A PRACTICAL APPROACH TO DSP ALGORITHMS USING FPGA DEVICES

by

Piraj Fozoonmayeh
B.A.Sc., Electronic Engineering, Simon Fraser University, 2008

THESIS SUBMITTED IN PARTIAL FULFILLMENT OF

THE REQUIREMENTS FOR THE DEGREE OF

MASTER OF APPLIED SCIENCE

in the

School of Engineering Science

SIMON FRASER UNIVERSITY

Spring 2011

Approval

Name: **Piraj Fozoonmayeh**

Degree: **Master of Applied Science**

Title of Thesis: **A Practical Approach To DSP Algorithms Using FPGA Devices**

Examining Committee:

Chair: **Dr. Sami Muhaidat**
Assistant Professor, School of Engineering Science

Dr. Rodney Vaughan
Senior Supervisor
Professor, School of Engineering Science

Dr. Jie Liang
Supervisor
Associate Professor, School of Engineering Science

Dr. Shawn Stapleton
Internal Examiner
Professor, School of Engineering Science

Date Defended/Approved: April 14, 2011

Declaration of
Partial Copyright Licence

Abstract

Digital Signal Processing (DSP) is a basis for FPGA designs and is the core technology of many electronics devices. However, it requires study at a practical level in order to complete its deployment. This thesis examines the practical implementation of different DSP algorithms using FPGA devices. In our first contribution, we investigate the performance of a quadrature demodulator when implemented on FPGA. In the second contribution, we design a switch in the transmitter of wireless system operating under orthogonal frequency division multiplexing (OFDM). Here we devise a simple FPGA design to utilize feedback from the receiver aimed at reducing interference present in the system. In the third contribution, we present an efficient FPGA design for real-time convolution/correlation, as an essential component in the electronics in most wireless systems. Throughout the thesis, our emphasis is on the practical aspects of DSP.

Keywords: Field Programmable Gate Arrays; Digital Signal Processing ; Orthogonal frequency-division multiplexing

Dedication

I would like to dedicate this thesis to my mother, Fariba, for all of her love and support throughout the years.

Acknowledgements

I would like to send my sincere gratitude to those who helped me in the completion of my thesis. Special thanks to my supervisor, Dr. Rodney Vaughan for his excellent mentorship and efforts in searching for support. He has been very patient and helpful with a paranoid student like me.

I would like to acknowledge all of Dr. Vaughan's research group (especially Jane, Ali, Alireza, Jin Yun, Maral and Milad) for their continuous help during the last few years that I have been in the masters program. I would like to thank Dr. Lesley Shannon, Dr. Shawn Stapleton and Mr. Patrick Leung for inspiring me in the practical field and collaborating with me to solve some of my technical problems.

I would like to thank my mom and my sister for all the love and support they have given me during my lifetime.

Table of Contents

List of Figures

List of Tables

Glossary

HDL	Hardware Design Language
ASIC	Application Specific Integrated Circuit
FPGA	Field Programmable Gate Arrays
GUI	Graphical User Interface
SoC	System on Chip
ADC	Analog to Digital Converter
DAC	Digital to Analog Converter
ESB	Embedded System Block
FIR	Finite Impulse Response
OFDM	Orthogonal frequency-division multiplexing
DSP	Digital Signal Processing
Sysgen	System Generator
MSB	Most Significant Bit
LSB	Least Significant Bit
ISI	Inter Symbol Interference
Quad Demod	Quadrature demodulator
VGA	Variable gain amplifier
MUX	Multiplexer

1: Introduction

The need for complex computation in multimedia and high-speed communication has increased the demand for hardware implementation of digital signal processing (DSP) algorithms.

Before 1990, most DSP algorithms were implemented using Integrated Circuits (ICs) mounted on Printed Circuit Boards (PCBs). Demand for miniaturization has encouraged designers to implement System-On-Chips (SOCs) using Application Specific Integrated Circuits (ASICs).

However, one of the most challenging aspects of using SOCs is that it is very difficult to make post-fabrication changes. Verification of the design functionality is very time consuming and it cannot be undertaken on the chip. These shortcomings have made FPGAs a better choice than ASICs for implementation of DSP algorithms. Since these devices can be reprogrammed easily, verification of design functionality can be undertaken on the chip. Another advantage of the FPGA is the flexibility in parallel computing which enables it to do massive parallel operations. [1]

1.1 Objective

The objectives are to study the step of how theoretically derived designs connect with implementation, and to provide insight into performance limits of

today's FPGA devices. With emphasis on the practical aspects of modern DSPs, our key contributions are three applications for wireless communications.

1.2 Design Environment

Mathwork's Simulink® provides one of the best design environments for implementing DSP algorithms. This environment supports a System Generator which has the most advanced DSP blocksets available. System Generator provides a high level design environment where designers can use graphical programming (G programming) to implement their algorithms. This type of programming is not efficient in terms of area (real estate on the chip) when implementing a custom design, but it is very time efficient. In the following three chapters, we use the System Generator as our main design tool (although we use ISE® and Xilinx® Platform Studio (XPS) for generation of the bitstream). In the last chapter, we use XPS and ISE as our main design tools to compare these tools to System Generator.

1.3 Background

Introduction to OFDM

Orthogonal frequency division multiplexing (OFDM) is a multicarrier communications technique that has been widely deployed in wireless communications systems. According to [2], the first OFDM system was proposed in 1966. The core idea of OFDM is simple: a high rate data stream is converted to several parallel lower-rate streams that are then modulated by separate carrier

frequencies using fast Fourier transforms (FFT). Moreover, with modern DSP, the FFT operations are fast and efficient in terms of hardware usage, thus making OFDM attractive from a practical point-of-view. Furthermore, by using parallel streams, a wideband channel spectrum is broken down into several narrow-band sub-channels that experience flat fading. This avoids the need for equalizers for each sub-band. Such benefits have made OFDM a central technology for many wireless standards. OFDM is already used in many broadcast standards such the Digital Video Broadcasting (DVB) in Europe. Most wireless standards such the IEEE 802.11a, wireless local area network (WLAN), and HiperLAN/2 are OFDM-based. It is expected that many future standards (including cellular) will be based on OFDM, such as the 3GPP Long Term Evolution standard.

In practical systems, the benefits of OFDM signalling come at an expense, a particular example is its sensitivity to certain forms of interference. To understand this, we elaborate on the types of interference relevant to OFDM. This serves as a prelude to our contributions.

1.4 Thesis Organization

We design some of the subsystems of an OFDM system throughout the chapters. Figure 1.1 shows an overall view of the OFDM system and the subsystem that we intend to design. The receiver is on the right side, and the transmitter is on the left of the Figure 1.1. We intend to design the quadrature

demodulator in the receiver. In addition to the quadrature demodulator in the receiver, we implement an algorithm that controls the external variable gain amplifier (VGA). The VGA is denoted with a triangle containing a plus sign in Figure 1-1. We also intend to design a switch in the transmitter, which changes some subcarrier positions of the symbols, explained further below.

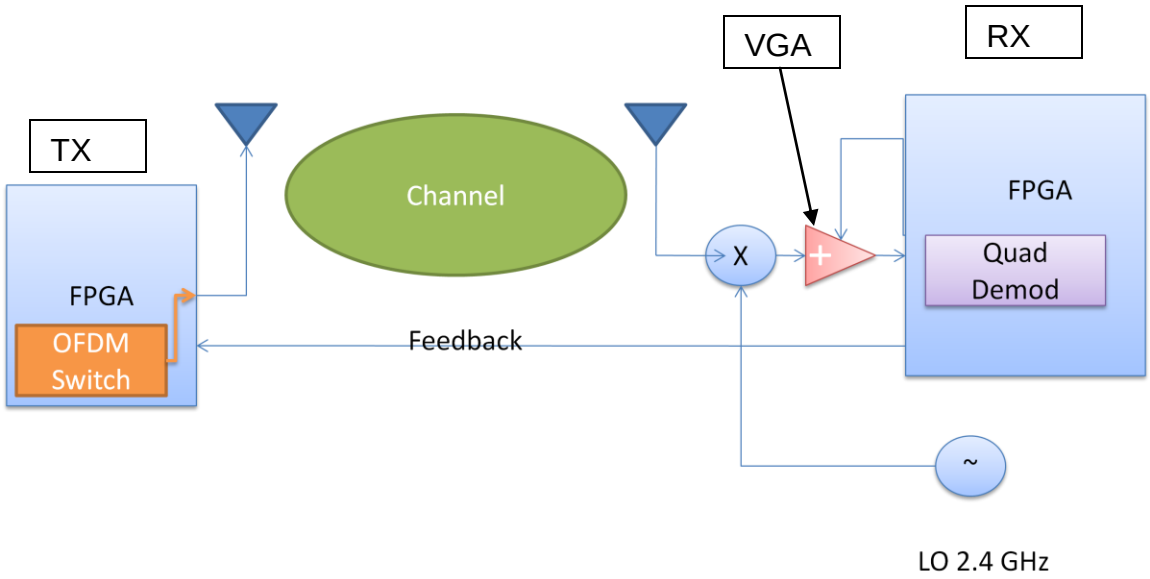

Figure 1-1 Overall View of the OFDM System

This thesis has four chapters. Chapter 1 gives an introduction and background of SOCs, FPGAs and background to OFDM. Chapter 2 discusses the design of the quadrature demodulator and tests it in real time using a Linksys wireless router. Chapter 3 discusses the design of the OFDM switch that manipulates the subcarrier positions within the OFDM symbol to improve the redundancy and throughput of 802.11 algorithms operating over a frequency-selective channel. Chapter 4 discusses the design of a convolver and a

correlator. The convolver could be used, for example, to model a linear time-varying channel in Figure 1-1. We also discuss the limitations of today's FPGA chips in the implementation of these algorithms.

2: Quadrature demodulator with support functions

Modern digital radios are a combination of RF, analog and digital signal processing. The DSP section of the radio is implemented by a DSP-specific microprocessor; however, the FPGA has become a viable alternative to microprocessors for certain radio systems [3]. The purpose of this chapter is to investigate the design of some subsystems of a digital radio and implement these designs in an FPGA. These subsystems will be designed using the System Generator environment, implemented in Xilinx XtremeDSP [3] evaluation boards, and tested by interfacing with off-the-shelf 802.11 hardware. Multiple-input and multiple-output (MIMO) systems are being used with OFDM. This requires DSP calculations such as array multiplications, Fast Fourier transforms (FFTs), and inverse FFTs (IFFTs). Such systems typically need a lot of memory, and so a good design goal is to use an algorithm that occupies minimal memory. We demonstrate the difference in memory usage between presented algorithms.

Our setup is shown in Figure 2-2. The OFDM signal is acquired from a Linksys router and it is fed to a RX/TX switch. The RX/TX switch detects when the Linksys router is in transmit mode and feeds the OFDM signal to a mixer which down-converts the signal from 2.4 GHz to 25 MHz. The down-converted signal is fed to the FPGA and the output from the FPGA-implemented algorithm is viewed with an oscilloscope. Figure 2-1 shows the technique that we used in designing

the quadrature demodulator [4]. Figure 2-1 illustrates the subsystems of our design and the signal routing.

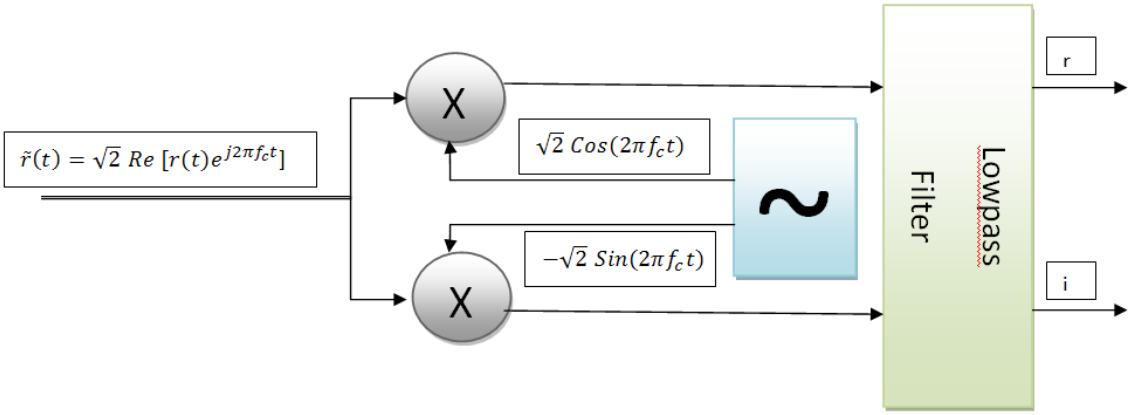

Figure 2-1 Quadrature demodulator technique

The following equations show how we retrieve the real part of the original signal from its modulated version.

$$\sqrt{2}\,\boldsymbol{Re}\left[\boldsymbol{r(t)e^{j2\pi f_c t}}\right] \times \sqrt{2}\boldsymbol{Cos(2\pi f_c t)} \qquad \textbf{(2-1)}$$

$$= \boldsymbol{2\ Re}\left[\boldsymbol{r(t)e^{j2\pi f_c t}}\right]\boldsymbol{Re}\left[\boldsymbol{e^{j2\pi f_c t}}\right]$$

Using identity $\boldsymbol{Re[\alpha]Re[\beta]} = \frac{1}{2}[\boldsymbol{\alpha\beta^*} + \boldsymbol{\alpha\beta}]$

$$= \boldsymbol{Re}\left[\boldsymbol{r(t) + r(t)e^{j4\pi f_c t}}\right]$$

We can see that the final term has a double frequency term, which can be eliminated using a low pass filter. The imaginary part can be retrieved using a similar identity.

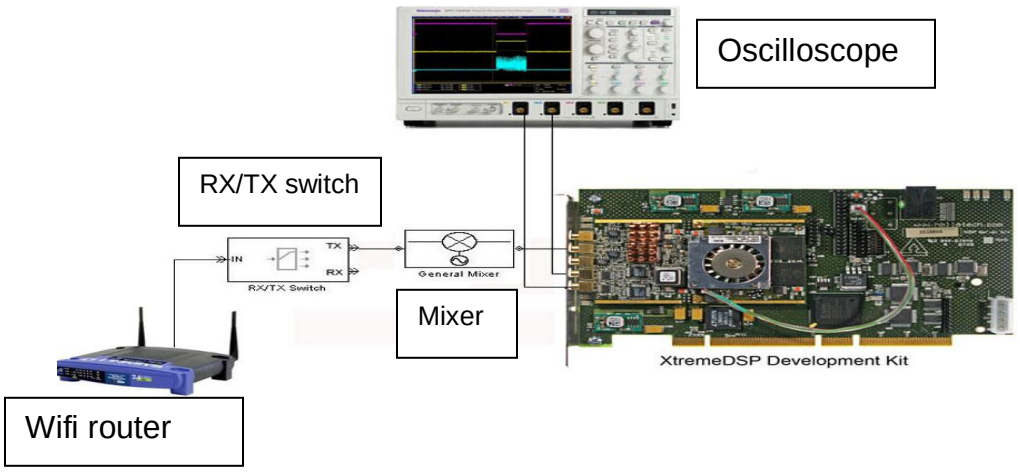

Figure 2-2 Overall setup. The photo of the extreme DSP kit is from [3]

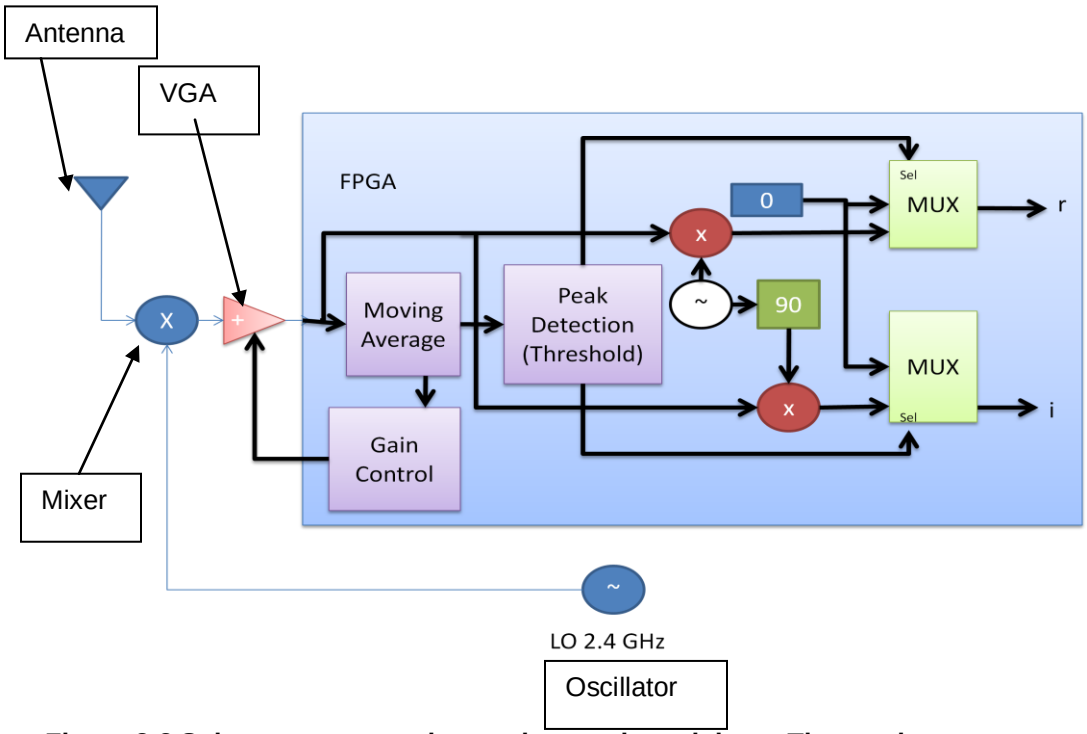

Figure 2-3 Subsystems around a quadrature demodulator. The moving average and detection is for detecting the presence of an OFDM signal.

This chapter contains algorithms used to implement the following:

1. Sine wave generation: This DSP block is capable of generating sine/cosine waves for multiplying the OFDM signal when a signal is detected. [4]

2. Signal detection: The DSP block that detects the presence of an OFDM signal is a combination of signal rectification block, moving average blocks (implementation of windowed averaging) and signal threshold detection blocks (Conventional threshold and Schmitt trigger)

3. Automatic gain control: An external variable gain amplifier interfaced with the DSP. The DSP circuit module should detect the presence of an OFDM signal and scale the input signal to be centred in the dynamic range of the analog-to-digital convertor (ADC) within a specified length of time.

2.1 Sinosoid Wave Generation

We need to design a block which is capable of generating sine/cosine at a certain carrier (intermediate) frequency. We multiply these sinusoids by the received (intermediate frequency) signal [4].

We found, from the literature, that the most common technique for sinusoidal function generation is to use a lookup table for the sinusoidal values [5]. We attempted a different design below using another approach for a look up table, presented in Section 2.1.1. We started this design by using a ROM block for the look up table.

2.1.1 Using ROM Block and MATLAB M-file

An easy way to generate a sign wave is to store the sample values in the ROM block and read it out at different speeds to acquire different frequencies. We use equation (2-2) below to generate the sample values of a sine wave which we

store in the look up table. In order to generate wave at different frequencies, we have to read the stored values in the look up table at different speed. This is done using a Digital Clock Management block (DCM). The samples are stored in a ROM block. A counter clocks the ROM look up table, which in turn will read out the content of the indexes of the RAM block. Figure 2-4 illustrates the sine wave generation algorithm that was implemented in the System Generator program. This algorithm is constructed by combining the individual blocks from System Generator's library.

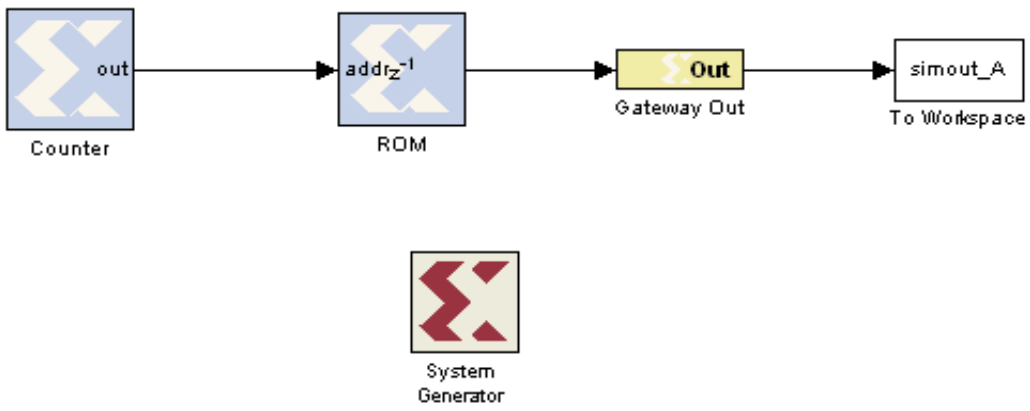

Figure 2-4 Sine wave generation using MATLAB and ROM Block, from System Generator.

The following MATLAB code shows the generation of the single cycle of a sine wave in MATLAB:

$$\boldsymbol{Ts = \frac{(2\pi)}{1024} = 0.0061}$$

$$\boldsymbol{x = 0{:}\,0.0061{:}\,2\pi;}$$

$$\boldsymbol{A = Sin(x)} \qquad (2\text{-}1)$$

Ts is the spacing between samples. *x* is the sample location (The magnitude of block ROM should be 2^n therefore 1024 samples). A is the actual value of the sample.

2.1.2 Using Direct Digital synthesizer

We introduce a second design which uses a Direct Digital Synthesizer (DDS) block in the System Generator. This block can generate sine, cosine, or quadraturerature outputs [6]. The interface can be configured to accept system-level parameters such as the output frequency and SNR level (size of look up table) of the generated waveforms.

The mechanism of operation of the DDS is as follows: the samples of the desired signal are stored in the look-up table. A digital integrator generates the clock. The design is depicted in Figure 2-5 and the way it works it laid out in [6].

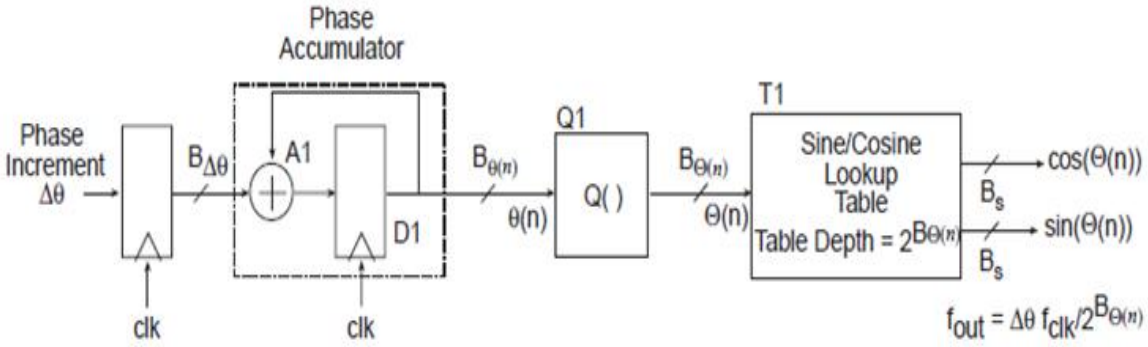

Figure 2-5 Phase Truncation DDS [6]

According to [2], the phase slope of the sinusoid (frequency) is calculated in the integrator (components D1 and A1) which produces an index to the look-up table T1. The quantizer, Q1, is simply a slicer that takes the high-precision phase angle θ(n) and generates a lower precision representation of θ(n) to the right of Q1 in the figure above. These values act as the index in the lookup table [6].

The desired frequency of the signal is determined by the following formula [2]:

$$\boldsymbol{Fout} = \frac{\boldsymbol{f_{clk}} * \Delta\Theta}{\boldsymbol{2^{B_\Theta(n)}}} \qquad (2\text{-}2)$$

where $B_\theta(n)$ is the number of bits in the phase accumulator and ΔΘ is the phase increment and f_{clk} is the frequency of the FPGA clock. Figure 2-6 illustrates the DSS block in the System Generator.

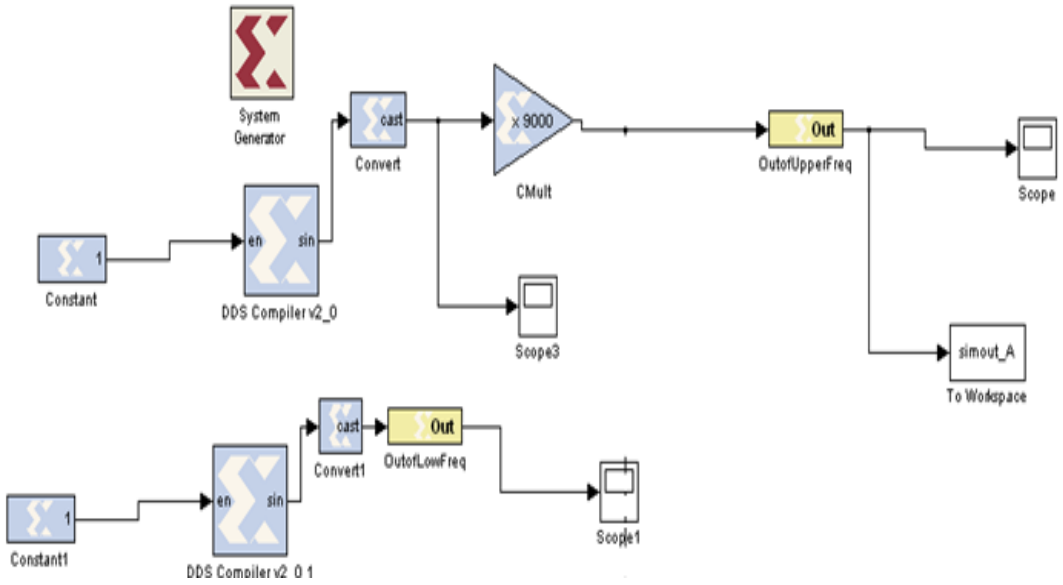

Figure 2-6 System Generator view of two DSS's for two different frequencies

The frequencies are set inside each DSS block. The constants, depicted in Figure 2.5, need to be set to 1 to activate the DSS blocks. Since the amplitude of the signal that was attained from the ADC might be different from the amplitude of the generated signal by the DDS, an additional gain block can be added for future scaling of the samples, as we can see in the upper DSS in Figure 2-6.

Disadvantages of using DSS

"Spurs" (spurious frequencies) are a problem imposed by the DSS block. Still, the predictability of the spurious frequencies allows us to develop a filter to suppress them.

Advantages of using DSS over Using a ROM Block [6]

1. 1. The look-up table in the DDS is more efficient use of block memory FPGA resources, than the look-up table of the previous algorithm of Figure 2.3.
    a. The Spur-Free Dynamic Range (SFDR) can be set from 18 dB to 120 dB
    b. Higher accuracy can be obtained in several ways:
2. Using conventional phase truncation waveform synthesis architectures.[6]
3. Adding a high-precision synthesizer with fine frequency resolution.[6]
4. Using 4-bit to 20-bit two's complement output sample precision.[6]
5. 4. A user interface that is easy to use.

2.2 Detection of Signal

In the literature, a common technique to detect a signal is to use amplitude thresholding [7]. In order to use amplitude thresholding we need to design a block to implement the threshold.

The solution is a moving average (windowed average) block. We have implemented two techniques for the moving average. We have implemented different techniques to optimize the performance of the signal detections. We also study the advantages and disadvantages of each technique.

2.2.1 Moving Average

In the literature, the most common technique to design a moving average is to use a FIR filter. A disadvantage of the FIR filter is that current FIR Compiler blocks only accept up to 1024 samples, and a moving average can require much

more than this. During our studies, two different techniques have been implemented for a moving average.

2.2.1.1 Having N-1 delays for a window size N

In this technique a window of size N-1 is implemented using N-1 single delay blocks. After the implementation of this window, the samples are added and this sum is divided by the window size. This is done by the System Generator's add/sub block and shift register block. The reason for choosing shift registers is discussed in the next section. There are two disadvantages to using this scheme:

1. Implementing a large window requires many delay blocks and many additional blocks, which is not easy to implement.
2. The memory usage is inefficient.

Figure 2-7 illustrates a window of size 4. As can be seen from the figure, there are 3 delay blocks (named delay 9,7, and 6), each of these blocks is configured to produce 1,2, and 3 clock cycles of delay, respectively. We add the output of these delay blocks to the current sample of the ADC, by doing this; we have added 4 consecutive samples. Shifting the summation twice to the right means that we have divided this result by 4, which gives us the average value of 4 samples in our window.

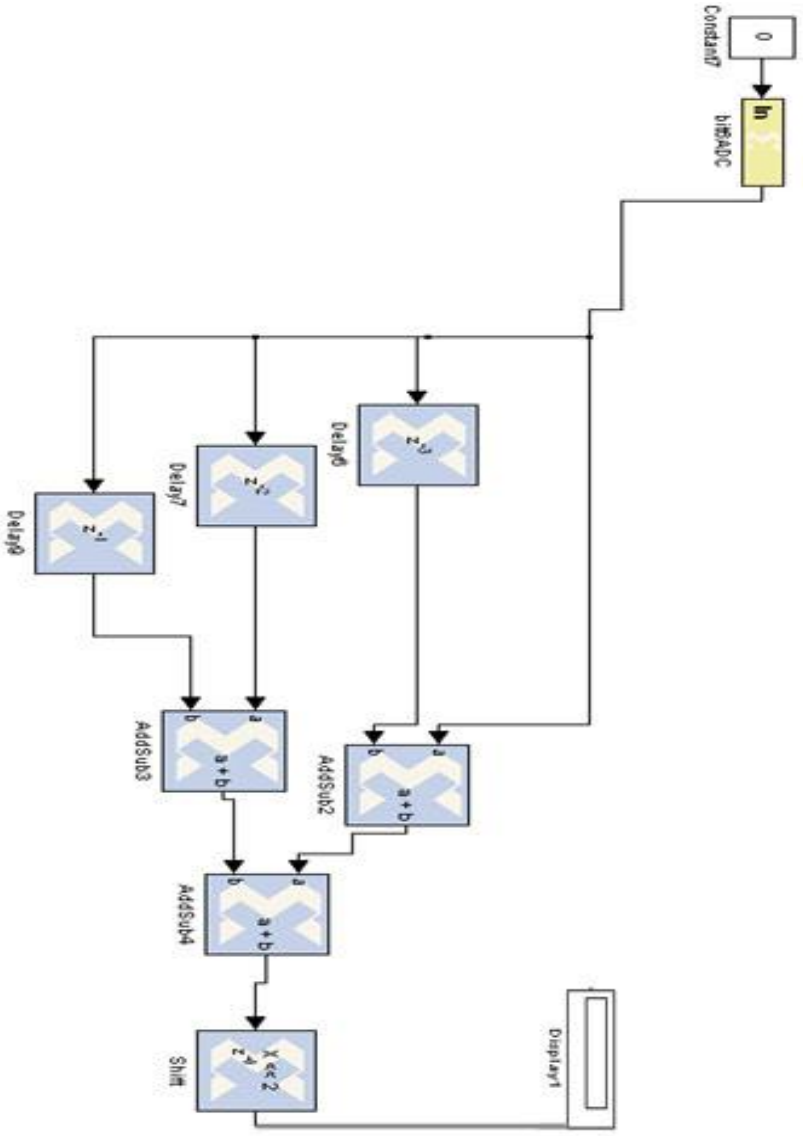

Figure 2-7 Four Samples Moving Average

2.2.1.2 Using two delay blocks

In order to improve the resource efficiency, we implement a new algorithm, which uses the previous running mean (the average from a previous window) to calculate the average of the total window. Figure 2-8 illustrates this algorithm.

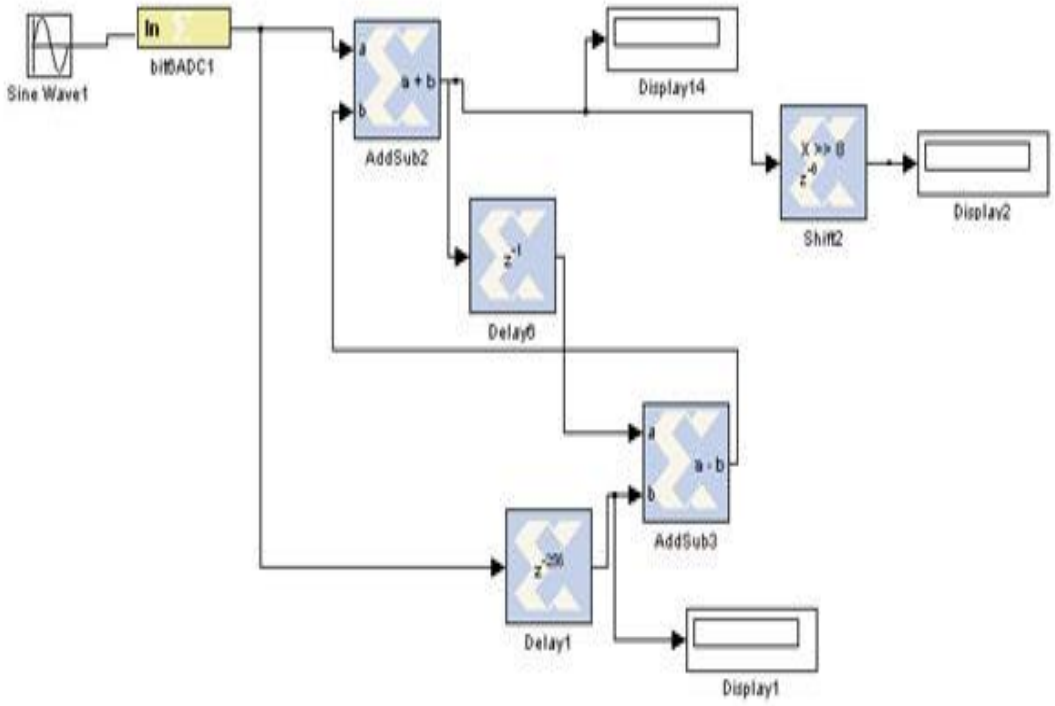

Figure 2-8 256 Samples Moving Average

Figure 2-8 is an example of a window of size 256. During the first cycle, the ADC value is fed to AddSub2's port *a*. The value of port *b* of this block is zero (this is because the output from AddSub3 is zero, as Z^{-1} (1 cycle delay) and Z^{-256} (256 cycles delay) are zero on the first cycle), so its output is equal to *a*. During the second cycle, the 2nd ADC sample is ready at port *a*, and port *b* has the value of the output of Addsub3 (which is equal to the previous sample only), therefore the 1st sample has been added to the 2nd sample. At this point, the sum of the two samples are fed to AddSub3 (note that port *b* of AddSub3 will stay at zero until the 257th cycle, and at the 257th cycle it will have the value of the 1st sample). During the 3rd cycle, port *a* of Addsub2 will have the 3rd sample of the

ADC and port *b* will have the sum of the two previous samples. In other word, during the 3rd cycle, the first three sample values of the ADC have been added. This summation will continue until the 257th cycle, where delay1 starts to have *a* value, and this value is the ADC reading of the first cycle (on 257th cycle it will have the first sample (Z^{-256})). This value will be subtracted from port *a* of addsub3 (the value that is fed at addsub3 at cycle 257 is the sum of the first 257 samples), so at this point the summation of the samples in the window is complete. By using a shift register with a shift value 8, we divide the summation of 256 samples by 8. The reason for using the shift register is that there is no straightforward division block in the System Generator. We can use a combination of different blocks to implement division, but this adds noise to our signal. It is worth noting that each System Generator block usually has two options for its output type:

1. Full: System Generator determine the output type
2. User Defined: User defines the output type.

Since the System Generator does not understand the window size in the above algorithm, it does not know how many bits it should assign for the output of AddSub2, which results in a compilation error. The way to fix this error is to use the user defined output option and manually assign the number of bits, taking the maximum value of the summation into account.

2.2.2 Threshold

2.2.2.1 Detection by using OR gate for the top bits and using System Generator threshold block

The signal from the ADC is in two's complement format. Therefore using OR gates on the magnitude portion (does not include the sign bit – these are called top bits) with itself only works for positive numbers. This is seen from the following 2's complement figure (Figure 2-9). Although using a NOR gate will give us correct results for both positive and negative numbers, it does not work when the signal is zero.

sign bit									
0	1	1	1	1	1	1	1	=	127
0	0	0	0	0	0	1	0	=	2
0	0	0	0	0	0	0	1	=	1
0	0	0	0	0	0	0	0	=	0
1	1	1	1	1	1	1	1	=	−1
1	1	1	1	1	1	1	0	=	−2
1	0	0	0	0	0	0	1	=	−127
1	0	0	0	0	0	0	0	=	−128

8-bit two's-complement integers

Figure 2-9 Two's complement format [5]

Figure 2-10 illustrates the System Generator's view for detection of a positive signal. We have the bashed bits (an operation that extracts the bits, which allows us to look at each top bits separately) fed to the logical OR block, which will tell us if any of the top bits are set. In the next step, we concatenate the result from

the OR block, with 5 zeros (result from OR block goes to MSB). This will give us the number 32 if any of the high bits of input are set. Since threshold block of System Generator is preset at zero, we rescale this by using an add/sub block (AddSub). The result from the threshold controls the select input of a multiplexer which in turn shows if the signal is detected.

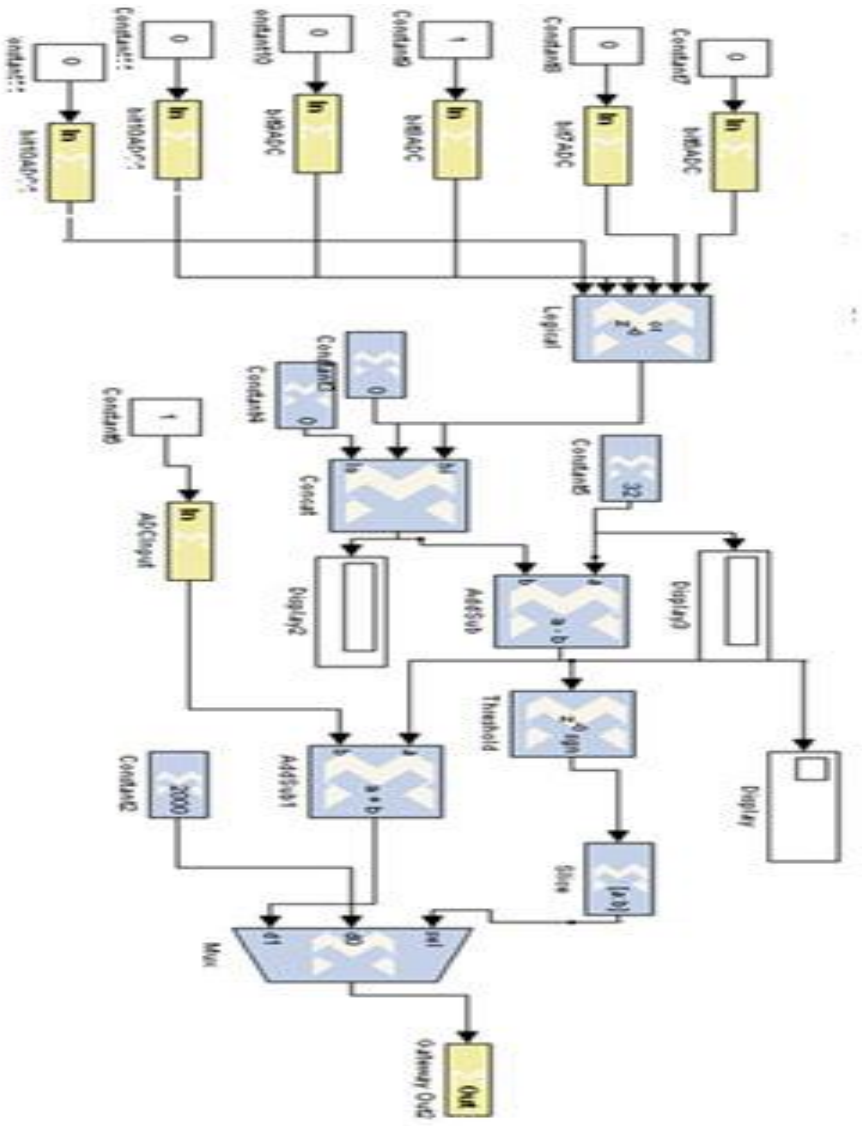

Figure 2-10 Detection of Positive Signal

A more complicated algorithm can be used to improve the quality of our detection, namely using the sign bit to control the result of OR gates. This algorithm involves using a multiplexer that has the sign bit at its selected input. When the signal is negative, we have the high bits of the 2's complement input equal to zero, so by NOT gates, the high bits and using the sign bit to control the multiplexer we can solve the issue of having a negative signal. Figure 2-11 illustrates this algorithm.

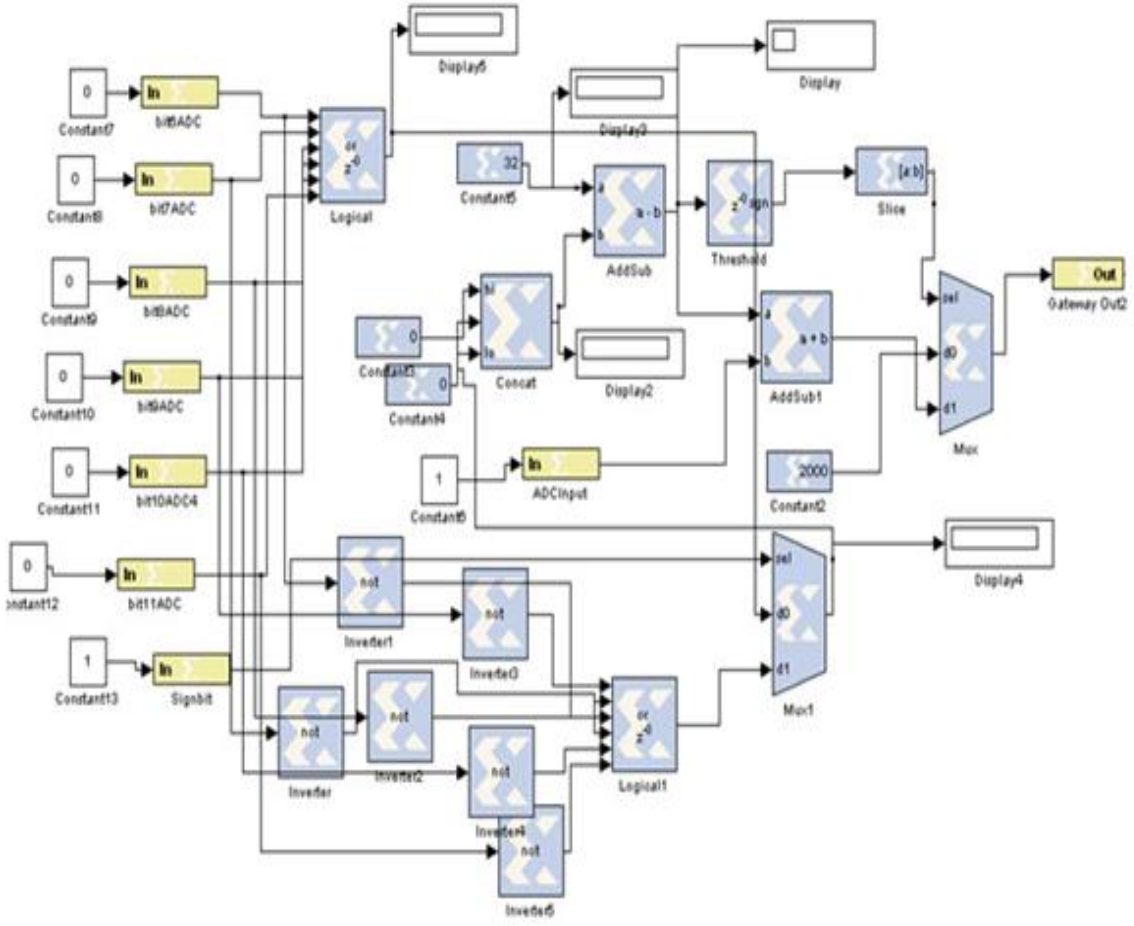

Figure 2-11 Positive and Negative Detection

Since the threshold block in the System Generator has a preset threshold at zero, we need to subtract a constant from our signal to change the threshold to our desired number. There are some disadvantages to using the ORing technique:

1. The threshold level can only be set to powers of two; for example, if the desired threshold is at 300 (i.e., not power of two), we need to OR bit 8 with bit 13, meaning that we have to put the threshold to the wrong value of 256. This adds some error to our detection.
2. The memory usage is inefficient.

2.2.2.2 Detection using relational blocks

We developed a new technique to overcome the disadvantages of the previous techniques. In this technique, we use the relational block of the System Generator. This will give us more accuracy and the final algorithm will be more user-friendly for the designer. In order to have a more user friendly program, bit bashing (this is not an operation, it just removes the notional decimal point) followed by a concatenation algorithm has been implemented. This is because the reading from the ADC is 14 bits wide, where the 14^{th} bit is the sign bit and the remaining 13 bits are decimal binary values (so the ADC reading is in 2's complement which represents a value between -0.99 and +0.99). Therefore, setting fractional thresholds is not easy, so by using bit bashing and concatenation we can use a fixed point number as our threshold.

Some explanation of the jargon may be useful here. Bit bashing extracts the bits, therefore by bashing our sample input completely, we get about 14 separated bits. Concatenation is a technique that allows us to cast bits (i.e.,

reassemble them), therefore by casting the bashed bits we get the scaled version of ADC. (This operation is just to make the system user's life easier and does not change anything in our samples.)

Figure 2-12 illustrates this bit bashing and concatenation in System Generator. We can see from Figure 2.11 that the threshold is set at 400 in port *b* of the Relational2 block, and port *a* is the fixed point ADC value (which can be between -8192 and 8192). If the ADC value is more than 400, the relational block will output the value 1 and since this output controls our multiplexer, the value 0.99 will be outputted to the DAC.

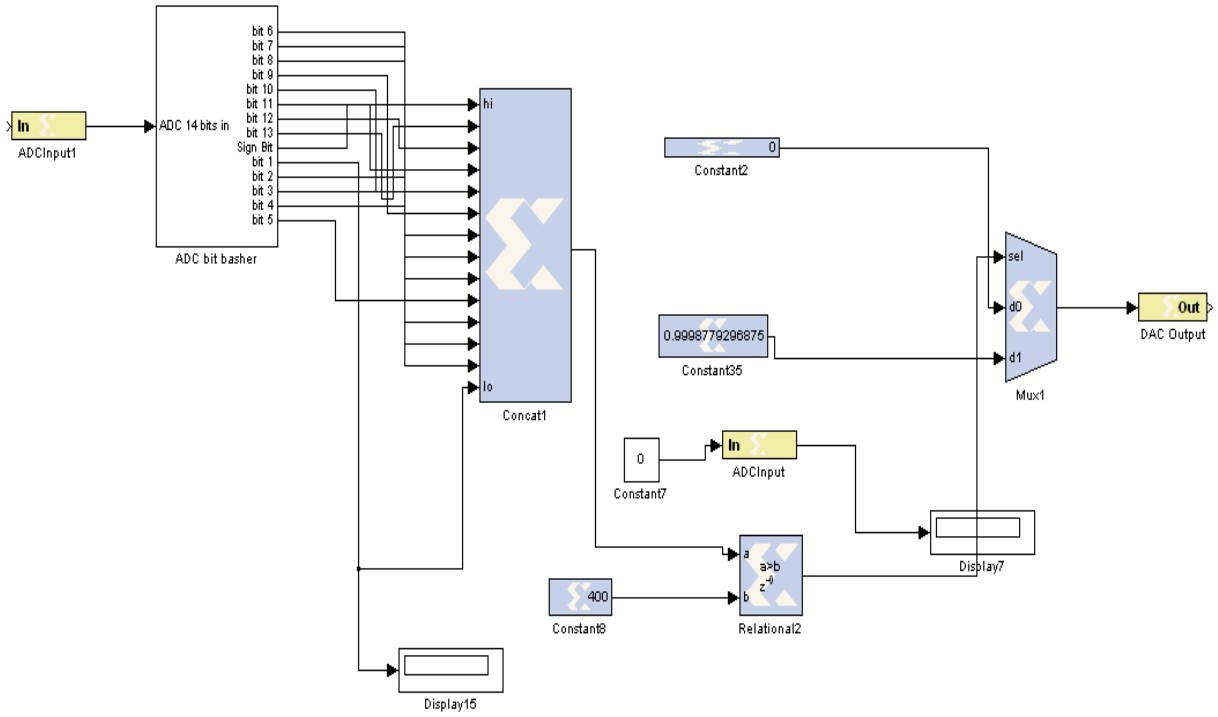

Figure 2-12 Binary point to Fixed Point Conversion

We prefer to use this method over the previous ones because:

1. The threshold can be set at any desired level.
2. Easy calculation of the threshold level, since we have scaled the fractions to the fixed points.
3. The memory usage is more efficient.

2.2.3 Rectification of the ADC signal

As discussed in the previous sections, there is a problem when we had a negative signal in our detection algorithm (*viz*., catering for negative polarity signals requires more memory). In order to overcome this issue, we can invert the negative part of our signal for our detection. The following three algorithms have been implemented.

2.2.3.1 Squaring the Signal

In this method, we multiply the ADC reading by itself. Figure 2-13 illustrates this method.

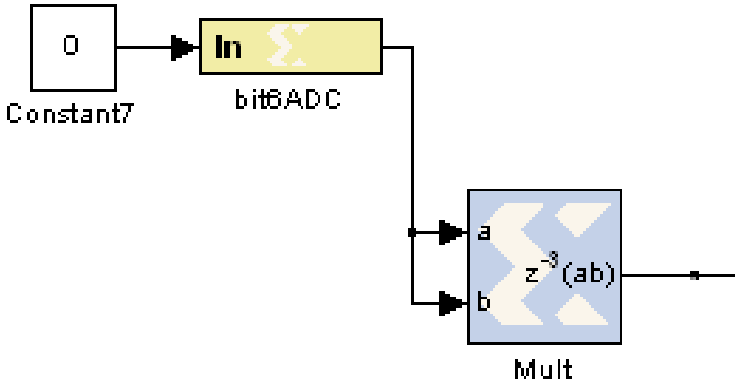

Figure 2-13 Signal Squaring

This block basically squares the input of the ADC. Some disadvantages of this method are as follows:

1. The Mult block adds noise. The reason for this noise is that the System Generator does not use an “Adaptive Algorithm” (one that caters for negative numbers) in the Mult block when the signal is negative.
2. The memory usage is inefficient in comparison to the other algorithm (discussed below). Figure 2-14 illustrates the Resource Estimator block of the System Generator for this algorithm.

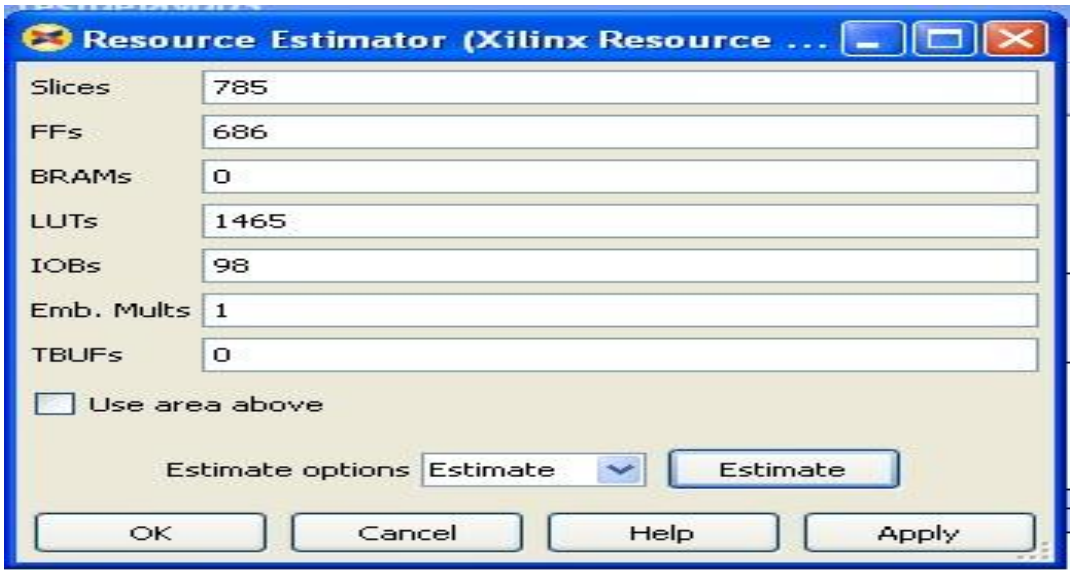

Figure 2-14 System Generator's Resource Estimator

3. The output of the Mult block will be 26 bits wide, as opposed to the 13-bit input signal. Such non-linear operators should be avoided because the threshold values need scaling as well.
4. This algorithm adds delay because of extra computation time.

2.2.3.2 Multiplication of the signal by -1 and using a multiplexer

The selection port of the multiplexer is fed by the sign bit of ADC data, therefore when the signal is negative it chooses d1 (see Figure 2.15), which is the ADC reading multiplied by -1 (which will result in a positive number). This method also has the disadvantage of adding noise to our signal, as discussed above. Figure 2-15 shows this method in detail.

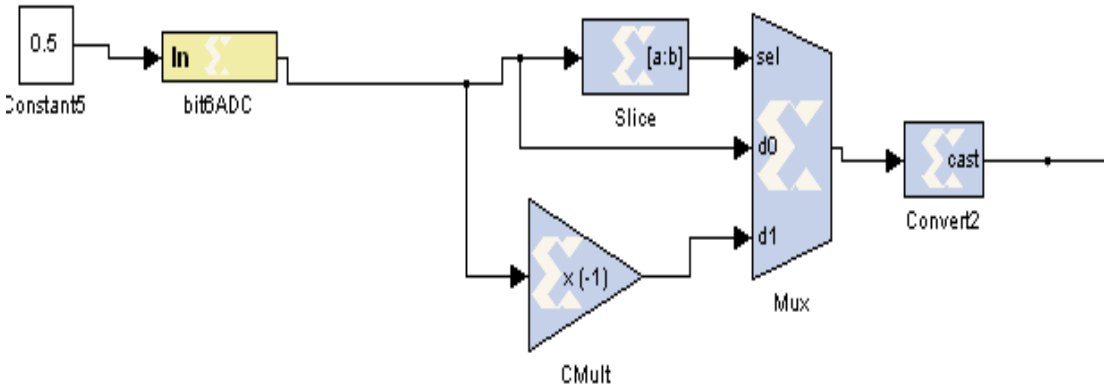

Figure 2-15 Rectifying Signal Using CMult

2.2.3.3 Using NOT, Add/Sub block and a multiplexer

If we use NOT gate on all the bits of a negative 2's complement and add 1 to them [7] , we get the positive representation of a negative number. Figure 2-16 illustrates this scheme:

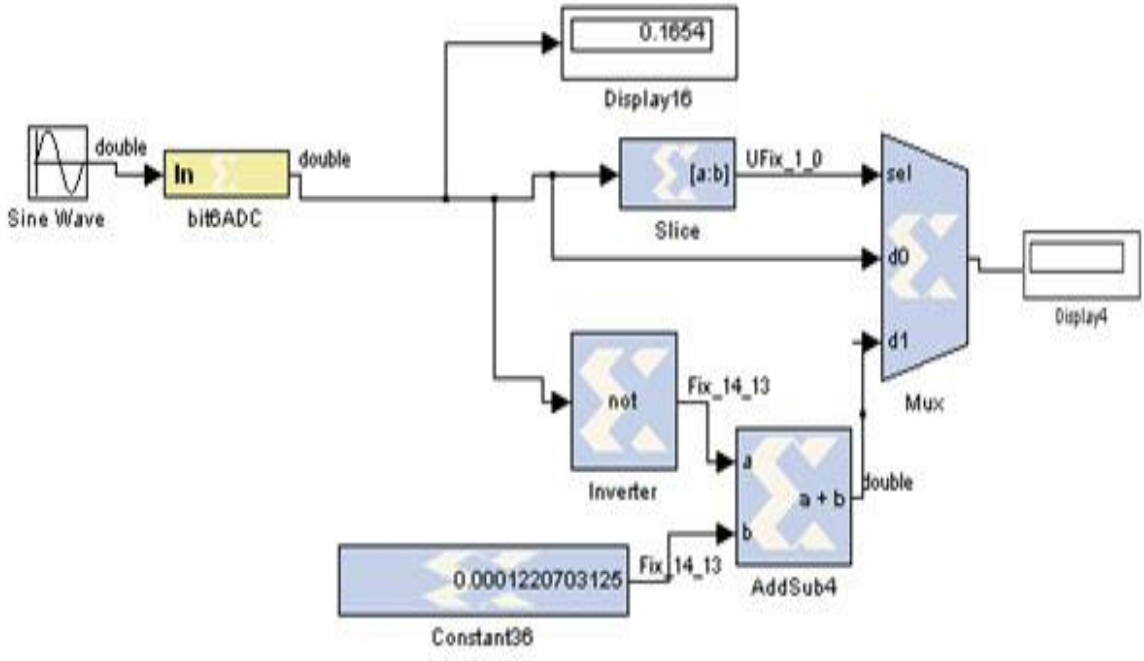

Figure 2-16 Rectifying Signal using NOT Gate

This method is preferred as it adds no noise and uses less memory space. Figure 2-17 illustrates the Resource Estimator block of the System Generator for this algorithm.

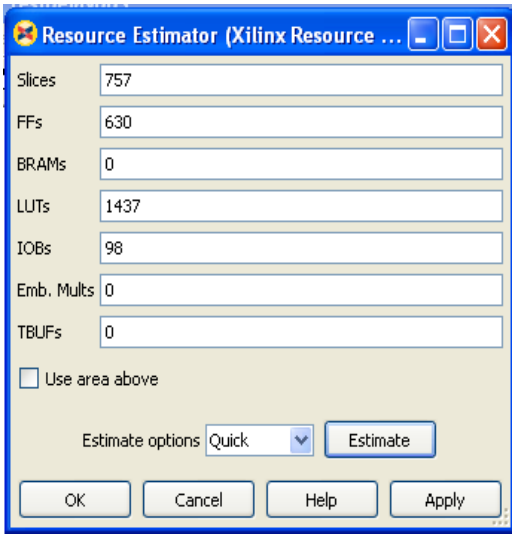

Figure 2-17 System Generator's Resource Estimator when using NOT gate

2.2.4 Results of the detection algorithm

Figure 2-18 illustrates the relational detection algorithm with a 256 sample window size on a sine wave (Using NOT, Add/Sub block and a multiplexer)

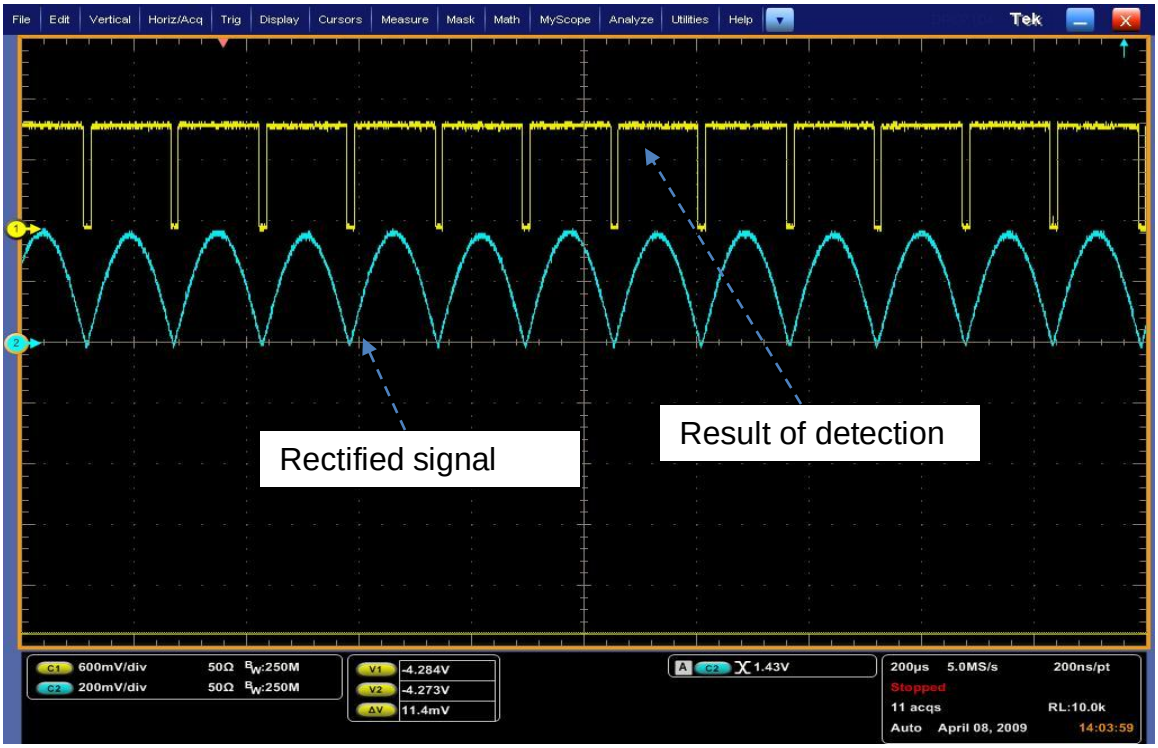

Figure 2-18 Detection of Sine Wave Signal

The detection threshold has been set at 40 mV, and the frequency of the sine wave has been set to 100KHz. This would allow 256 samples to sit in the area under our threshold level. The upper signal is the threshold output, and is high when the signal presence is detected by thresholding.

Figure 2-19 shows the OFDM symbol (lowest) that we will use in our detection. The blue signal (upper) is the hardware detection of the OFDM symbol. The middle signal is the rectified version of the OFDM symbol.

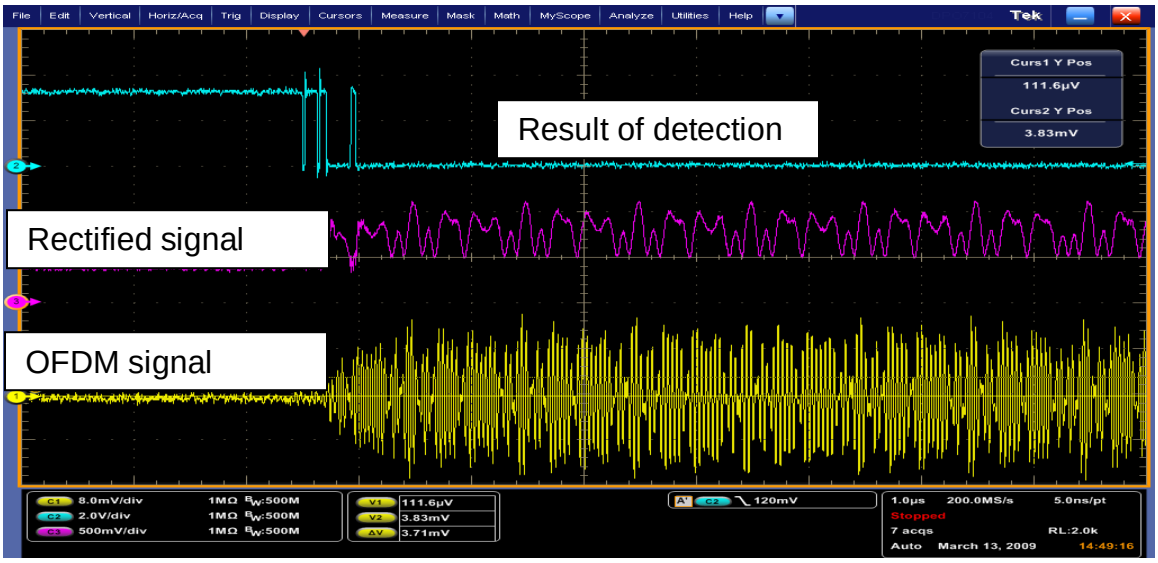

Figure 2-19 OFDM Symbol

Figure 2-20 illustrates our FPGA detection of the OFDM symbol. The lowest trace is the rectified OFDM signal used in our detection, the middle line is our FPGA detection and the upper line is the hardware detection.

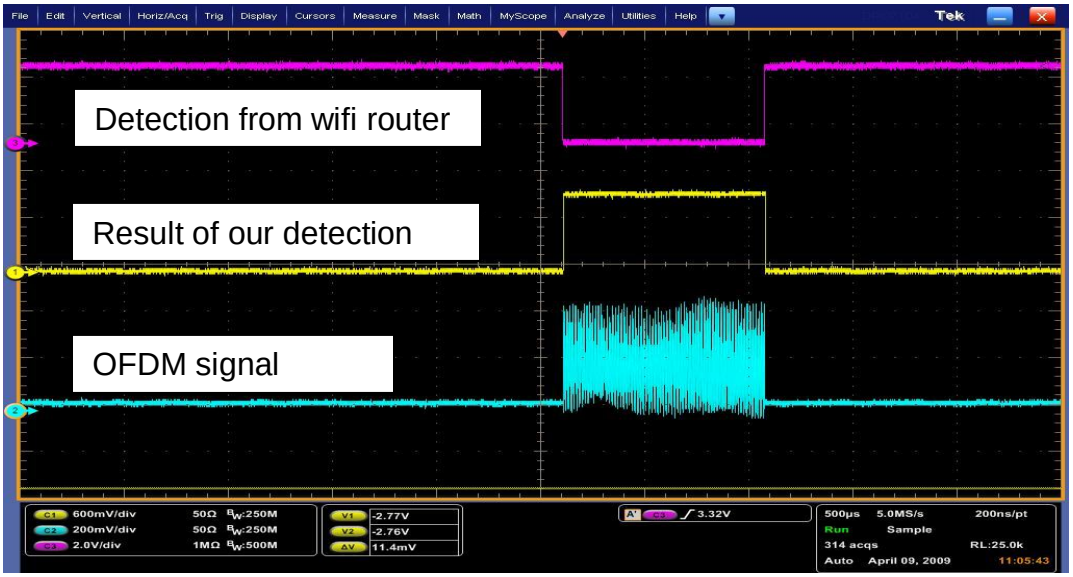

Figure 2-20 Detection of OFDM Symbol

2.3 Automatic Gain Control

An external variable gain amplifier is needed to amplify the signal when the signal level is low. This solution will reduce quantization noise. The DSP circuit module detects the presence of the OFDM signal and scales the input signal to be centered in the dynamic range of the ADC, within a specified length of time. The detection algorithm is discussed in the previous sections. The gain control algorithm consists of an MCode block, which is basically a complicated version of the System Generator's Relational block, and a large multiplexer. Figure 2-21 illustrates the inside of the MCode block.

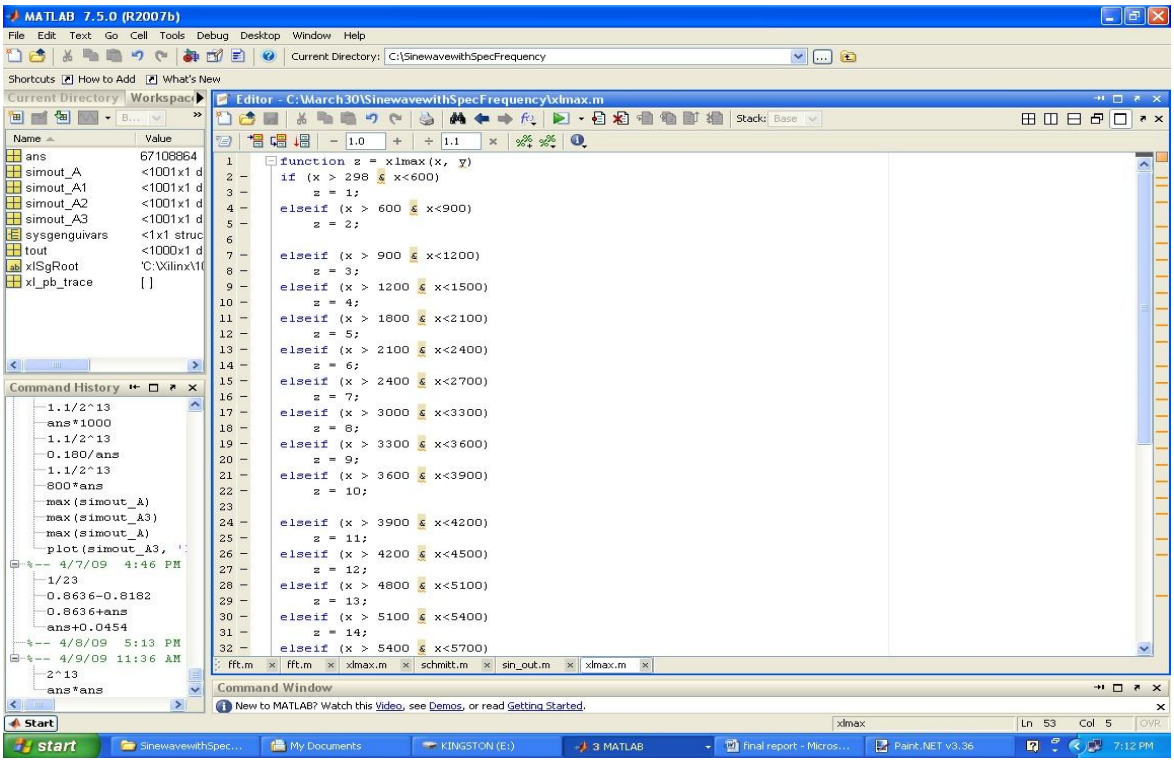

Figure 2-21 MATLAB MCode

The reason for using a big multiplexer is that the MCode block can only output fixed point numbers, therefore bit bashing and concatenation of the ADC

reading is very important in this algorithm. In the following figure, the samples in our input gateway are fixed point numbers, meaning that the ADC values have passed through bit bashing and concatenation blocks before they were fed to this input gateway. The MCode block evaluates the input sample value and outputs a fixed point number which controls the multiplexer. The multiplexer has 24 inputs. Each of these inputs represents a level for our DAC. Since the dynamic range of the DAC is between 0 and 1, each consecutive level is $\frac{1}{24}$ different from the last. Figure 2-22 illustrates this algorithm.

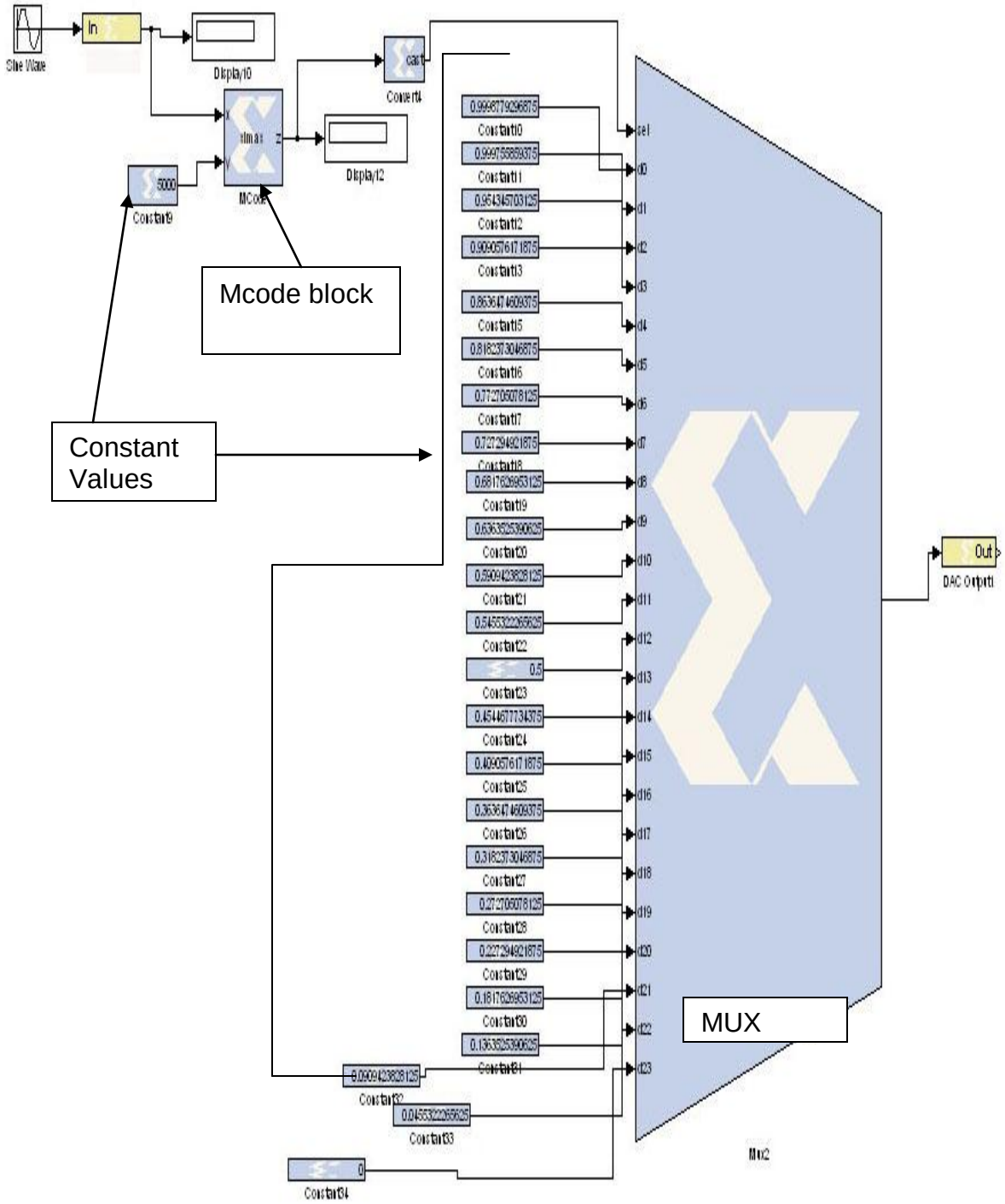

Figure 2-22 Variable Gain Control

2.4 Schmitt trigger

The Schmitt trigger algorithm has been developed to have different threshold levels for our signal detection. Figure 2-23 illustrates this function:

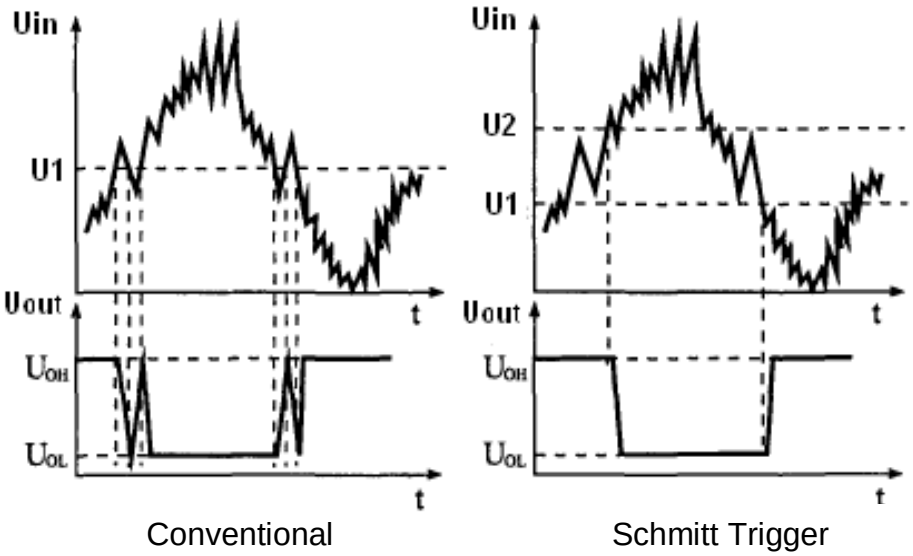

Figure 2-23 Conventional threshold detection vs. Schmitt Trigger threshold detection. From [4]

Here, the input gateway has fixed point values (the ADC values have passed the Bit Bash and Concat block). A delay block is used to calculate the slope of the signal. The slope is used in the MCode block to determine the threshold level. Figure 2-24 shows the System Generator's view of our Schmitt trigger algorithm:

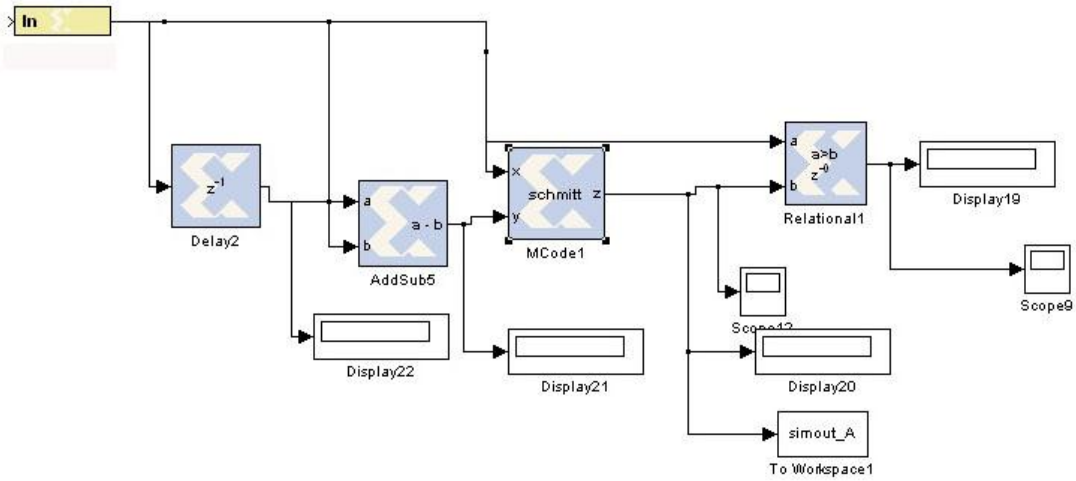

Figure 2-24 Schmitt Trigger in System Generator

3: OFDM Switch Design

Interference in OFDM-based systems [2]

An important practical drawback to OFDM is its sensitivity to narrowband interference [2]. Since data is transmitted over each subcarrier, one or more information symbols are likely to be destroyed when these subcarriers experience narrowband interference at the receiver. As a result, the bit error rate (BER) and capacity performances are jeopardized. The source of narrow-band interference is generally somewhere outside the primary communication system; for example a microwave oven operating in the 2.4GHz band of a Wi-Fi based OFDM system.

In addition to outside interference, the actual OFDM system itself may create interference. This form of interference generally manifests in two forms: i) inter-carrier interference (ICI) and ii) inter-symbol interference (ISI). ICI is caused by temporal variations of the channel that result in the loss of orthogonality between the subcarriers. This is due to the Doppler shift associated with the temporal variations of the channel. Both ISI and ICI are undesirable since they lead to an error floor in conventional receivers.

Fortunately, while the nature of interferences is different, the general techniques for dealing with them are quite similar. For example, as summarized in [2], there are several approaches to dealing with ICI:

- **Tuning system parameters**. The system designer may decide to tune the frequency spacing, symbol time or even the pulse shaping parameters to trade-off ICI with BER performance.

- **Self-interference cancellation.** ICI may be greatly reduced using simple frequency coding techniques at the transmitter. For example, repetition coding may be used to send replicas of symbols on multiple subcarriers. Although this method is very effective for the mitigation of ICI, it clearly leads to a reduction of the spectral efficiency of the system.

- **Temporal equalization.** In these techniques, equalization is done in the time domain, before going through the FFT block in the receiver. The objective here is to maximally reduce the ICI after equalization.

- **Forward-error correction.** The capability of any forward-error-correcting code can be used to eliminate the errors caused by the ICI, and correlative coding can improve the ICI as well.

- **Redundant frequency coding.** This technique is performed at the transmitter by using more carriers than is strictly necessary. Similarly to the principle of the cyclic prefix, which allows elimination of the ISI by redundancy in the time domain, the redundancy in the frequency domain allows elimination of the ICI.

ISI, on the other hand, is caused by insufficient length of the *cyclic prefix (CP)*. An insufficient duration of the CP may arise for various reasons, for example, a system might shorten the CP length to improve the spectral efficiency. In this case, if the CP length is shorter than the maximum excess delay of the channel, ISI occurs and each OFDM symbol affects the subsequent symbol. This effect can be mitigated by decision feedback techniques [2].

Our contribution

In this work we investigate one *feedback technique* inspired by a variation of the redundant frequency coding technique used for ICI mitigation explained above. We will use this technique for reducing *narrow-band interference*; however it may easily be extended to include ICI or ISI reduction. Specifically, we

assume that the location of dominant interfering subcarriers is revealed to the transmitter (for example via feedback from the receiver). As a case study, we consider a system transmitting a frame of 64 OFDM subcarriers of which numbers 53, 16, 14, 21, 32, and 27 have been identified (by the receiver) as having substantial narrow-band interference. The transmitter simply nulls the data subcarriers (no transmit power) in these locations and relocates the data subcarriers to beginning and end carrier positions of the frame. Figure 3-1 shows the function of the OFDM switch.

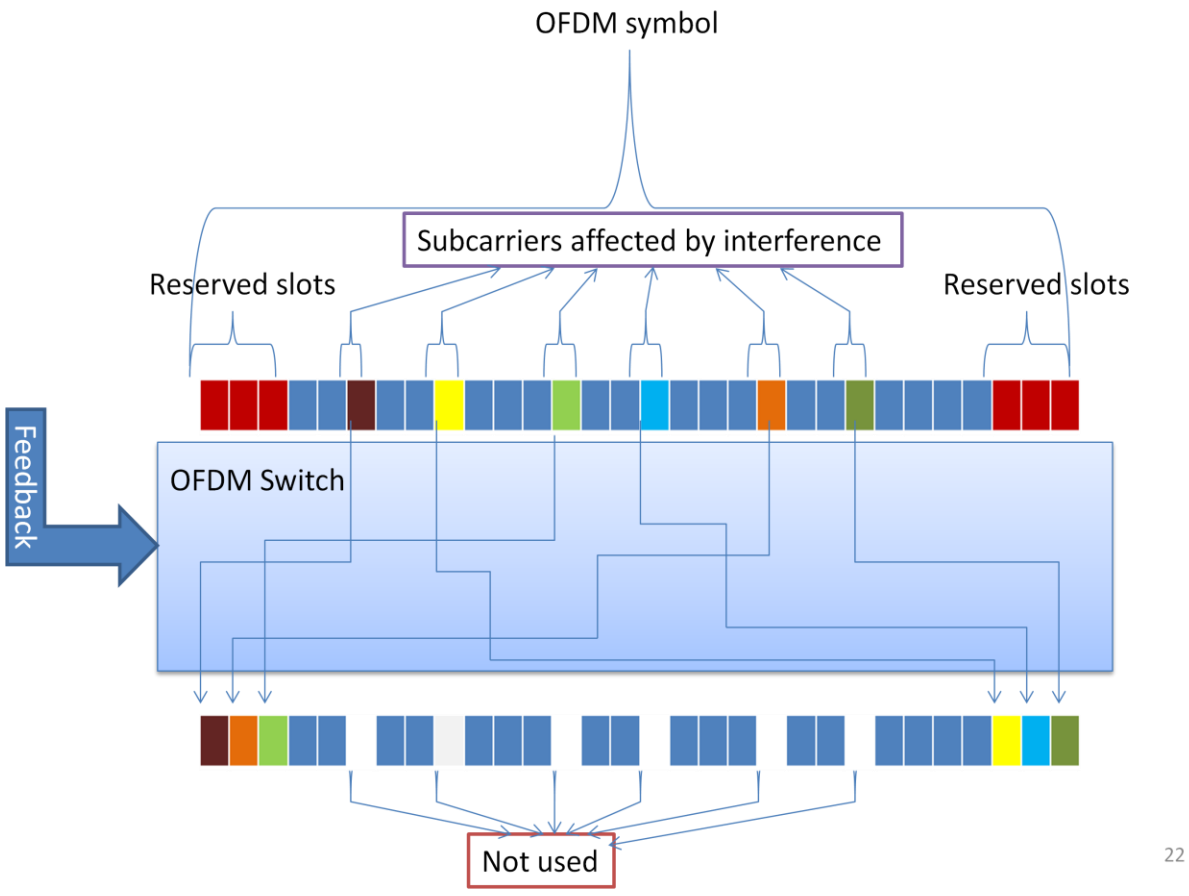

Figure 3-1 Function of the OFDM switch

3.1 Write subcarriers to the RAM then reorder when read

The design consists of 7 inputs, 6 of which are the location (index) of the detected interferences in the OFDM signal, and the seventh input acts as a synchronizer, that will tell the RAM block that there are data ready to be written to it. This input would be fed to the counter which is in charge of indexing addresses to the RAM block in addition to the “Write Enable” port of the RAM block. Each of the cell blocks of the RAM will be filled with its corresponding subcarrier symbol and since there are 64 subcarriers in the OFDM symbol (RAM should have depth of 64), it would take 64 cycles to get all 64 subcarrier symbols stored to the RAM. Our FPGA uses a clock at 100 MHz; therefore the “write to the RAM” process would take

$$64 \times \frac{1}{100\ Mhz} = 640\ ns = 0.64\ \mu s$$

After storing all the 64 subcarrier symbols, the data is read from the RAM in a specific order; therefore the “Write Enable” port should be set to false or zero to read the RAM content. In order to read the RAM in a specific order, we have to manipulate the order of the index that is fed to the RAM, and this is done by designing a state machine using an MCode block with 7 inputs and 2 outputs. The first input would be a free running counter with 6 bits resolution (0-64 counter), and the remaining inputs are the location of the detected interferences. The MCode block will check the counter value to the value of the destinations of the interferences (these destinations are in the order 2,1,61,62,0,63 (these values are simply an example)). If the counter value is equal to the destinations

values, the first output of the block will be the value of the inputs, otherwise the first output will be the value of the counter. The second output is for nulling the position of the interferences; therefore, we set the second output to 1 whenever the counter is equal to any of the interference positions, and later this would be used to null those positions. The read from the RAM would take about:

$$64 \times \frac{1}{100\ Mhz} = 640\ ns = 0.64\ \mu s$$

There is also one cycle delay in the RAM block, therefore the switching would take about 640ns+640ns+10ns=1290ns=1.29 µs.

3.1.1 Resource usage

If we have enough time between each of the 64 subcarrier symbols, we can use the above design to reorder these subcarriers. However, if the time between each 64 subcarriers is small, we need to use 2 or more of the above blocks to construct parallel processes (Figure 3-2). Figure 3-3 show the resources that are used when there is enough time between 2 consecutive signals. Figure 3-4 shows the overall view of this design.

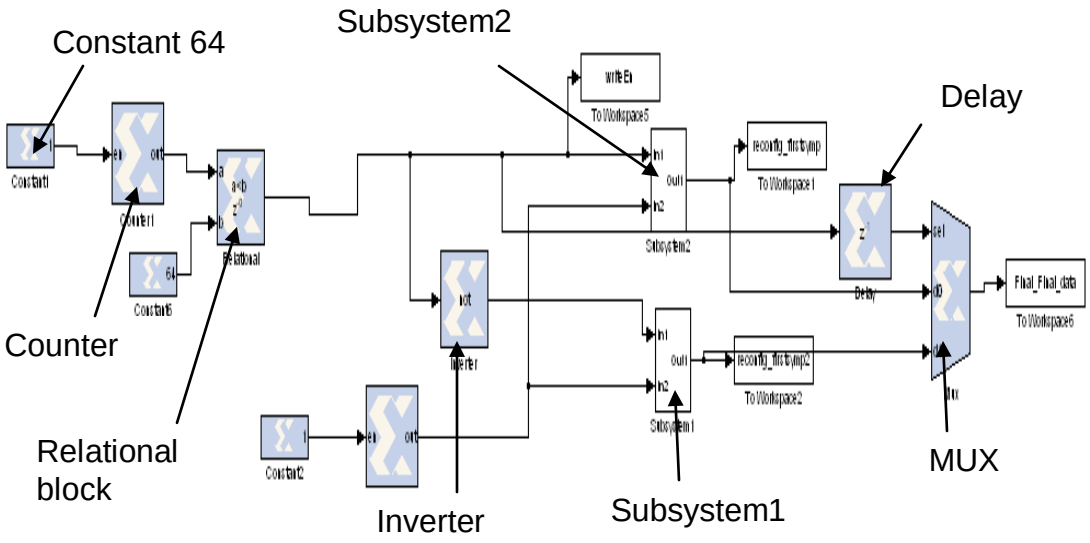

Figure 3-2 Pipelining between 2 OFDM switch

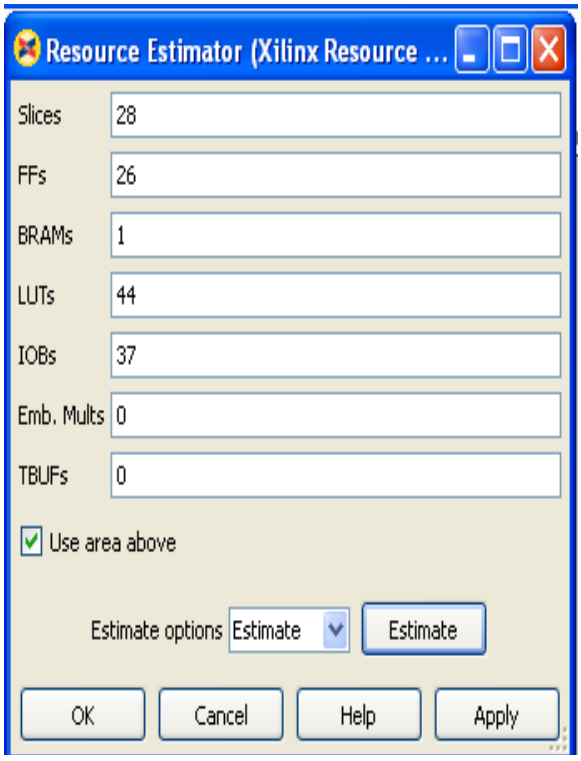

Figure 3-3 Resource usage for simple index manipulation

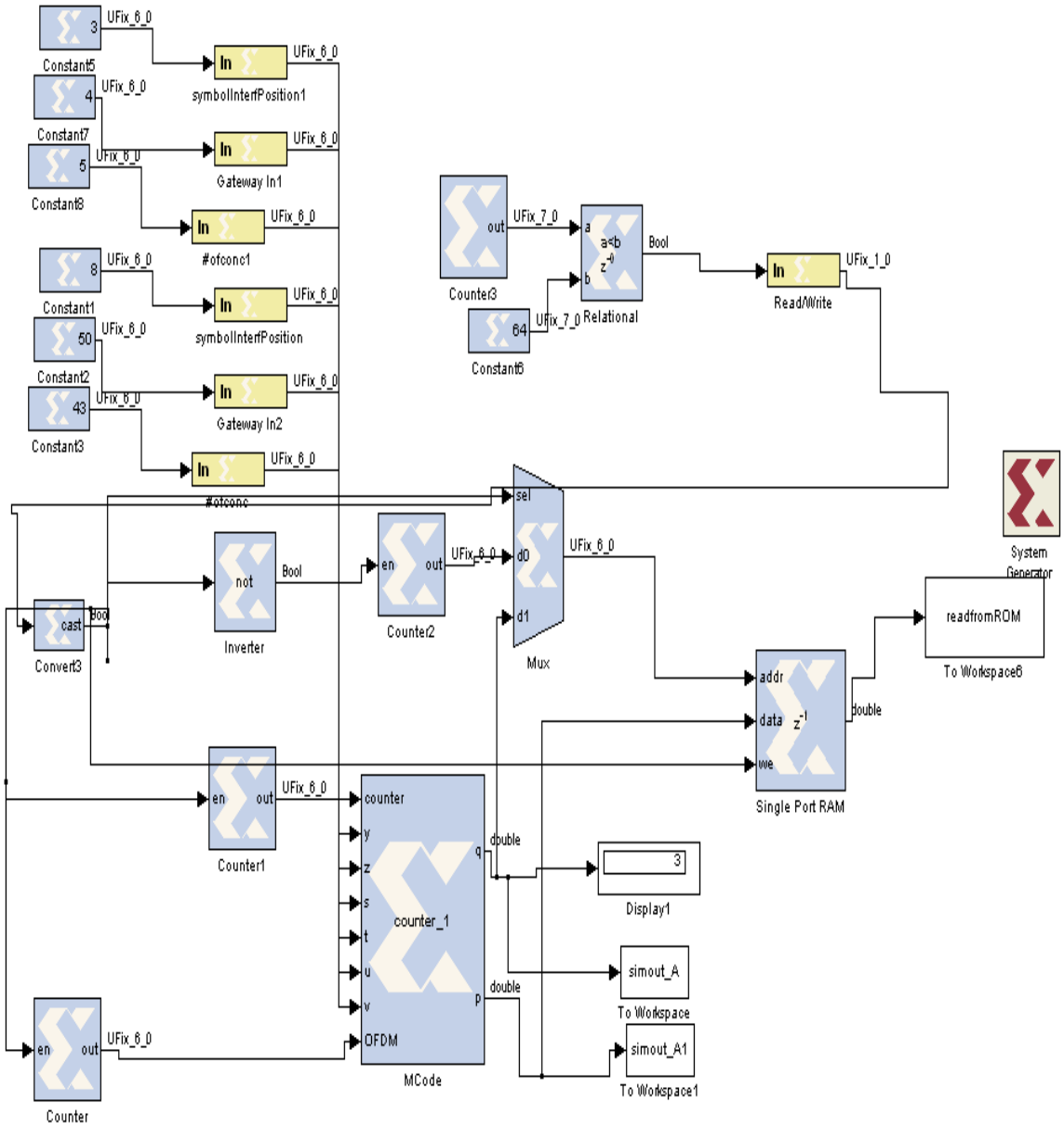

Figure 3-4 Index manipulations for symbol reconfiguration

3.2 Write the reordered subcarriers to RAM block and then Read

In this method, the subcarriers get reconfigured before they are written to the RAM block. This method would perform better than the previous method,

because we do not write subcarriers 0,1,2,61,62,63 to the RAM. In those time slots (counter 0,1,2,61,62,63), we nullify (make them zero) the positions of the subcarriers that were affected by interference. This will eliminate the delay that the multiplexer imposed in the previous section. In addition to the better performance, this module will store the reconfigured data in the RAM and the RAM can be read by our request, so the data is not lost if the next algorithm loses the data. All the possible scenarios have been taken into account in the MCode block (eg when some inputs are equal to zero). Figure 3-5 shows our overall design, which is called an OFDM Switch.

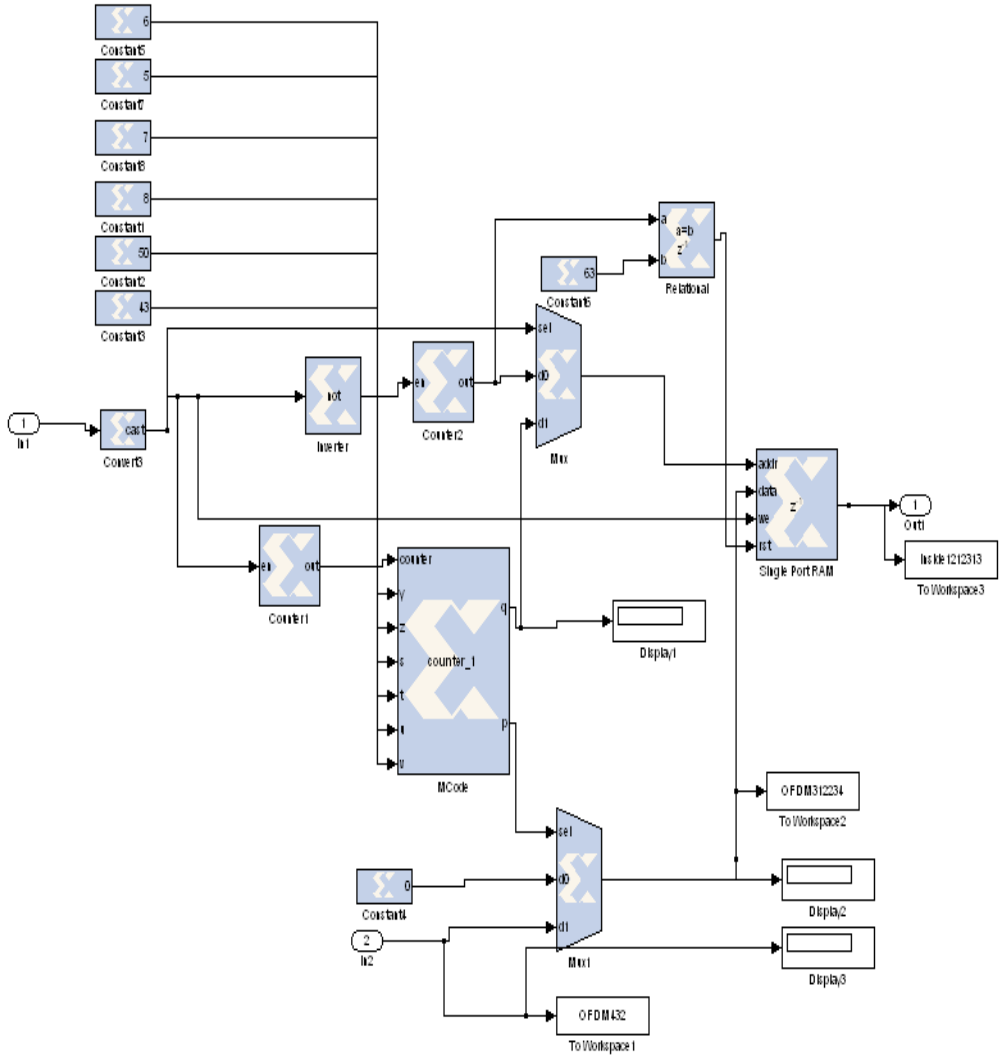

Figure 3-5 Improved index manipulations for symbol reconfiguration

Further design changes have been undertaken in order to use OFDM signal. One major change is that we do not feed the OFDM signal to the MCode Block. This is because MCode block will only accept fixed point numbers and the values from the output of this are also a fixed point numbers, therefore saturation and rounding of the OFDM signal cannot be tolerated. This issue can be fixed by using an extra multiplexer (which uses a fixed point number from the MCode Block) for nullifying (making the values of the positions of the subcarriers that are

affected by the interference zero (zero transmit power) the locations of the interferences. As we can see, the MCode block is only used for address (index) manipulation of the subcarriers. The mechanism of the MCode block is controlled by the counter, which is in turn controlled by a Read/Write signal. When the Read/Write signal is set, the counter starts feeding 0-64 to the MCode block, and the MCode block will start outputting addresses from output q in addition to output p, which controls the multiplexer for carrier symbol nulling when the interference is present. When the clock value is 0,1,2,61,62,63, it means that the OFDM symbol at that instance is one of the reserve slots; therefore, we can use these 6 time slots to nullify the positions of the interference. We do this by outputting the interference positions (inputs y,z,s,t,u,v of the Mcode block) at output q and outputting “0” at output p (which controls the multiplexer for nullifying). When the counter is at value y,z,s,t,u,v, it means that the OFDM subcarriers at those instances are the ones that have been affected by the interference, therefore we need to move these subcarriers to the reserve slots. We do this by outputting the value 2,1,61,62,0,63 at output q, which puts the bad OFDM subcarriers in the reserve slots. When the Read/Write signal is zero, the RAM will go into read mode and a simple counter reads the values of the RAM cells (0-64). In the case that all the signals (each with 64 OFDM subcarriers) start one after another, and pipelining to 2 different OFDM switches can be implemented, we will be able to process consecutive signals. Figure 3-6 illustrates this pipeline. [8]

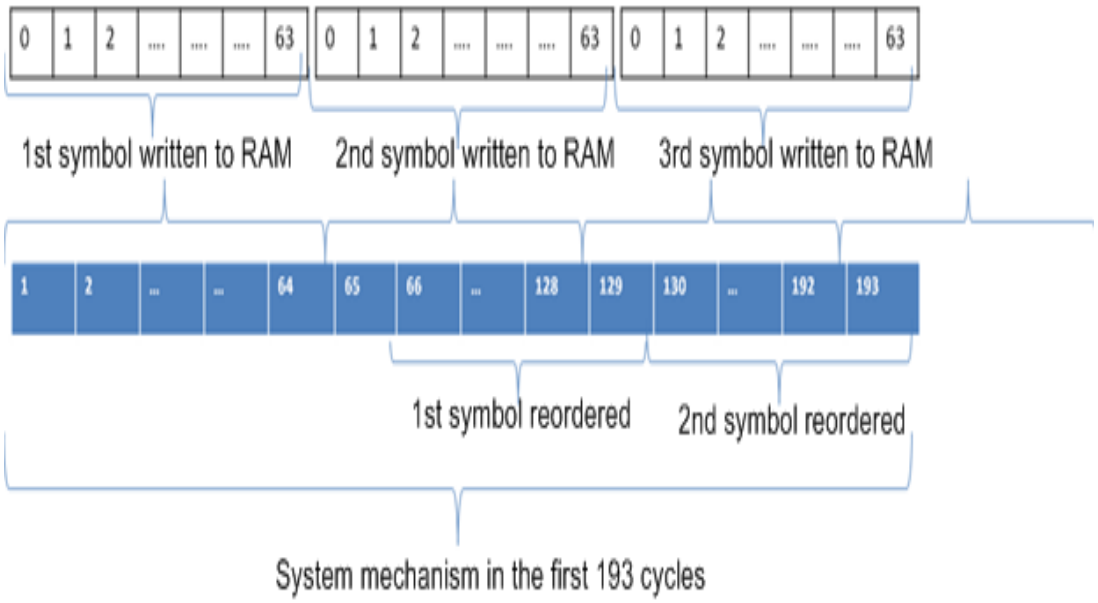

Figure 3-6 Symbols with timing

Figure 3-7 shows the mechanism of the state machine.

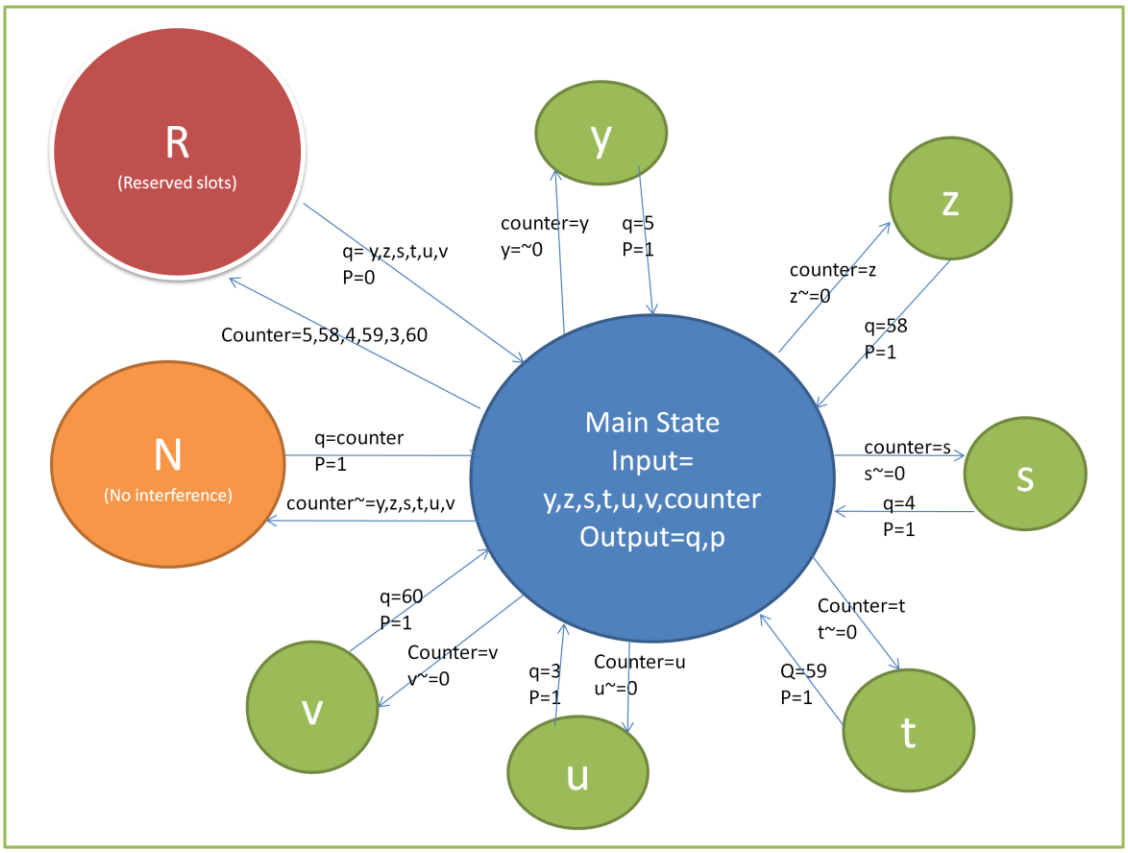

Figure 3-7 Mechanism of the state machine

Further synchronization and initialization of the blocks has been implemented by adding a RAM reset/Initialization port, and controlling it by the control counter. Figure 3-8 shows the Xilinx resource estimator.

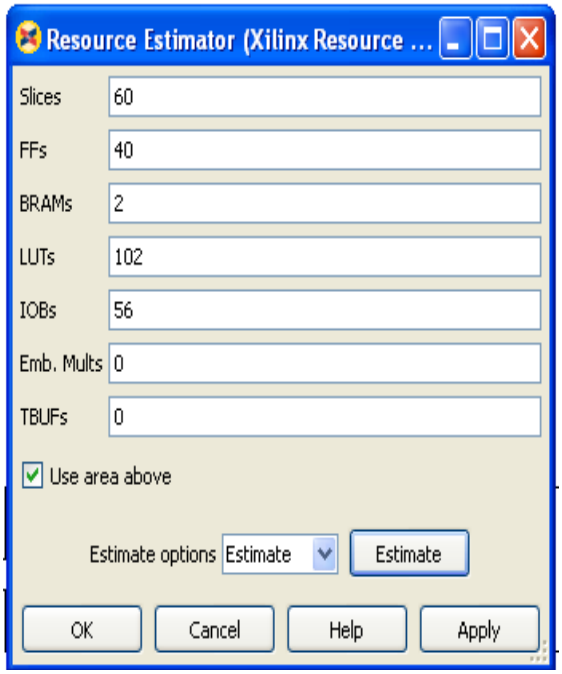

Figure 3-8 Resource Estimatation

3.3 Optimization

3.3.1 Mcode block (OFDM Sub Carrier Addr Reconfigure) relocation

Further optimization has been implemented by taking the MCODE block (OFDM Sub Carrier Addr Reconfigure) out of each subsystem that we implemented for pipelining consecutive signals. This is achievable because the Mcode block is used when the subsystems are in write mode, and since only one of the subsystems need to be in write mode at any given time, we can take the Mcode block out and use it for both subsystems. Figure 3-9 shows the pipelining between RAM blocks and the single Mcode block implementation.

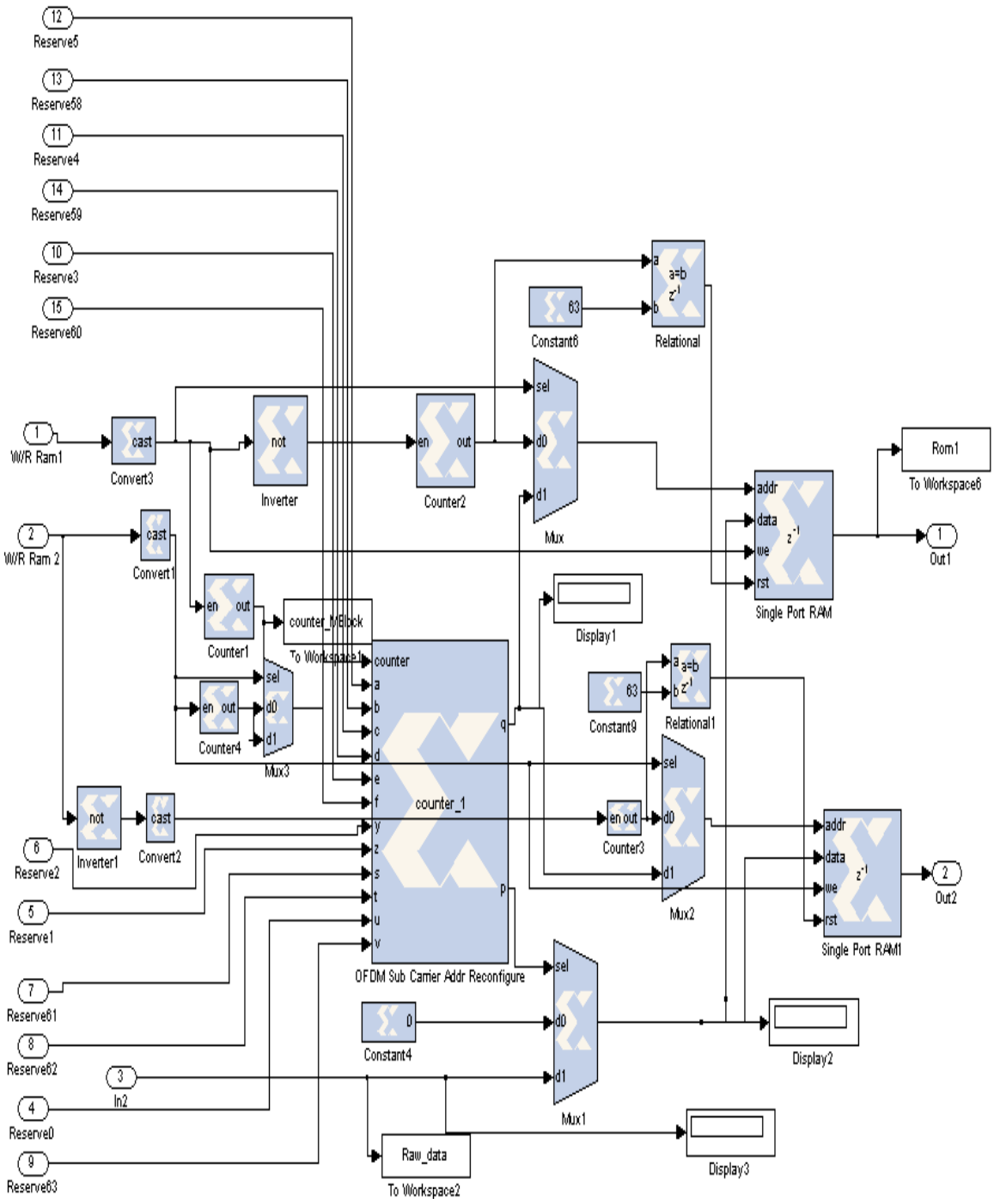

Figure 3-9 Pipelining between RAM blocks and single Mcode block

Figure 3-10 shows the Xilinx resource estimator of the system after optimization. As we can see the system uses the same number of FF, BRAM, SLICES, FFS and LUTs.

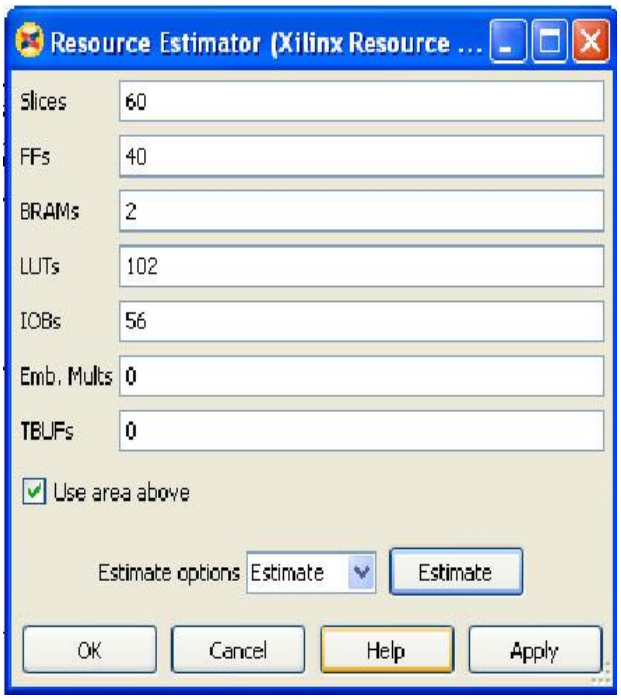

Figure 3-10 Resource Estimation

3.3.2 Implementation of the OFDM Switch using DUAL PORT RAM

Instead of using 2 BRAMs, we have used a Dual Port RAM. This is done with almost the identical structure as 2 BRAMs; the only difference that was noticed in this design is that we will have some limitation in terms of accessing cells while reading from them. The Dual Port RAM can be used in 3 different modes:

1. Read after write
2. Read before write
3. No read on write

The Read after write mode will not work in our design since it reads out each cell right after the data is written to it. Therefore, since we reconfigure the addresses in order to reconfigure the subcarriers, we need to read the content of the whole RAM from 0-63.

The Read before write is a good option to choose, because this option would read all the values from 0-63 before writing to them, and as a result the system would be less prone to data collision. However, the limitation would be that if the signals have some delay between them, it means that the first signal would be available at the output right after the second signal is at the input of the system. If the delay between these signals is variable, then we would have a different timing in the output.

The No read on write mode will impose write-read collisions if the OFDM signals have no delays between them. This is because the BRAM 1 starts reading out the reordered subcarriers while the system starts writing the new subcarriers to BRAM2, therefore reading from a cell and writing to the same cell will cause data collision, which will show an output “nan” (Not a Number). Figure 3-11 shows the dual port RAM implementation approach:

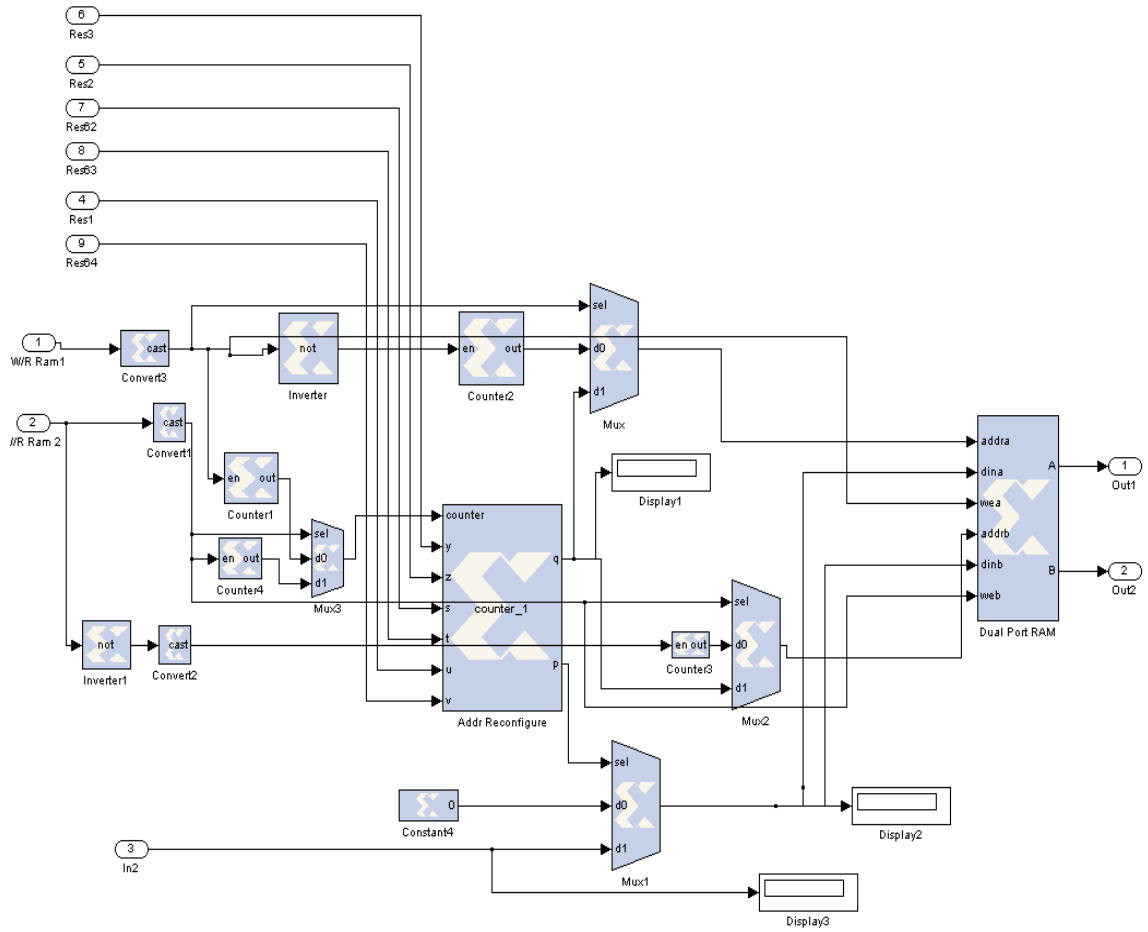

Figure 3-11 OFDM Switch using Dual port RAM

3.4 Testbed

A Testbed has been developed in Visual Studio C# Express [30] in order to test the system with random numbers. Figure 3-12 shows the screen shot of the GUI of the testbed.

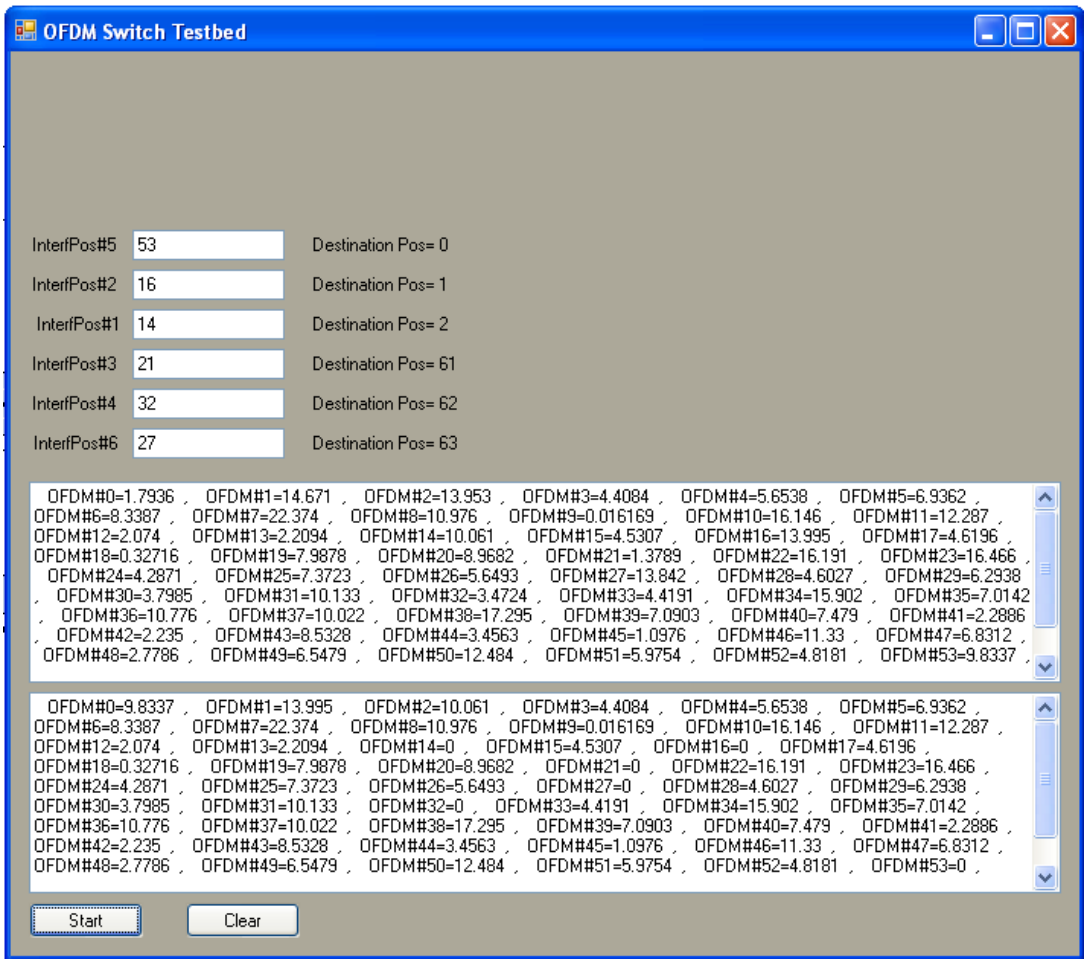

Figure 3-12 C# Testbed

3.4.1 Comparing the System Generator result with our testbed

Figure 3-13 shows the result from our testbed and the result from simulation.

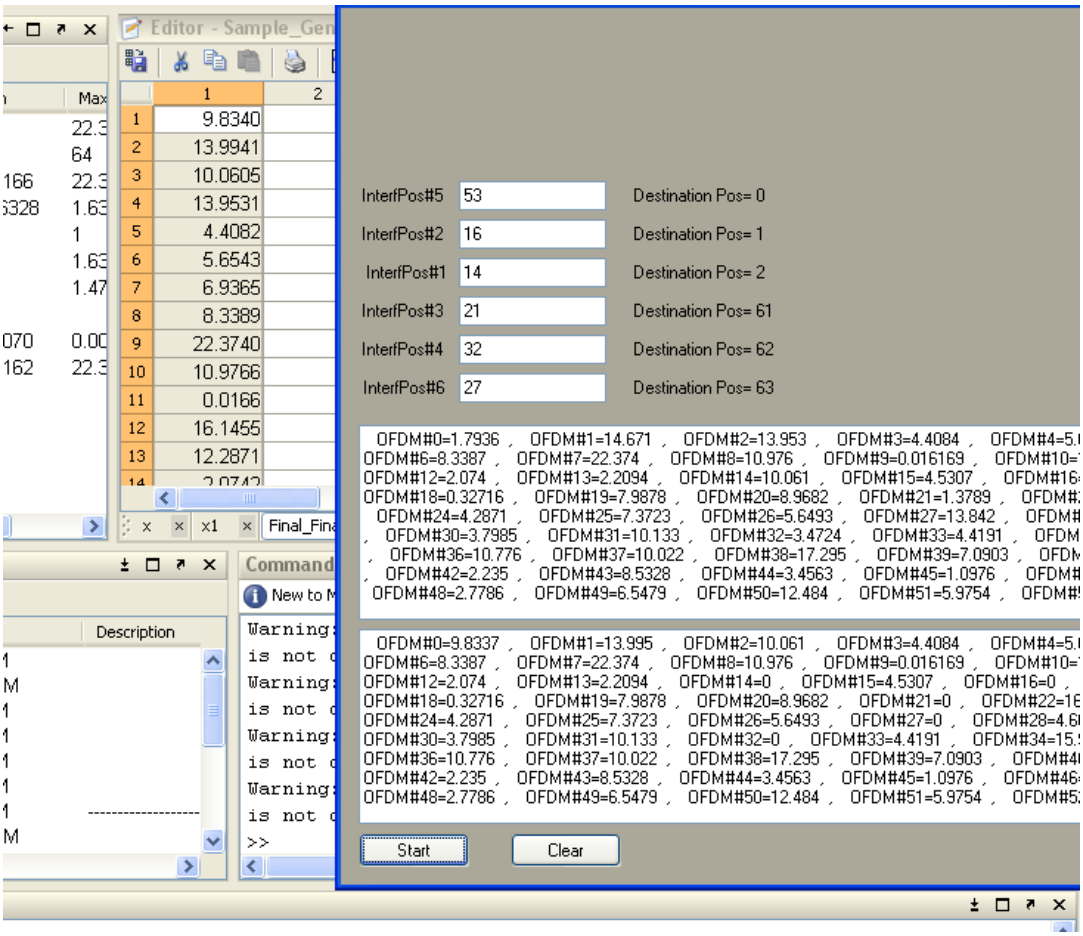

Figure 3-13 Result from testbed and result from simulation

3.5 Development for 802.11a

Since the 802.11a contains 12 reserved subcarrier spots (6 at each end of the spectrum), we have extended our design in order to use these 12 spots.

3.6 Mechanism of the OFDM Switch

The “12 Symbol OFDM Switch” needs about 15 inputs. The first two inputs are the W/R enabled signals for the 2 block RAMs. In the test program we have defined a counter that runs from 0-127 with an explicit sample period of 20 MSPS

and a relational block with the constant 64 inputted to its port b. This algorithm generates “1” in cycle 1-64, therefore the first set of OFDM subcarriers are available for the first Block RAM; since it is fed to “W/R Ram 1”. Furthermore having value 1 fed to W/R Ram 1 puts the first Block RAM in the write mode. The above algorithm generates 0 from the cycles 65-128, which puts the first Block RAM in Read mode. Since the negated version of this signal is fed to the “W/R Ram 2”, Ram 2 will always be in the opposite mode from RAM 1, thereby making the system capable of receiving the consecutive 64 OFDM subcarriers. The next input is the OFDM subcarriers. In our test program we have used a counter to generate the OFDM data. The next 12 inputs to the OFDM Switch are the locations of the interference. These inputs have to be between 0-63.

3.6.1 Returning the OFDM subcarriers to their original locations

At the receiver side we need to put the subcarriers back to their original positions. The main algorithm for the state machine stays the same, therefore after moving to each state, the system goes back to the main state. The only difference in this case is the function of the state machine (Mcode block). As we can see from the following figure, the inputs to the state machine are the same, so the positions of the subcarriers with interference and a counter are the input to the state machine. The only difference is that when the counter is at the reserved symbol position, the output q will have the address of the location of the interference and output p will be “1” which chooses the reconfigured OFDM signal over “0”. This is because when the counter is equal to the position of the

reserved subcarriers, the OFDM signal at that point carries the value that was stored in the symbol that was affected by interference; therefore, by outputting the location and knowing the value of it, we can write that value to its original location. Another difference is that when the counter is equal to the position value of the interferences, the system outputs the values of the reserved subcarriers at its output q and 0 at its output p. This will nullify the reserved locations in our final OFDM signal. Figure 3-14 illustrates the overall function of this state machine.

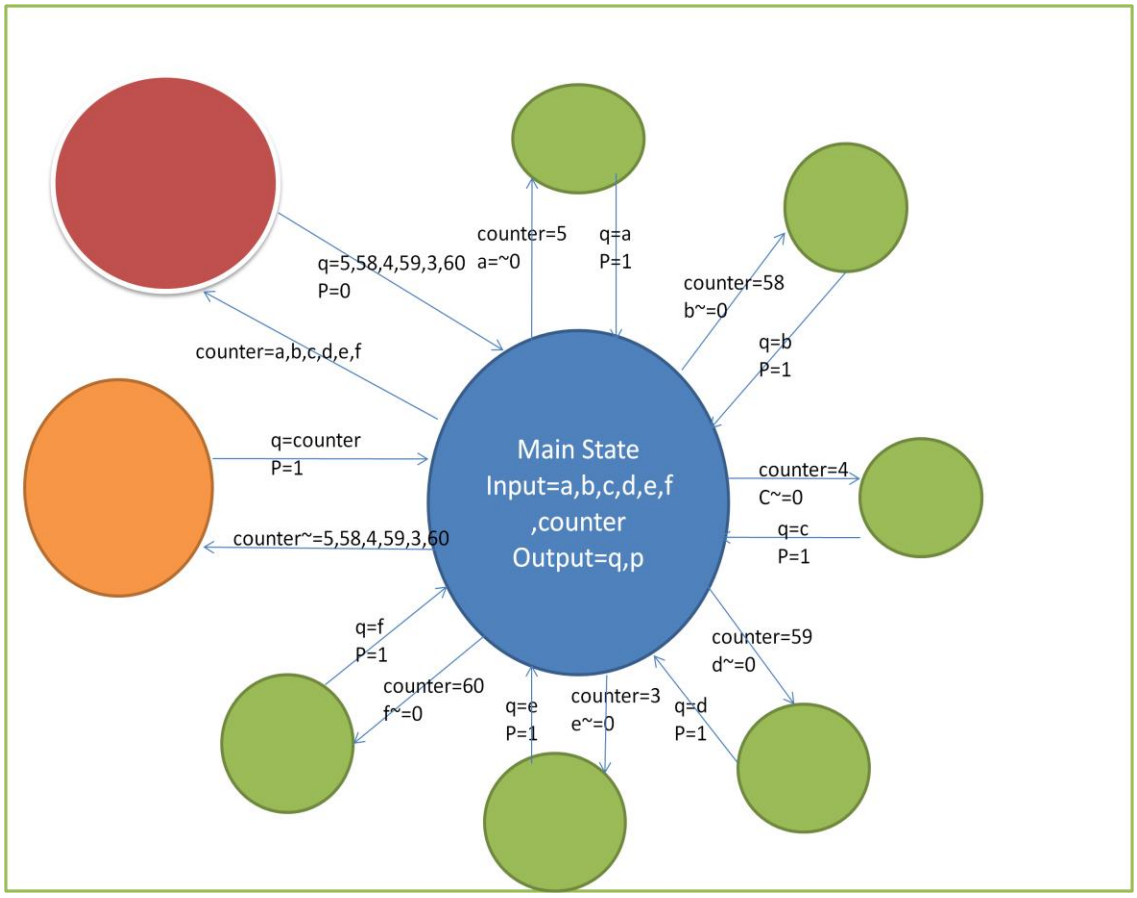

Figure 3-14 Subcarriers to their original position

3.7 Synchronous Reset Generator

The synchronous reset generator is not required if the system receives the OFDM signal on the first cycle. If an OFDM signal is not present in the first cycle, then the reset signal is fed to the RAM Blocks and the counters. When the OFDM signal is delayed by one or more cycles then the system needs to be reset in order for the receiver to be synchronized. Figure 3-15 shows the design that uses the Synchronous Reset Generator.

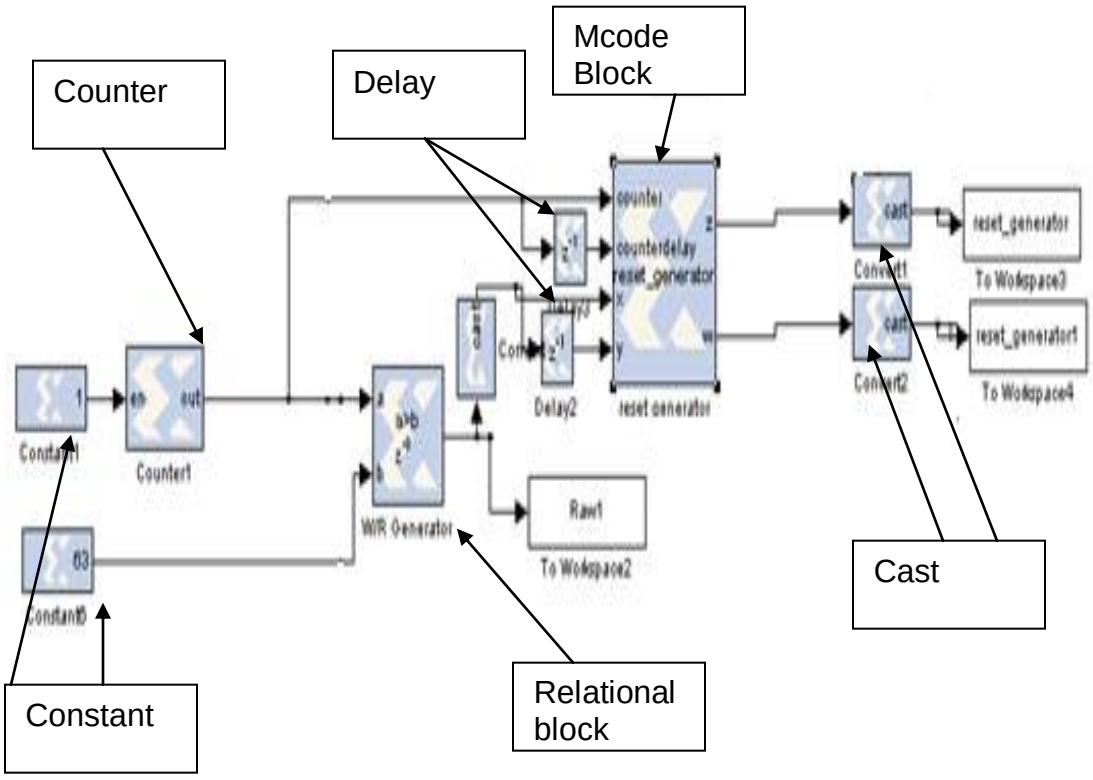

Figure 3-15 Synchronous Reset Circuit

Figure 3-16 illustrates the state machine that generates the reset signal for our algorithm.

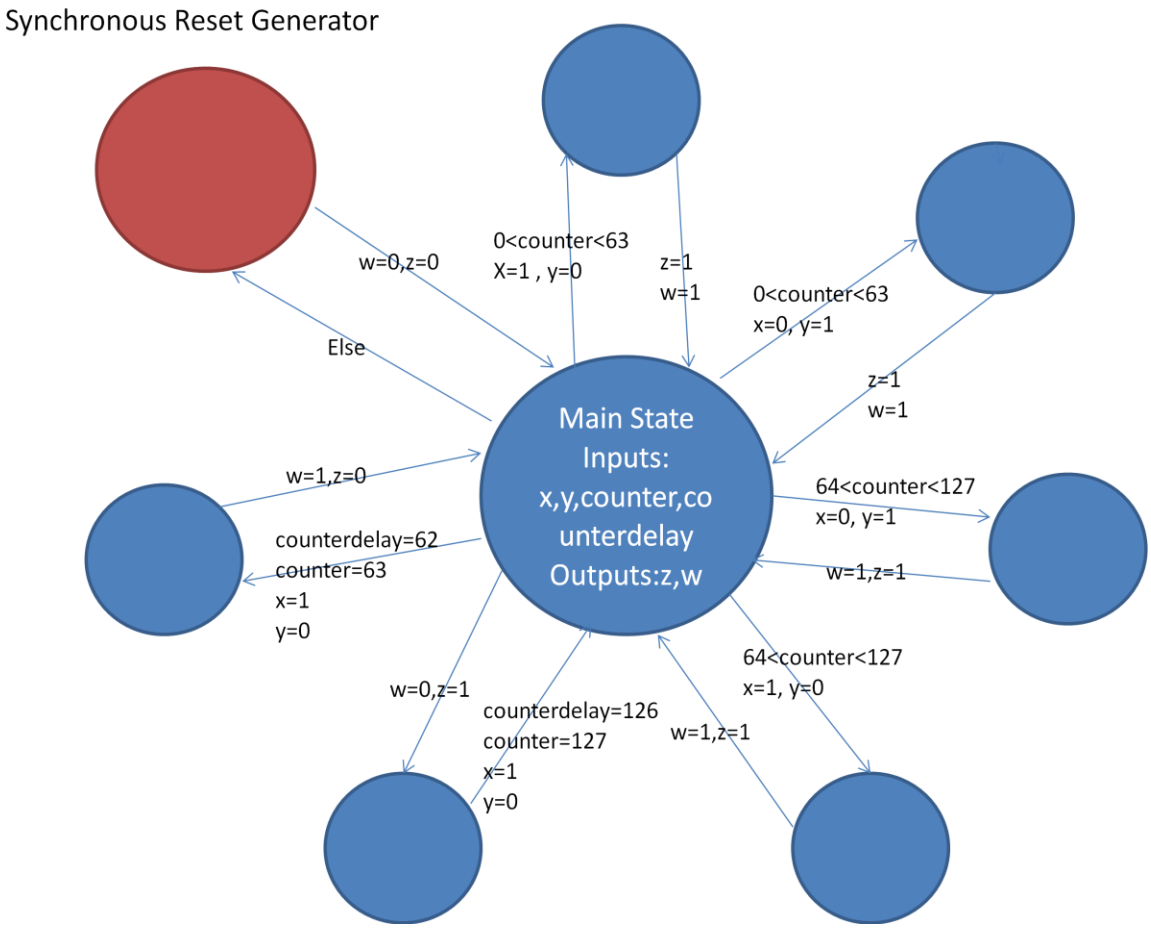

Figure 3-16 Synchronous reset state machine

4: Convolver and Correlator

Another DSP subsystem implemented is a convolver. A convolver is central to the concept of modelling channels, including time varying channels. The following is based on [9]. A communications channel is conventionally modeled as a *continuous-time* impulse response such as

$$\boldsymbol{h}\,(\boldsymbol{t};\boldsymbol{\tau}) = \sum_{\boldsymbol{l}=1}^{\boldsymbol{L}} \boldsymbol{\alpha}_{\boldsymbol{l}}(\boldsymbol{\tau}_{\boldsymbol{l}})\boldsymbol{\delta}(\boldsymbol{\tau} - \boldsymbol{\tau}_{\boldsymbol{l}}) \tag{4-1}$$

Where α_l is the first signal and δ is the second signal and τ_l is the time.

Note that the tap discretization is fixed in time (τ_l), and these are called time bins.

An input signal to this channel is convolved with the channel function above. Such a mixing of signals therefore requires similar de-convolving operations at the receiver in order to recover the original signal (a process also known as equalization). Even in systems with a simple receiver structure, including those that are preceded by OFDM protocol, convolution modules are important as they present a means of *post-processing* signals, for example to reduce residual interference [9] at the receiver. This is particularly important for a channel that introduces ICI or ISI (this is the usual reason for deploying OFDM) as explained in Chapter 3. Given the high transmission rates, the receiver must perform the equalization quickly, and more importantly, in *real-time* fashion, i.e., the latency should be low. This presents a major challenge as some DSP

applications such as convolution and correlation have very intensive computations; our Xtream DSP board, as with most dedicated DSP modules, has very powerful computation capabilities. In this chapter, we begin by designing a convolver with different taps and then we study the complexity of our design and the capabilities of different FPGA families to implement these designs.

4.1 Convolver

As we will see in Section 4.4, we can design the convolution function using the FIR Compiler Core, but this core is not suitable for designing the real-time moving window convolver/correlator. This is because of the limitation of the FIR compiler core to incorporate a moving window. The option of having a reloadable weight is limited to 2 accumulators. When either accumulator is full, the next set of data will be fetched to the other accumulator, meaning these two accumulators are working independently. (When accumulator B is being loaded with new weights, accumulator A is being use for the convolution computation.) Therefore, it is not obvious that a moving window convolver can be implemented using the FIR Compiler.

Our proposed design involves 2 delay lines and multipliers and summation blocks. Figure 4-2 illustrates our design (note that the delay lines are aligned in opposite directions to each other, which allows us to time-reverse one of our signals).

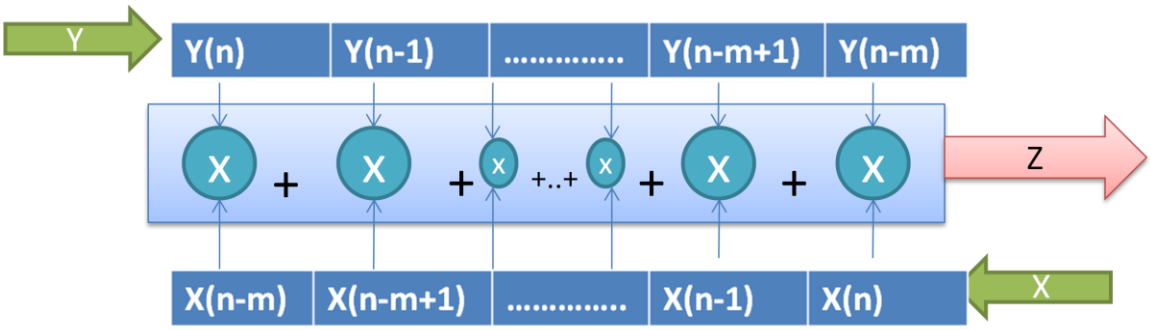

Figure 4-1 Convolver design

The design starts from a 16 tap convolver. By replicating this and adding an add-sub block, we get the 32, 64,128,256 and 1024 tap convolver.

Our design is optimized for operation speed. The total amount of delay for our system is sum of delays in the delay line (size of the window) plus 4 cycle delays for the multiplication stage.

Each block from the system generator consumes different resources for different configurations, therefore it is vital to study each block's resource usage. Table 4-1 illustrates the resources used in different settings for the delay block. Table 4-2 illustrates the resources used in different settings for the multiplication block. Table 4-3 illustrates the resources used in different settings for the add/sub block.

Table 4-1 Resource used in different settings for a single delay block.

z^{-1}	4 Bits Inputs	8 Bits Inputs	16 Bits Inputs	24 Bits Inputs
Slices	2	4	8	12
FFs	4	8	16	24

Table 4-2 Resource used in different settings for multiplication block.

Mult	4 Bits Inputs	8 Bits Inputs	16 Bits Inputs	24 Bits Inputs
Slices	8	16	24	105
FFs	16	32	48	173
LUT	8	16	24	170
Emb Mult	1	1	1	4

Table 4-3 Resource used in different settings for addition block.

Add/Sub	8 Bits Inputs	9 Bits Inputs	10 Bits Inputs	11 Bits Inputs
Slices	5	5	6	6
Add/Sub	12 Bits Inputs	13 Bits Inputs	14 Bits Inputs	15 Bits Inputs
Slices	7	7	8	8
Add/Sub	16 Bits Inputs	17 Bits Inputs	18 Bits Inputs	19 Bits Inputs
Slices	9	9	10	10
Add/Sub	20 Bits Inputs	21 Bits Inputs	22 Bits Inputs	23 Bits Inputs
Slices	11	11	12	12
Add/Sub	24 Bits Inputs	25 Bits Inputs	32 Bits Inputs	33 Bits Inputs
Slices	13	13	17	17
Add/Sub	34 Bits Inputs	35 Bits Inputs	36 Bits Inputs	37 Bits Inputs
Slices	18	18	19	19
Add/Sub	38 Bits Inputs	39 Bits Inputs	40 Bits Inputs	41 Bits Inputs
Slices	20	20	21	21
Add/Sub	48 Bits Inputs	49 Bits Inputs	50 Bits Inputs	51 Bits Inputs
Slices	25	25	26	26
Add/Sub	52 Bits Inputs	53 Bits Inputs	54 Bits Inputs	55 Bits Inputs
Slices	27	27	28	28
Add/Sub	56 Bits Inputs	57 Bits Inputs	58 Bits Inputs	59 Bits Inputs
Slices	29	29	30	30
Add/Sub	60 Bits Inputs	61 Bits Inputs	62 Bits Inputs	63 Bits Inputs
Slices	31	31	32	32

The parameters of our interest in resource estimation are the slices. Each slice contains two flip-flops, two LUTs and associated Mux, as well as carry and control logic which can easily be mapped to each FPGA family's available resources. Table 4-4 shows the type of delay, multiplication block and add-sub block used in the 16 tap convolver.

Table 4-4 Type of delay, multiplication block and add-sub block used in the 16 tap convolver. (The z-1 denotes a single delay)

16 Taps Convolver	4 Bits Signal	8 Bits Signal	16 Bits Signal	24 Bits Signal
15 Delays	z^{-1}4 bits inputs	z^{-1}8 bits inputs	z^{-1} 16 bits inputs	z^{-1} 24 bits inputs
16 Multiplications	4 bits inputs	8 bits inputs	16 bits inputs	24 bits inputs
8 Add/Sub	8 bits inputs	16 bits inputs	32 bits inputs	48 bits inputs
4 Add/Sub	9 bits inputs	17 bits inputs	33 bits inputs	49 bits inputs
2 Add/Sub	10 bits inputs	18 bits inputs	34 bits inputs	50 bits inputs
1 Add/Sub	11 bits inputs	19 bits inputs	35 bits inputs	51 bits inputs

The 32 tap convolver has two 16 tap convolvers and an extra add/sub block; therefore, with this type of formulation we can calculate the number of blocks in the 32, 64,128,256, 512 and 1024 tap convolvers, and in turn calculate the resources used for each of these designs. With the above formulation we can generate Table 4-5 for the 32 tap convolver, Table 4-6 for the 64 tap, and so on.

Table 4-5 Type of delay, multiplication block and add-sub block used in the 64 tap convolver.(The z-1 denotes a single delay)

32 Taps Convolver	4 Bits Signal	4 Bits Signal	4 Bits Signal	4 Bits Signal
30 Delays	z^{-1}4 bits inputs	z^{-1}8 bits inputs	z^{-1} 16 bits inputs	z^{-1} 24 bits inputs
32 Multiplications	4 bits inputs	8 bits inputs	16 bits inputs	24 bits inputs
16 Add/Sub	8 bits inputs	16 bits inputs	32 bits inputs	48 bits inputs
8 Add/Sub	9 bits inputs	17 bits inputs	33 bits inputs	49 bits inputs
4 Add/Sub	10 bits inputs	18 bits inputs	34 bits inputs	50 bits inputs
2 Add/Sub	11 bits inputs	19 bits inputs	35 bits inputs	51 bits inputs
1 Add/Sub	12 bits inputs	20 bits inputs	36 bits inputs	52 bits inputs

Table 4-6 Type of delay, multiplication block and add-sub block used in the 16 tap convolver.(The z-1 denotes a single delay)

64 Taps Convolver	4 Bits Signal	4 Bits Signal	4 Bits Signal	4 Bits Signal
60 Delays	z^{-1}4 bits inputs	z^{-1}8 bits inputs	z^{-1} 16 bits inputs	z^{-1} 24 bits inputs
64 Multiplications	4 bits inputs	8 bits inputs	16 bits inputs	24 bits inputs
32 Add/Sub	8 bits inputs	16 bits inputs	32 bits inputs	48 bits inputs
16 Add/Sub	9 bits inputs	17 bits inputs	33 bits inputs	49 bits inputs
8 Add/Sub	10 bits inputs	18 bits inputs	34 bits inputs	50 bits inputs
4 Add/Sub	11 bits inputs	19 bits inputs	35 bits inputs	51 bits inputs
2 Add/Sub	12 bits inputs	20 bits inputs	36 bits inputs	52 bits inputs
1 Add/Sub	13 bits inputs	21 bits inputs	37 bits inputs	53 bits inputs

The same methodology can be applied for more complicated systems. Using these tables and Table 44, 45 and 46, we can calculate the total resource usage of each design.

Figure 4-2 summarizes the complexity of the system at a different bit resolution for a different number of taps. The value of the slices for the 64 bits input signal of the 1024 tap convolver is a guess. This is because this operation is very resource intensive, and the design computer “freezes” when trying to estimate the resources for this convolver. The figure offers insight for the convolver design complexity. The number of bits used relates to the signal fidelity, any limitations on power consumption (not considered here), and the processor capability. The number of bits also depends on the avaiable capability of the processor. Current technology seems to be limited to 32 bit (fixed point) operations. The 64-bit implementation is not possible on current (Virtex 4) hardware.

As the graph indicates, the best Virtex 4 chip can handle up to 1024 taps at 16 bits. We cannot draw strong conclusions for the Virtex5 and Virtex 6 family. These families have a far more advanced architecture than the Virtex 4 in that they use far fewer resources even with the same design. The Virtex 5 slice contains four LUTs and four flip flops, and the Virtex-6 slice contains four LUTs and eight flip-flops. However, the major difference comes from the fact the DSP48E1 slice of these two families contains a 25x18 multiplier, whereas in the Virtex 4 family it is only 18x18. In addition to these, there are new IP cores which

work only with Virtex 5 and 6. Future processors will be even more powerful, the trend appears to be continuing to follow Moore's law.

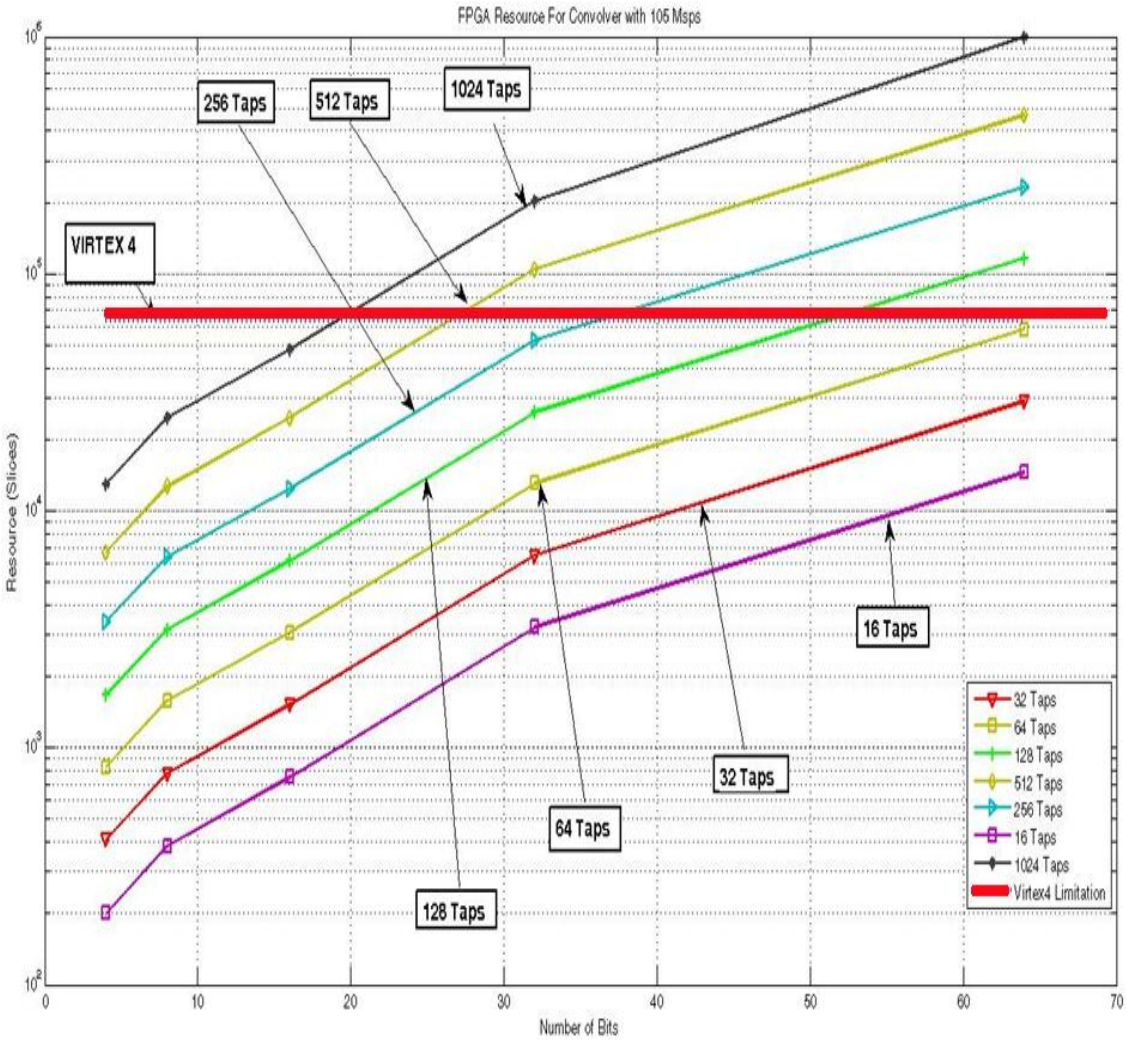

Figure 4-2 FPGA resources usage for convolver (105 MSPS)

4.2 Optimization in term of Area using a Multi-Rate system

Although the previous design is straight forward, it can accept samples at 105 MSPS (Maximum Speed of the board) and there is no room for improvement in resource usage. It is of interest to investigate the performance for lower

bandwidth (slower sampling rate) input signals. Here, we consider 96 KSPS, used widely in high fidelity audio systems. Between consecutive input samples, there is a large delay in terms of the 105MHz clock rate.

The Number of Cycles Between consecutive samples can be calculated using the following equation:

$$\textbf{Number of samples} = \frac{\textbf{105 MSPS}}{\textbf{96 KSPS}} = \textbf{1093.75} \qquad (4\text{-}2)$$

Having 1093 cycles in hand will allow us to have 1093 operations. Therefore, we can reduce the number of multiplication blocks and add-sub blocks using just one multiplier instead of several as above. The 1024 tap convolver of the previous section is the best candidate for this design. Figure 4-3 illustrates this new design.

As shown in the above figure, input signals are fed to subsystem1 and subsystem2. Subsystem2 contains 1024 delays and 33 multiplexers, and each of the delay blocks are using a slower clock rate (96 KSPS) to buffer the input signals. The main complication comes from the fact that the maximum number of inputs for the System Generator's multiplexer is 32, and since we have 1024 taps, we need thirty-two 32-to-1 input multiplexers; this gives 32 output signals as well as another 32-to-1 multiplexer to have a signal output for the multiplication stage. The main issue with this design is timing and controlling the Selection signal of the multiplexers at the right clock cycle. To do this, we design a state machine that generates the right code to control each of the multiplexers using an Mcode block. We feed the Mcode block with a counter that runs from 0 to 1023. The Mcode block uses the counter value to output two control signals, one

for the thirty-two input multiplexers and one for the single output multiplexer. Figure 4-4 illustrates a small part of the mechanism of our state machine.

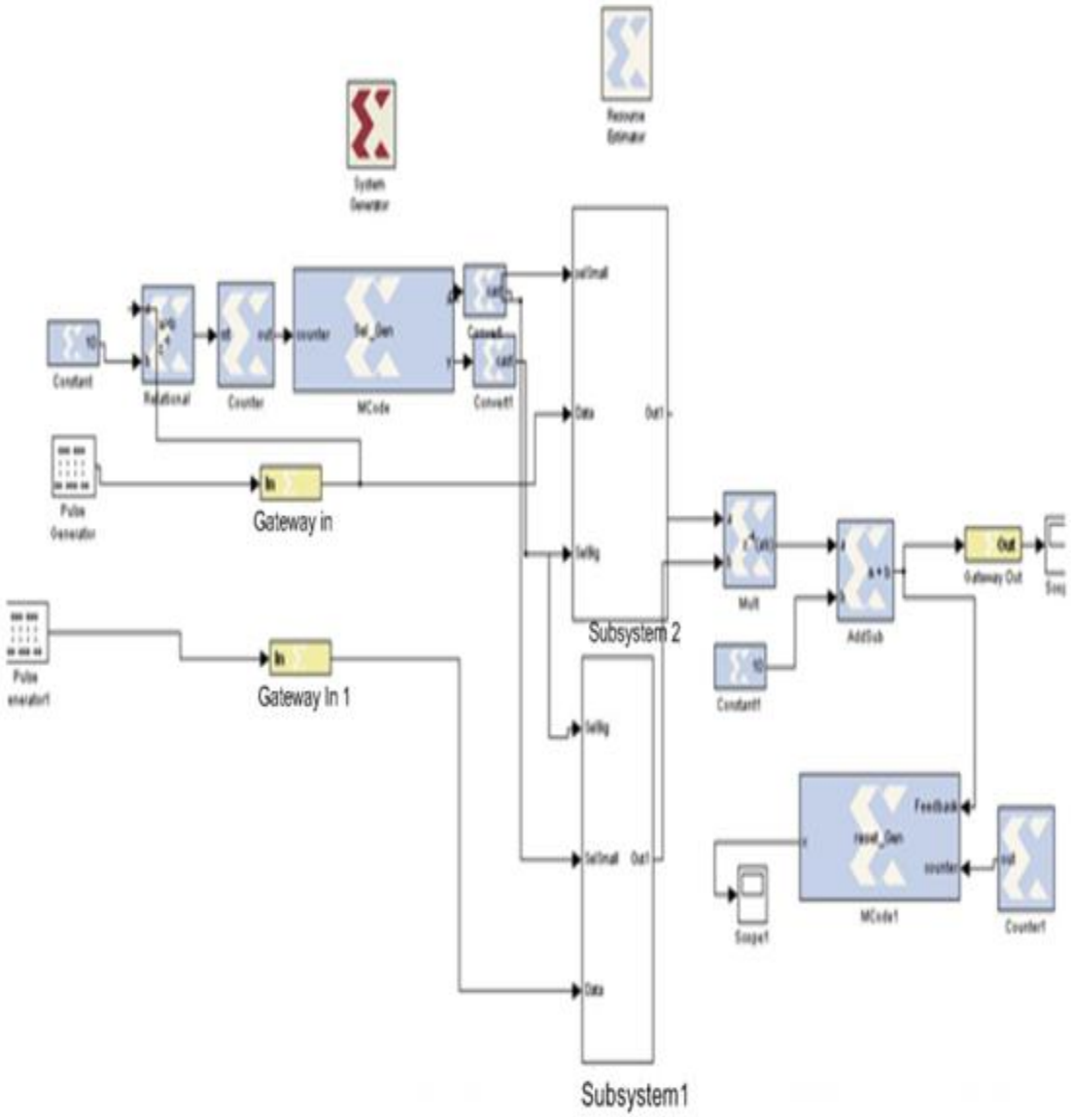

Figure 4-3 Convolver design (96 KSPS)

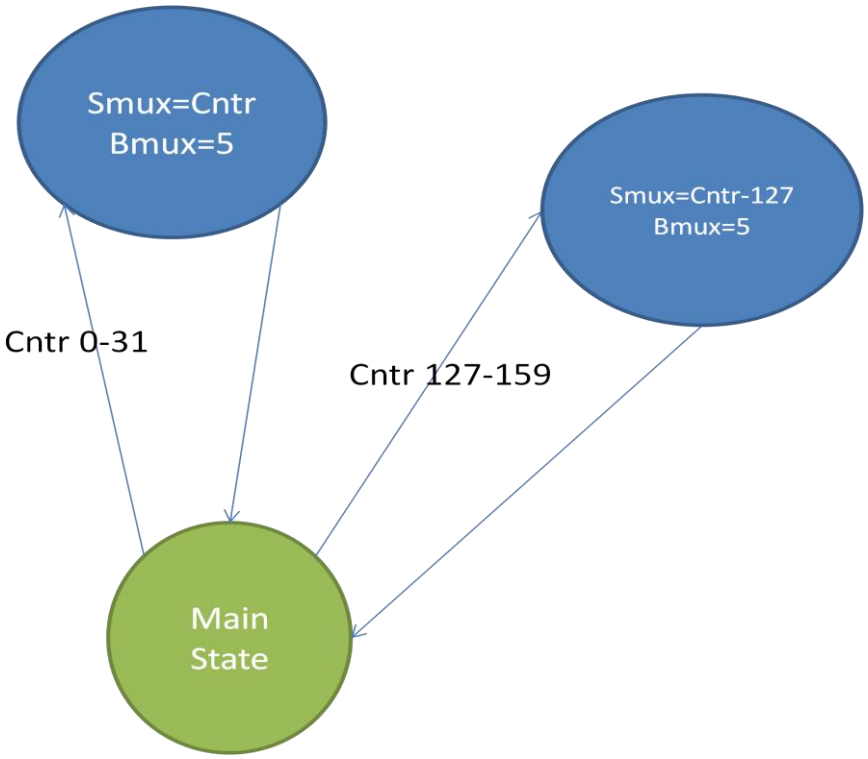

Figure 4-4 partial view of the state machine

We tested our design by running a resource estimation. By comparing the numbers from our new design to the number that we got for the design of the previous section, we see that we improved the design by a factor of two in terms of area. Figure 4-5 illustrates this finding.

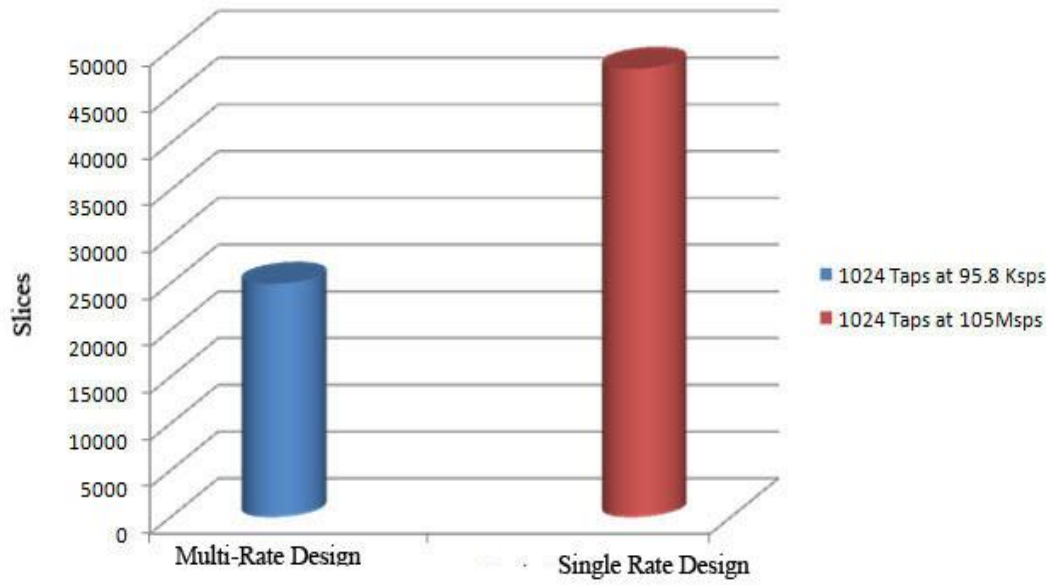

Figure 4-5 Resource comparison between the multi-rate design of this section and the single rate design of the previous section.

4.3 Correlator

The basic difference between a convolver and correlator is that one of the input signals of the convolver must be time-reversed, whereas this is not required in the correlator. Therefore, almost all the previous-in-time multiplications can be re-used. The only new multiplications required for each new sample cycle, is for the new 2 samples. Subtracting the product of the old sample is also required. The mechanism is similar to the moving average described Chapter 3. Figure 4-6 shows the mechanism of the correlator block, and Figure 4-7 illustrates the system generator view of this block.

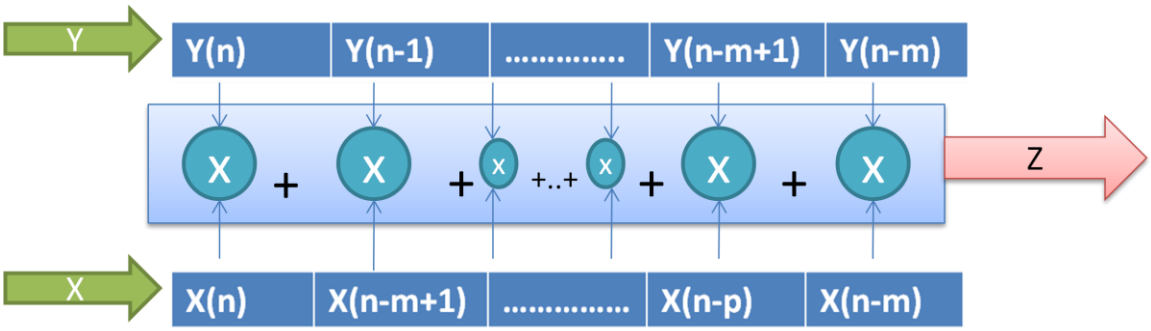

Figure 4-6 Correlator mechanism

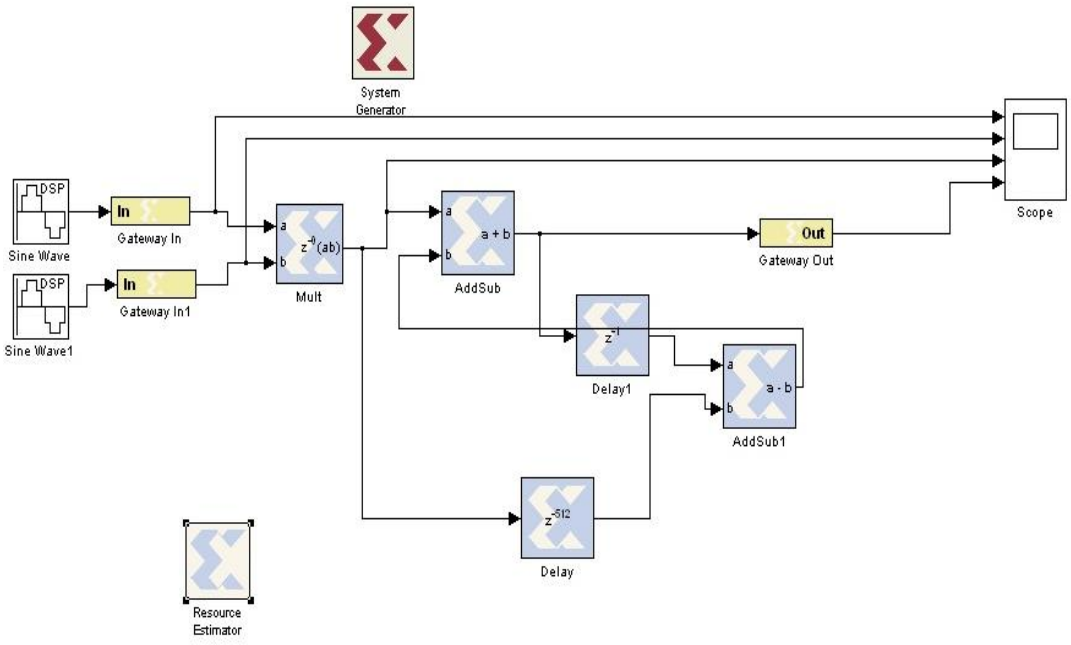

Figure 4-7 System Generator view of correlator design

The re-use of most of the products (see Figure 4.6) makes a difference when it comes to speed and area of the DSP chip. The correlator block can run at system clock rate, and uses little system resources. Figure 4-8 illustrates the resource usage of this block. This can be compared to the resource usage of the convolver block of Figure 4-2 . By comparing these two figures we can see that since correlator takes a lot less resources. If we compare the design of the

convolver with the correlator when the input signals are 16 bits wide, we can see that the Virtex 4 family is capable computing a 65,536 tap correlation, but only 1024 taps when it comes to convolver. This huge difference is because one of the signals in the convolver must be reversed; therefore, each new sample needs a lot more calculations.

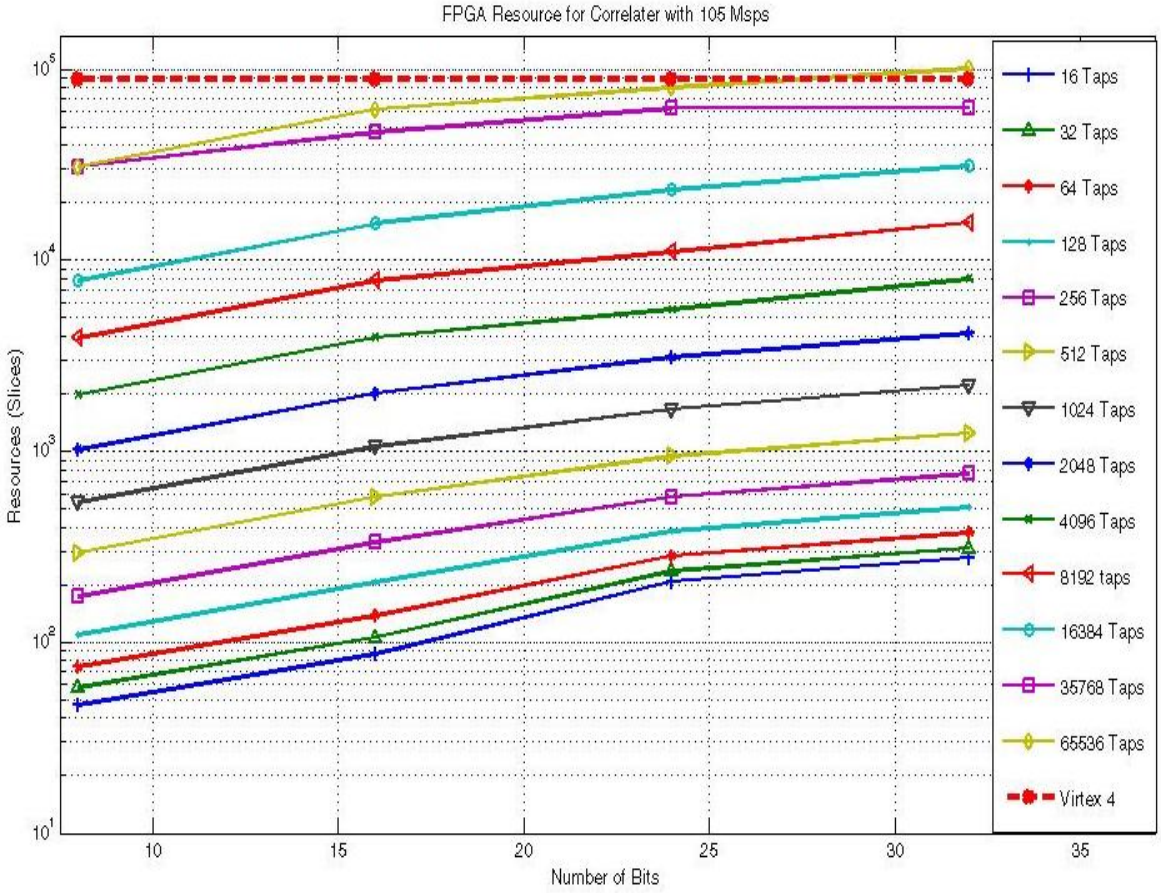

Figure 4-8 FPGA resources usage for correlator (105 MSPS)

4.4 Real-time Convolution Function using FIR filter

The goal of this chapter is to design a FIR filter that is capable of accepting different weights. The weights (impulse response of the filter) and raw data will be the inputs to our system. The system follows the formula below,

where a(n) is the filter coefficient value which will be provided by the BRAM block and x(k-n) is the input sample sequence.

$$y[n] = \sum_{k=0}^{1023} a[n]x[k-n] \qquad (4\text{-}3)$$

The reason that the filter coefficient is provided by a BRAM block is to allow us to manipulate the output of the BRAM (for example, reading in the constant coefficients backwards).

We design this convolver using a single-rate FIR compiler. The FIR Compiler Core is not designed for convolving 2 real-time signals; rather, this core is designed to be used as a single-rate filter where the coefficient does not change as much as the real-time signals. Therefore, we have to modify our design to work with this core.

The issue that we face is the timing delay between the reload of the new coefficients. As we can see from Figure 4-9, there is a cycle delay before the coeffient is loaded. This delay will cause omission of a single coefficient in every window of data (ie 1024) which will of course grow with time.

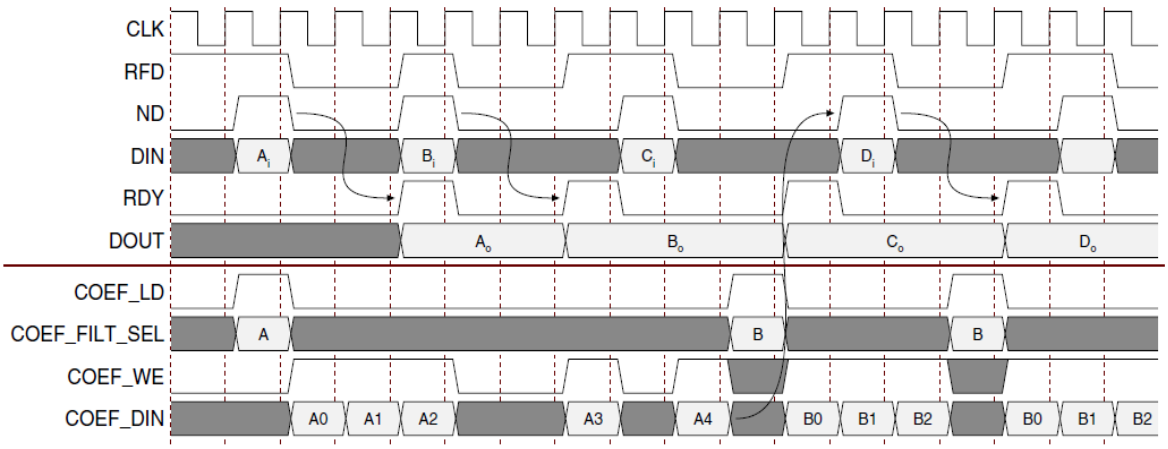

Figure 4-9 FIR Compiler Timing

In order to solve this issue, our design should have 2 FIR Compilers (FIR filters) working in parallel. This will let us set the reload signal of FIR 2 when the last coefficient is being written to FIR 1. Figure 4-10 illustrates our complete design.

Each FIR Compiler can have up to 1024 coefficients. We push our design to its limits by having 1024 coefficients loaded to each FIR Compiler. The easiest way to test our design is to convolve two rectangle functions, which will result in a triangle function. In order to test our design we used a pulse generator with 2048 samples (signal *x*) where half of these samples are set to one. In order to test the reload function we need to have a different coefficient value (Signal *a*). As illustrated in Figure 4-11, the core expects the coefficient to be in reverse order.

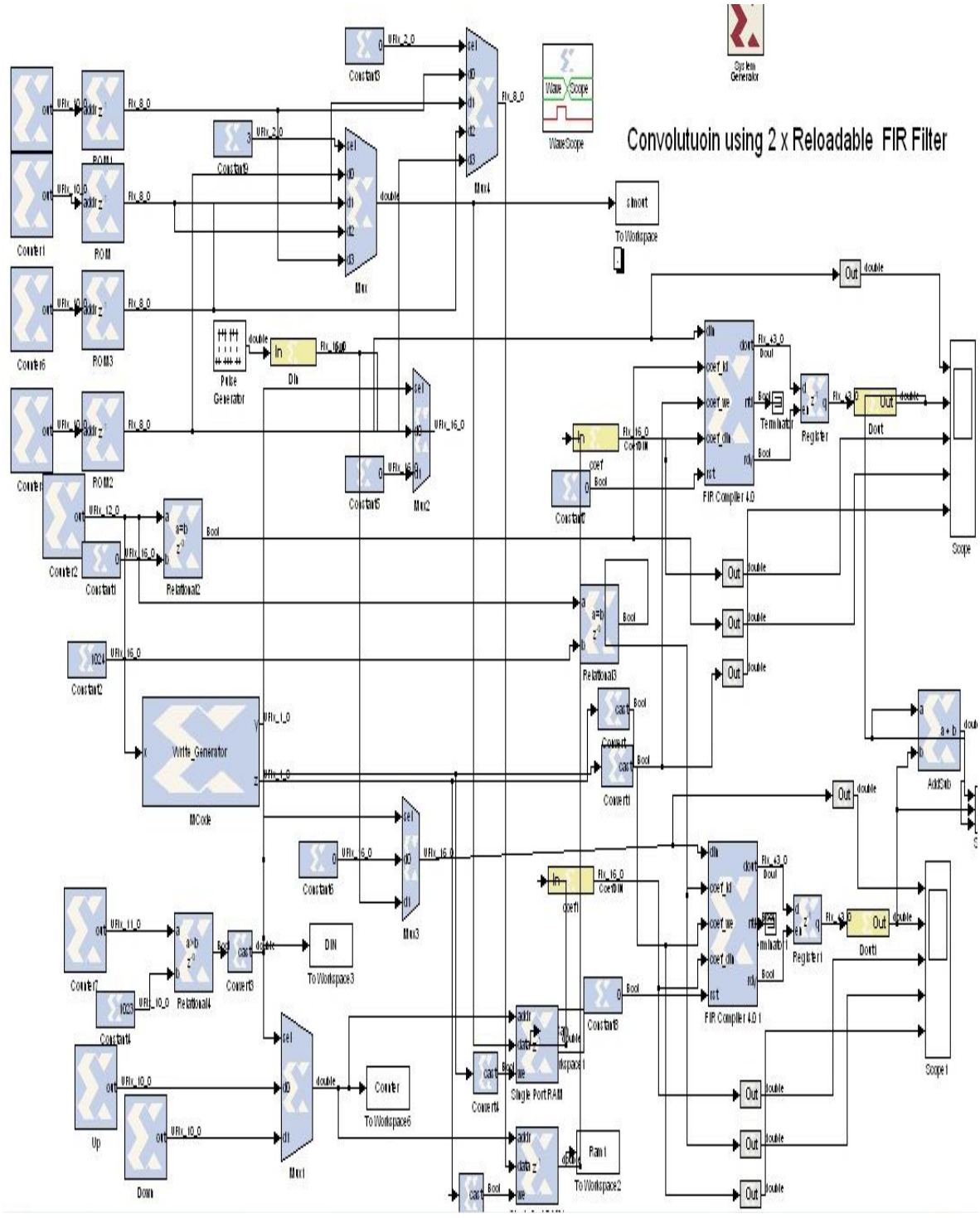

Figure 4-10 Convolver using FIR Compiler

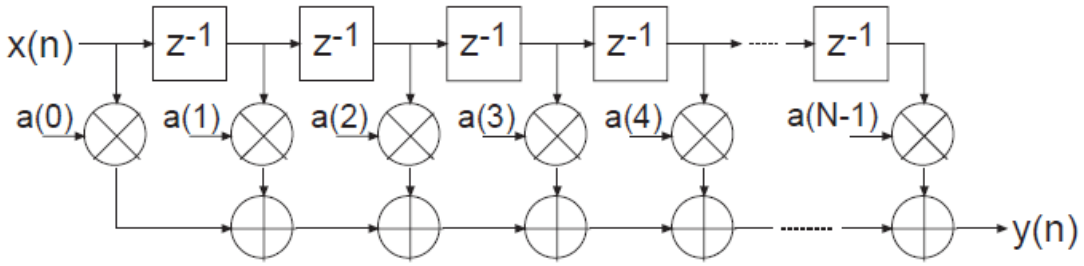

Figure 4-11 FIR Compiler Function

In order to reverse signal a, we use a RAM Block to buffer it and we reverse each window by manipulation of the RAM index. The manipulation of the RAM index is done by using 2 counters, one that counts up and another that counts down. The first RAM Block uses the count up value to write signal a into itself and uses the countdown to read out the reverse window values. The second RAM Block works exactly the opposite way. This scheme will save us area by eliminating the use of 3 extra counters, 1 extra MUX, 1 extra Converter block and 1 extra relational block.

We used MCode block to generate the Write and load signals for our FIR Compilers. The input signal of the MCode block is driven by a counter. The Mcode block creates a simple state machine. Figure 4-12 illustrates the functionality of this block.[10],[11],[12].

Trialling the convolver with general signals (say audio signals) was not possible because of scaling problems. Algorithms are available for the scaling, but were not implemented here.

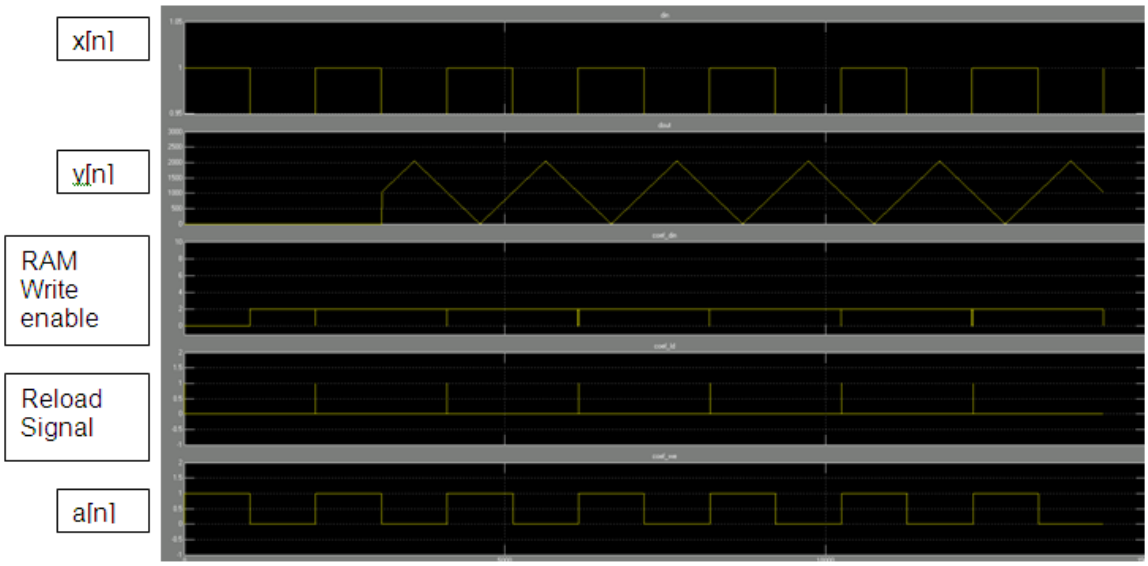

Figure 4-12 Result of Convolving 2 two Rectangular wave signal

5: Conclusion

In this thesis, the implementation of DSP algorithms on FPGA devices using two different design tools is presented. It is demonstrated how the hardware implementation of DSP algorithms enables high performance and high accuracy.

SoC technologies has enabled designers to implement highly complex designs very efficiently. The advances in FPGAs make them viable for increasingly large and complex designs. However, the superiority of FPGAs over other technologies such as ASICs, becomes emphasized when we consider designs that can be reconfigured.

We presented three applications in wireless communications: i) demodulation ii) *interference management in OFDM-based systems*, iii) *convolver/correlator structures for OFDM based systems*. We have developed an algorithm that extracts the original signal from its modulated carrier wave. This type of algorithm is used in radio receivers and other systems such as modems [29]. We also have developed a switch for rearranging the subcarrier positions of an OFDM system at the transmitter, based on feedback from the receiver. The objective here was to mitigate the effects of narrow-band interference inherent to OFDM systems. The receiver identifies the locations of subcarriers that experience strong interference (so the data is undetectable) and reports the subcarrier indices to the transmitter. The transmitter *switch* then relocates the

data located on these subcarriers to unused subcarrier locations near the band edge.

The *convolver/correlator* structures described in the thesis are for fast real-time implementation, and considerable effort was applied to their practical optimization. They may be used in applications such as high data-rate communication systems, for example, those using OFDM technology. In particular, fast variations in OFDM channels (i.e., fast time-varying channels) may lead to inter-mixing of signals at the receivers in OFDM-MIMO. This leads to undesirable levels of self-interference, which degrade the system performance. Fast digital processing is essential for combating interference in this setting. The correlator structure presented in this thesis may find applications in algorithms that strive to mitigate interference. Other applications are also foreseeable in this setting. For example, using real-time correlators for *time-processing* of signals prior to the FFT operation at the receiver, an operation also known as windowing in literature [28]. We believe this area to be a promising direction for future work.

Appendices

Appendix A.

Introduction to XtremeDSP Development Kit

Hardware

The XtremeDSP board consists of a motherboard module with a daughter card. The Motherboard in the XtremeDSP kit –IV is called BenONE-Kit Motherboard which hosts 2 FPGAs: the first is the Spartan-II, which is used for interfacing with the board, and the second is called Virtex-II, which is used for clock management. The daughter card module is called BenADDA DIME-II module, which hosts a Virtex-4 User FPGA (XC4VSX35). This module contains 2 independent analog-to-digital converter ADC (AD6645) channels that have a resolution of 14 bits and can sample up to 105 Mega Samples per Second (MSPS). In addition to the ADC, the daughter card has 2 independent digital-to-analog converters (DACs) that also have a resolution of 14 bits, but the sampling rate can go up to 160 MSPS. Figure A-1 shows an illustration of the Xtreme DSP kit [23],[24].

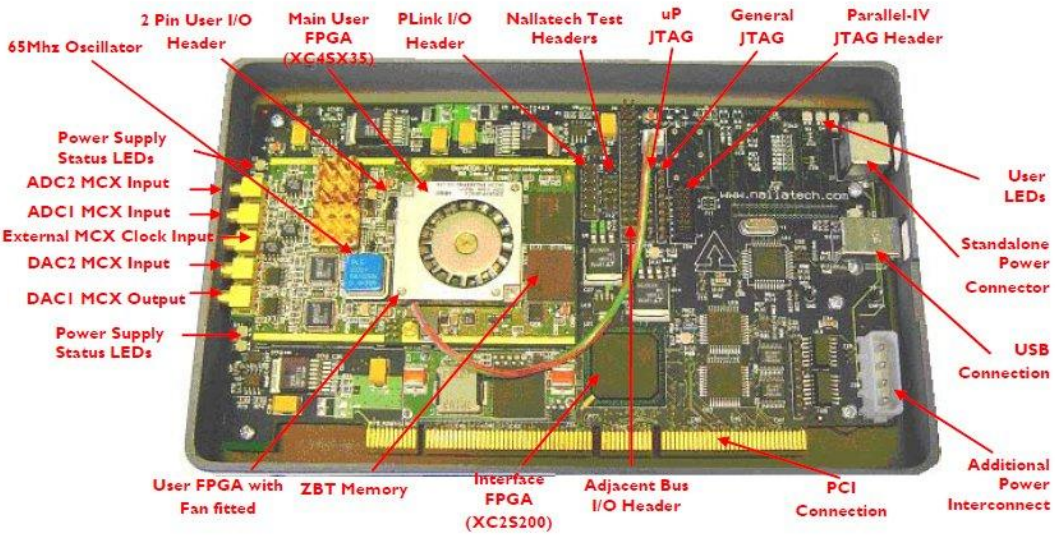

Figure A- 1 the Xtreme DSP kit

ADC

The ADC has the following capabilities [23]

1. 14-bits resolution in 2's complement format.
2. 105 MSPS sampling rate.
3. Single-ended 50Ω impedance analog inputs.
4. 3rd order filtering on analog inputs (-3dB point at 58 MHz).
5. The maximum signal magnitude for obtaining best performance is 2.2 Vpp.
6. Can attain between 11 to 12 bits using an onboard 105 MHz oscillator.

The best achievable signal-to-noise ratio (SNR) for this module is 74.5 dB at which point the maximum number of bits that we can attain is 12.1.

The system can accept an input signal swing of up to 2.2 Vpp. A larger input will be clipped. The quantized level of each signal can be calculated as follows

$$Each\ Level = \frac{1.1}{2^{13}} = 1.3428 * 10^{-4}\ Volt \qquad \text{(A-1)}$$

This means that the smallest voltage difference that the ADC can distinguish is about 0.134 mV.

DAC

The DAC has the following capabilities [3]:

• 14-bit DAC resolution with offset-binary input

• 160MSPS max input data rate.

• LVPECL clock inputs from the XC2V80-4CS144 Clock FPGA.

• Internal phase-locked loop (PLL) clock multiplier device feature.

• Single ended (DC coupled) 50Ω outputs via micro coaxial connector (MCX) connectors as standard.

Appendix B

Programming the Xtreme DSP Kit

We need a bit stream file to program our FPGA using a program called FUSE.

The bit stream file can be obtained in three different ways:

1. Compiling VHDL code in a compiler such as ISE, and generating necessary files.
2. Compiling and generating necessary files in the System Generator.
3. A combination of 1 and 2; in other words, making blocks in NGC format and compiling and generating bit steam files in ISE.

Reference List

[1] L. Shannon and P. Chow. “Maximizing System Performance: Using Reconfigurability to Monitor System Communications” IEEE international Conference on Field Programmable Technology, Pages 231-238, December 2004.

[2] F. Molisch, M.Toeltsch and S. Vermani, “Iterative Methods for Cancellation of Intercarrier Interference in OFDM Systems”, IEEE Transactions on Vehicular Technology , Vol. 56, No. 4, Jul 2007.

[3] Xilinx Xtreme DSP Board. Online: http://www.xilinx.com/univ/XUPV2P/Documentation/ug069.pdf

[4] J. G. Proakis, M. Salehi, “ Digital Communications “ , McGRAW Hill International Edition, 2008.

[5] Sine Wave Generation. Online:

http://www.actel.com/documents/Fusion_Waveform_TB.pdf

[6] Xilinx LogiCORE IP DDS Compiler v4.0 Online:

http://www.xilinx.com/support/documentation/ip_documentation/dds_ds558.pdf

[7] An FPGA-Based Adaptive Computing Implementation of Signal Detection. Online:

http://klabs.org/richcontent/MAPLDCon99/Presentations/F1_Zaino_S.PDF

[8] ISE Counter. Online:

http://www.xilinx.com/support/documentation/ip_documentation/c_counter_ds215.pdf

[9] D. Tse, “Fundamentals of Wireless Communication”, Cambridge University Press, 2005.

[10] Xilinx FIR Compiler. Online: http://www.xilinx.com/support/documentation/ip_documentation/fir_compiler_ds534.pdf

[11] FIR Filter Designer Pro. Online: http://www.ittc.ku.edu/~rvc/projects/firfilter/normal/

[12] T. Rissa, R. Uusikartano and J. Niittylahti, “Adaptive FIR Filter Architectures for Run-Time Reconfigurable FPGAs,” Proceedings 2002 IEEE International Conference on Field Programmable Technology, Pages 52-59 , Dec 2002

[13] ISE 10.1 Tutorial. Online: http://www.xilinx.com/itp/xilinx10/books/docs/qst/qst.pdf

[14] Xilinx Inc, “Getting Started with EDK”, Xilinx Inc, September 2, 2003.

[15] Xilinx Inc, “Platform Studio User Guide”, Xilinx Inc, February 15, 2005.

[16] Xilinx Inc, “Xilinx University Program Virtex-II Pro Development System Hardware Reference Manual”, Xilinx Inc, March 8, 2005.

[17] Audio DSP Algorithm. Online: http://www.harmonycentral.com/Effects/effects-explained.html

[18] OPB_AC97 Core. Online:

http://www.eecg.toronto.edu/~pc/courses/432/2004/projects/

[19] FSL V20. Online: http://www.xilinx.com/support/documentation/ip_documentation/fsl_v20.pdf

[20] Xilinx Inc, “MicroBlaze Processor Reference Guide”, Xilinx Inc, October 5, 2005, pp. 11-40.

[21] Wikipedia on Random number generator. Online: en.wikipedia.org/wiki/Pseudorandom_number_generator

[22] ISE FIFO Generator . Online: http://www.xilinx.com/support/documentation/ip_documentation/fifo_generator_ds317.pdf

[23] Xtreme DSP Board. Online: http://www.xilinx.com/univ/XUPV2P/Documentation/ug069.pdf

[24] Nallatech Limited, “XtremeDSP Development Kit-IV User Guide”. Online: www.xilinx.com/support/documentation/boards_and_kits/ug_xtremedsp_devkitIV.pdf

[25] Markus Nentwig, “SNR measurement using a spectrum. Online: http://www.elisanet.fi/~d635415/webroot/SNR3/index.html

[26] D. Harris, S.Harris, "Digital Design and Computer Architecture", Elsevier , 2007, Page 18.

[27] An FPGA-Based Adaptive Computing Implementation of Signal Detection. Online: http://klabs.org/richcontent/MAPLDCon99/Presentations/F1_Zaino_S.PDF

[28] F. J. Harris “On the Use of Windows for Harmonic Analysis with the Discrete Fourier Transform” Proceedings. IEEE , Vol. 66, No. 1, Pages 51-83, Jan. 1978

[29] F. Yu “FPGA Implementation of a Fully Digital FM Demodulator” IEEE The Ninth International Conference on Communications systems, Pages 446-450, Sep. 2004

[30] Visual Studio Express. Online: http://www.microsoft.com/express/Windows/