converts outputs of '1' to -8192 in 2's complement and converts '0' outputs to 8192(all '0' followed by a '1'). The reason for using ± 8192 instead of ± 1 is because the round off noise inside the FFT core will drown out really small magnitude numbers, leading to highly inaccurate results. This architecture also protects the FFT core from overflow and allows considerably fewer bits to accurately and completely represent C_A(F).

5.2 FFT 4,096 Domain

5.2.1 Overview of Operations in the FFT 4,096 Domain.

As shown in figure 5.5, this FFT 4,096 Domain serves to: 1) regulate the storage of the FFT of the C/A code (C_A(F)) and 2) synchronize the C_A(F) signal with the FFT of the down converted data (D_C(F)).

The first 2,048 samples of 4,096 C_A(F) samples are stored in 2048_RAM via the demux, and within 155 μs of asserting global start, the 2,048 FFT output samples from these data are stored and ready for use. The demux then switches to the alternate (lower) output and stays in this mode for the remainder of the acquisition process until a new satellite is selected.

As shown in Figure 5.5, the Complex Multiplie**r** (cplx mult**)** multiplies D_C(F) and C_A(F) and generates the output D_S(F). This multiplication is a point multiplication equivalent to circular correlation of the base band y (n) and c_a (n) in the time domain. The actual conversion of D_C(F) to complex conjugate takes place in the complex multiplier. The multiplier is modified so the subtraction and addition operations are reversed after operands are collected. If the two operands are *(a+jb)* and *(c+jd)*, then a straightforward complex multiplier would yield: *(ac-bd)* for the real and *(bc+ad)* for the imaginary. If the second operand is the complex conjugate of the first, i.e. *(c-jd)*, then the result is *(ac+bd)* and *j(bc-ad)*.

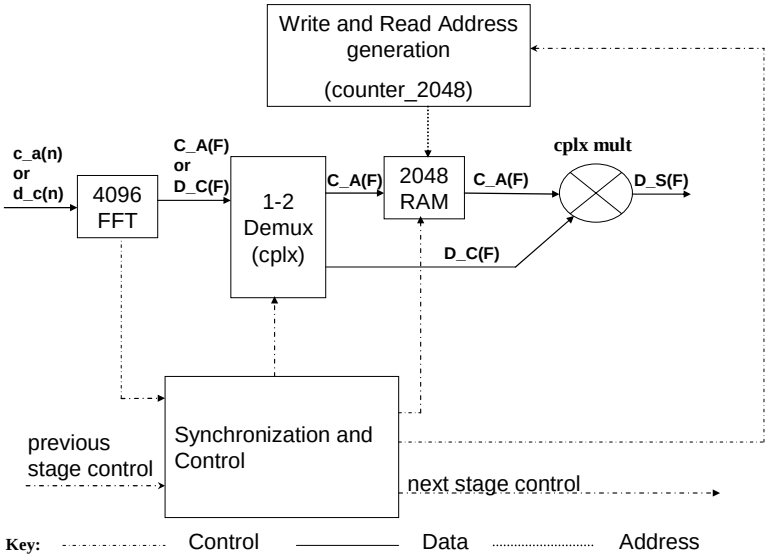

Figure. 5.5. Block diagram of 4,096 FFT domain

5.2.2 Datapath And Control Operations In The FFT 4,096 Domain

5.2.2.1 FFT_4096

The FFT_4096 module is a pipelined streaming I/O 4,096 point FFT engine generated by the Xilinx Logicore Software. The core can, "simultaneously perform transform calculations on the current frame of data, load input data for the next frame of data, and unload the results of the previous frame of data. The user can stream in input data and, after the calculation latency, can continuously unload the results" [30]. The core is designed to stream in both real and imaginary inputs and stream out complex outputs after a 4,096 clock latency. This IP core is unscaled, meaning that for each butterfly stage, more bits are added to the datapath. This means that for this particular implementation the output width is 13 bits greater than the input width. The reason for this choice of implementation is based on the MATLAB simulations.

The input and output are serial, and once the module is started, it will continuously compute FFTs of data in 4,096 frames. The only control required is to activate the core for each new 4,096 frame of d_c(n). This control comes from the previous stage. The algorithm requires only the first half of the output D_C (F) for use in the next stage, but the core needs to output the results of the entire transform before the next frame output can be accessed. This causes an unavoidable delay. Because one processing unit is effectively 4,096 * 10 samples long, the 4,096 FFT must process the unit in 10 frames.

The FFT_4096 module is a Xilinx generated IP core, version v3.2. It has a large number of inputs and outputs, but the ones relevant to the design are as follows; Inputs: *xn_re, xn_im*, and *fft_start*. Outputs*: fft_done, overflow* (optional)*, xk_re*, and *xk_im.* The *xn_re* and *xn_im* are the 15-bit real and imaginary streaming inputs. For continuous

data processing, the *fft_start* can either be held high or pulsed every 4,096 cycles. This will result in processing the data in frames of 4,096 [31]. The instructions given on the datasheet read, "Simply assert START at any time to begin data loading. After the data frame is loaded, the core will proceed to calculate the transform and then output the results... Input data (*xn_re, xn_im*) corresponding to a certain XN_INDEX should arrive three clock cycles later than the XN_INDEX it matches. In this way, XN_INDEX can be used to address external memory or a frame buffer storing the input data." [32] Figure 5.6 shows the timing described in the previous instructions. Note that when the first input data samples are launched, the index is at 03, or on the 4th clock if start is launched on the 0th clock.

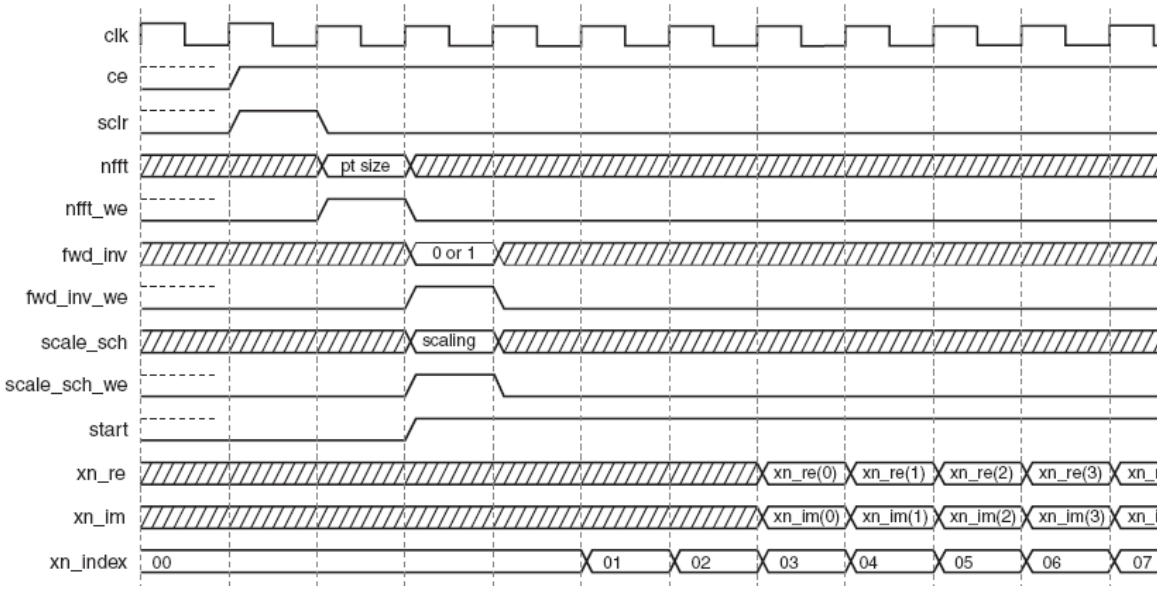

Figure 5.6: Synchronization of Beginning of Data Frame with index for FFT.

Figure 5.7 is a schematic symbol of the 4,096 FFT module showing the core pinout.

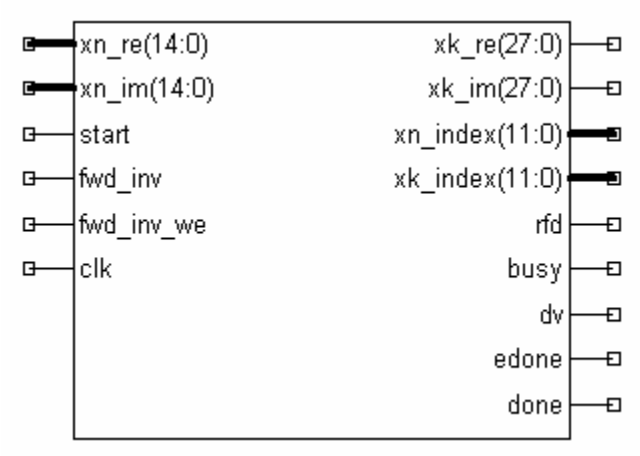

Figure 5.7 FFT_4096 symbol (overflow not shown)

After a latency of 4,096 clocks, FFT results begin to stream out from the core. As shown in figure 25, at this point the *fft_done* signal goes high one cycle before the core starts unloading. The *fft_done* signal then goes low and stays low until the next frame is completed (not shown).

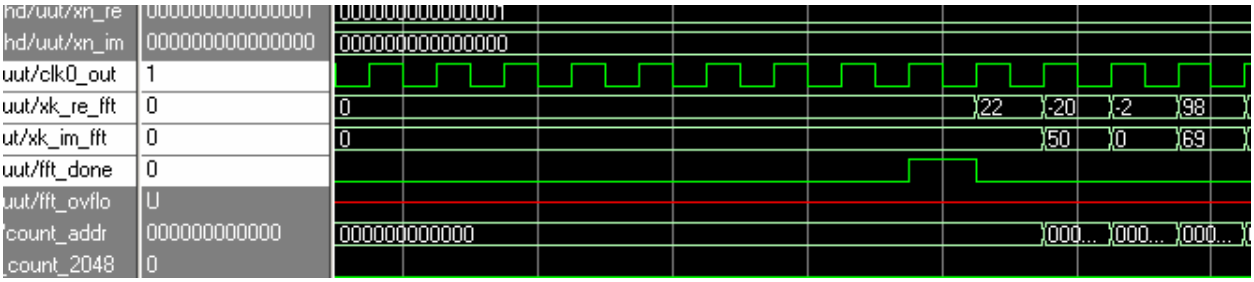

Figure 5.8, Unloading results after *fft_done* has pulsed

If an overflow had occurred during the processing of data, the *overflow* signal would go high and the user could then decide what to do with the output xk_re and xk_im samples. Because there is no scaling selected for this IP core, The *xk_re* and *xk_im* samples are 13 bits wider than the *xn_re* and *xn_im* samples in order to accommodate growth of the datapath within the core. Because of the allowance of datapath growth, there is no need for an *overflow* pin.

5.2.2.2 Counter_2048

This counter has an *enable* input and two outputs: a terminal count flag, *tc_0*, which goes high when the counter reaches a terminal count of 2,048, and a 12-bit *count_out*. The counter does not roll over and needs to be reset when it reaches its terminal count. This counter provides the write and read address for the complex RAM in this domain.

5.2.2.3 RAM_2048

This RAM is actually a pair of RAMs configured to act as one complex RAM to store real and imaginary data. It serves to store the first half of the complex C_A(F) output from the 4,096 FFT. The RAM is 2,048 deep and 9 bits wide.

5.2.2.4 Cplx_mult_1

This Complex multiplier has a modified function relative to a standard complex multiplier. It performs complex conjugate multiplication instead of regular complex multiplication. The working of this multiplier was explained earlier in this section. The real and imaginary inputs of this multiplier are *ar, br, ai,* and *bi.* It is written in VHDL using generics so that the operands have independent and variable widths. The real and imaginary outputs are termed pr and, *pi* respectively. This multiplier performs the complex multiplication in one clock, but can be modified for further pipelining using Attributes in the User Constraints File or in the VHDL code itself.

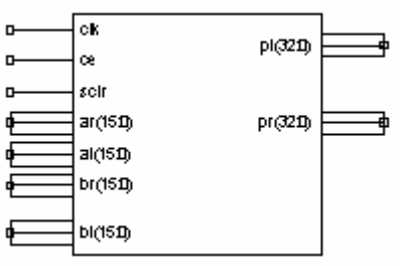

Figure 5.9 Complex multiplier Schematic

5.2.2.5 Demultiplexer_2 (DEMUX 1_2)

This complex demultiplexer is connected to the *xk_re* and *xk_im* outputs of the **FFT_4096** module and is used to route the data from the FFT either to the **2048 RAM** for C_A(F) storage or to the **CPLX_MULT_1** for the complex conjugate multiplication operation of C_A(F) and D_C(F)*.

5.2.3 Controllers in the FFT_4096 Domain

5.2.3.1 IFFT_Cont (IC)

This controller is concerned with regulating the storage of the first 2,048 C_A(F) samples in RAM 2048 and also controlling the multiplication of the stored C_A(F) samples with the D_C(F) samples that will eventually stream out of the FFT_4096. This module also initiates the start of each frame of 2,048 IFFTs. When the *fft_done* signal goes high, it does so one clock before the valid output begins. If this is the first high signal of *fft_done*, the controller will then signal on the next clock for the 2,048 counter to be enabled as well as for the write enable to go high for writing to the 2,048 RAM. 2,048 samples of data can then stream into the RAM. We are only going to use the first half of the FFT output for further processing. Once the count reaches 2,047, the *we* for the RAM is disabled and the rest of the FFT data is "lost".

IC's control inputs are *fft_done* and *ter_count_2048*. IC's control outputs are *start_ifft, start_counter_2048, reset_counter*, *we_ca_ram,* and *ca_vs_dc_sel.*

If *fft_done* goes high for the first time, then the module interprets this to mean that the FFT of the C/A code is complete and via *start_counter_2048,* it triggers **counter_2048** to start generating the write address for **RAM 2048**. At the same time *we_ca_ram* goes high enabling the write to the RAM. When *ter_count_2048* goes high, then IC disables and resets the counter and disables the write to the RAM. The *ca_vs_dc_sel* output controls whether the demux is steering data to the RAM or to the multiplier. At the point that the counter reaches its terminal count, the *ca_vs_dc_sel* changes to a '1' in preparation to steer the next set of streaming *xk_re* and *xk_im* outputs from the **FFT_4096** to the **cplx_mult_1**. From then on, all fft_done pulses are taken as a signal to complex conjugate multiply the D_C(F) and C_A(F) samples and stream the results to the IFFT_2048 module.

The next time *fft_done* goes high, the module asserts the *start_counter_2048* signal to the **counter_2048** to start generation of the read address for the **RAM_2048**. At this point the *xk_re* and *xk_im* samples streaming from the FFT represent D_C(F) samples. These samples are routed to the complex multiplier by the demux and point multiplied with the prestored C_A(F) samples streaming from the RAM. Only the first 2,048 samples of the D_C(F) samples from the FFT core are complex conjugate multiplied with the stored data from the 2,048 RAM. These multiplication results are buffered by 3 clocks and fed into the 2,048 IFFT core for further processing. If we count the clock on which *ifft_start* is clocked in as 0, then at clock 3, the first valid

multiplication results should be available at the ifft inputs. The need to buffer the input data by 3 clocks is unavoidable because of the use of an IP core with this requirement.

The product of the the complex conjugate multiplication is termed D_S(F) according to the algorithm. The **IC** module also asserts *start_ifft,* causing the **IFFT_2048** module to begin processing of this D_S(F) frame. Once the counter reaches its terminal count of 2,048, the **IC** module resets the counter and goes to a predetermined state where it waits for the next *fft_done* signal whereupon it repeats the procedure. This **IC** module is the only control module required for this domain. It sends the *ifft_start* signal to the next domain, the 2,048 IFFT domain to trigger its operations.

5.3 IFFT_2048 Domain.

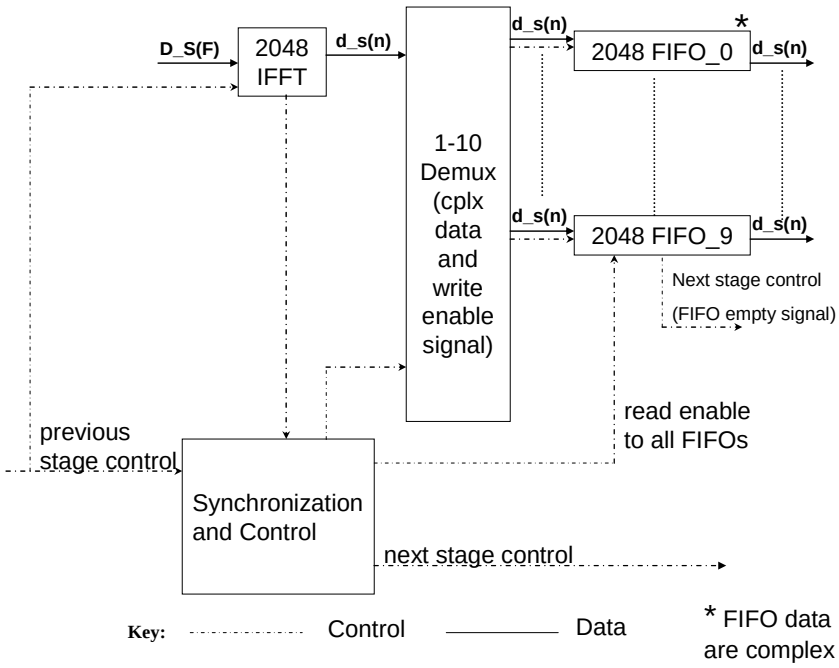

Figure 5.10 The IFFT_2048 Domain

5.3.1 Overview of Operations the IFFT_2048 Domain

As shown in Figure 5.10 this IFFT 2,048 domain serves to buffer the 2048 IFFT output samples d_s(n) in a series of 2,048 FIFOs and then feed the data in parallel to the 10 point DFT in the next domain. As referred to in the step 10 of the algorithm, there are a total of 10 frames of 2,500 samples for each downconversion frequency used. These data can be represented by a 10*2,048 d_s(n) matrix. This matrix is then accessed one column at a time and a 10-point DFT is performed on each column.

5.3.2 Details of Datapaths in the IFFT_2048 Domain

5.3.2.1 IFFT_2048

This is a Xilinx IP core very similar to the pipelined streaming **FFT_4096** core used in the previous domain. The main difference is that this one is scaled. The input that controls the amount of scaling is a 12-bit bus called *ifft_scale_sch*. There are 11 butterfly stages in a 2048 FFT, and for pipelined streaming architecture, the scaling schedule is specified with two bits for every pair of radix-2 stages. For example, a scaling schedule of N = 2048 could be [2 2 2 3 2 1]. [31] The last stage just has one bit for scaling because 2,048 is not a radix-4 number. Another difference between the 4,096 FFT and the 2,048 IFFT is that the scaled IFFT has the same number of bits for its *xn* inputs and *xk* outputs. Because of this limited number of bits, the latter can overflow if not scaled properly. There is therefore an *ifft_ovflo* flag that alerts the user that an overflow occurred during calculation of the IFFT. The user could then decide what to do with the output *xk_re* and *xk_im* samples.

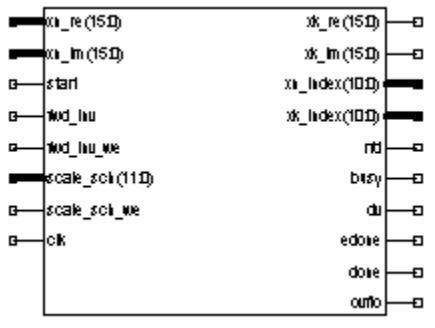

Figure 5.11 Schematic of IFFT_2048 module

The engineer therefore has to be judicious in picking a scaling schedule. If there is not enough scaling, the core can overflow. If there is too much scaling, the *xk* outputs may lose their numerical relation, making the output almost meaningless. MATLAB or C simulations are one reliable way to test scaling or truncation schedules before they are implemented in hardware. The MATLAB simulations show that after the generation of D_S(F), the data path is less sensitive to scaling and truncation, allowing the user to scale fairly aggressively towards the back end of the system.

5.3.2.2 FIFO_2048

At this point, a few words about FIFOs might be in order. The following is taken from a digital design text [33]: "A FIFO (First In First Out) is a type of memory that is commonly used to buffer data that is being transferred between different systems of different parts of a system, which are operating at different speeds or with different delays. The FIFO allows the transmitter to send data while the receiver is not ready. The data then fills up the FIFO memory until the receiver begins unloading it"

This particular FIFO is a Xilinx IP core called Asynchronous FIFO v6.1. As shown in figure 5.12, it has two independent clock domains, *wr_clk* for writing and

rd_clk for reading. There are a number of outputs also. The ones relative to this design are the *full* and the *empty* flags. The *full* signal is used to signal to the control to disable writing to the FIFO and the *empty* flag is used to signal to the control to stop reading from the FIFO. In this thesis design there are 10 of these FIFOs (from 0 to 9), each of depth 2,048. Each FIFO collects the 2,048 d_s(n) outputs for each 2,048 IFFT. Once all FIFOs are full, the FIFO data are read out in parallel and fed to the next domain.

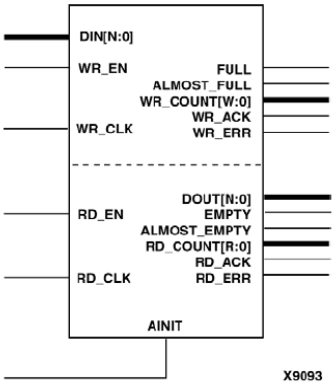

Figure 1: Core Schematic Symbol

Figure 5.12 FIFO 2,048 IP Core

5.3.2.3 DEMUX 1_10

This demux serves to route of d_s(n) data and the *wr_en* signal to the appropriate FIFO This FIFO has a 4-bit *demux_select* input and has a variable data path width based on the amount of truncation post IFFT. Although the demux, can theoretically handle 16 different routes, only outputs 0 to 9 are used, so during synthesis the remaining 6 outputs and their associated logic would not be implemented, leading to a conservation of FPGA resources.. Since the *wr_en* is routed to the same FIFO as the d_s(n) data, there is no need to have a separate 1 to 10 FIFO for *wr_en and* d_s(n)

5.3.3 Controllers in the IFFT_2048 Domain

5.3.3.1 IFFT_OP_CONT (IOC)

IFFT_OP_CONT (IOC) is the only control for this domain. Its relevant inputs are *ifft_done, ifft_dv* and the relevant outputs are *we_ifft_fifo, fifo_writes_done* and a 4-bit output *ifft_demux_cont*. The *ifft_done* input, which is actually an output from the IFFFT 2,048, triggers this controller to route the d_s(n) output and the *we_ifft_fifo* signal, a write enable signal, to the appropriate FIFO via the demux. The *we_ifft_fifo* signal is kept high as long as the *ifft_dv* input from the IFFT is high, ensuring that all d_s(n) samples are written into the FIFO. The routing is controlled by the *ifft_demux_cont* value. Once the *ifft_dv* flag goes low, the IOC changes state and increments *ifft_demux_cont.* **IOC** then "waits" until ifft_done goes high again and then it regulates the writing of the next 2,048 samples of d_s(n) to the next FIFO in the bank. This process is repeated until the 10th state is reached, at which point the *fifo_writes_done* flag goes high, indicating to the next domain that the 10 point DFTs can now start. IOC then goes back to its init state and "waits" for the next set of IFFT outputs from the next downconversion frequency.

5.3.4 Timing Issues Post IFFT_2048

After the last fifo_2048 has been completely written to, at the front end, the initial buffer has already gotten nearly 40,000 samples read from it for the next downconversion frequency. An analysis of the time needed for complete sorting reveals that there is already a 90 μs delay between completing a downconversion frequency and beginning a new data stream to the FFT. There is also the built-in latency of 4,096 clocks for the FFT input, 4,096 clocks for the output, and timing analysis and simulation, it is concluded that

there is sufficient time for the sorting algorithm to complete before new downconversion data is written to the FIFOs. Figure 5.13 shows that the sorting in complete before the 5th FIFO begins to take data from the 5th 4096 FFT.

Simulation showing that sorting is complete before next frequency is completely written in.

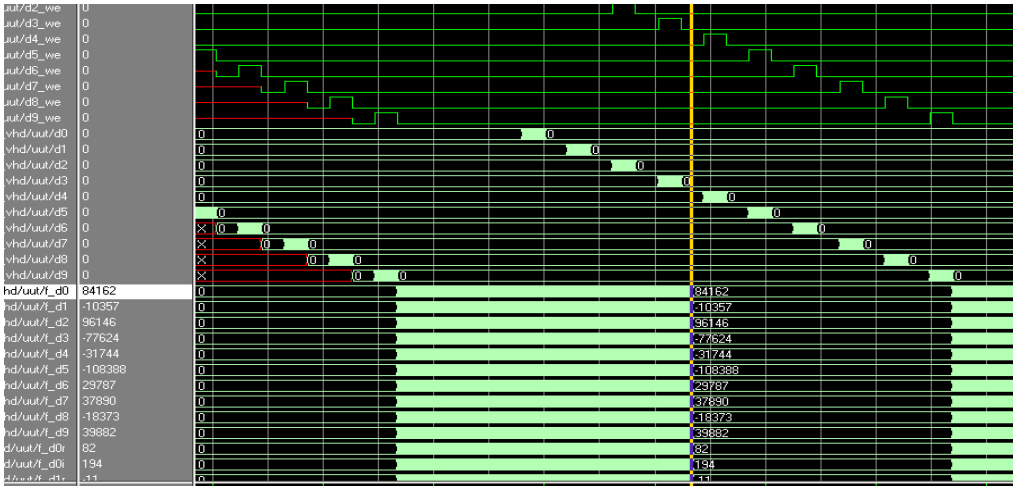

Figure 5.13 Simulation showing that FIFO collisions are avoided.

5.4 The 10 Point DFT/ Sorting Domain

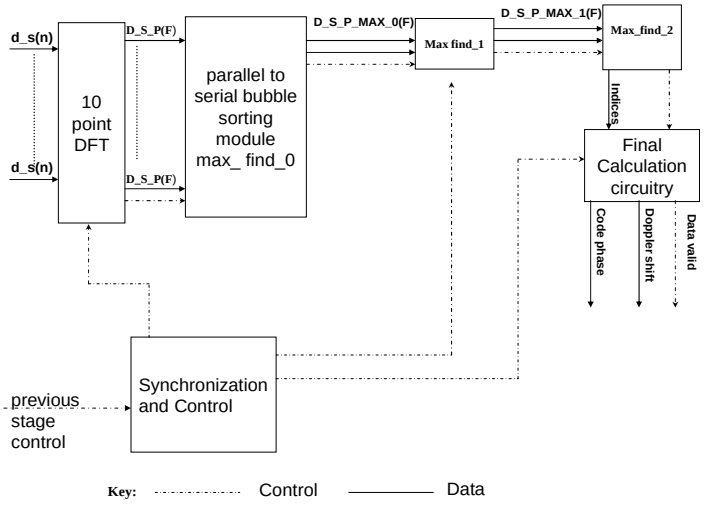

Figure 5.14 Block Diagram of 10 Point DFT/Sorting Domain

5.4.1 Overview of Operations in the 10 Point DFT/ Sorting Domain.

Once all the FIFOs are full, as signaled by the *fifo_writes_done* signal from the previous domain, this domain then proceeds to calculate a series of 10 point DFTs on the synchronized parallel outputs of the FIFOs. A total of 2,048 ten-point DFTs are calculated for each downconversion frequency. The result of these 10-point DFTs is sorted to obtain the global maximum. The control circuitry of this domain ensures that: 1) the 10 point DFT starts after the FIFOs from the previous stage are filled and 2) the bubble sorters are flushed after each batch of data

5.4.2 Details of Datapaths in The 10-Point DFT/Sorting Domain

5.4.2.1 DFT_10

This module has 10 variable width inputs named *in0r, in1r , in2r, in3r ,in4r, in5r, in6r, in7r, in8r,* and *in9r;* it also has 10 variable width imaginary inputs named *in0i ,in1i ,in2i ,in3i ,in4i, in5i, in6i ,in7i, in8i*, and *in9i*. The outputs are: *out0r, out1r, out2r, out3r, out4r, out5r, out6r, out7r, out8r, out9r, out0i, out1i, out2i, out3i, out4i, out5i, out6i, out7i, out8i*, and *out9i.* The DFT is enabled via an *enable* input signal. The 10 point DFT module also has a *data_valid* flag that goes high for one clock to indicate that the data on the outputs is valid. The *data_valid* flag goes low until the next 10-point DFT is enabled and subsequently calculated. This parallel/pipelined architecture allows rapid calculation of the 10-point DFT, greatly speeding up the calculations relative to a purely serial implementation. The price paid for such a rapid implementation is increased arithmetic hardware. Below is the synthesis report for DFT_10, illustrating the resources used by this module. As can be seen from the table, this design is very multiplier intensive.

Table 5.1 Synthesis Results of FFT_10 showing FPGA resources utilized

FPGA Resources	# in design	Total # on FPGA	% of total
Number of Slices	5,498	13,696	40
Number of Slice Flip Flops	1,009	27,392	4
Number of 4 input LUTs	10,127	27,392	37
Number of MULT18X18	112	644	83

The 10 point DFT module carries out a discrete Fourier transform in 3 clocks. It accepts 10 real and 10 imaginary inputs in parallel and after 3 clocks, presents the valid 10 point transform (10 real and 10 imaginary numbers) in parallel at its outputs. The module performs a total of 2,048 10-point DFTs for each 50,000 input unit.

A 10-point DFT is very uncommon, since most DFTs are implemented as FFTs and are thus commonly powers of 2. This DFT was hand-coded specifically for this circuit. The Divide and Conquer DFT algorithm outlined earlier in the paper was functionally decomposed into three pipelined modules and implemented in VHDL. This design choice leads to the 9 clock latency. The simulation results for a sample run of the DFT_10 module are shown in figure 5.15.

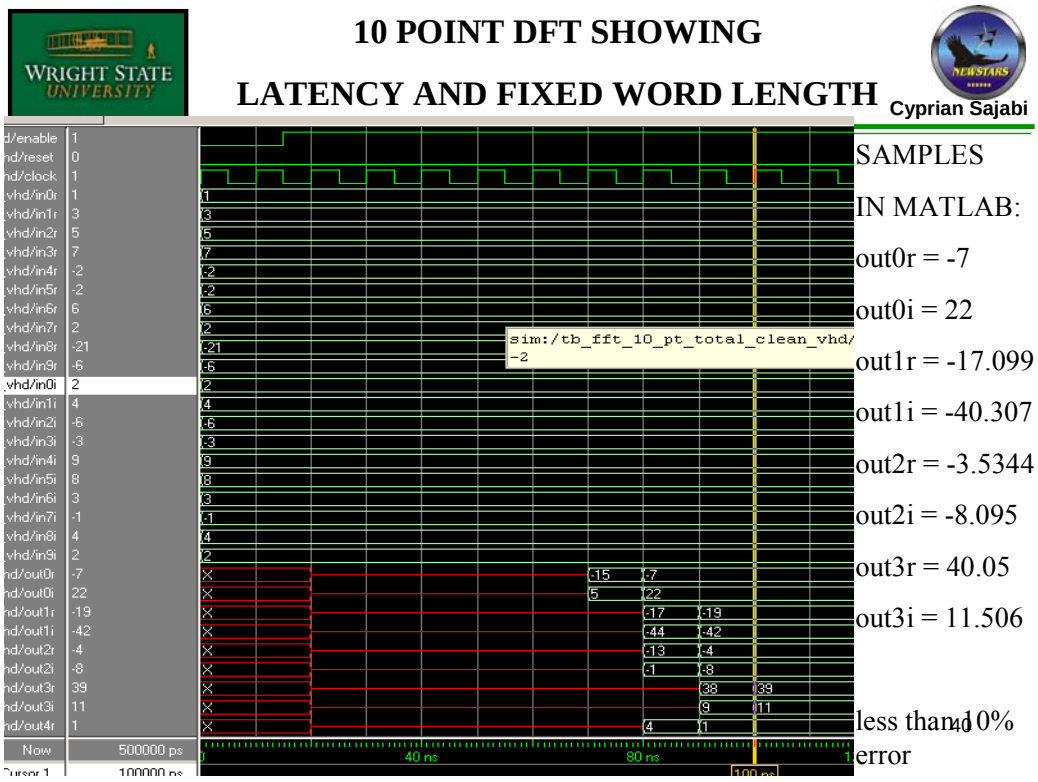

Figure 5.15 DFT_10 Module Sample Simulation

5.4.2.2 Max_find_0

This module has an *enable* input and 10 N-width real and imaginary inputs named *din0r, din1r, din2r, din3r, din4r, din5r, din6r, din7r, din8r, din9r, din0i, din1i, din2i, din3i, din4i, din5i, din6i, din7i, din8i,* and *din9i.* The outputs are a flag named *max_0_done*, a four-bit vector named *max_0_index*, and a variable-width vector named *max_0*. The width of *max_0* vector is 2*N + 1 because it represents the sum of the squares of the real and imaginary inputs.

The **max_find_0** searches the outputs of the 10 point DFT to find the maximum absolute value of each set of ten and presents this value (*max_0*) along with an associated row index (*max_0_index)*, ranging from 1 to 10 to the max_find_1 module. For each down conversion frequency, the **max_find_0** carries out a total of 2,048 bubble sorts to

give a total of 2,048 maxima. Within max_find_0 are a parallel to serial converter, a serial bubble sorter and a control module. Max_find_0 requires 10 clocks to perform its function. At this point the data have been converted to the unsigned 2*N +1 -bit numbers.

5.4.2.3 Max_find_1

Max_find_1 serially sorts through the 2,048 outputs of **max_find_0** to obtain a local maximum for a given down conversion frequency. The inputs to max_find_1 are: *enable*, *d_in*, a 2*N +1-bit input, and a 4-bit input *row_index_in*, connected directly to the *max_0_index* output of the max_find_0 module. The outputs are a 2*N+1-bit output *max_out,* a 4-bit *max_out_row_indx* and an 11-bit *max_out_col_indx* vector. Max_find_1 serially sorts through the d_in inputs by comparing the new *d_in* value with a current maximum. The *enable* signal going high causes the module to perform a comparison of the absolute value of its inputs with it currently stored maximum. If the new value is greater than the current maximum, it swaps them and makes the new value the current maximum. In addition, **max_find_1** clocks in the value of *row_index_in* that is associated with the new current maximum and presents this value on its output as *max_out_row_indx.* Max_find_1 generates an additional column index (*max_out_col_indx*) ranging from 1 to 2,048 to accompany the row index from max_find_0. There is now a single unsigned 2*N +1 -bit number, a row index, and a column index stored as a result of processing 50,000 points of input data. Max_find_1 continues this process for all 10 downconversion frequencies.

5.4.2.4 Max_find_2

Max_find_2 serially sorts through the 10 numbers from Max_find_1 to find a global maximum, associating a third index, a stack index, with the final maximum. Max_find_2 has an *enable*, a 2*N+1-bit input named d_in, a 4-bit input called row_index_in, and an 11-bit input called *col_index_in.* The outputs are max*_out_2_row_indx* (4-bits), *max_out_2_col_indx*(11 bits), *max_out_2_stack_indx* (4-bits), and *max_out_2* (2N+1-bits). As with max_find_1, when this module is enabled, it performs a comparison of the value on its *d_in* input with the value stored in its register. If the new value is greater than the stored value, it performs a swap of information, presenting the associated column, row, and stack index of the new maximum. Otherwise the old values remain on the outputs as the current maximum. Since there is no more sorting to be done, the current maximum is not needed as an output, but is needed internally for comparisons.

5.4.2.5 Final Calculation Circuitry

The Final Calculation Circuitry takes the three indices as inputs and via some simple calculations decodes them to obtain the carrier frequency and the relative phase of the C/A code of the incoming code. These data are continually fed into an Early Prompt Late module that keeps the relative phase of code adjusted and keeps the down conversion frequency as close to the Doppler shifted frequency as possible within the resolution limits of the system. There are two functional modules within the final calculation circuitry: Final_decision_module_0 (FDM_0) and Final_decision_module_1(FDM_1).

5.4.2.5.2 FDM_0

This module is used to calculate the Doppler shift of the carrier. The relevant inputs are the *max_find_1_done* flag (taken from max_find_1), and the 4-bit inputs *row_index_in* and *stack_index_in* (taken from max_find_2).

The outputs are the *data_valid* flag and the 14-bit output *freq*. Each time a down conversion frequency's outputs are completely sorted, the *max_find_1_done* flag pulses. The **FDM_0** module keeps a count of how many times *max_find_1_done* pulses, When the count reaches 10, it means that sorting of all downconversion frequencies is complete, and the final calculations can now proceed. The FDM_0 module performs some simple signed integer arithmetic using *row_index_in* and *stack_index_in* to calculate *freq.* The value of freq has a possible range of ± 5000 Hz, so it needs 14 bits to represent the possible values it can take on. Simulations show that the entire acquisition cycle requires about 5.5 ms at 100 MHz.

5.4.2.5.2 FDM_1

This module is used to calculate the code phase. The relevant inputs are the *max_find_1_done* flag (taken from max_find_1), and the 11-bit input *col_index_in* and *stack_index_in* (taken from max_find_2).

The outputs are the *data_valid* flag and the 13-bit output *ini_ca*. The FDM_1 module calculates its numerical output value at the same time the FDM_0 module calculates its numerical output value. The FDM_1 module performs some simple unsigned integer arithmetic using *col_index_in* to calculate *ini_ca.* The value of *ini_ca* has

a possible range of ± 4096 samples, so it needs 13 bits to represent the possible values it can take on.

5.4.3 Controllers in the DFT_10/Sorting Domain

5.4.3.1 Max_find_1 Control

This controller is responsible for a number of functions in this domain:

1. It enables max_find_1 for one clock after each column is sorted by max_find_0.
2. It controls the resetting of max_find_1 after each downconversion frequency is completed.
3. It signals to the final calculation circuitry when each downconversion frequency is completely sorted, allowing them to keep a count of how many frequencies are completed.

Max_find_1 control has two control inputs named *mx_0_done* and *fifo_2048_empty*. It has three control outputs called *max_find_1_done, max_find_1_enable,* and *max_find_1_reset.*

When the *mx_0_done* input goes high, the *max_find_1_enable* output goes high for one clock and triggers the max_find_1 module to perform a comparison operation. This pattern continues until *fifo_2048_empty* goes high, at which point the max_find_1 controller resets max_find_1 via the *max_find_1_reset* signal. In this state the max_find_1 controller also pulses *max_find_1_done* for one clock, signaling to the final calculation circuitry that a downconversion frequency is completed. The max_find_1_control then goes back into its init state and waits for another pulse from *mx_0_done*.

5.4.3.1.2 Max_find_2_control

This module serves to reset the max_find_2 module after the *data_valid* flag goes high on max_find_2. It creates a delay of 4 clocks after the *data_valid* signal goes high so any downstream elements that require the phase and frequency data can have free access to it within the 4 clock window. This time is arbitrary and may be increased or decreased as needed. Max_find_2_control has one functional input called *data_valid*, and an output called *max_find_2_reset*. When *data_valid* goes high, this control module goes through a series of 4 Moore type state transitions, the last of which causes the *max_find_reset* signal to go high. Subsequently, the controller goes into the init state and "waits" for the next *data_valid* assertion.

5.5 Simulation results of the Processor

Figure 5.16 shows the simulation results of the entire acquisition process. The highlighted signals are the signals of interest. A comparison with MATLAB shows a match for both *ini_ca* and *freq*. As can be seen, the *data_valid* signal goes high when the acquisition results are ready. This *data_valid* signal will be used by downstream tracking elements to capture the acquisition results and use them in tracking the signal. The results for the code phase, *ini_ca* are in unsigned format, whereas the results for the Doppler shift, *freq* are in 2's complement format. These results are ready within 5.6 ms of the frontside buffer being filled up.

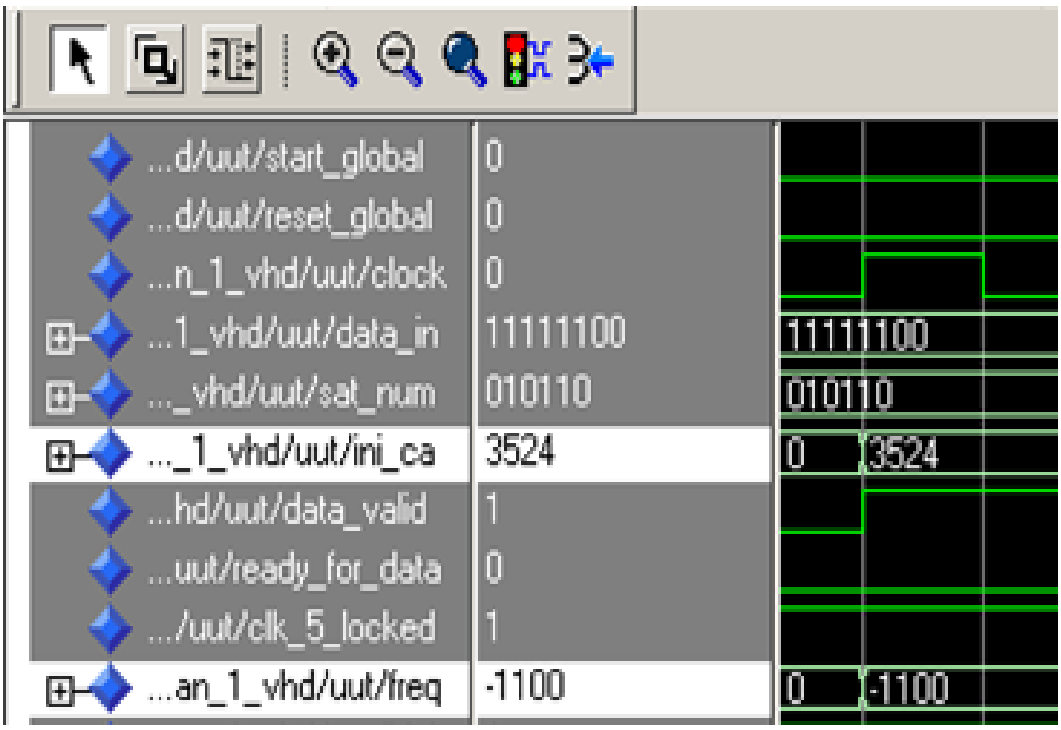

Figure 5.16 Behavioral simulation showing successful emulation of MATLAB code.

6 Finite Word-length Considerations.

6.1 Growth of the Bits in the Datapath Due to Addition and Multiplication

Discrete time systems are implemented in hardware or software, but in either case, finite-word-length has to be taken into consideration. Even with the flexibility of software based systems, there are limits on the size of numbers they can handle. The system used in this thesis is a hardware fixed-point system, and finite-word-length is a major consideration in the implementation.

In this section, we consider what happens when arithmetic takes place in a fixed-point system. An example will illustrate some challenges faced by digital designers using a fixed point system. We will assume that there is a basic understanding of unsigned binary mathematics for this example:

Let's say we want to add two 4-bit unsigned numbers. If the numbers are small enough, the result can also be represented by 4-bits, and the datapath can keep its original width. For example $(6 + 4)_{10} = (0110 + 0100)_2 = 1010_2$. But the following sum cannot be represented by the 4-bit datapath, and an extra bit needs to be added: $(8 + 9)_{10} = (1000 + 1001)_2 = 10001_2$. Because of this possibility, the datapath needs to be expanded by one bit as we add numbers and move the data "downstream". In fact, the maximum expansion for addition datapaths is one bit.

Let us look at the case with multiplication: As an example $(6 * 4)_{10} = (0110 * 0100)_2 = 11000_2$. The following sum cannot be represented by the 4-bit datapath, and an extra bit needs to be added. In the boundary case of: $(15 * 15)_{10} = (1111 * 1111)_2 = 11100001_2$., the required data path actually doubles in size. Because of this possibility, the data path needs to be doubled in size as we multiply numbers and move the data

"downstream". In fact, the expansion for multiplication datapaths is bounded by the sum of the datawidths of the inputs.

It can then be seen that if we have a very long signal processing algorithm, the data-path will keep growing without bounds. This is sometime referred to as data-path "explosion". There are a number of measures, such as rounding and truncation, that can be taken to counteract this effect. In this thesis, truncation was used because of its relative simplicity.

6.2 Effects of Truncation on the Datapath.

The following discussion will illuminate the topic of truncation and its effect on the datapath. We usually need to quantize a number from a given level of precision to a lower level in order to keep the datapath from "exploding". Truncation introduces an error whose value depends on the number of bits in the original number versus the number of bits after truncation. In the two's-complement representation, the negative of a number is obtained by subtracting the corresponding positive number from 2. Let us consider a fixed-point representation in which a two's complement number *x* is quantized from b_u bits to b bits. Thus the number

$$x = \overbrace{0.1011...1}^{b_u}$$

represented with b_u bits prior to quantization is represented as

$$x = \overbrace{0.1011...1}^{b}$$

after quantization, where $b < b_u$. If *x* represents a sample of an analog signal, the b_u may be taken as infinite [34]. The value of x would then be truncated, and the truncation error is defined as

$$\mathbf{E_t = Q_t\,(x) - x} \qquad \mathbf{(6.1)}$$

The effect of truncation by one bit on a two's complement number is to decrease the magnitude of the number by a factor of 2 for radix 2 numbers. For non radix-2 numbers, truncation by one bit leaves a result that is equivalent of dividing by 2 and rounding down to the nearest integer. For negative numbers, the result depends on the numbers used, but is bounded by half the magnitude of the number ± 1. The following examples illustrate a few cases of truncation of 6-bit two's complement numbers by one bit:

Table 6.1 Examples of Truncation Error for positive and negative numbers

Decimal (*x*)	Two's complement	Truncated by one bit	New Decimal $Q_t\,(x)$	Truncation Error $Q_t\,(x) - x$
20	010100	01010	10	-10
19	010011	01001	9	-10
-20	101100	10110	-9	11
-19	101101	10110	-9	10

The above table suggests that truncation error for two's complement representation is essentially symmetric about zero, which is desirable to minimize D.C bias in the results. Another important point about truncation is that if the initial numbers

are very small, then truncation can lead to results that are ambiguous. Truncation is acceptable if it preserves the ratios of the signal values within the noise limits of the system in question. One can therefore test the effects of truncation by simulation before implementing it in hardware.

6.3 Resizing without changing value.

In some cases, it is possible to change the size of the datapath without changing the signal values. This happens when there are many more bits than are needed to represent the possible range of numbers on a bus. For example, an 11-bit bus can represent numbers from -1024 up to 1023 in two's complement. However, if it is known *apriori*, that the range of numbers on a bus will be ± 100 , then an 8-bit bus is sufficient to represent the numbers with full precision. In this case, another method of reducing the bus width, called **resizing** can be used, and we can still keep the complete accuracy of the numbers. A VHDL function called **std_resize** was written to carry out bus resizing to a smaller size. Figure 6.1 illustrates the action of **std_resize.**

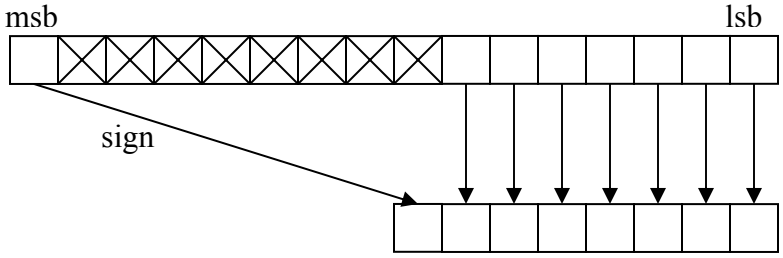

Figure 6.1 Action of std_resize to smaller size

6.4 Truncation and resizing schedule for C/A processor

Table 6.2 illustrates a possible truncation and resizing schedule for the C/A code processor based on MATLAB simulations.

Table 6.2 Theoretical versus implemented truncation and resizing schedule

Signal	Max Val/Bits	Theoretical Truncated Range/Bits	Implemented Truncated Range/Bits	Name In Algorithm (If Applicable)
primary input bits	+-84/8	+-42/7	+-42/7	y(n)
bits from cplx exp	+- 256/9	+-64/7	+- 256/9	c_e(n)
bits into 1st cplx mult	9 and 8	7 and 7 bits	7 and 9 bits	y(n)*c_e(n)
bits out of of 1st cplx mult	+-10752/16	+-2624/14	+-5248/15	d_c(n)
bits of C/A FFT	+-280/10	+-140/9	+-280/10	C_A(F)
bits to 2048 ram	+-280/10	+-140/9	+-280/10	C_A(F)
40,960 ram bits	+-10,752/16	+-2,624/14	+-5,248/15	d_c(n)
bits to 4,096 FFT	+-10,752/16	+-2,624/14	+-5,248/15	d_c(n)
bits out of 4,096 FFT	+-852,910/21	+-30/6	+-200/9	D_C(F)
demux bits	+-10,752/21	+-2624/14	+-200/9	D_C(F) or C_A(F)
bits into 2nd cplx mult	10 and 21	6 and 9	10 and 9 bits	C_A(F) * D_C(F)
bits out of 2nd cplx multiplier	+- 113,986,498/28	+-882/12	+-32,000/17	D_S(F)
Bits out of 2,048 IFFT	+- 1024901/21	+-7/4	+-16/10	d_s(n)
bits of 2,048 FIFO	+- 1024901/21	+-7/4	+-16/10	d_s(n)
bits to 10 pt FFT	+- 1024901/21	+-7/4	+-16/10	d_s(n)
bits from 10-pt FFT	+- 9377207/24	+-9/5	+-22/30	D_S_P(F)
bits of sorting algorithm	+- 1024901/21	+162/8 (only positive)	+400/25 (only positive)	D_S_P_MAX(F)

Implemented truncation and resizing detailed summary:

- **y(n) 7-bits (*truncated* from 8 to 7 bits)**
- **c_e(n) 9-bits(no truncation)**
- **y(n) 7-bits * c_e(n) 9-bit inputs to multiplier**
- **16 bit multiplier output d_c(n) (*truncated* to 15 bits)**
- **15-bit 40,960 ram stores d_c(n)**
- **15-bit FFT_4096 input d_c(n)**
- **28-bit FFT output for either D_C(F) or C_A(F)**
- **28-bit FFT *resized* to 10 bits for C_A(F)**
- **28 bit FFToutput *truncated* to 17 bits for tmp1f, then *resized* to 9 bits for D_C(F)**
- **9-bit 2,048 RAM**
- **9-bit D_C(F) * 10-bit C_A(F) input to complex conjugate multiplier**
- **multiplier output D_S(F) of 21 bits *truncated* to 17 bits**
- **17-bit ifft output d_s(n) *resized* to 10 bits for 10-pt FFT**
- **10(*2)-bit 2048 fifo for d_s(n)**
- **10-bit 10-point FFT input d_s(n)**
- **30-bit fft_10 output D_S_P(F) *truncated* to 12 bits**
- **25-bit comparison circuits to find D_S_P_MAX(F)**

The truncation schedules given above are by no means final. They are simply one of many possible configurations that gave the required accuracy for this algorithm.

7 HARDWARE REQUIREMENTS FROM XILINX REPORTS

From the Xilinx XST Synthesis Reports, the table below shows the hardware requirements for the major components in the processor. The selected device is the Virtex-II Pro type 2vp70ff1152-6. Below is a summary of the resources available on this FPGA and the percentage used by the processor:

Table 7.1 Overall Virtex-II Pro 70 FPGA Resource Usage of Processor

Resource	**Total**	**In design**	**Percentage usage**
Slices	33,088	7,149	21%
Slice Registers	66,176	11,915	16%
4 input LUTs	66,176	12,233	18 %
Bonded IOBs	644	35	3%
BRAMs:	328	50	48%
GCLKs	16	6	37%
MULT18X18	328	160	49%

After Synthesis, the maximum clock rate of the design was 140.174 MHz. Resource sharing was enabled during the synthesis run. This resource sharing for some arithmetic operations slowed down the maximum clock frequency, but allowed reduced device utilization. For improved clock frequency resource sharing could be disabled. After Place and Route, the maximum clock frequency was 105 MHz. As seen in table 7.2, the 10-point DFT consumes the most multiplier resources at 112, whereas **Dual-Port RAM1** consumes the most memory resources. The **IFFT_2048** and **FFT_4096** consume the most slices, LUTs and flip-flops.

Table 7.2 Analysis of Individual Synthesis Results of Major Components

Component	Number of Slices	Slice flip flops	4-input LUTs	Block RAMs	MULT18X18
FIFO_2048 (written in VHDL)	45	36	83	3	0
IP core generated FIFO_4095	45	88	106	5	0
DFT_10	2,976	3,439	5,245	0	112
Dual Port RAM 0	27	0	0	49	0
Dual Port RAM 1	21	0	0	96	0
FFT_4096	5,227	7,703	6,977	39	27
IFFT_2048	4,477	7,736	6,286	17	40
DDS	43	57	61	0	0

These figures give some ideas of areas for further size optimization. A word of caution: the synthesis results cannot be totaled up to give the complete synthesis figures because many each slice may "donate" resources to more than one module in the overall design. For example, a slice may "donate" a register to one entity and a 4-input LUT to another entity. This is one reason why the total slice count from the synthesis of the whole design in table 7.1 is less than what would be obtained by summing up the slice count from the individual synthesis results in table 7.2. Another reason the slice count changes is that unused outputs are "ripped" out by the translation and mapping algorithms, leading to a greatly reduced design in terms of resource usage.

8. COMPARISONS WITH CURRENT DESIGNS

Below is table 8.1 showing a comparison with some current designs in terms of Signal-to-noise ratio and acquisition times. It is difficult to make meaningful comparisons among all systems because they may use different technology, units of measure and testing conditions. To the extent possible, a comparison between the different systems and the one implemented in this paper has been attempted. As it can be seen, the acquisition times range from 5.6 ms on the AFRL design to 2 seconds on the Trimble Resolution T Receiver. Sensitivity ranges from -140dBm with the Teletype GPS CF v3.0 Receiver to 40 dB in work done by P. Rinder and N Bertelsen. Two caveats are in order here: firstly, the term "acquisition" may carry different meanings to different systems and secondly that comparing hardware and software approaches is often fraught with risk. At any rate, the AFRL GPS processor seems to be the fastest in acquisition time and in the top three in terms of sensitivity, making it an attractive design. The FPGA implementation gives a flexibility not given by an ASIC and approaching that of a general purpose processor. Another point to note is that the AFRL processor also is well positioned for future work with P and M code because it is designed for 10 ms data records.

Table 8.1 Comparisons with some Current GPS Receivers

COMPARISONS WITH SOME CURRENT GPS RECEIVERS

Cyprian Sajabi

Reference/ Product	Sensitivity	Acquisition time	Data Length	Approach
TeleType GPS CF v3.0 Receiver	-140 dBm	100 ms	N/A	ASIC
Trimble Resolution T Receiver	-136 dBm	2,000 ms	N/A	DSP chip
High-Performance GPS (HPGPS), STMicroelectronics	~40 dB outdoors -125 dBm indoors	1 ms outdoors 100 ms indoors	1 ms	DSP chip
Van Diggelen , F. Abraham, C. GPSWorld,	~40 dB outdoors -125 dBm indoors	~5 seconds indoors 100 ms outdoors	1ms	Massively Parallel Correlator Bank
Rinder, P. Bertelsen, N. 2004	~40dB(outdoors)	100 ms	1 ms	Software On a Laptop
AFRL 2006	-135 dBm	5.6 ms	10 ms	FPGA [47]

9. CONCLUSIONS AND FUTURE WORK

9.1 Conclusions

Acquisition is the most important step to a GPS receiver because one must lock onto the C/A code in order to despread the GPS signal. The acquisition generates two important parameters: the carrier frequency and the initial phase of the C/A code. In general, acquisition performed on long data will increase the receiver sensitivity. This thesis describes the implementation of a weak signal C/A code acquisition circuit that uses 10 ms of data to perform a frequency domain based operation to realize its function. The novel "subsampling" technique demonstrated in this paper is an alternative to averaging correlation as a way to map an ideal software approach acquisition to hardware.

In this thesis, a C/A code processor was designed and simulated successfully. It is found to work at the required 100 MHz clock frequency and acquires the C/A code and Doppler shift in 5.5 ms. Truncation and data-path resizing were effectively used to keep the datapath from "exploding". The design was synthesized and projected to fit onto the Virtex-II pro 70 FPGA.

9.2 Future Work

The limitations of this system lie primarily in the massive hardware requirements. In synthesis, the processor maps onto the Virtex-II pro 70 FPGA. The algorithm itself is highly efficient relative to the purely time-domain approach. The places where hardware could be minimized are the FFT modules (especially FFT_10) ,and the FIFOs, because the design uses 10 FIFOs. **IFFT_2048** and **FFT_4096** are Xilinx IP core "black boxes", so the only changes that can be made here is to change the architecture to Burst mode,

which will cut down on the resource usage. Such architecture changes in FFT_4096 and IFFT_2048 are not trivial, because they would necessitate further changes in the overall timing and scheduling of the processor.

In addition, major re-engineering would need to be carried out to have a design that does not use **Dual Port RAM 1**, but if this could be done, the design would use about 96 less block rams and could conceivably fit in a Virtex-II pro 30 part. The use of shift and add architecture rather than dedicated multipliers would reduce the number of MULT18x18 used and allow the design to fit on the Virtex-II pro 30 part. The amount of truncation and resizing, especially early on in the algorithm, will determine the datapath width of the size of the downstream components. Further studies need to be carried out to determine the optimal truncation schedule for this particular architecture, while maintaining the -20 dB signal-to-noise ratio.

APPENDIX

A.1 MATLAB CODE FOR C/A PROCESSOR BLOCKS

The MATLAB code for the processor is broken up into a number of functions. This appendix contains the basic blocks and their basic functional description within the design. All code was provided courtesy of Air Force Research Lab(AFRL)

A.1.1 C/A Code Generation

This function, called "cacode" is responsible for generation of the reference C/A code, which is the Pseudo Noise (PN) sequence that will be used as a template to match the incoming signal. The function takes one parameter, the satellite number, and generates a unique PN sequence for each satellite number input. This PN sequence is a series of -1 and 1 in MATLAB that map onto 1 and 0 in hardware. The code is generated at 1.023 MHz in hardware, corresponding to one sample per generator clock. The MATLAB function is given below:

```
function [ss_ca]=cacode_1(svnum);
% modified by JT May 8 02 to generate satellite ca this program contains WASP
%function ca=cacode2(svnum,fs,numsamp);
%  function to generate any of the 32 GPS C/A codes at a user
% specified sampling frequency.
% Input Arguments:
% svnum - the Satellite's PRN number
%        1-32 is traditional GPS PRN numbers
%        38 is PRN for WAAS INMARSAT AOR-E (refer to WAAS MOPS for shift)
%        39 is PRN for WAAS INMARSAT AOR-W
%        40 is PRN for WAAS INMARSAT Reserved
%        41 is PRN for WAAS INMARSAT IOR
%        42 is PRN for WAAS INMARSAT POR
%
% fs - desired sampling frequency
% numsamp - number of samples to generate
%          (sequence can extend for longer than a single code period (1 ms)
% Output Argument
% ca - a vector containing the desired output sequence
% D. Akos - WPAFB AAWP-1
%this start the sequence one sample into the first chip (not at leading edge of first chip)!
%dma comment 17-5-00
%this has been fixed as of 9-Dec-2000 but not fully completely tested, but the first sample in the code should
correspond to the first chip now!!!  Also added the ability to find the GEO for WAAS by using PRN numbers above 32
%dma comment 9-Dec-2000
% Constants
%svnum=26;
```

```
coderate=1.023e6;  %no consider given to the neglible Doppler effect on code

% the g2s vector holds the appropriate shift of the g2 code to generate
% the C/A code (ex. for SV#19 - use a G2 shift of g2shift(19,1)=471)
g2s = [5;6;7;8;17;18;139;140;141;251;252;254;255;256;257;258;469;470;471; ...
     472;473;474;509;512;513;514;515;516;859;860;861;862;863;950;947;948;950; ...
     145;52;886;1012;130];
g2shift=g2s(svnum,1);
% Generate G1 code
    %   load shift register
        reg = -1*ones(1,10);%-1 corresponds to 1
    for i = 1:1023,
       g1(i) = reg(10);
       save1 = reg(3)*reg(10);%xor
       reg(1,2:10) = reg(1:1:9);
       reg(1) = save1;
    end,
 % Generate G2 code
    %   load shift register
         reg = -1*ones(1,10); %-1 corresponds to 1
    for i = 1:1023 ; %start in state 1 and add 1023 to end up in state 1
       g2(i) = reg(10) ;% first is state 0
       g2_rows(i,:) = reg;% addded by cyprian sajabi
       save2 = reg(2)*reg(3)*reg(6)*reg(8)*reg(9)*reg(10);%xors
       reg(1,2:10) = reg(1:1:9);
       reg(1) = save2;
    end
    %    Shift G2 code
    g2tmp(1,1:g2shift)=g2(1,1023-g2shift+1:1023);
    g2tmp(1,g2shift+1:1023)=g2(1,1:1023-g2shift);
%  Form single sample C/A code by multiplying G1 and G2
ss_ca = -g1.*g2tmp; %xnor operation
```

A.1.2 C/A Code Sampling

This function, called “digold_trk” is responsible for simulating the sampling of the reference C/A code, at a sampling frequency specified by the design. In this case, the sampling frequency is 5 Mhz. The arguments for this function are: the number of samples to generate, sampling frequency, Doppler shift, time offset, and satellite number. Digold_trk calls the “cacode” to generate the one-sample-per chip PN sequence, which is then sampled by the next lines of the function to return 5,000 PN code samples.

```
function code2 = digold_trk(n,fs,fd,offset,sat);
% code  - gold code
% n     - number of samples
% fs    - sample frequency in Hz;
% offset - delay time in ns second must be less than 1/fs can not shift left
% sat   - satellite number;


gold_rate = 1.023e6;     %gold code clock rate in Hz.
```

```
f1=1575.42e6;
fs=fs*(f1-fd)/f1;
ts=1/fs;
tc=1/gold_rate;
code_in = cacode(sat);  %generate C/A code
offset=offset*f1/(f1-fd);
b = 1:n;
if(offset<0)
tmp=(ts*b+offset);
   tmp(1)=4999*ts;
   tmp(2)=ts;
else
 offset=rem(offset,1e-3);
 tmp=(ts*b+offset);
   end
c = ceil(tmp/tc);
c=rem((c-1),1023)+1;
code2 =code_in(c);
```

A.1.3 Generation of real-world data.

The file, called “main” prompts the user to input a satellite number, the amount of offset into the data required, and the required signal to noise ratio. The file then opens and reads a data file taken from sampled and quantized real-world data. The file takes in 50,000 data samples and based on the signal-to-noise ratio entered by the user, it will add the required white noise to the signal. The “main” program then calls a function called “acq10ms”, passing to it the noisy signal, the satellite number, and the number of Doppler frequencies to be used. The MATLAB code for “main” is given below.

```
clear all; close all; clc
global fs n nn ts fc nsat x thresh thresh_flag srchbnr blknbr blksz readnbr vec;%fcode fdata;
sat=input('enter satellite number [a b c ..] = ');
intodat=input('enter initial point into data (should be mult of n) = ');
db=-input('enter S/N in db = ');
nsat=length(sat); %*** total # of satellites
fs=5e6; ts=1/fs; nn=[0:n-1]; fc=1.25e6;
offset=0;
filename ='drm10062.dat';% sat 23 Data taken from the east side of the bridge near the window
fid=fopen(filename,'r');
fseek(fid,intodat,'bof');
ferror(fid)
readnbr=50000;
 [x,readnbr] = fread(fid,readnbr,'schar');
mn=mean(x)
std=sqrt(mean((x-mn).^2))
x=x/std;
std=sqrt(mean((x-mn).^2))
mn=(10^(db/20))*std
no_frq=10;
```

```
%randn('seed',1234599753);
%x=x/mn+randn(1,length(x))';--courtesy of David Lin
x=x/mn;%no noise added
status = fclose(fid);
%[ini_ca,freq]=acq10ms_clean_dav_trunc(x,no_frq,sat)
%[ini_ca, freq]=acq10ms(x,no_frq,sat)
```

A.1.4 C/A code acquisition function.

The function called "acq10ms" is called by "main" and it in turn calls "digold_trk" to generate the sampled PN sequence. This function performs acquisition on the noisy signal from "main" using the PN template from "digold_trk" and returns the C/A code phase and Doppler frequency. There are also a number of truncation constants in this function that can be used to emulate what would happen during truncation in hardware. The MATLAB code for "acq10ms" is given below.

```
% Take 10 ms data to perform coherent acq
%function [timefrq]=acq10ms(sat,no_frq,xin);
% sat: sat number
%no_frq: no. of frq used to down convert each one separates by 1 KHz
%xin: input data 10 ms
%function [ini_ca, freq]=acq10ms_clean_dav_trunc(xin,no_frq,sat);
close all
clc
xin = x;
%truncation schedules
xin_t= 2;%input data   2 7 bits
rf_t =1;%sinusoid1          7 bits
sig1_t = 2;%2 downconverted signal1 4 bits(9 bits min),2
caf_t =8192;% 1 fft of cacode, not scaled in hardware2 (needs 9 bits)
tmp1f_t = 2048;%post fft 2048,8192 7 bits
prodf_t = 16;%16%after complex multiply before ifft16 16 bits
out_t = 1;%after ifft1 16 bits
outf_t =4;% 4after 10 point fft4
zero_pad = zeros(1,1596);
fs=5e6;
ts=1/fs;
fo=1.25e6;
%n=fs/1000;
n = 4096;%for our hardware purposes
n10=10*5000;
nn=[0:n10-1];
frq_corse=1000;
ca = digold_trk(5000,fs,0,0,sat); % ref signal no Doppler 1 ms data
ca = ca*8192;
%ca = (ca*-0.5)+ 0.5;% convert back to '1' and '0' does not work.
% ca_rnd = half(ca);
% ca_rnd = [ca_rnd zero_pad];
ca_rnd = random_5000to4096_dav(ca);
%ca_rnd = 8192*ca_rnd;% scale so FFT rounding is negligible
caf=fft(ca_rnd);
```

```
caf=(caf(1:2048));
caf = floor(caf/caf_t);%truncation of fft of c/a code
foo=fo-no_frq/2*frq_corse;
xin = floor(xin/xin_t);%truncation of primary input unavoidable
%for jj=1:no_frq; % no of 1KHz down conversion, 10 in our case
for jj=1:4; % no of 1KHz down conversion, 10 in our case
  foo=foo+1000;
            rf=exp(j*2*pi*foo*ts*nn)*256;
   rf= floor(rf/rf_t);%truncation of sinusoid
   sig1=rf.*xin'; %down converted signal
   sig1 = floor(sig1/sig1_t);%truncation of downconverted signal
  tmp1=reshape(sig1,5000,length(sig1)/5000);%break signal into 10 columns
  %tmp4 = zeros(4096,10);
  tmp4 = zeros(4096,10);
  for i = 1:10;%columnwise conversions
  tmp2 = tmp1(:,i)';   %convert to row
  %tmp2 = half(tmp2);% average to 2500 samples
  %tmp2 = [tmp2 zero_pad];
  tmp2 = random_5000to4096_dav(tmp2);%take 4096 "random" samples
  tmp4(:,i)= tmp2';%convert to column
  end
  %tmp1 =  zeros(4096,10);tmp1 = tmp4;
  tmp1 =  zeros(4096,10);tmp1 = tmp4;
tmp1f=fft(tmp1);%fft on each column with zero padding to 4096
tmp1f=floor(tmp1f/tmp1f_t);% truncation before ifft
out=[];out_pre_ifft_matrix=[];
            for ii=1:10; % ***** convolution approx 10 ms columns
      %tmp1f_t = tmp1f';%transpose
                        tmp2=tmp1f(1:2048,ii)';% take 1st half fft of each column then transpose
      prodf = (tmp2.*caf);%complex conjugate
      prodf = floor(prodf/prodf_t);
      tmp3=ifft(prodf);
      out_pre_ifft =(tmp2.*caf);
      out_pre_ifft_matrix=[out_pre_ifft_matrix;out_pre_ifft];% just for VHDL testing purposes
                        out=[out;tmp3];%from tmp3
   end
   out = floor(out/out_t);%truncation after ifft
            outf(:,:,jj)=fft(out);% fine frq ,% ten point dft
end
outf = floor(outf/outf_t);%truncation after 10 point fft
clc
[mxamp,mxcoarse]=max(max(max(outf)))%mxcoarse is the stack index
[mxamp,init_ca]=max(max(outf(:,:,mxcoarse)))
[mxamp,mxfine]=max(outf(:,init_ca,mxcoarse))%mxfine is the row index
init_ca%column index
init_ca=(2048-init_ca)*2*(5000/4096)
%init_ca = init_ca*2*(4096/5000)% correction factor
%ini_ca=(2048-init_ca)*2*(5000/4096)
% init_ca = (2048-init_ca)*2*(5000/4096)
% ini_ca = init_ca*2
if(mxfine <= 5)
   freq=(mxcoarse-no_frq/2)*1000+(mxfine-1)*100
else
  freq=((mxcoarse-no_frq/2)*1000)+((mxfine-11)*100)
end
plot(abs(outf(:,:,mxcoarse)'));
figure
plot(abs(outf(mxfine,:,mxcoarse)));
```

A.1.5 Subsampling Matlab Function.

The function called “random_5000to4096” is called by “acqu10ms” This function performs subsampling on 5,000 input samples and returns 4,096 samples. It uses a series of nested loops to extract the required samples in a manner that retains the autocorrelation properties of the C/A code. The MATLAB code for “random_5000to4096” is given below.

```
 function [tem]=random_5000to4096_dav(xin);
tem=[];
   i=1;
    for k=1:5;
for j=1:91;
   tem=[tem xin(i:i+3)]; %4 samples i = 1
   i=i+5; %skip 1 i = 7
   tem=[tem xin(i:i+4)];%5 samples i=7
   i=i+6; %skip 1, i=12
end
i=i-1;% retard by 1, i = 1001
end
   tem=[tem 0];% zero pad with one zero
```

REFERENCES

[1] Lathi, B.P. "Modern Digital and Analog Communication Systems-Third Edition." pp. 413 - 417. Oxford University Press. 1998.

[2] http://www.losangeles.af.mil/smc/pa/fact_sheets/gps_fs.htm

[3] D. Lin and J.B.Y Tsui "A Software GPS Reciver for Weak Signals," IEEE MTT-S Digest 2001

[4] http://en.wikipedia.org/wiki/Spread_spectrum

[5] Dixon, R.C. "Spread Spectrum Systems with Commercial Applications Third Edition" pp 6-12, Wiley Inter-Science, 1994.

[6] Proakis, J.G. and Manolakis, D.G, "Digital Signal Processing, Principles, Algorithms and Applications," P 443, Prentice Hall 1996.

[7] Proakis, J.G. and Manolakis, D.G, "Digital Signal Processing, Principles, Algorithms and Applications," P 449-450, Prentice Hall 1996.

[8] Proakis, J.G. and Manolakis, D.G, "Digital Signal Processing, Principles, Algorithms and Applications," P 460, Prentice Hall 1996.

[9] Xilinx Virtex-II-Pro Datasheet, page 3 http://www.xilinx.com/products/silicon_solutions/fpgas/virtex/virtex_ii_pro_fpgas/capabilities/index.htm

[10] Elbirt. A.J. & Paar. C "An FPGA Implementation and Performance Evaluation of the Serpent Block Cipher" ACM/SIGDA International Symposium on FPGAs, 33-40.

[11] Lin, D. M. and Tsui, J. B.-Y. "Acquisitions Schemes for Software GPS Reciever," Proceedings of ION GPS 98, Part 1, pp. 317-326, September 1998.

[12] Lin, D.M., Tsui, J.B.Y. "A Software GPS Receiver for Weak Signals" IEEE MTT-S Digest 2001.

[13] Lee, B.H, & Kuo, S.M. "Real Time Digital Signal Processing, Implementations, Applications and Experiments with the TMS320C55x" John Wiley & Sons LTD, (New York) 2001 p. 330

[14] Proakis, J.G. and Manolakis, D.G, "Digital Signal Processing, Principles, Algorithms and Applications," P 423, Prentice Hall 1996.

[15] Manandhar, D., Suh, Y. & Shibasaki, R. "GPS Signal Acquisition and Tracking- An Approach towards Development of Software-based GPS Receiver." p. 2. The Institute of Electronics, Technical Report of IEICE. 2004.

[16] Psiaki, M.L. "Block Acquisition of Weak GPS Signals in a Software Receiver." p. 1 Princeton University Research Project Funded by NASA under cooperative agreement number NCC5-563 . Upublished.

[17] Tsui, J.B.Y. "Fundamentals of Global Positioning System Receivers, A Software Approach," J. Wiley & Sons, (New York, 2000), p. 4.

[18] Tsui, J.B.Y. "Fundamentals of Global Positioning System Receivers, A Software Approach," J. Wiley & Sons, (New York, 2000), pp. 73-75.

[19] Tsui, J.B.Y. "Fundamentals of Global Positioning System Receivers, A Software Approach," J. Wiley & Sons, (New York, 2000), p. 39

[20] Tsui, J.B.Y. "Fundamentals of Global Positioning System Receivers, A Software Approach," J. Wiley & Sons, (New York, 2000), p. 85, p.135.

[21] Tsui, J.B.Y. "Fundamentals of Global Positioning System Receivers, A Software Approach," J. Wiley & Sons, (New York, 2000), p. 136

[22] P.E Howland, D Maksimiuk and G Reitsma, IEEE Proc.-Radar Sonar Navigation Vol 152, No. 3, June 2005

[23] Tsui, J.B.Y. “Fundamentals of Global Positioning System Receivers, A Software Approach,” J. Wiley & Sons, (New York, 2000), p. 113

[24] Tsui, J.B.Y. “Fundamentals of Global Positioning System Receivers, A Software Approach,” J. Wiley & Sons, (New York, 2000), p. 111

[25] http://www.electronicproducts.com/ShowPage.asp?FileName=maxim.feb2006.html

[26] Dixon, R.C. “Spread Spectrum Systems with Commercial Applications Third Edition” pp. 327-330, Wiley Inter-Science, 1994.

[27] Van Nee, D., Coenen, A., “A New fast GPS code acquisition technique using FFT,” Electronic Letters, vol 27, pp. 158-160, January 17, 1991.

[28] J. Starzyk and Z. Zhu, “Averaging Correlation for C/A Code Acquisition and Tracking in Frequency Domain,” MWSCS Conference, Fairborn, OH, August 2001.

[29] A. Alqeeli Abdulqadir, “Global Positioning System Signal Acquisition and Tracking Using Field Programmable Gate Arrays, “ doctoral Dissertation, Ohio University, Athens, OH, November 2002.

[30] Dixon, R.C. “Spread Spectrum Systems with Commercial Applications Third Edition” Pp 160-166, Wiley Inter-Science, 1994.

[31] Xilinx FFT IP core DataSheet, page 18 http://www.xilinx.com/ipcenter/catalog/logicore/docs/xfft.pdf#search='xilinx%20fft%202005

[32] Xilinx FFT IP core DataSheet, page 9 http://www.xilinx.com/ipcenter/catalog/logicore/docs/xfft.pdf#search='xilinx%20fft%202005

[33] Ziedman, B 1999 Prentice Hall “Verilog Designer’s Library” p239

[34] Proakis, J.G. and Manolakis, D.G, "Digital Signal Processing, Principles, Algorithms and Applications," pp 555-556, Prentice Hall 1996.