HARDWARE IMPLEMENTATION OF A VARIABLE COEFFICIENT FINITE IMPULSE RESPONSE FILTER USED IN AUDIO PROCESSING

Thesis

Submitted to

The School of Engineering

UNIVERSITY OF DAYTON

In Partial Fulfillment of the Requirements for

The Degree

Master of Science in Electrical Engineering

By

Eric John Balster

UNIVERSITY OF DAYTON

Dayton, Ohio

May 2000

HARDWARE IMPLEMENTATION OF A VARIABLE COEFFICIENT FINITE IMPULSE RESPONSE FILTER USED IN AUDIO PROCESSING

APPROVED BY:

Dr. F. A. Scarpino, Professor
Electrical and Computer Engineering
University of Dayton
Committee Chairman

Dr. R. C. Hardie, Associate Professor
Electrical and Computer Engineering
University of Dayton
Committee Member

Dr. W. W. Smari, Assistant Professor
Electrical and Computer Engineering
University of Dayton
Committee Member

Dr. D. L. Moon, Associate Dean
Graduate Engineering and Research
University of Dayton

Dr. B. E. Cherrington, Dean
School of Engineering
University of Dayton

ABSTRACT

HARDWARE IMPLEMENTATION OF A VARIABLE COEFFICIENT FINITE IMPULSE RESPONSE FILTER USED IN AUDIO PROCESSING

Name: Balster, Eric, J.
University of Dayton, 2000

Advisor: Dr. F. A. Scarpino

The graphic equalizer has been a component of audio enhancement for many years. A typical equalizer has several active analog filters working in parallel to break up the input music into various frequency components. The user adjusts these signal component amplitudes, and the adjusted signal components are summed to create the equalized output signal. In this way, the user of the graphic equalizer may dynamically change the quality of music listening.

However, many problems exist in the areas of analog filtering and traditional equalizer design. Imprecise analog component values and tolerances give variance in cutoff frequencies, which may produce an undesired audio output. Also, electrical noise may disturb the analog signals and create unwanted tones in the adjusted output signal.

Digital filtering, however, is more precise in frequency cutoffs and electrical noise no longer becomes a problem. Also, only one digital filter needs to be designed to replace the functionality of the many analog filters used in traditional graphic equalizers. If a digital filter can vary its coefficient values, then it can be reconfigured to produce any frequency response desired. The dynamic nature of a digital filter with variable coefficients enables it to replace the many analog filters used in traditional equalizer design.

Most digital filters designed today are software programs that cannot operate without the use of a personal computer (PC), and many of these designs cannot operate in real time. These software digital filters must have the audio signal sampled, digitized, and downloaded into the computer's disk space. The software then calculates the desired output. Only after computation can the user listen to the filtered output.

Therefore, for a digital equalizer to become practical in the audio enhancement industry, it must be developed in hardware for a real time application. That is, there must exist minimal delay between the unfiltered input audio and the equalized audio output. This body of work is the development of a real-time hardware digital audio equalizer. The equalizer is made up of one digital filter with variable coefficient values.

The variable coefficient digital filter designed in this body of work receives its coefficient values from the serial port of a personal computer, and the coefficient values of the filter can be modified dynamically at any time by the user. In this way, the user can modify the frequency response of the filter to change the sound quality of music listening.

ACKNOWLEDGMENTS

I sincerely thank my advisor Dr. Frank Scarpino for introducing me to the art of advanced digital design and signal processing. His vast expertise and guidance enabled me to overcome many obstacles throughout the design and implementation of this effort. I also would like to thank him for his professional advice and guidance throughout my graduate and undergraduate years. I would like to thank Dr. Russell Hardie and Dr. Waleed Smari for reviewing this thesis.

I also thank both my parents for their love and support throughout my life and education. They have truly been both role models and advice givers throughout my important life decisions.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

INTRODUCTION

The hardware development of a 27-tap, 8-bit wide variable coefficient finite impulse response (FIR) filter has been designed, implemented, and demonstrated. The hardware implementation of this type of filter is used for real-time audio enhancement. Most audio equalizers today are comprised of a parallel combination of analog filters. The implementation of a variable coefficient FIR filter can replace the analog filters of traditional audio equalizers. Also, the dynamic nature of variable coefficients allows the user to change the frequency response of the filter, thus changing the quality of music listening.

The coefficient values of the filter are loaded into the hardware through the serial port of a personal computer. The software necessary for the coefficient loading on a Microsoft NT, 95, or 98 operating system is designed, implemented, and demonstrated. The coefficients are loaded serially through a COM port cable and converted to a parallel data word by a UART (Universal Asynchronous Receiver/Transmitter) receiver.

The hardware development of the UART receiver is designed, implemented, and demonstrated. The UART receiver sends an interrupt signal to the hardware FIR filter after converting a serial data stream to a parallel data word. The FIR filter acknowledges the interrupt signal and shifts the new coefficient value into the filter.

These coefficient values are used to filter digital input data from an audio source such as a CD player or Radio. The audio signal is applied to the filter, and the output of the filter is sent to an amplifier and on to a speaker. Depending on the filter coefficients, the filter can be adjusted to one of a virtually limitless number of frequency responses. The result is a dynamically adjustable digital audio equalizer used for audio enhancement.

CHAPTER 1

SAMPLING OF AN ANALOG SIGNAL

In order for an analog signal to be processed digitally, it must first be converted into a digital signal. This is accomplished by sampling and digitizing the analog signal. The sampling of an analog signal is given by:

$$x_d(n) = x_c(nT_s), \qquad (1.1)$$

where x_d is the sampled discrete signal, x_c is the original continuous signal, T_s is the sampling period, and n is an integer. Figure 1-1 shows an analog and discrete sine wave.

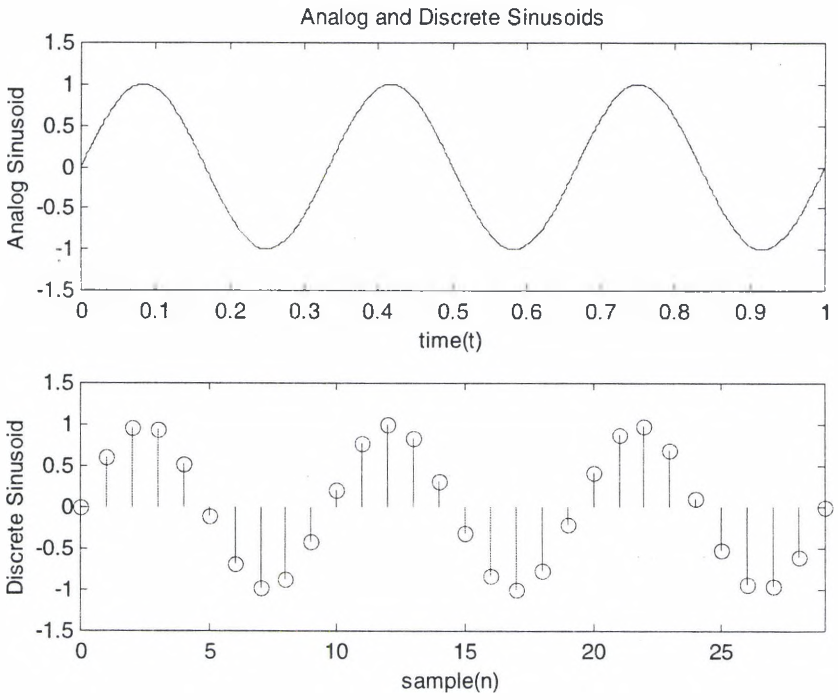


Figure 1-1 --- Analog and Discrete Sinusoids

The analog sinusoid is continuous and therefore exists for all time. The discrete sinusoid, however, is not continuous but only exists for integer values n. This discrete sinusoid can then be digitized into an array of values for processing by a digital computer. Equation 1.2 shows a digitized sinusoid.

$$y(n)=[\ldots -0.9868,-0.8835,-0.4199,0.2150,0.7622,0.9985,0.8277 \ldots]. \qquad (1.2)$$

For an analog signal to be processed digitally, it must first be converted into an array of numerical values by sampling and digitizing. This process is referred to as analog-to-digital conversion.

The Nyquist sampling theorem states that any input signal must be sampled at least twice its maximum frequency content for no information to be lost. In other words, if an analog signal is band limited to 5 kHz, the sampling frequency of that signal must be at least 10 kHz. The Nyquist sampling theorem is given in Equation 1.3.

$$f_s \geq 2 f_{\max}, \qquad (1.3)$$

where f_s is the sampling frequency and $f_{\max}$ is the maximum frequency content of the sampled signal. The Nyquist theorem states that if a signal is sampled at or above f_s, it can be reconstructed with no information lost. The maximum frequency content of the sampled signal is generally referred to as the Nyquist frequency, and the sampling frequency is referred to as the Nyquist rate. The Nyquist theory is covered in more detail in (2)*Oppenheim and Schafer*, p. 146.

However, if a signal is sampled below the Nyquist rate, aliasing occurs. Figure 1-2 shows an example of an aliased signal.

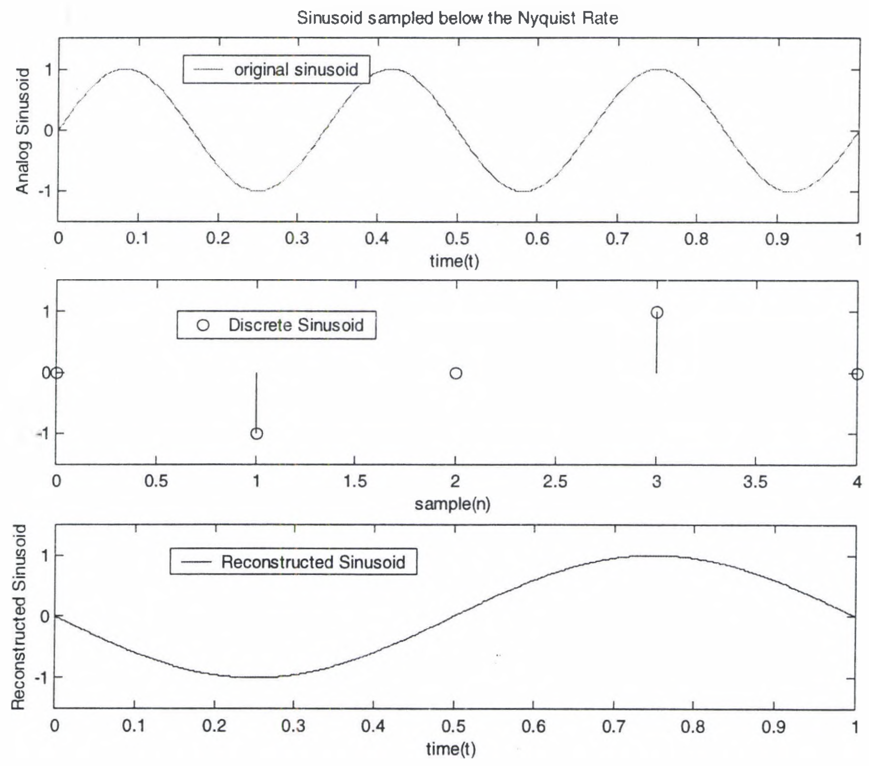


Figure 1-2 --- Sinusoid Sampled Below the Nyquist Rate

From Figure 1-2, it is readily seen that sampling a signal below the Nyquist rate will cause aliasing.

After the signal has been discretized, it can now be represented as a sequence of numbers as shown in Figure 1-1. This sequence has frequency content much like its analog signal counterpart. However, because it has been sampled, the original frequency of the signal has been lost. A digital frequency ω is defined as:

$$\omega = \frac{\Omega}{f_s} = 2\pi \frac{f}{f_s}, \tag{1.4}$$

where Ω is the frequency of the analog signal, and f_s is the sampling rate. Because the maximum frequency that can be sampled without aliasing is equal to one half the sampling frequency, the maximum digital frequency allowed is calculated.

$$\omega_{\max} = \frac{2\pi(\frac{f_s}{2})}{f_s} = \pi . \tag{1.5}$$

Digital frequencies always range between $-\pi$ and π; $\omega = 0$ is DC and $\omega = \pi$ is the maximum digital frequency.

Conversely, the analog frequency that corresponds to the digitally frequency must sometimes be calculated. Taking Equation 1.4 and solving for f the equation becomes:

$$f = \frac{f_s \omega}{2\pi}, \tag{1.6}$$

where f is the analog frequency of the signal, ω is the digital frequency, and f_s is the sampling frequency.

CHAPTER 2

FIR FILTER DESIGN OVERVIEW

An FIR (Finite Impulse Response) filter is constructed by producing a weighted sum of input samples at the output. As a waveform is input into the system it is first sampled and digitized, then many samples of the waveform are multiplied by their respective coefficients and summed together to produce a desired output. Figure 2-1 shows a block diagram of a simple FIR filter.

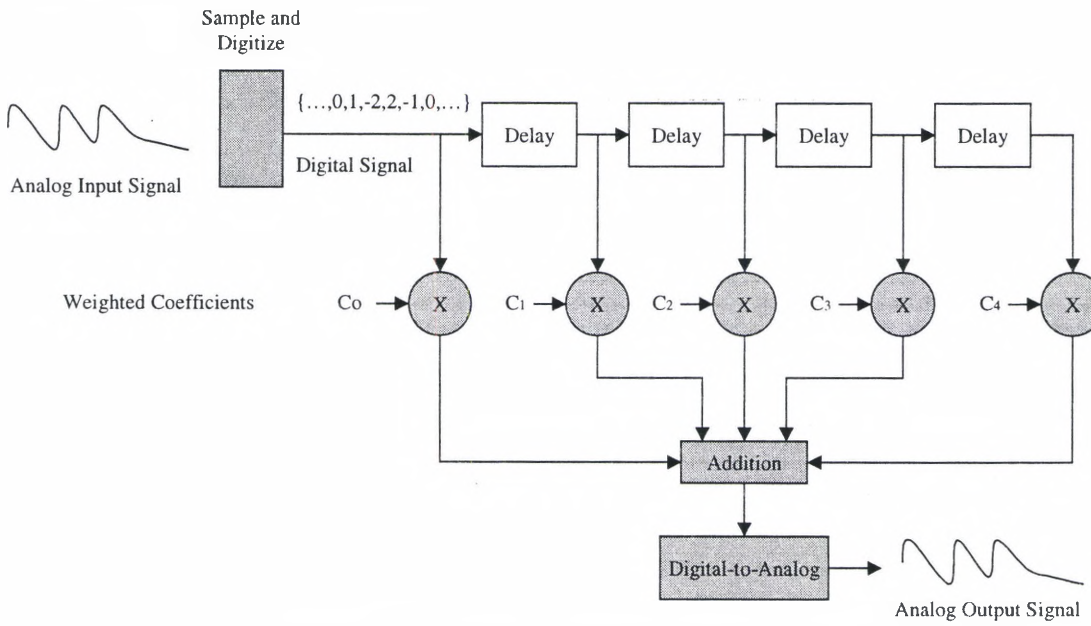


Figure 2-1 --- Five Tap FIR Filter Architecture

The Weighted Coefficients of the filter are calculated to pass or attenuate certain frequencies associated with the input signal. The equation that describes an FIR filter is given by:

$$y(n) = \ldots + C_2 x(n+2) + C_1 x(n+1) + C_0 x(n) + C_{-1} x(n-1) + C_{-2} x(n-2) + \ldots \quad (2.1)$$

Equation 2.1 is the Linear Constant Coefficient Difference Equation (LCCDE) of an FIR filter and fully describes the system.

Moving Average Filter:

A specific type of FIR filter is the Moving Average Filter. This filter's coefficients are all equal, thus averaging the input signal samples. The LCCDE of a 5-tap Moving Average Filter is given by:

$$y(n) = \frac{1}{5}[x(n+2) + x(n+1) + x(n) + x(n-1) + x(n-2)]. \qquad (2.2)$$

The behavior of this filter can be analyzed by calculating the impulse response of the system. The impulse response of the five-tap Moving Average Filter is easily calculated by inputting a delta function as the input waveform. The impulse response of a five-tap moving average filter is given by:

$$h(n) = \frac{1}{5}[\delta(n+2) + \delta(n+1) + \delta(n) + \delta(n-1) + \delta(n-2)]. \qquad (2.3)$$

Figure 2-2 displays the impulse response of the filter.

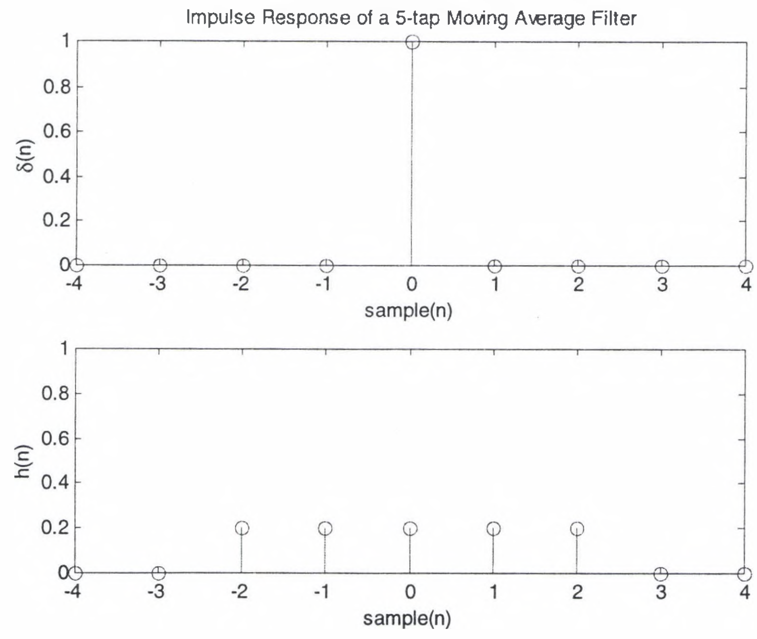


Figure 2-2 --- Impulse Response of a Five Tap Moving Averaage Filter

The impulse response of a linear system gives great insight into the behavior of the system. In fact, with the impulse response of a linear system, the output of the system due to any input is completely predictable.

In filter analysis, it is often beneficial to obtain the frequency response of the system. The frequency response of a system is calculated by taking the Discrete-Time Fourier Transform (DTFT) of the impulse response. The DTFT of a signal is given by:

$$X(\omega) = \sum_{n=-\infty}^{\infty} x(n) e^{-j\omega n} \quad . \qquad (2.4)$$

Because the moving average filter only has five taps, the variable n only sums from –2 to 2. Therefore, the frequency response of the moving average filter is given by:

$$H(\omega) = \frac{1}{5}(e^{j2\omega} + e^{j\omega} + e^{j0} + e^{-j\omega} + e^{-j2\omega}) \quad . \qquad (2.5)$$

Equation 2.5 may be further simplified by utilizing Euler's equation. The frequency response of the 5-tap moving average filter is:

$$H(\omega) = \frac{1}{5} + \frac{2}{5}\cos(2\omega) + \frac{2}{5}\cos(\omega) . \tag{2.6}$$

Plotting the frequency response of the filter gives further insight to its behavior. Figure 2-3 shows the frequency response of the moving average filter.

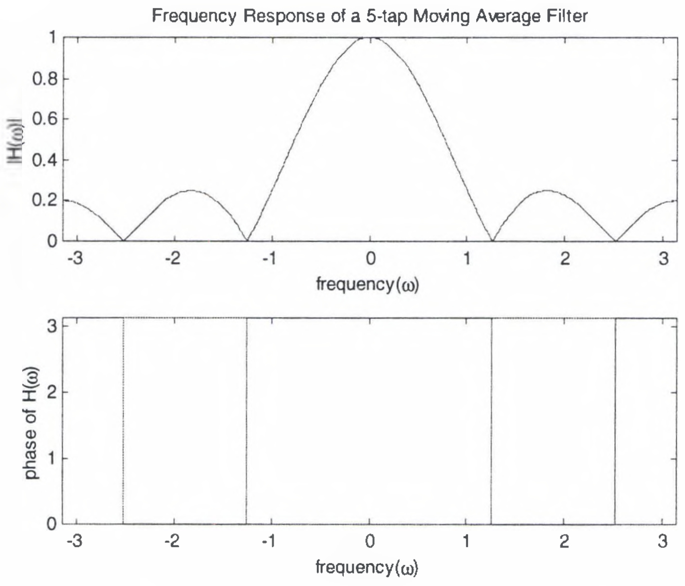


Figure 2-3 --- Frequency Response of a 5-Tap Moving Average Filter

It is easily seen that this type of filter attenuates higher frequencies while passing the lower frequencies with approximate unity gain. Also, from Equation 2.6 it is readily seen that the frequency response of this filter is purely real. However, many filters have frequency responses that have both real and imaginary parts. Therefore, most frequency responses are plotted by their magnitudes and phases. The magnitude response of the system is of most importance, and the phase response of the system will be discussed later.

Windowing Filter Design

For the majority of filter designs, a particular frequency response is desired, and the filter's LCCDE is then calculated. This design method is referred to as windowing. For example, a filter is to be implemented that will pass all signal frequencies from 0 to $\frac{\pi}{4}$ radians/sample; all other frequencies are to be attenuated. Figure 2-4 gives the ideal frequency response of the system.

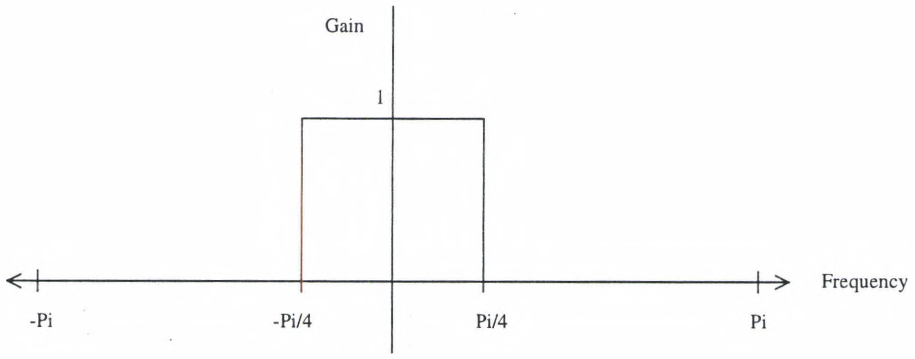


Figure 2-4 --- Frequency Response of an Ideal Low-Pass Filter

Now that the frequency response has been determined, the impulse response of the low-pass filter (LPF) can be determined, by taking the Inverse Discrete Time Fourier Transform (IDTFT) of the frequency response. The IDTFT of a signal is given by:

$$x(n) = \frac{1}{2\pi} \int_{-\pi}^{\pi} X(\omega) e^{j\omega n} d\omega . \tag{2.7}$$

The impulse response of the filter is then:

$$h(n) = \frac{1}{2\pi} \int_{-\frac{\pi}{4}}^{\frac{\pi}{4}} e^{j\omega n} d\omega . \tag{2.8}$$

The limits of the integral are changed because the integral is non-zero from the interval $-\frac{\pi}{4}$ to $\frac{\pi}{4}$. Also, in this interval, the value of $H(\omega)$ is one. Equation 2.8 can be further simplified by Euler's equation. The frequency response of the ideal filter is given by:

$$h(n) = \frac{1}{\pi n} \sin(\frac{\pi n}{4}) , \qquad n \neq 0$$

$$= \frac{1}{4} , \qquad n = 0 \ . \tag{2.9}$$

However, by plotting the impulse response of the ideal LPF, some difficulties are found. Figure 2-5 shows the impulse response of the ideal LPF.

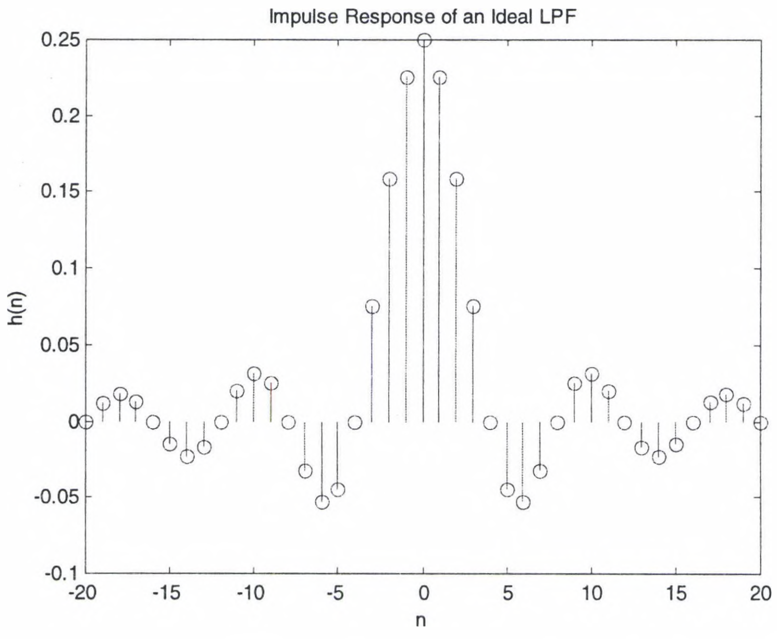


Figure 2-5 --- Impulse Response of an Ideal Low-Pass Filter

This impulse response is problematic because it is infinitely long, thus requiring an infinite number of filter taps. However, the value of the function does decrease as *n* goes to infinity, so the impulse response of the filter may be truncated at a certain value of *n*. Also, the impulse response of the system starts before time n=0. This means that the system starts reacting to the impulse before the impulse is input into the system! This type of system is referred to as a non-causal system.

Truncating the impulse response, and then shifting the response to the right can combat both of these problems. The modified impulse response of the system is displayed in Figure 2-6.

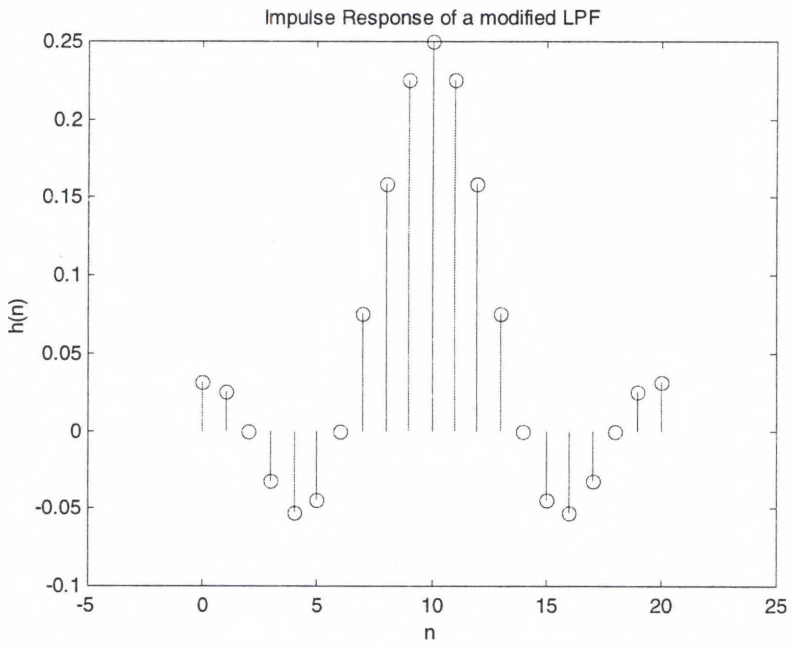


Figure 2-6 --- Impulse Response of a modified Low-Pass Filter

As shown in the example of the moving average filter, the impulse response values are equal to the tap coefficients for the filter. Therefore, if the impulse response of a system has 21 different values (as the one in Figure 2-6), then the filter must have 21 taps. This is called the filter length. However, truncating and shifting the impulse response of the system does affect the performance of the filter. For the modified filter in Figure 2-6, the frequency response is shown in Figure 2-7.

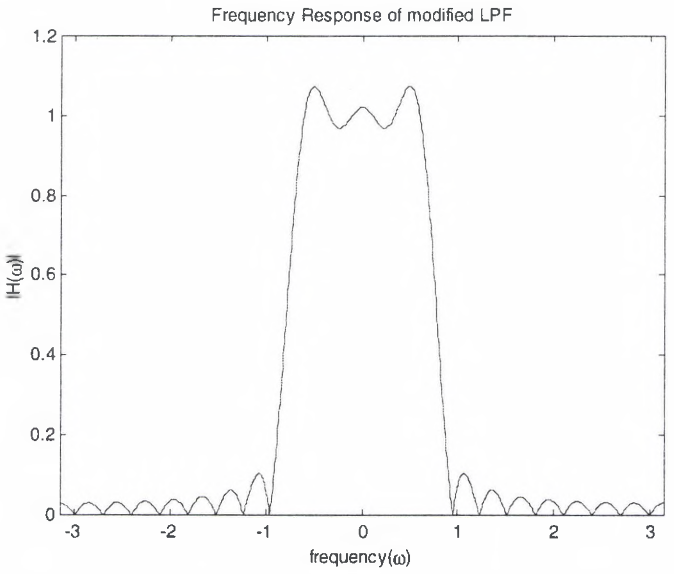


Figure 2-7 --- Frequency Response of the Modified Low-Pass Filter

It is readily seen that the frequencies in the pass-band are not all passed with unity gain, and the frequencies in the stop-band are not completely attenuated. Therefore, an ideal filter is not physically realizable, but the frequency response of a real system can come arbitrarily close to an ideal response, depending on the filter length.

By determining the ideal frequency response of a system, the ideal impulse response is found. The ideal impulse response is then truncated and shifted to become the impulse response of the system and the coefficient values of the filter. Appendix A contains derivations for the ideal impulse responses for low-pass, high-pass, band-pass, and band-stop filters.

Other windowing methods:

The rectangular windowing method of designing FIR filters is good for calculating the correct frequencies in the pass and stop bands, but as seen in Figure 2-7 the result from truncating and shifting the coefficients can be undesirable. Therefore, other methods of windowing have been developed to eliminate the unwanted pass-band ripples, and further attenuate the stop-band. One windowing method is the Hamming window. The Hamming window is definied as:

$$w(n) = 0.54 - 0.46\cos(\frac{2\pi n}{N-1}), \qquad n = 0,1,\ldots,N-1. \tag{2.10}$$

The Hamming window and rectangular window coefficients are multiplied together to calculate the new coefficients of the filter. The coefficients for a Hamming window filter are given by:

$$h(n) = w(n)d(n-M)\ , \tag{2.11}$$

where d is an array of rectangular window coefficients, and M is half the filter length. Applying the Hamming window coefficients to the rectangular low-pass filter given in Figures 2-6 and 2-7, gives the frequency response a smoother characteristic. The Hamming window is given in (3)*Orfanidis,* p. 549. Figure 2-8 shows the frequency response of the FIR filter with Hamming window coefficients.

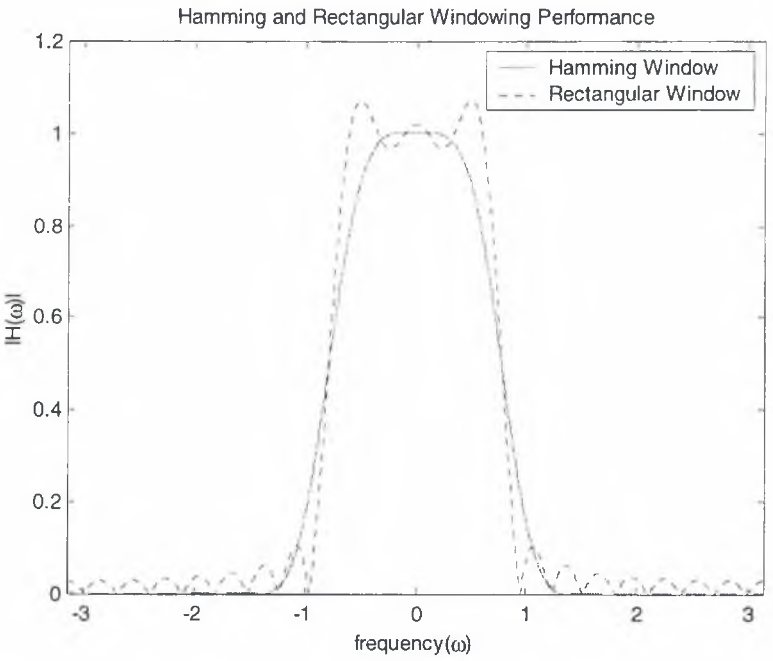


Figure 2-8 --- Frequency Response of a LPF with Hamming Coefficients

Figure 2-8 clearly shows that the Hamming coefficients serve as a better filter than the rectangular coefficients. The pass-band ripple is almost completely eliminated. Also, if the frequency responses of the two filters are plotted on a dB (decibel) axis, the stop-band attenuation of both the filters can be analyzed. Figure 2-9 shows a dB plot of the two frequency responses.

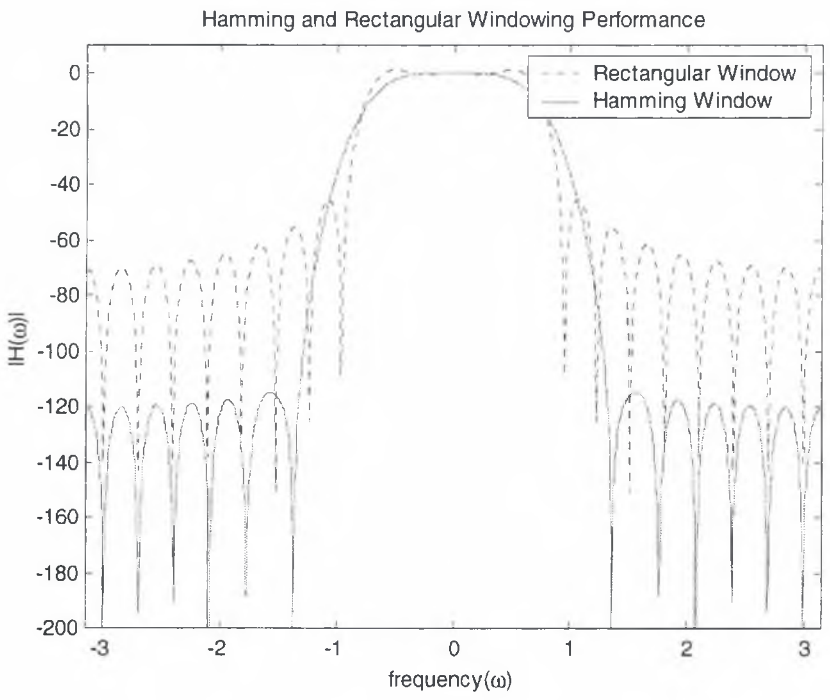


Figure 2-9 --- dB Plots of Hamming and Rectangular Window Frequency Responses

As seen in Figure 2-9, a Hamming window filter attenuates the stop-band by approximately 40 dB more than the Rectangular Window with the same number of taps. Because of its flat pass-band, and greater attenuation of the stop-band, the Hamming window filter is chosen almost exclusively over the Rectangular window filter.

However, another popular windowing method further exceeds the Hamming window filter's performance. The Kaiser window method is applied like the Hamming window method in that the coefficients are multiplied by the Rectangular window coefficients. The Kaiser Window is defined as:

$$w(n) = \frac{I_o(\alpha\sqrt{1-(n-M)^2/M^2})}{I_o(\alpha)} , \tag{2.12}$$

where I_0 is the modified Bessel function of the first kind and 0th order, and alpha is approximately equal to 7. The Kaiser window design is found in (3)*Orfanidis,* p. 553. When applying the Kaiser window coefficients to the rectangular window coefficients, the frequency response is drastically improved. Figure 2-10 shows the LPF frequency response with all three types of filter coefficients.

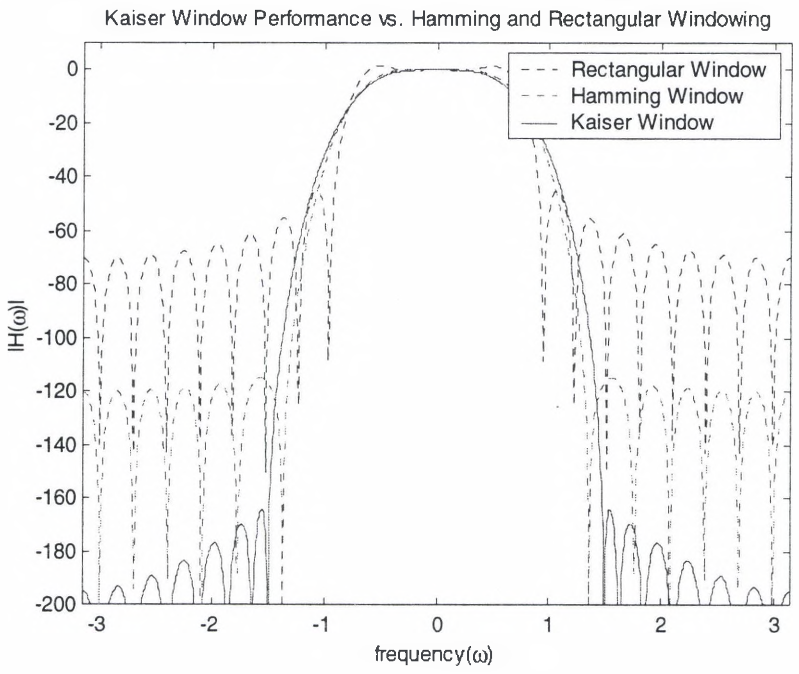


Figure 2-10 – Kaiser, Hamming, and Rectangular Window Frequency Responses

As seen in Figure 2-10, the Kaiser window is the best method for calculation of FIR filter coefficients. Appendix B contains *Matlab* software that calculates Rectangular, Hamming, and Kaiser Window coefficients for FIR filter design.

Equalization:

An audio equalizer is easily developed from windowing filter designs. An equalizer is comprised of a low-pass filter, high-pass filter, and several band-pass filters.

These types of filters are easily designed with the windowing methods described above. The Kaiser windowing method is chosen because of its maximally flat pass-band, and great attenuation in the stop band. A three-band equalizer is developed.

The final implementation of the hardware FIR filter is a 27-tap filter. Therefore, when designing the filter with the *Matlab* software, only 27 taps are chosen. First, the cutoff frequencies of the filters are to be chosen. Keeping in mind that the ear is logarithmically sensitive to frequency, the frequencies given in Table 2-1 are chosen.

Table 2-1 --- Analog Cutoff Frequencies in an Audio Equalizer

Filter Type	Low-Frequency Cutoff	High-Frequency Cutoff
Low-Pass	0 Hz	2000 Hz
Band-Pass	2000 Hz	5000 Hz
High-Pass	5000 Hz	22050 Hz

The high-pass filter only passes to 22050 Hz because the filter's sampling rate has been chosen to be 44100 Hz, the same sampling rate as CD (Compact Disk) players. And the largest frequency possible in digital process is up to one half the sampling rate as shown in Equation 1.3. The analog cutoff frequencies have been chosen, but the digital frequencies need to be calculated before any design work can begin. A digital frequency is calculated by Equation 1.4, and the digital cutoff frequencies of this design are given in Table 2-2.

Table 2-2 --- Digital Cutoff Frequencies in an Audio Equalizer

Filter Type	Low-Frequency Cutoff	High-Frequency Cutoff
Low-Pass	0	0.284952
Band-Pass	0.284952	0.712379
High-Pass	0.712379	π

Using the *Matlab* software given in Appendix B, the design of the audio equalizer is very straightforward. The cutoff frequencies are given as well as the filter length, and the filter tap coefficients are given. Figure 2-11 gives the frequency responses of the filters used in the equalizer.

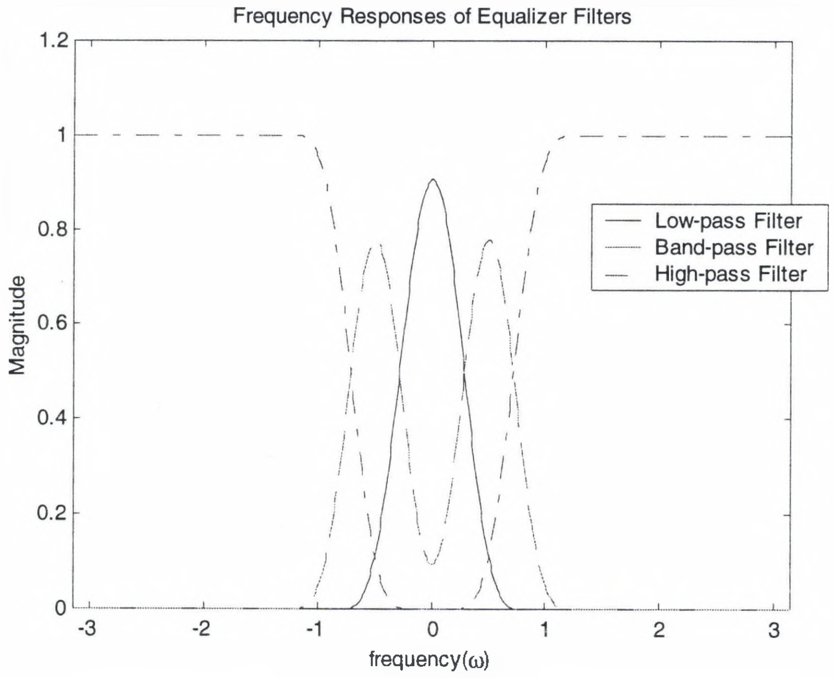


Figure 2-11 --- Equalizer Filters' Frequency Responses

To produce the desired frequency response, the scaling factors are then multiplied by the impulse responses of the respective filters. For this equalizer design, the low frequency response will be boosted by a factor of three, and the high frequency response will be boosted by a factor of two. These impulse responses can then be added together to form the coefficient values of the filter. The coefficient values of the equalizer are given by:

$$E(n) = 3 * L(n) + B_p(n) + 2 * H(n), \tag{2.13}$$

where $L(n)$ is the low-pass filter coefficients, $B_p(n)$ is the band-pass filter coefficients, and $H(n)$ is the high-pass filter coefficients. Figure 2-12 shows the frequency response of the equalizer.

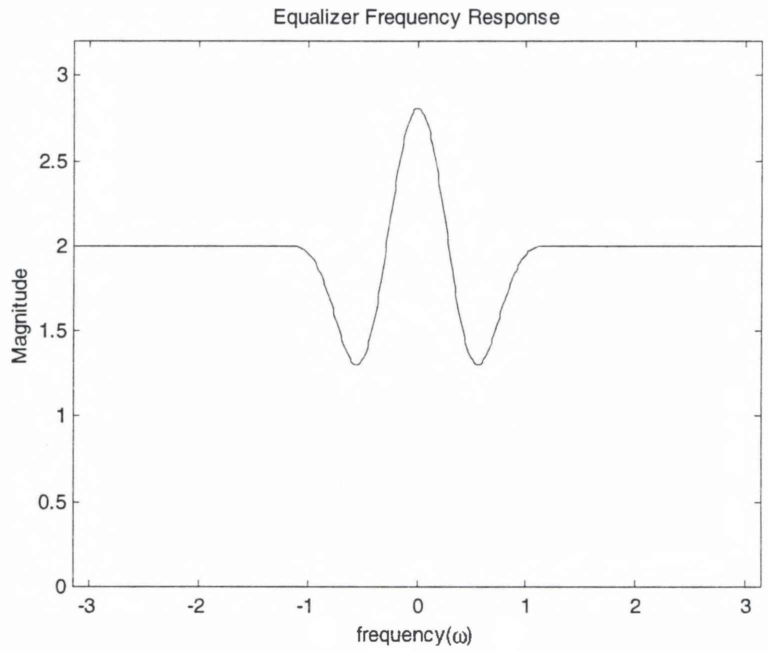


Figure 2-12 --- Equalizer Frequency Response

As seen from Figure 2-12, the low-frequency gain is approximately equal to three, the mid-frequency gain is approximately equal to unity, and the high frequency gain is approximately equal to two.

As said previously, the coefficient values of the filter were calculated by a weighted sum of the three filter impulse responses. Figure 2-13 gives the impulse responses of the three filters, and the impulse response of the equalizer.

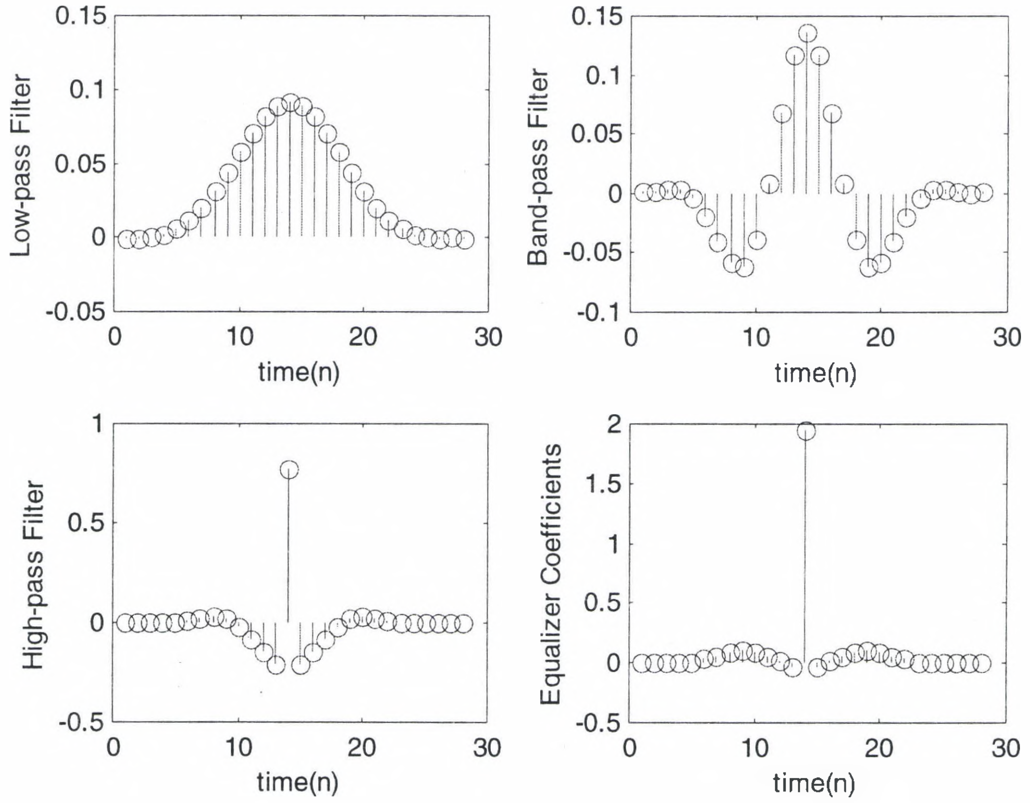


Figure 2-13 --- Impulse Responses of Equalizer Filters, and Equalizer Coefficients

As seen from Figure 2-13, the equalizer coefficients are generated by a weighted sum of low-pass, band-pass, and high-pass filter coefficient values.

Equalization is a good way to develop a desired frequency response. First, the filters' cutoff frequencies are established. Then, the windowing filter coefficients for each of the filters are calculated. After the filter coefficients have been determined, each filter's coefficients are multiplied by a gain value determined by the designer. Then all of the filter coefficients are added together to form the final coefficient values of the equalizer.

Linear Phase:

Phase distortion is often a problem with most filter designs. However, if a filter has a linear phase response, then the output signal phase is not distorted at all. The only filter with linear phase is an FIR filter.

However, not all FIR filters have linear phase. In order for an FIR filter to have linear phase, its impulse response must be symmetric about some point. Figure 2-6 shows an impulse response of an FIR filter with linear phase.

Linear phase is equivalent to a pure delay of the input signal. Therefore, when using and FIR filter with symmetric taps, the filtered output signal is only delayed by a certain amount of time. This is especially beneficial for audio processing. The output of the FIR filter is a pure amplitude scaling of certain frequency components. Phase distortion is not a concern. All of the windowing designs discussed in this chapter are linear phase designs.

CHAPTER 3

HARDWARE ARCHITECTURE DEVELOPMENT OF FIR FILTER COMPONENTS

The basic architecture of an FIR filter is given in figure 2-1. From figure 2-1, it is shown that an FIR filter is comprised of an array of additions and multiplications. Both of these processes are accomplished in hardware using AHDL (Altera Hardware Description Language).

Ripple Addition:

The ripple adder is the simplest method of addition implemented in hardware. However, the price to pay for the simple architecture of the ripple adder is that of latency. Ripple adders, as given in their name, ripple the carry output from the least significant bit to the most significant bit creating large gate delays. Most audio signals are sampled at a rate many magnitudes slower than modern digital systems' clock speeds. For example, CD (Compact Disk) quality music is sampled at a rate of 44.1 kHz, the same sampling rate of the filter in this design, and the maximum clock speeds of most FPGA (Field Programmable Gate Array) devices are approximately 50 MHz. Therefore, long gate delays from the ripple adders are not a foreseeable problem.

The 1-bit full adder can be created using a truth table design. The truth table and logic design of the 1-bit full adder can be found in (5)*Scarpino*, p. 11. Three inputs are used; the two variables *A* and *B*, and the carry input *Cin*. The outputs are the sum of the inputs *S* and a carry output *Cout*. The truth table for the 1-bit adder is given in Table 3-1.

Table 3-1 --- Truth Table for a 1-Bit Full Adder

A	B	Cin	S	Cout
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

From the truth table given in Table 3-1, it is determined that the output sum *S* is determined by the exclusive *or* of the three input terms. That is, the output *S* is equal to 1 only when an odd number of the input terms is equal to one.

The carry out term of the 1-bit full adder is equal to one when two or more of the input terms are equal to 1. Therefore, the logic diagram for a one bit full adder is given in figure 1.

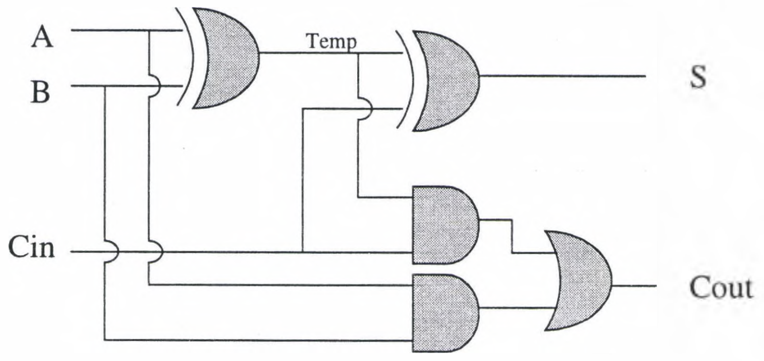


Figure 3-1 --- Logic Diagram of a 1-Bit Full Adder

The AHDL code for implementing the one-bit full adder is given in Figure 3-2.

```
1          Subdesign 'adder'
2          (
3          Cin, A, B          :Input;
4          S, Co              :Output;
5          )
6          Variable
7          Temp               :Node;
8          Begin
9          Temp = A xor B;
10         % Create Sum Term %
11         S = Cin xor Temp;
12         % Create Carry Out Term %
13         Co = (A and B) or (Cin and Temp);
14         End;
```

Figure 3-2 – AHDL Design for implementing a 1-Bit full adder

Lines *1* through *5* of Figure 3-2 declare all of the I/O of the full adder. Lines *6* and *7* give any signals that are not inputs or outputs. And lines *9* through *13* give the logic necessary for the creation of a one-bit full adder. Note that variable *Temp* is a node that is neither an input or output. It is relatively easy to see that the logic in the AHDL code given in Figure 3-2 exactly matches the logic diagram given in Figure 3-1.

A simulation of the one-bit full adder AHDL code is run to verify the validity of the design. Figure 3-3 gives the simulation results of the AHDL code given in Figure 3-2.

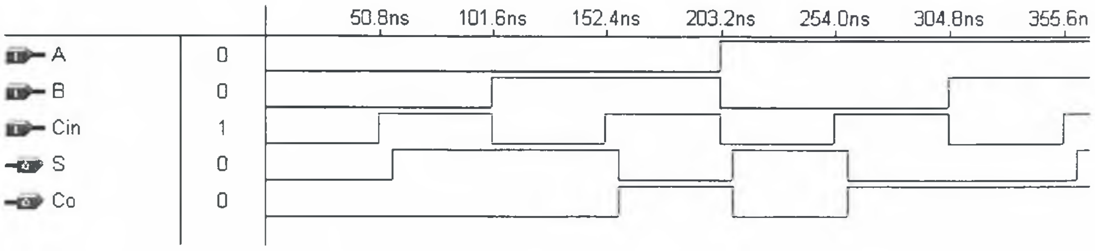


Figure 3-3 --- Simulation Results of a 1-Bit Full Adder

The simulation results show that the one-bit full adder functions exactly as the truth table in Table 3-1 states.

A ripple adder can be created by linking a number of one-bit full adders together and attaching the carry output of the lower bit adder to the carry input of the higher order bit. The Logic Structure of a four-bit ripple adder is given in figure 3-4.

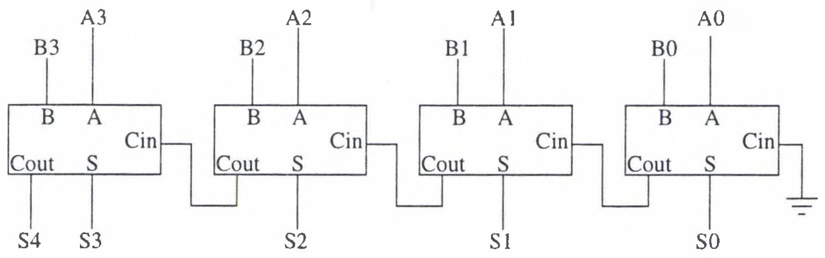


Figure 3-4 --- Logic Diagram of a 4-bit Ripple Adder

As seen in Figure 3-4, an adder of any bit width can be designed by linking together a number of one-bit full adders. However, as also seen in Figure 3-4, the output of the ripple adder is potentially one bit larger than the input values. This becomes a problem when dealing with signed addition.

The two's complement representation of negative numbers is the standard in digital arithmetic. A negative number, in two's complement notation, is equivalent to the positive binary representation of the number negated and added to one. This representation will produce the correct sum when adding both negative and positive numbers. Table 3-2 gives the two's complement representation of several negative numbers.

Table 3-2 --- Binary and Two's Complement Representation of Numbers

Positive Number	Binary Representation	Negative Number	Two's Complement Binary Rep.
1	00000001	-1	11111111
2	00000010	-2	11111110
3	00000011	-3	11111101
5	00000101	-5	11111011
10	00001010	-10	11110110

In Table 3-2, eight-bit binary representation is used. This data length is used for illustration, and many different data lengths are used in the final AHDL application of the FIR filter.

The problem with addition of negative and positive number comes with the value of the most significant bit at the output. For example, the addition of 5 and –3 is 2. However, figure 3-5 gives the result when the ripple adder is used:

```
  5              0101
 -3    ____    + 1101
-14             10010
```

Figure 3-5 --- Ripple Adder Results From Two's Complement Addition

As seen in Figure 3-5, the ripple adder produces a –14 as the sum. This result comes because of the carry output of the most significant adder. Noticing when a situation like this will occur can combat this problem. When two positive numbers are added together, the most significant digits of the two addends by definition are zero. Therefore, when the addition takes place, the rippling of the adder will never reach the most significant digit of the sum. Conversely, when two negative numbers are added together, the most significant digits of the addends by definition are one. The addition of these two arguments is necessarily going to generate a carry output of one. Figure 3-6 exhibits the addition of two positive and two negative numbers.

```
  5        0101               -5        1011
 +3      + 0011               -3      + 1101
  8       01000               -8       11000
```

Figure 3-6 --- Ripple Addition of Two Positive, and Two Negative Numbers

From Figure 3-6, it is obvious that the ripple error in addition only occurs when adding one positive and one negative number. However, when adding one positive and one negative number, the magnitude of the result is always less than the greater of the addends' magnitudes. Therefore, a sign extension is not necessary. This valuable information is used to create an automatic sign extension to ripple adders. Table 3-3 shows a truth table for an automatic sign extension to an N-bit ripple adder.

Table 3-3 --- Truth Table For the Automatic Sign Extension.

A_N(MSB)	B_N(MSB)	S_{N+1}(ExtendedBit)
0	0	0
0	1	S_N
1	0	S_N
[illegible]	1	1

As seen from Table 3-3 and Figure 3-5, the extended sign bit should equal the next most significant bit of the output of the ripple adder when the signs of the addends differ. Figure 3-7 shows the logic diagram of the automatic sign extension.

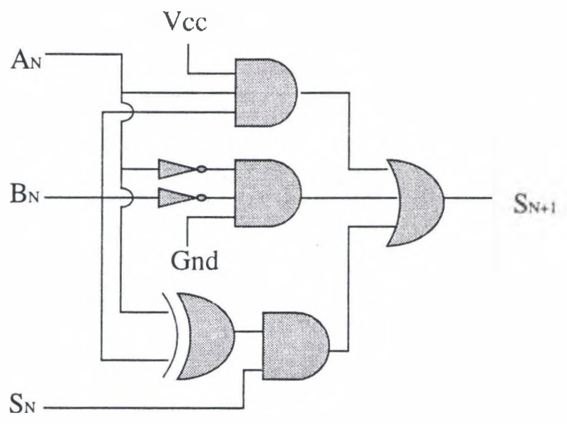


Figure 3-7 --- Logic Diagram of the Automatic Sign Extension

The AHDL implementation of the automatic sign extension is used in all of the adders used in the design of the FIR filter. The AHDL code for the design of an 8-bit adder is given in Figure 3-8.

```
% 8-bit Adder %
include "adder.inc";
Subdesign 'dspadd8'
(
dataa[7..0], datab[7..0]                          :Input;
result[8..0]                                      :Output;
)

Variable
Adder8[7..0]                               :Adder;

Begin
% ***** series of carry assignments ***** %
Adder8[0].Cin=Gnd;
Adder8[7..1].Cin=Adder8[6..0].co;

% ***** Addend assignments ***** %
Adder8[7..0].A=dataa[7..0];
Adder8[7..0].B=datab[7..0];

% ***** Sum assignments ***** %
result[7..0]=Adder8[7..0].s;
```

```
23          % ***** Create Automatic Sign Extension ***** %
24          result[8] = (dataa[7] and datab[7] and Vcc) or
25                      ((not dataa[7]) and (not datab[7]) and gnd) or
26                      ((dataa[7] xor datab[7]) and result[7]);
27          End;
```

Figure 3-8 --- AHDL Design of an 8-bit Ripple Adder With Automatic Sign Extension

In the design of the 8-bit adder, the design of the one-bit full adder is included in line *2* of Figure 3-8. The 8-bit adder is comprised of eight one bit full adders linked together, as shown in figure 3-4. Lines *4* through *7* give the I/O declarations of the design. Lines *9* and *10* create eight instances of a one-bit full adder by declaring eight variables of the name *adder*, and lines *13* through *22* connect the adders to the proper I/O and to each other in accordance with Figure 3-4. Lines *23* through *26* are the creation of the automatic sign extension. It is easily shown that the code exactly matches the logic diagram given in Figure 3-7.

A simulation of the 8-bit adder AHDL code is run to verify the validity of the design. Figure 3-9 gives the simulation results of the AHDL code given in Figure 3-8.

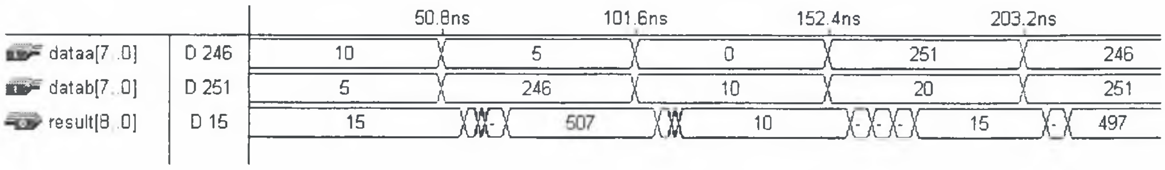


Figure 3-9 --- Altera Simulation Results of the AHDL 8-Bit Ripple Adder

Although it seems that the 8-bit adder is not giving the correct results, the Altera simulator does not understand two's complement representation. The negative numbers in the simulation look like large positive numbers, and the results do not look correct. However, the results of the adder are correct two's complement representation sums. Also, Figure 3-9 shows the ripple delays of the adder. These delays occur from the way the adders are linked as shown in Figure 3-4.

Many times in digital design, the designer must extend the bit-length of each of the addends before addition to ensure that an overflow error does not occur (as shown in Figure 3-5). However, with the implementation of the automatic sign extension to each of the ripple adders implemented in the final design, the sum is automatically sign extended to ensure no occurrence of an overflow condition.

Serialized Multiplication:

Multiplication can be accomplished in two distinct ways: One, with a pure combinational structure which gives the correct output with two input arguments; and two, with a sequential operation of additions and shifting. The process used in this design is the latter sequential structure.

The decision to instantiate a sequential, or *serial* multiplier comes with the difference in logic size. A combinational multiplier takes up much more space on logic devices than a serial multiplier. There is a price to pay however; combinational multipliers produce the product in one clock cycle, whereas serial multipliers take many clock cycles to produce the product.

As said previously, the sampling rate of this filter is only 44.1 kHz. The maximum clock rates of most programmable logic devices range from 20 to 100 MHz; or, at worst scenario, approximately 450 operations can be accomplished per sampling period. From a preliminary calculation, serialized multiplication does not seem to be a significant problem to the success of the design.

Serial Multiplication is accomplished in much the same fashion as multiplying two numbers by hand. Figure 3-10 shows the multiplication of numbers 12 and 13, and how their binary equivalents can be multiplied in the same fashion.

```
     12                      01100
   x 13                  x   01101
   ----                  ---------
     36                      01100
 +  12                      00000
 ------                    01100
    156                   01100
                        + 00000
                       -----------
                          010011100
```

Figure 3-10 --- Multiplication of Decimal and Binary Numbers

As seen in Figure 3-10, multiplication of binary numbers is much like multiplication of decimals. However, there is one advantage of multiplying binary numbers. The only numbers to multiply together are 1 and 0. This makes the process of serialized multiplication simple. Looking again at Figure 3-7, the multiplier (top number) is either copied or not copied to the addition stage of the multiplication, corresponding to the value of the multiplicand (bottom number). Serial multiplication exploits this ease of multiplying binary numbers. The logic design of the serial multiplier is given in (4)*Patterson and Hennessy,* p. 204, and the architecture is given in Figure 3-11.

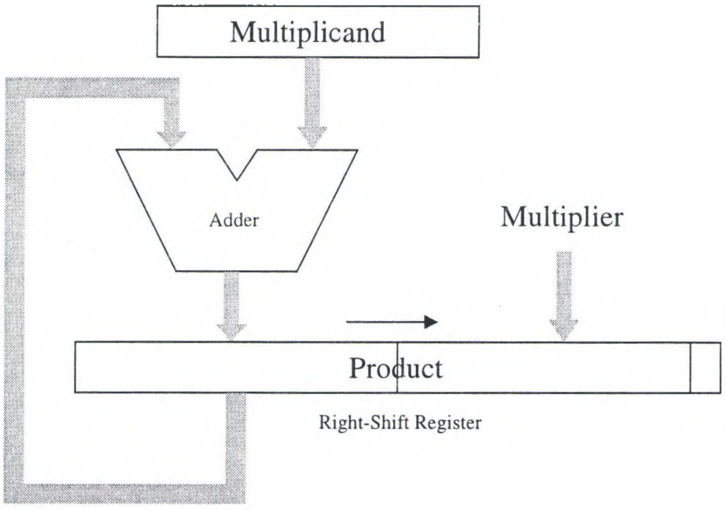


Figure 3-11 --- Architecture of a Serial Multiplier

The architecture of the serial multiplier operates in the same way as given in Figure 3-10. Because the Multiplicand is either copied or not copied to be summed at the output, the Least Significant Bit (LSB) of the Multiplier is used to determine whether the multiplicand is added or not added to the product. The multiplier is stored in the least-significant bits of the product to save logic space. The multiplication of the numbers 12 and 13 are further illustrated in Table 3-4.

Table 3-4 --- Serial Multiplier Process

Iteration	Multiplicand	Product	Description
0	01100	00000 01101	Look at LSB, Add
1	01100	01100 01101	Shift
2	01100	00110 00110	Look at LSB, do nothing
3	01100	00110 00110	Shift
4	01100	00011 00011	Look at LSB, Add
5	01100	01111 00011	Shift
6	01100	00111 10001	Look at LSB, Add
7	01100	10011 10001	Shift
8	01100	01001 11000	Look at LSB, do nothing
9	01100	01001 11000	Shift
10	**01100**	**00100 11100**	**Result**

Table 3-4 shows the operation of a 5-bit serial multiplier. As seen in Table 3-4, 10 clock cycles are necessary to multiply two 5-bit numbers, and the product is a 10-bit number. This multiplication process does not function properly with negative multiplication, however. Therefore, before the two numbers are multiplied together, negative numbers must be changed to positive, and the product is changed back to a negative number, if necessary.

Serial multiplication is desired in the implementation of the FIR filter because it requires less hardware resources then a combinational multiplier. The cost of serial multiplication, however, is time. A 5-bit serial multiplier takes 10 clock cycles to perform the multiplication; an 8-bit multiplier takes 16 clock cycles, etc. However, if time-efficient multiplication is not a concern, as it is with the implementation of an audio filter, then a serial multiplier gives the correct results with the least amount of logic gates.

Signed Multiplication:

As said previously, the serial multiplier used in this design cannot multiply negative numbers. Therefore, the multiplier and multiplicand must first be converted to positive numbers, multiplied, and the product is converted to a negative number if necessary. Through this process of signed multiplication, though, the bit length of the product is no longer equal to 2x the bit length of the multiplicand and multiplier. Because the process is the multiplication of signed numbers, the output length is reduced by one bit. Therefore, an 8-bit signed multiplier produces a 15-bit signed product. Figure 3-12 shows the multiplication of 127 and 127, the largest 8-bit signed value.

```
       01111111
     x 01111111
       01111111
      01111111
     01111111
    01111111
   01111111
  01111111
 01111111
00000000      _
011111100000001
```

Figure 3-12 – 8-Bit Signed Multiplication of Two Large Positive Binary Numbers

Figure 3-8 shows the result of multiplying the largest positive 8-bit number with itself. The product, then, is the largest positive product of 8-bit multiplication. The product is 15-bits wide, including the sign bit.

This phenomenon occurs when multiplying signed numbers because of the application of the sign bit. Two's complement representation of numbers adds one bit to the length of the word to include sign information. Therefore, both the multiplier and the multiplicand have an extra bit added to their bit length to include sign information. In the example of Figure 3-12, both the multiplicand and the multiplier are 7-bit numbers with a sign bit added. Therefore, the product should be a 14-bit number with a sign bit added. The result is a 15-bit product.

The AHDL design of the serial multiplier is given in Figure 3-13. The multiplier used in the final design of the FIR filter is a 9-bit multiplier. The reason for using a 9-bit serial multiplier will be discussed later.

```
% 9- bit Multiplier %
include "dspadd9.inc";
subdesign 'mult'
(
Clock                                   :input;
Mplier[8..0]                            :input;
Mcand[8..0]                             :input;
Result[16..0]                           :output;
)

variable
% 9 bit multiplier Variables %
result[16..0]                           :dff;
% ---------------------------- %
Mplierbus[8..0]                         :dff;
Resbus[17..0]                           :dff;
count[4..0]                             :dff;
multDone                                :dff;
multstart                               :dff;
Np, Nc                                  :dff;
Pos, Neg                                :node;
Load, shift                             :node;
add                                     :node;%
addMult                                 :DSPadd9;

begin
% Create timing generator for multiplication %
count[4..0].clk = clock;
multDone.clk = clock;
multstart.clk = clock;

Count[4..0].d = (Count[4..0].q + 1) and !multdone;
% Start multiplication at count zero %
multstart.d = multDone.q;
% Multiplication is complete at count 19 %
multDone.d = count[4] and !count[3] and !count[2]
                                 and count[1] and count[0];

Np.clk = clock; % Multiplier register %
Nc.clk = clock; % Multiiplicand register %
% Load Multiplication flags at startup %
Np.d = (Mplier[8] and multstart) or
       (Np.q and !multstart);
Nc.d = (Mcand[8] and multstart) or
       (Nc.q and !multstart);

Neg = Np.q XOR Nc.q; % Negative product flag %
Pos = !Neg; % Positive product flag %

% Create input Register to hold Multiplier %
Mplierbus[8..0].clk = clock;
Mplierbus[8..0].d = (Mplier[8..0] and Load and !Np) or
            % Change Negative number to positive %
            ((!Mplier[8..0] + 1) and load and Np) or
            (Mplierbus[8..0] and !Load);

% Create Output Register
```

```
to handle addend and shifting %
result[16..0].clk = clock;
Resbus[17..0].clk = clock;

% Load, add, Shift and hold architecture
for resultant bus %
Resbus[16..9].d = (AddMult.result[7..0] and Add) or
                  (Resbus[17..10] and shift) or
                  (gnd and Load) or (Resbus[16..9] and
                  !add and !shift and !load);
Resbus[17].d = (addMult.result[8] and Add) or
                  (gnd and (shift or load))
                  or (Resbus[17] and
                  !add and !shift and !load);
% 9 least significant bits hold Multiplicand %
Resbus[8..0].d = (Resbus[8..0] and Add) or
                 (Resbus[9..1] and shift) or
                 (Mcand[8..0] and load and !Nc) or
                 % Change Negative number to positive %
                 ((!Mcand[8..0] + 1) and load and Nc) or
                 (Resbus[8..0] and !add and
                 !shift and !load);

% Connect Multiplier adder inputs %
addMult.dataa[8..0] = Mplierbus[8..0];
addMult.datab[8..0] = Resbus[17..9];

% Create Multiplier Controller %
% Load data when counter is equal to zero %
if (Count[4..0] == H"01") then
      Load = Vcc;
      Shift = gnd;
      Add = Gnd;
elsif ((Count[0] == gnd) and (Resbus[0] == Gnd)) then
      Load = gnd;
      Shift = gnd;
      Add = Gnd;
elsif ((Count[0] == gnd) and (Resbus[0] == Vcc)) then
      Load = gnd;
      Shift = gnd;
      Add = Vcc;
Else
      Load = Gnd;
      Shift = Vcc;
      Add = gnd;
end if;

result[16..0].d = (Resbus[16..0] and multDone and Pos) or
                 ((!Resbus[16..0] + 1) and multDone and Neg) or
                 (result[16..0] and !multDone);
End;
```

Figure 3-13 --- AHDL Design of a 9-Bit Serial Multiplier

Lines *1* through *31* of Figure 3-13 are the I/O and variable declarations. The variable of type *dff* is a D-type flip-flop, which is a one-bit register that updates on the rising edge of the clock input. Lines *33* through *39* are the design of the multiplication counter. Lines *40* through *50* are the design of the negative multiplication flags. Because this multiplier cannot multiply negative numbers together, the numbers must first be converted to positive numbers, and the product is changed back to a negative numbers, if necessary. Lines *51* through *80* are the design of the multiplier register and the resultant register. From the block diagram shown in Figure 3-7, it is shown that the multiplicand is stored in the least significant bits of the resultant register in the beginning of the multiplication process. The load flag in the AHDL code determines the beginning of the multiplication process. The *shift* flag in the AHDL code shifts the data in the resultant register to the right by one bit, and the *add* flag adds the multiplier with the most significant bits of the resultant bus. The result of the addition is placed back into the most significant bits of the resultant bus. Lines *86* through *105* are the design of the multiplication controller. The multiplication controller determines whether the multiplier should *load*, *shift, add*, or do nothing. Lastly, *106* through *108* produce the desired output of the mutliplier.

A simulation of the 9-bit multiplier AHDL code is run to verify the validity of the design. Figure 3-14 gives the simulation results of the AHDL code given in Figure 3-13.

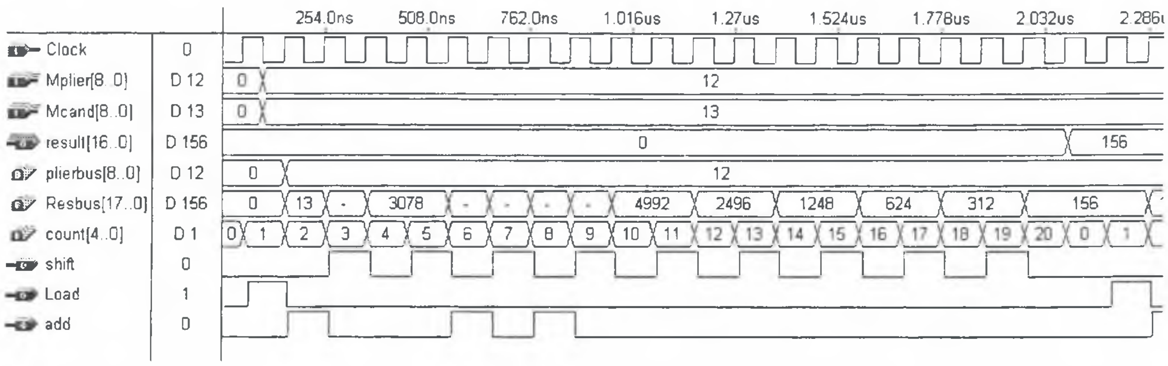


Figure 3-14 --- Altera Simulation Results of a 9-Bit Serial Multiplier

As seen in Figure 3-14, the serial multiplier works exactly as specified by Table 3-4.

CHAPTER 4

HARDWARE ARCHITECTURE DEVELOPMENT OF AN FIR FILTER

The architecture of any design depends on the final application of that design. As said previously, in the design of an audio FIR filter, maximum I/O speed is not an issue. The input and output signals are clocked at a rate of 44.1 KHz for compact disk quality sound. This sampling rate is many orders of magnitude less than the typical clock rate of standard FPGA and EPLD devices. Therefore, the focus of this design is to minimize gate counts by serializing most processes. The serialization of processes will increase the number of system clock cycles per sampling period while reducing the number of logical gates used in the design.

Folding:

The multiplication of input data with coefficient data is very expensive in terms of gate count. The introduction of serial multiplication helps lessen the gate count, but with one multiplier per filter tap, the gate count grows linearly for each additional tap added to the design. One design technique to combat this problem is to use a folded filter design.

Folding is a design method that will reduce the number of multiplications by a factor of two. Figure 1-7 shows the impulse response of a typical low-pass filter. The impulse response is symmetric about a center point, and the impulse responses generated by virtually all windowing methods are also symmetric. This symmetry can be exploited to reduce the number of multiplications in a FIR filter. The LCCDE of a 5-tap FIR filter is given by

$$y(n) = C_2x(n+2) + C_1x(n+1) + C_0x(n) + C_{-1}x(n-1) + C_{-2}x(n-2), \qquad (4.1)$$

where C_n is the impulse response of the system. Knowing that the impulse response of the filter is symmetric about a center point, the LCCDE of the filter can be simplified to

$$y(n) = C_2[x(n+2) + x(n-2)] + C_1[x(n+1) + x(n-1)] + C_0x(n). \qquad (4.2)$$

Thus, the number of multiplications of the filter is reduced from five to three. The hardware architecture of a folded FIR filter is given in Figure 4-1.

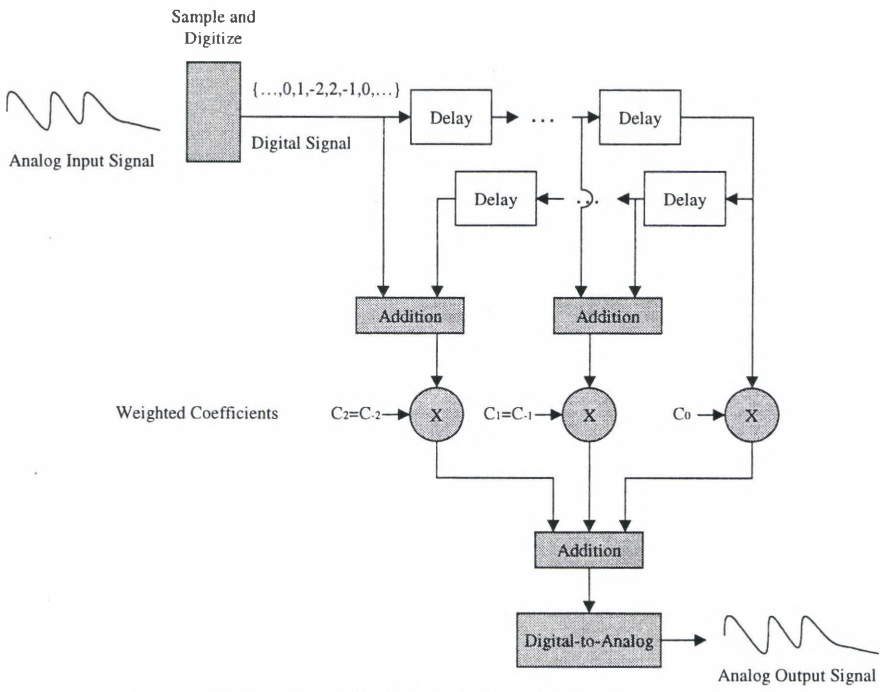


Figure 4-1 --- Folded FIR Filter Architecture

As shown in Figure 4-1, a folded filter design can reduce the number of multiplications by a factor of two. However, the designed filter coefficients must be symmetric, and with most windowing and other FIR filter designs symmetric filter coefficients are generated.

Folding also increases the bit width of the data input into the multiplier. The input data into the filter is 8-bits wide. However, when two eight bit numbers are added together, the result is a possible 9-bit number. Therefore, the multiplier used with a folded filter must be a 9-bit serial multiplier.

Multiplication Multiplexing:

The number of multiplications in the filter design has been reduced to half the number of filter taps, but another design strategy can further reduce the number of multipliers in the design to one.

Multiplexing the multiplications performed in the filter operation can create an FIR filter with only one multiplier. However, the clock speed of the final implementation will increase due to the serialization of the design. Each filter tap data and coefficient will be time multiplexed, thus increasing the number of operations performed per sampling period. Figure 4-2 shows the general architecture of an FIR filter that utilizes time-multiplexed multiplication.

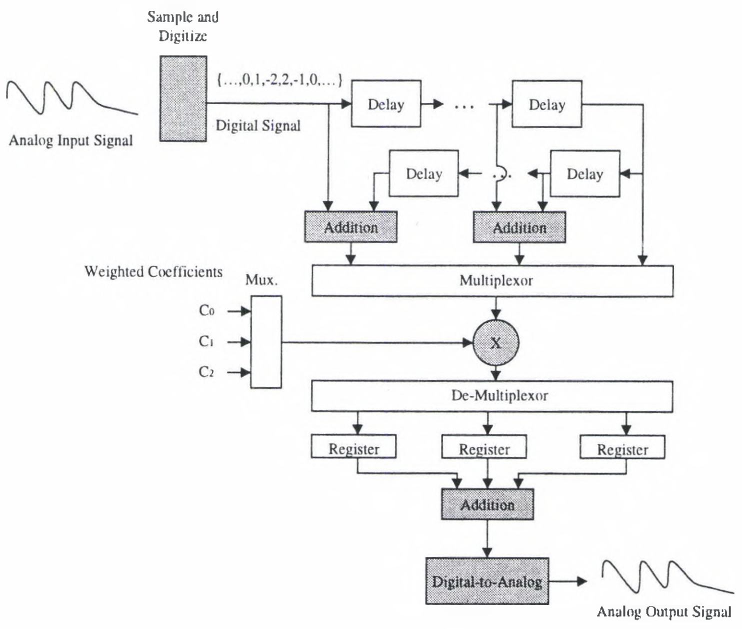


Figure 4-2 – FIR Filter Architecture With Time Multiplexed Multiplication

The instantiation of multiplexed multiplication minimizes the number of logic gates needed to implement the filter design by utilizing the speed of modern programmable logic devices.

Data Width Minimization:

The input data width of the filter is set to 8-bits for input data precision and computational simplicity. However, with the additions and multiplications performed the data width produced at the output is increased. Because the output data width is fixed to 16-bits, the data width must be truncated at some point in the design. However, the truncation points in the design are instrumental in reducing the gate count of the implemented filter.

From Chapter 3 it is shown that the signed addition performed in the design automatically sign extends the sum by one bit. Also, the multiplication of two 8-bit signed numbers produces a 15-bit signed product. Therefore, the data width increases through the flow of the design. Figure 4-3 shows the data widths throughout the FIR filter design architecture.

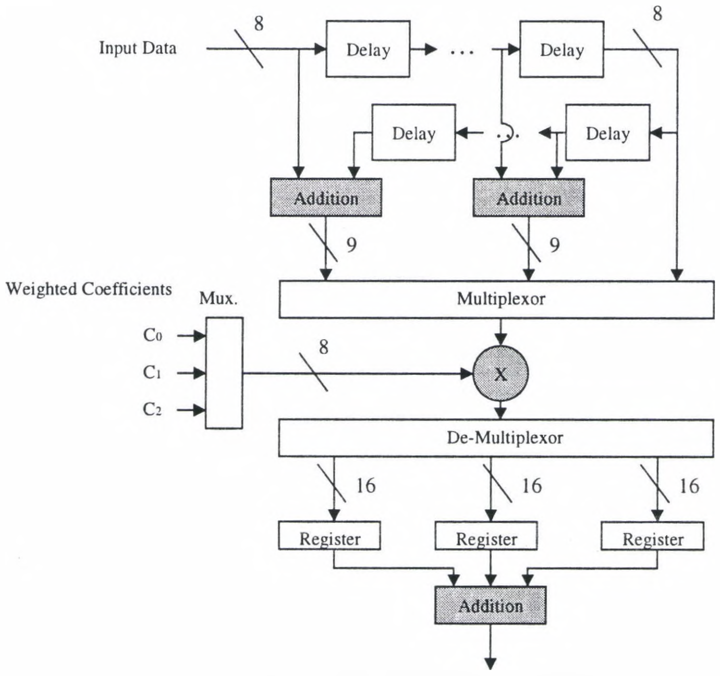


Figure 4-3 --- FIR Filter Data Widths

As seen from Figure 4-3, the width of the data grows to 16-bits before the products are added together to produce the sum. Also, knowing that the addition of two numbers automatically increases the data width by one bit, the output of the filter will have to be truncated to 16-bits. The amount of truncation, however, depends on the length of the filter. The last addition stage of the filter is comprised of several layers of adders. The number of adders in the last stage depends on the length of the filter. For example, a folded 21-tap filter will produce 11 products that will need to be summed to produce the output. The sum of the products will need to be sign extended by four bits to ensure that an overflow error does not occur. The final sum will then be 20 bits wide, and the 20-bit result is then truncated to 16 bits.

But the truncation of data can occur earlier in the design. The products produced by the serial multiplier can be truncated by a certain bit width to ensure a 16-bit result from the final addition process. The product registers in the design can then be reduced in width thus reducing the filter gate count.

The final FIR filter design is a 27-tap filter. This means that the folded filter design has 14 product registers to hold the products for addition. Therefore, fourteen 16-bit integers are added together to form the filtered output, and each adder must sign extend one bit to ensure that an overflow error does not occur. Figure 4-4 shows the final addition stage in the FIR filter design.

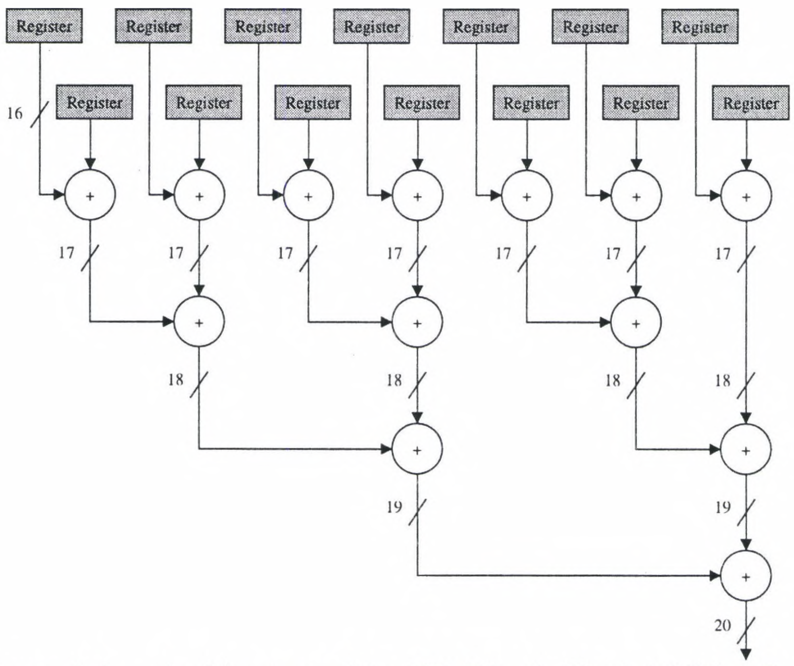


Figure 4-4 --- Architecture of the FIR Filter's Final Addition Stage

As seen in Figure 4-4, the final addition stage adds 4 bits to the output, creating a 20-bit number. The four least significant bits may be truncated to form the desired 16-bit output, or the multiplication registers may be truncated by four bits *before* the final addition stage. The latter design method is chosen to minimize the gate count of the final design. Figure 4-5 shows the final addition stage of the FIR filter with the product registers truncated before addition.

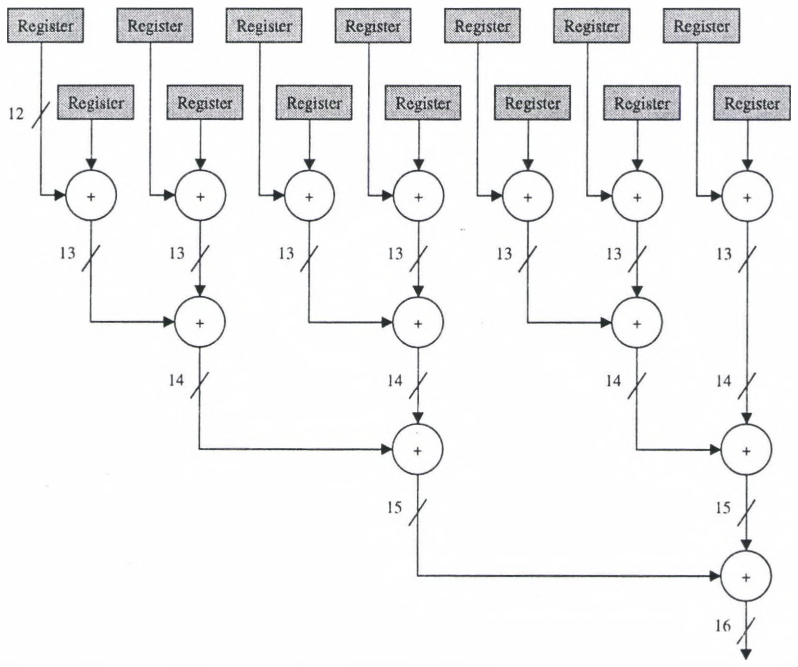


Figure 4-5 --- Final Addition Stage of the FIR Filter Truncated to a 16-bit Output

The truncation of the data early in the design is a lossy truncation. Information may be lost in the process. However, simulation results and testing of the hardware have determined that the losses due to this design method are negligible.

System Clock Calculation:

The FIR filter designed has 27-taps with a folded design. With the advent of a 9-bit serialized multiplier, each multiplication of a coefficient value with a data value takes 18 clock cycles. However, adding in a clock cycle each for converting the input data values to positive numbers, loading in the multiplier and the multiplicand, and registering the product at an output, the number of clock cycles for multiplication become 21.

Because of the folded design, only 14 multiplications have to take place per sampling period. Therefore, the number of clock cycles needed per sampling period becomes:

$$Cycles = 14\frac{Multiplications}{SamplingPeriod}\cdot 21\frac{ClockCycles}{Multiplication} = 294\frac{ClockCycles}{SamplingPeriod}. \quad (4.3)$$

From Equation 4.3, it is given that 294 clock cycles are needed per sampling period. However, The ripple adders used to sum up all the partial products and produce the output of the filter can create great gate delays. A timing analysis of the ripple adders shows that the longest ripple delay of the adders is greater than one clock cycle. Therefore, the sum from the adders may be registered to the output before the adders have produced the correct sum.

To combat this problem, a wait cycle is introduced into the filter design. Instead of 14 multiplications in the design, 15 multiplications will take place. However, the fifteenth multiplication cycle does not compute any results. This cycle is used to ensure that the ripple adders have had enough time to produce the correct resultant sum. Therefore, the number of clock cycles needed for the FIR filter becomes:

$$Cycles = 294\frac{ClockCycles}{SamplingPeriod} + 21\frac{ClockCycles}{SamplingPeriod} = 315\frac{ClockCycles}{SamplingPeriod}. \quad (4.4)$$

Therefore, 315 clock cycles are needed per sampling period.

Knowing that the sampling rate of the filter is 44.1 KHz, and that 315 clock cycles are needed per sampling period, the necessary clock speed of the filter is calculated. The system clock speed of the FIR filter is given by:

$$f_{clock} = 44100KHz \cdot 315 = 13.89MHz \ . \qquad (4.5)$$

The necessary system clock speed given by Equation 4.5 is reasonable for an FPGA design. Maximum clock rates of FPGA's range from the order of 20 MHz to 100 MHz, and the system clock speed calculated is well under that speed.

CHAPTER 5

SERIALIZED LOADING OF FILTER COEFFICIENTS THROUGH THE PC SERIAL PORT

The FIR filter coefficients are loaded into the hardware through the PC serial port. Therefore, the serial data from the computer must first me converted into a parallel data word by a separate piece of hardware. Once the parallel data word is obtained, the conversion hardware sends an interrupt signal to the filter hardware. The filter hardware detects the interrupt and shifts all coefficient values by one to allow the new coefficient to enter the filter.

The UART Communication Port Receiver:

The separate piece of digital hardware to convert the serial data stream to a parallel data word is the UART (Universal Asynchronous Receiver/Transmitter) receiver. The UART will convert an 8-bit word from a serial data stream to an 8-bit parallel word. Figure 5-1 shows the serial data stream of the two's complement representation of the number 5.

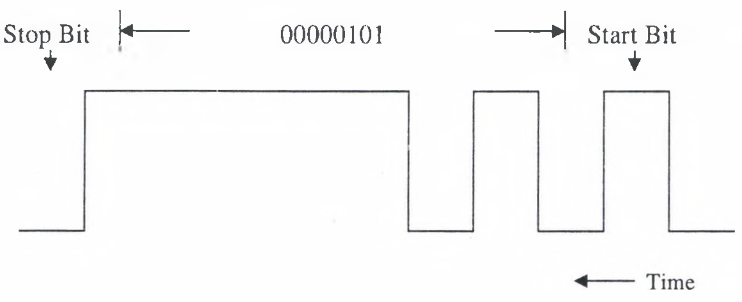
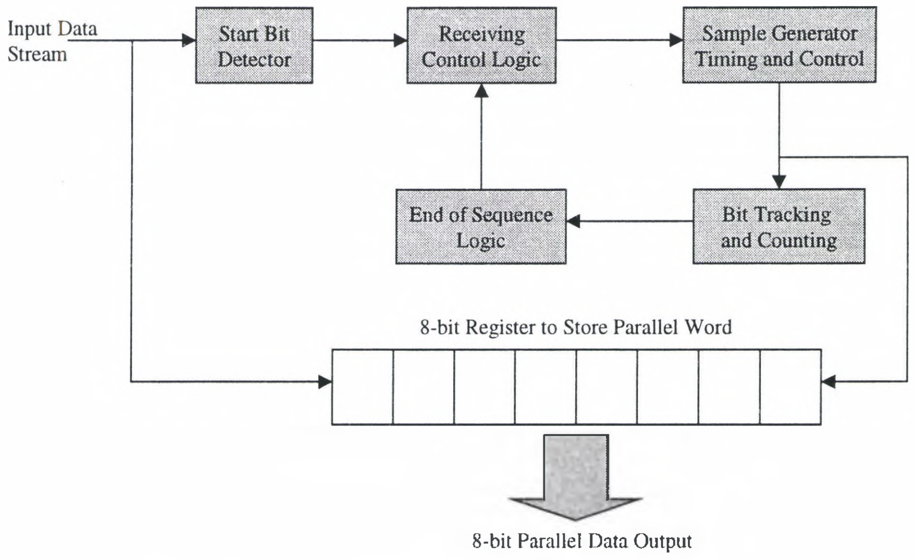


Figure 5-1 --- Serial Bit Stream for the Number 5

As seen in Figure 5-1, the serial bit stream out of the serial port uses a negative representation of numbers. Each logical "1" value sits at the low voltage level, and each logical "0" value sits at the high voltage level. The serial data is also given least significant bit first, and the 8-bit number is preceded by a start-bit at the high voltage level, and followed by a stop-bit at the low voltage level. The start bit and stop bit can be utilized to create an asynchronous receiver that will register the 8-bit data word.

The architecture of the UART receiver is given in Figure 5-2.



Input Data Stream
Start Bit Detector
Receiving Control Logic
Sample Generator Timing and Control
End of Sequence Logic
Bit Tracking and Counting
8-bit Register to Store Parallel Word
8-bit Parallel Data Output

Figure 5-2 --- Architecture of the UART Receiver

As the serial data stream enters the receiver, the *start bit detector* detects the rising edge of the start bit. From there, the Sample generator counts up to the center and edges of each bit in the data word. At the center of each bit, the 8-bit register samples the input stream, and shifts all the contents to the right. Each time the 8-bit register shifts to the right, the *bit tracking and counting* logic increments its counter. When the *bit tracking and counting* counter counts to nine, the *Receiving and Control Logic* are set false, and the system starts to listen for the next start bit.

The clock speed for this device is 3.6864 MHz, a common digital clock speed. However, the baud rate of the input data is 28,800 bits per second. Therefore, the number of clock cycles per bit is given by:

$$ClockCycles = 3686400\frac{Cycles}{Sec.}\frac{1}{28800}\frac{Sec.}{Bit} = 128\frac{Cycles}{Bit}. \tag{5.1}$$

Therefore, the input data stream is sampled every 128 clock cycles. The AHDL code to implement the UART receiver is given in Figure 5-3.

```
Subdesign 'Rcv288'
% ***** Clock frequency is 3.6864 MHz ***** %
% ***** clock period is 271 ns ***** %
% ***** grid size is 135.63 ns ***** %
% ***** data rate is 28,800 bps ***** %
% ***** DataIn period is 34.72 us ***** %
% ***** Each data period is 64 clock cycles ***** %
(
clock, DataIn                             :Input;
Dataout[7..0]                             :Output;
StopReceiving                       :Output;
)

Variable
DataDelay[1..0]                     :Dff;
Receiving                           :Dff;
InRegister[7..0]                    :Dff;
SampleCount[6..0]                   :Dff;
SampleCountReset                    :Dff;
BitCounter[3..0]                    :Dff;
BitClock                            :Dff;
StopReceiving                       :Dff;
BitEdge, Posedge                    :node;
DataOut[7..0]                       :node;

Begin
% ***** Connect Clocks ***** %
DataDelay[1..0].clk = Clock;
SampleCount[].clk = Clock;
InRegister[].clk = BitClock;
SampleCountReset.clk = Clock;
Receiving.clk = Clock;
StopReceiving.clk = Clock;
BitCounter[].clk = BitClock;
BitClock.clk = BitEdge;

% ***** Connect Input Data Line ***** %
DataDelay[1].d = DataIn;
InRegister[7].d = !DataIn;
InRegister[6..0].d = InRegister[7..1].q;
Dataout[7..0] = InRegister[7..0].q;

% ***** Perform Edge Detection ***** %
DataDelay[0].d = DataDelay[1].q;
PosEdge = DataDelay[1] & !DataDelay[0];

% ***** set the receiving signal ***** %
Receiving.d = PosEdge # Receiving.q &
                                    !StopReceiving;

% ***** Generate BitEdge Pulses ***** %
if (Receiving) Then
      SampleCount[].d = (SampleCount[].q + 1) &
                                    !SampleCountReset;
            Case SampleCount[] IS
                  WHEN H"3E" =>
                        BitEdge = Vcc;
                        SampleCountReset.d = Vcc;
                  WHEN OTHERS =>
```

```
60                BitEdge = Gnd;
61                SampleCountReset.d = Gnd;
62          End Case;
63   end if;
64
65   % ***** centers and edges ***** %
66   BitClock.d = !BitClock.q;
67   BitClock.clrn = Receiving;
68
69   % ***** increment bit counter ***** %
70   BitCounter[].d = BitCounter[] +1;
71   BitCounter[].clrn = Receiving;
72
73   % ***** set end of data word ***** %
74   if(BitCounter[] == H"9") Then
75      StopReceiving.d = Vcc;
76   else
77      StopReceiving.d = Gnd;
78   end if;
79
80   End;
```

Figure 5-3 --- AHDL Design for the UART Receiver

Lines *1* through *25* are the I/O and signal declarations. Lines *27* through *35* give the clock inputs to the d-type flip-flops used in the design. Notice, the clock inputs to the registers *InRegister, BitCounter,* and *bitclock* are not the system clock. Lines *37* through *41* give the connections to the output shift register. Since all the *inRegister* flip-flops are clocked by *bitclock*, the shift occurs on the positive transitions of *bitclock*. Lines *43* through *45* create the *posedge* signal. The *posedge* signal becomes positive on the positive transitions of the input data stream. The *posedge* signal triggers the *Receiving* signal to become positive until the *StopReceiving* signal becomes true. The logic of the signal *Receiving* is given in lines *47* through *49*. When the *Receiving* signal is true, the counter *Samplecount* counts every clock cycle. Notice that *Samplecount* resets when it counts to 63. *Samplecount* is used to create *bitclock*, which in turn is used to shift the *InRegister* shift register. Lines *64* through *71* give the logic for the clock *bitclock* and for the counter *bitcounter*. When *bitcounter* counts to nine, the receiver is finished converting the serial input into a parallel data word, and *StopReceiving* becomes true which stops both counters *Samplecount* and *bitcounter*. The code for the *StopReceiving* logic is given in lines *73* through 78. The receiver then continues to listen for the next start bit.

Figure 5-4 gives the simulation results of the UART receiver to ensure proper functionality.

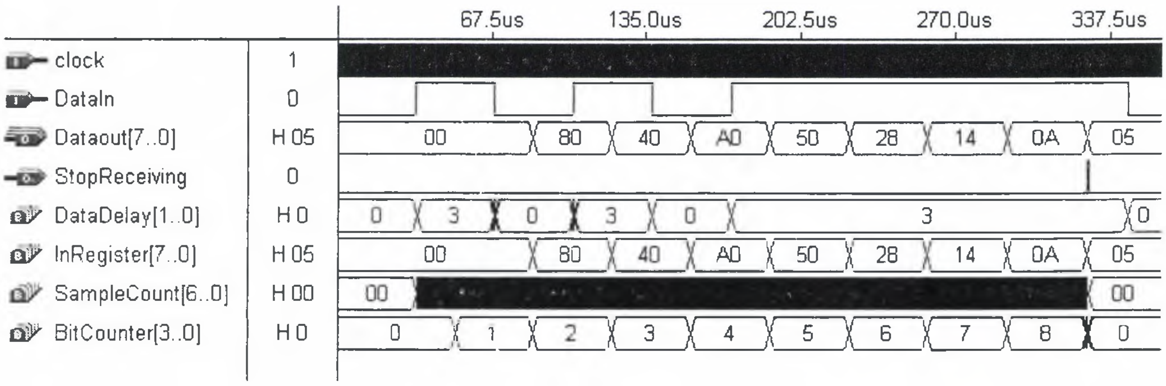


Figure 5-4 --- Altera Simulation Results of the UART Receiver

As seen from the simulation results, the number 5 is detected from the serial data stream, and the *StopReceiving* signal becomes true to reset the system back into listening mode. The UART receiver is covered in more detail in (5)*Scarpino, p. 123.*

Edge Detection for Loading Coefficients:

The *StopReceiving* signal from the UART receiver is used in the FIR filter design to shift in coefficient values asynchronously. The asynchronous handshaking between the UART receiver and the FIR filter are necessary for a more robust filter design.

Because the communication between the FIR filter and the UART receiver are asynchronous, the FIR filter can be operated at a number of different clock frequencies. It is desirable for the UART to operate at a fixed clock rate to communicate with the PC serial output at a fixed 28800-baud rate. However, the filter becomes more versatile if the system clock can be operated at several different speeds and still communicate with the UART which operates at a fixed clock speed. This type of handshaking can only occur with an asynchronous communication design between the UART receiver and the FIR filter.

The asynchronous design is accomplished by designing a positive-edge detection system for the output signal *StopReceiving* of the UART. The edge detection of the *StopReceiving* signal is accomplished by the use of two flip-flops *coeffdelay[1]* and *coeffdelay[0]*. The logic for the edge detection is given in Figure 5-5.

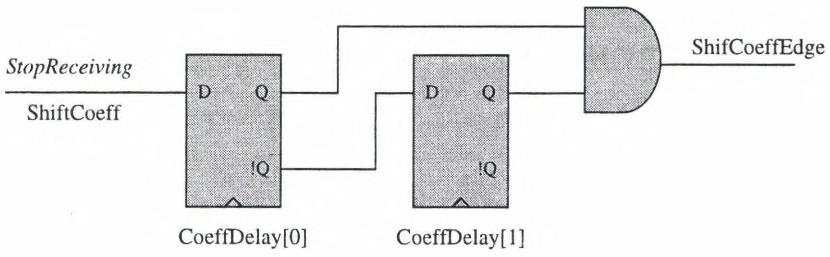


Figure 5-5 --- Edge Detection Logic for FIR Filter Coefficient Shifting

As seen from Figure 5-5, when the *StopReceiving* signal rises from a logic "0" to a logic "1", the *ShiftCoefEdge* signal become a logic "1" for one clock cycle. The coefficient values are shifted when the *ShiftCoefEdge* signal is a logic "1" value. The AHDL code for the coefficient values and shifting is given in Figure 5-6.

```
% Create Receiver/Registers to shift in Coefficients %
% create edge detection for coefficient shift %
coeffdelay[1..0].clk = clock;
coeffdelay[0].d = shiftcoef;
coeffdelay[1].d = !coeffdelay[0].q;
shiftcoefedge = coeffdelay[0].q and coeffdelay[1].q;

% Tap coefficients are clocked by clock %
Tapcoef[13..0][7..0].clk = clock;
```

```
10    % Shift coefficients when input Shiftcoefedge
11    %signal is high %
12    Tapcoef[13..1][7..0].d = ((Tapcoef[12..0][7..0].q)
13                             and shiftcoefedge) or
14                             ((Tapcoef[13..1][7..0].q)
15                             and !shiftcoefedge);
16
17    Tapcoef[0][7..0].d = (coeffin[7..0] and shiftcoefedge) or
18                         (Tapcoef[0][7..0] and !shiftcoefedge);
```

Figure 5-6 --- AHDL Design for Coefficient Values and Shifting

The logic given in Figure 5-5 is designed in lines *3* through 6. Also, lines 8 through 18, give the design that shifts the coefficient values when the signal *shiftcoefedge* is true. Figure 5-7 gives the simulation results of the edge detection and coefficient shifting logic to ensure proper functionality.

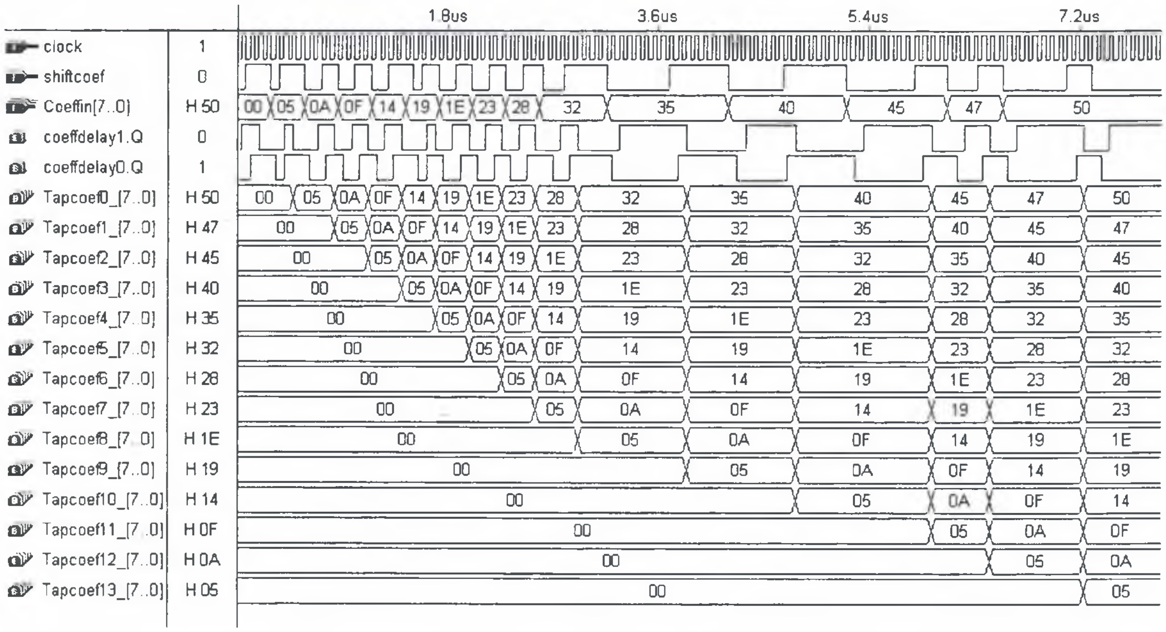


Figure 5-7 --- Altera Simulation Results of the Edge Detection and Coefficient Shifting

As shown in Figure 5-7, the coefficient values of the filter shift during the rising edge of the *Shiftcoef* input signal. In the final implementation of the filter, the output signal of the UART, *StopReceiving*, is connected to the input signal of the filter, *Shiftcoef*.

Therefore, when the input data is done being converted from a serial bit stream to a parallel data word, the FIR filter is notified that a new coefficient is ready, and its contents are shifted into the filter. Also, notice in Figure 5-7 that the *Shiftcoef* signal does not transition uniformally, yet the coefficient shifting still operates properly. This is because of the asynchronous nature of the edge detection circuitry. Because of the edge detection, the coefficients from the serial port may be sent at a constant baud rate, yet the filter may operate at many different clock speeds.

Serial Port Communication Software Development:

A software program must be developed that will send the coefficient values serially through the serial port of the PC to the UART receiver. The program is written in the C++ language and sends fourteen 8-bit coefficient values through the serial port to the FIR filter processing board. Fourteen coefficient values are written to the board because the filter is 27-taps wide, and a folded filter design is implemented.

The software written for the loading of filter coefficient is written with Win32 system calls. These system calls are only guaranteed to work on Windows '95, Windows '98, or Windows NT machines. The software most likely will not operate with other operating systems.

The serial port must first be set to operate in the fashion that is given earlier in this chapter. That includes a baud rate of 28800 bits per second and a stop bit to ensure that the UART resets after each 8-bit data word. The start bit is implicit in serial port communications. These settings are adjusted using the *.dcb* (data control block) structure parameters of the serial port. Figure 5-8 gives the *.dcb* settings of the serial port.

```
1      /* Open the comm port. Com1 */
2      comHandle = CreateFile("COM1",
3                  GENERIC_READ|GENERIC_WRITE,
4                    0,      0,    OPEN_EXISTING,
5                   FILE_ATTRIBUTE_NORMAL, 0);
6
7      if (comHandle == INVALID_HANDLE_VALUE)
8      MessageBox(NULL,"Comm Port Not Opened", "Error",MB_OK);
9       /* Get the current settings of the COMM port */
10     success = GetCommState(comHandle, &dcb);
11     if (!success)
12       MessageBox(NULL,"Cannot Get Com1 State", "Error",MB_OK);
13     /* Modify the baud rate, etc. */
14     dcb.BaudRate = 28800;
15     dcb.ByteSize = 8;
16     dcb.Parity = NOPARITY;
17     dcb.StopBits = ONESTOPBIT;
18     dcb.fBinary = TRUE;
19     dcb.fParity = FALSE;
20     dcb.fOutxCtsFlow = FALSE;
21     dcb.fOutxDsrFlow = FALSE;
22
23     dcb.fDtrControl = DTR_CONTROL_DISABLE;
24     dcb.fDsrSensitivity = FALSE;
25     dcb.fTXContinueOnXoff = TRUE;
26     dcb.fOutX = FALSE;
27     dcb.fInX = FALSE;
28     dcb.ErrorChar = (char)NULL;
29
30     dcb.fNull = FALSE;
31     dcb.fRtsControl = RTS_CONTROL_DISABLE;
32     dcb.fAbortOnError = FALSE;
33     dcb.wReserved = 0;
34
35     /* Apply the new comm port settings */
36     success = SetCommState(comHandle, &dcb);
37     if (!success)
38       MessageBox(NULL,"Cannot Apply Com1 Settings",
39                  "Error",MB_OK);
40
41     /* Set the Data Terminal Ready line */
42     EscapeCommFunction(comHandle, SETDTR);
```

Figure 5-8 ---.*dcb* Settings for the Serial Port

Figure 5-8 is the C++ code that sets the serial port with the parameters that are necessary to communicate effectively with the UART receiver. Lines *1* through *5* show how the handle of the COM1 port is obtained. In Windows 95 and Windows NT, most I/O are treated like files and are read from and written to. In this case, the file handle of the serial port, COM1, is obtained using the *CreateFile* command. After the handle of the serial port is obtained, the settings of the port are modified via the *.dcb* structure. Lines *15* through *18* show that the baud rate is set to 28800 bits/second, the size of the bytes to be sent are 8-bits wide, no parity check is needed, one stop bit at the end of the data stream is needed, and the data to be sent is in binary format. Lines *19* through *28* are other settings that have to do with flow control and two-way communication. As seen in Figure 5-8, most of the other control parameters are set to false because they are not needed in this design. A more detailed description of the data control block structure and serial port communication can be found in (1)*Brain*, p. 690. Lines *35* through *39* send the data control block parameters to the serial port. The *SetCommState* Win32 call is used to write the control parameters to COM1.

After the file handle of the serial port is obtained, and the control parameters are set, writing to the serial port becomes simple. Using the file metaphor in Windows, the *WriteFile* system call is used to send the data to the serial port. However, the serial port looks for character strings from the user. Therefore, the integer coefficient values must first be type cast into characters before being written to the serial port.

The complete C++ program for writing coefficient values to the FIR filter via the serial port is given in Appendix C.

CHAPTER 6

ANALOG-TO-DIGITAL, DIGITAL-TO-ANALOG INTERFACE ISSUES AND PRINTED CIRCUIT BOARD FABRICATION

Audio signals must first be converted into digital signals before being filtered digitally, as shown in Chapter 1. Also, the digital output of the FIR filter must be converted back into an analog signal before being amplified and sent to a speaker. Most analog-to-digital converters must be controlled to give the appropriate output at the correct time. This chapter gives the AHDL development of an analog-to-digial controller for the *ADC1241* by National Semiconductor. The digital-to-analog converter used in this design is the *PCM54* by Burr-Brown. This chapter also gives the architectural design of the printed circuit board used for the final implementation of the hardware FIR filter.

Development of an A/D Controller:

The *ADC1241* analog-to-digital converter contains five logic control lines. These lines determine when the the A/D converter will read the analog input, convert the analog input to a digital two's complement number, and produce the digital number at the output. The five inputs into the A/D converter are *clock*, *ChipSelect (CS)*, *Write (WR)*, *Read (RD)*, and *Calibration (Cal)*.

From the *ADC1241* data sheet, it is determined that the A/D converter functions optimally at a clock speed of 2 MHz. Knowing that the system clock speed of the FIR filter is 13.89 MHz given in Equation 4.5, a counter can be generated to produce a 2 MHz clock. The A/D clock speed is divided into the system clock speed to determine the number of system clock cycles per A/D clock cycles. Approximately 7 system clock cycles are needed per A/D clock cycles.

The controller of the A/D converter must follow some guidelines given in the *ADC1241* data sheet. The timing diagrams for the A/D converter are given in Figure 6-1.

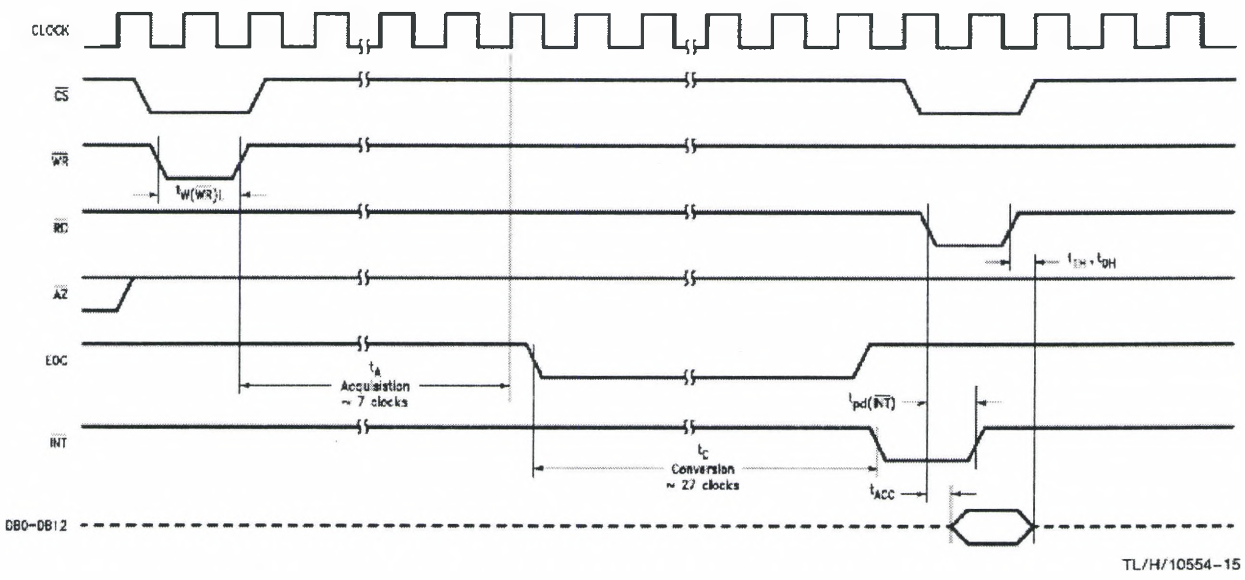


Figure 6-1 --- A/D Converter Input Controls Timing Diagram

The A/D controller is developed using the timing diagram given in Figure 6-1. As seen in Figure 6-1, all of the control lines into the A/D converter are active low signals. First, the *WR* signal is set low to write the analog input signal into the converter. After the necessary time is allotted for conversion is complete, the *RD* signal is set low to read the digital output. Each time either of the *RD* or *WR* signals are toggled low, the *CS* signal must also be active low.

Figure 6-1 shows that the acquisition of the analog signal takes 7 clock cycles, and the conversion of the analog signal to a digital work takes 27 clock cycles. Therefore, a complete analog to digital conversion takes 34 clock cycles.

However, recalling that 315 system clock cycles are necessary per sampling period, and 7 system clock cycles are needed for the A/D clock, the number of A/D converter clock cycles in a sampling period is given by:

$$cycles = \frac{315\frac{SystemClockCycles}{SamplingPeriod}}{7\frac{SystemClockCycles}{A/D.ClockCycles}} = 45\frac{A/D.ClockCycles}{SamplingPeriod}. \tag{6.1}$$

Therefore, the A/D controller counter counts from 0 to 44. As said previously, 34 A/D clock cycles are necessary for an A/D conversion. Therefore, the digital output of the A/D converter is available for several A/D clock cycles. Figure 6-2 gives the AHDL code for the A/D converter controller.

```
% Creation of the A/D Controller %

% Controller counter is clocked by clock %
adcount[5..0].clk = clock;

% all control outputs are clocked by clock %

adreset.clk = clock;
CK.clk = clock;
ChipSelect.clk = clock;
Read.clk = clock;
Write.clk = clock;

% A/D counter counts every seven clock cycles %
% giving the A/D converter controller a "clock" %
% frequency of 2 MHz %
% 2Mhz is a design specification of the A/D converter %
adcount[5..0].d = (adcount[5..0] + 1 and CK and
!adreset and res) or (adcount[5..0] and !CK and res);

% CK becomes valid during the 0, 7, and 14 counts of
the multiplication counter, count %
CK.d = ((!count[4] and !count[3] and count[2] and
 count[1] and count[0]) or (!count[4] and
                  count[3] and count[2] and count[1]
and !count[0]) or (!count[4] and !count[3] and
                  !count[2] and !count[1] and
                  !count[0])) and res;
```

```
28
29    % adcount counts from 0 to 44, (clock freq/sampling
30    freq/seven clock cycles = 45) %
31    adreset.d = adcount[5] and !adcount[4] and adcount[3]
32    and adcount[2] and !adcount[1] and !adcount[0];
33
34    if (res == gnd) then
35          ChipSelect.d = Vcc;
36          Read.d = Vcc;
37          Write.d = Vcc;
38    end if;
39
40    if (adcount[5..0] == 0) then
41          ChipSelect.d = gnd;
42          Write.d = Vcc;
43          Read.d = Vcc;
44    elsif (adcount[5..0] == 1) then
45          ChipSelect.d = gnd;
46          Write.d = gnd;
47          Read.d = Vcc;
48    elsif (adcount[5..0] < 37) then
49          ChipSelect.d = Vcc;
50          Write.d = Vcc;
51          Read.d = Vcc;
52    elsif (adcount[5..0] == 37) then
53          ChipSelect.d = gnd;
54          Write.d = Vcc;
55          Read.d = Vcc;
56    else
57          ChipSelect.d = gnd;
58          Write.d = Vcc;
59          Read.d = gnd;
60    end if;
```

Figure 6-2 --- AHDL Design for the A/D Controller

Lines *1* through *12* of Figure 6-2 connect the clock inputs of all flip-flops to the system clock. Lines *14* through *18* give the logic for the A/D counter *adcount*. The counter only counts on the positive levels of the A/D clock. Lines *20* through *27* given the logic for the A/D clock *CK*. *CK* is only valid when the multiplication counter *count* is a multiple of seven. Knowing that the multiplication counter counts to 21, the clock *CK* is exactly 7 times slower than the system clock. Lines *29* through *32* give the reset logic for *adcount* the A/D counter counts to 44. Therefore, the reset for the counter must be equal to 44. Lines *34* through *60* give the logic for the A/D control outputs. The output signal

ChipSelect is low every time the A/D converter reads or writes. Also, the output signal *Read* is set low after the *Write* signal is set low and the 34 conversion clock cycles have taken place.

A simulation of the A/D controller is run to ensure proper functionality. Figure 6-3 gives the simulation results of the A/D controller.

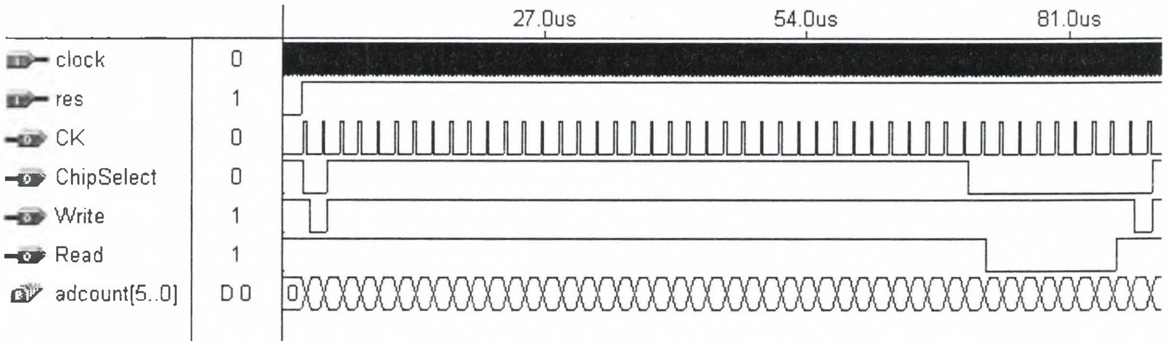


Figure 6-3 --- Altera Simulation Results of the A/D Controller

Figure 6-3 give one complete cycle of the A/D controller. It is shown that the simulation results given in Figure 6-3 match the timing requirement of the A/D converter given in Figure 6-1. A closer look at the simulation results of the A/D controller gives greater insight into its behavior. Figure 6-4 gives a closer look at the A/D controller simulation results.

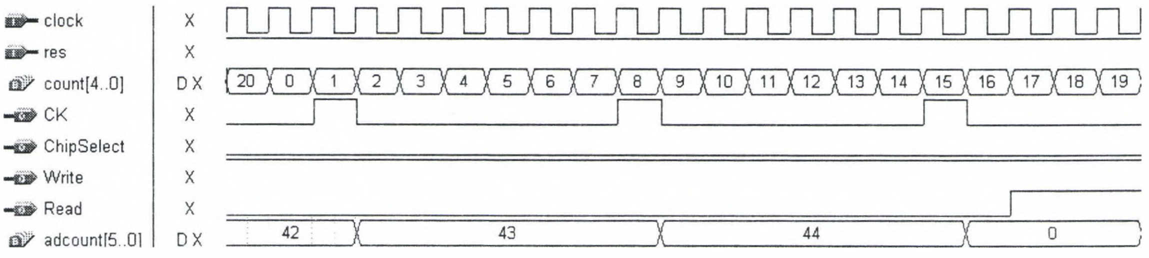


Figure 6-4 --- Further Altera Simulation Results of the A/D Controller

Figure 6-4 shows that the A/D clock *CK* is exactly 7 times slower than the system clock. Also, the A/D controller counter *adcount* does indeed reset after 45 clock cycles.

The concern for *CK* to reset after 7 clock cycles and for the A/D counter *adcount* to reset every 45 clock cycles is to ensure that the output of the A/D converter is valid data when the input data is ready to be read into the FIR filter. Because *CK* and *adcount* reset when they do, the A/D converter outputs valid data every 315 system clock cycles. The timing results of the A/D converter and the data loading is given if Figure 6-5.

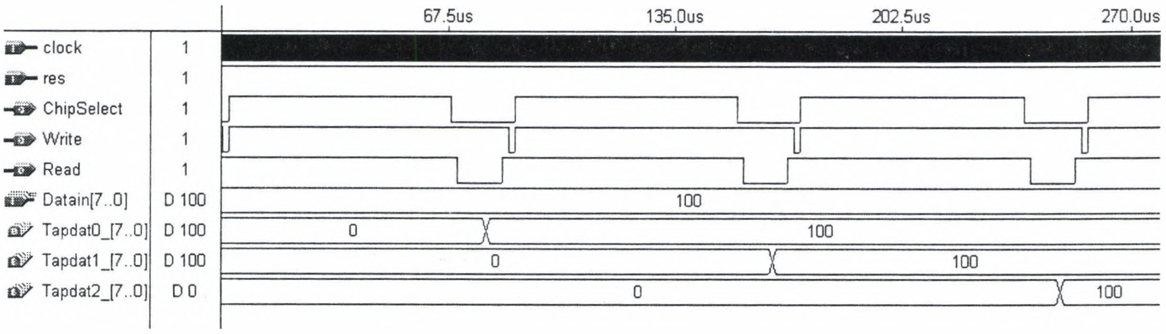


Figure 6-5 --- Timing Results of the A/D Controller and Data Loading

As seen in Figure 6-5, the input data from the A/D converter is loaded into the FIR filter when both the *Read* and *ChipSelect* signals are low.

Complementary Two's Complement Representation for the D/A Converter:

The Digital-to-Analog converter used in the design of the FIR filter is the *PCM54* developed by Burr-Brown Inc. The input/output codes for this D/A converter are given in the data sheet, and also given in Figure 6-6.

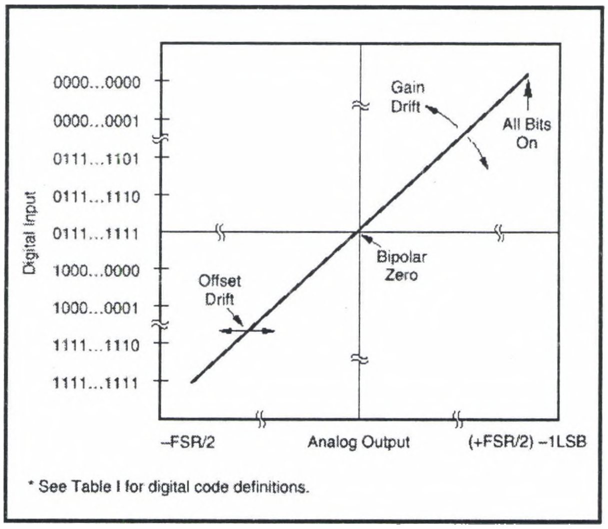


Figure 6-6 --- Input/Output for the D/A Converter

From Figure 6-6, it is shown that the digital representation of the analog output differs from the two's complement representation of numbers that is used in the design of the FIR filter. The type of digital representation that the D/A converter uses is referred to a complementary two's complement representation. Therefore, the output of the FIR filter must be converted from two's complement representation to complementary two's complement representation. Table 6-1 gives the correlation between the two representations of digital numbers.

Table 6-1 --- Two's Complement Representation vs. Complementary Two's Complement Representation of Numbers

Value	Two's Complement Representation	Complementary Two's Complement Rep.
127	01111111	00000000
126	01111110	00000001
2	00000010	01111101
1	00000001	01111110
0	00000000	01111111
-1	11111111	10000000
-2	11111110	10000001
-126	10000001	11111110
-127	10000000	11111111

As seen from Table 6-1, the complementary two's complement representation is the inverse of the two's complement representation with the exception of the most significant bit (MSB). Therefore, if the output of the filter is adjusted to invert all the output signals except for the MSB, the analog output of the filter will be the correct value. The AHDL code to implement the conversion between the two representations is given in Figure 6-7.

```
% Create Output Register%
% Output Register is clocked by clock %
Outdata[15..0].clk = clock;
% Output Register is updated when all processes are finished %
Outdata[15].d = (Add5.result[15] and
                (Multstart and Zero)) or
                (Outdata[15].q and
                (!Multstart or !Zero));

Outdata[14..0].d = (!Add5.result[14..0] and
                   (Multstart and Zero)) or
                   (Outdata[14..0].q and
                   (!Multstart or !Zero));
```

Figure 6-7 --- Complementary Two's Complement Representation Generation

Figure 6-7 gives the generation of complementary two's complement representation of the output, the signal *Add5.result* is the output of the final addition stage in the design. Therefore, when all the processes are finished, *Add5.result* is set equal to the output register *Outdata*. However, lines *5* through *8* set the MSB of the output equal to the result of the final addition stage, and lines *10* through *13* set the rest of the output bits equal to the negative of the result of the final addition stage. Therefore, the output is set to the complementary two's complement representation of digial numbers. The output of the filter is then connected directly to the digital inputs of the D/A converter. The AHDL code for the FIR filter is given in Appendix D.

Development of a Printed Circuit Board:

A printed circuit board is developed to house all the necessary components for the FIR filter. Table 6-2 gives the components necessary for the FIR filter.

Table 6-2 --- Necessary Components for the Development of the FIR Filter

	Component	Functionality
1	Altera FPGA (FLEX10K30)	FIR Filter
2	Altera EPC2 EPROM	FPGA Programming Device
3	Altera EPLD (MAX7032)	UART Receiver
4	ADC1241	Analog-to-Digital Converter
5	PCM54	Digital-to-Analog Converter
6	13.89 MHz Crystal Oscillator	Filter System Clock
7	3.6864 MHz Crystal Oscillator	UART System Clock
8	1458 Operational Amplifier	Pre-Amplifier

With the components given in Table 6-2, the FIR filter can be developed. The architecture of the printed circuit board designed for the development of the FIR filter is given in Figure 6-8.

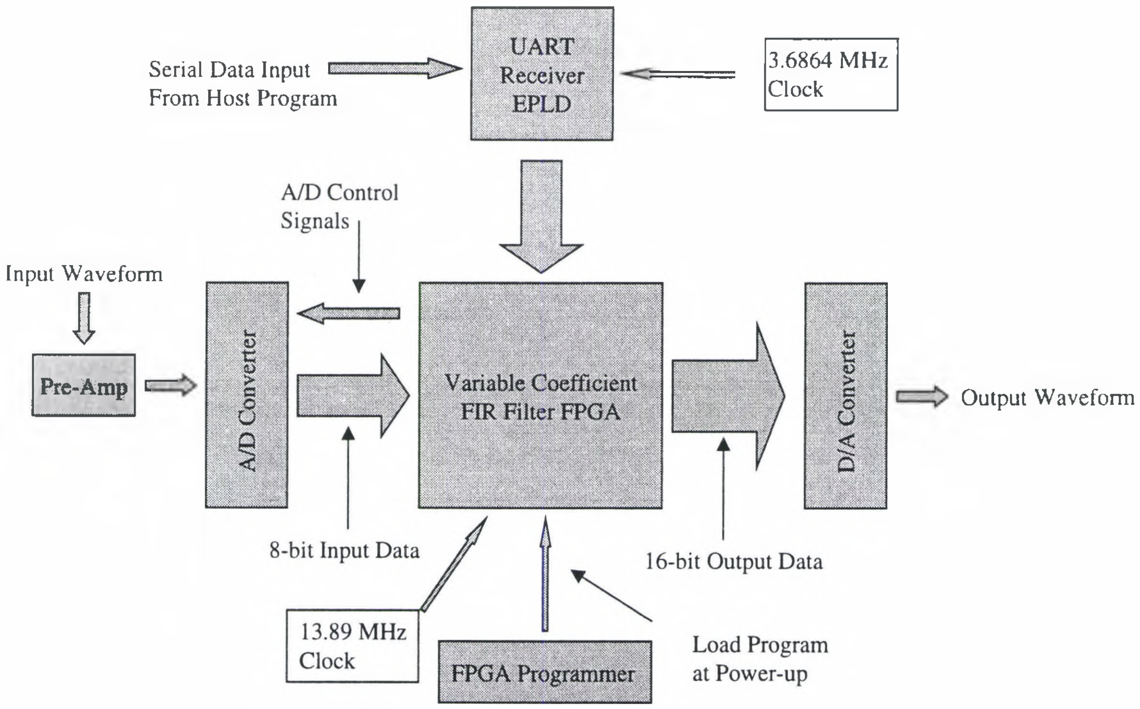


Figure 6-8 --- FIR Filter Printed Circuit Board Architecture

As seen in Figure 6-8, the input waveform is amplified before being converted into a digital signal by the A/D converter and the FPGA receives the FIR filter program on power-up of the processing board. The filter program then residing in the FPGA receives inputs from both the A/D converter and the UART receiver. The A/D converter provides the input data, and the UART receiver provides the filter coefficient data given by the host software. The filtered output of the system then gets input into the D/A converter and converted back into an analog signal. For musical applications, the analog output is amplified and sent to a speaker. The result is digitally enhanced audio. The final design of the variable coefficient FIR filter processing board is given in Figure 6-9.

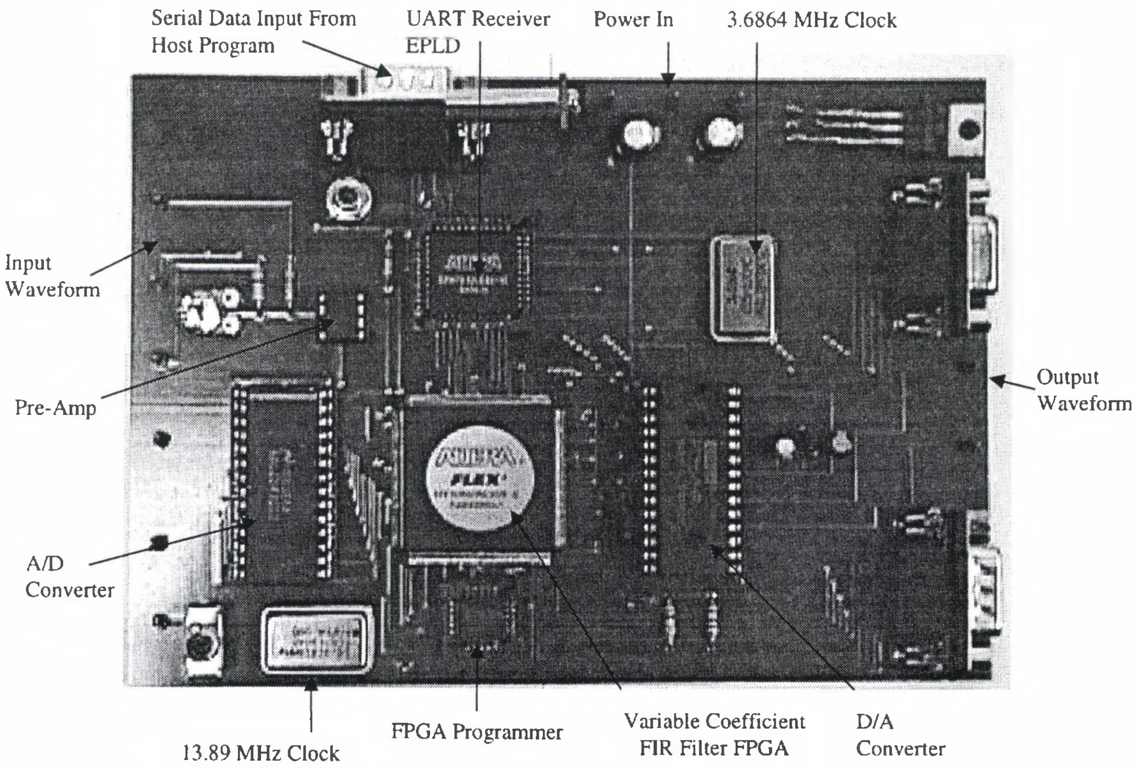


Figure 6-9 --- Variable Coefficient FIR Filter Processing Board

The processing board given in Figure 6-9 is designed and built. The sampling rate of the A/D converter is 44100 KHz, and the baud rate of the serial data input is 28.8 kbits/sec.

CHAPTER 7

MATHEMATICAL, SIMULATED, AND EXPERIMENTAL RESULTS OF FILTER PERFORMANCE

In order to verify the performance of the filter implemented onto the printed circuit board, several models of the filter's performance must be developed. In other words, some tools must be developed to calculate the desired filter output due to certain inputs. Then, the experimental results taken from the filter can be compared to the mathematical results and performance can be evaluated.

The mathematical tool used to calculate the ideal filter response is Matlab's *Simulink*. *Simulink* is a graphical tool that runs on top of the Matlab software. The results given from *Simulink* are compared both to the Altera simulation results, and the experimental results of the hardware.

Simulink Model for an FIR Filter:

As said previously, *Simulink* is a graphical software tool. The FIR filter developed in *Simulink* is then a graphical representation of the FIR filter's architecture. Figure 7-1 gives the graphical program developed in *Simulink*.

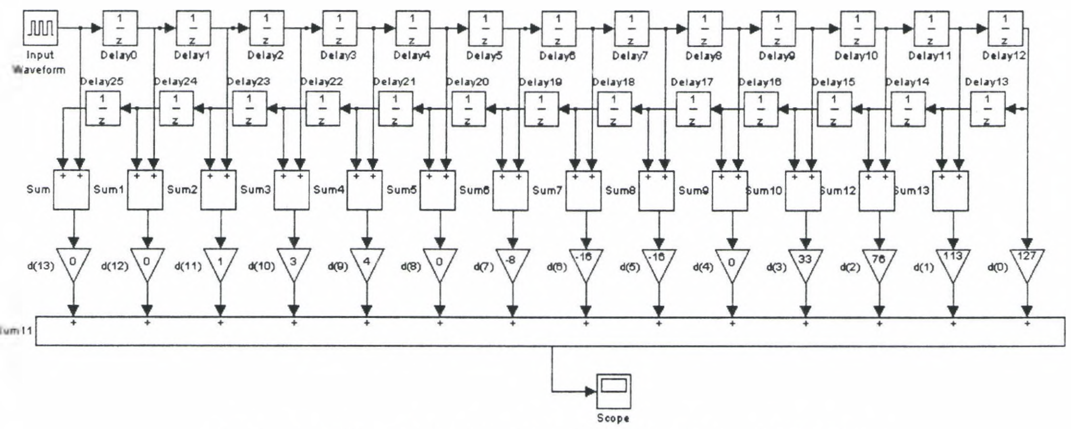


Figure 7-1 --- *Simulink* Model of a 27-tap FIR Filter

Figure 7-1 gives the *Simulink* model for a 27-tap folded FIR filter. Note, all of the delays used in the filter are unit delays of 22.676 μ s. Therefore, the sampling rate of the filter designed in *Simulink* is given by:

$$f_s = \frac{1}{22.676e-6s} = 44.1KHz \tag{7.1}$$

Not surprising, the sampling rate of the *Simulink* filter is the same as the sampling rate of the FIR filter implemented in Hardware. Also, the *Simulink* filter is a folded 27-tap filter, just as the hardware filter. Therefore, the filter developed in *Simulink* is a good mathematical representation of the hardware filter designed in AHDL. Actually, the only difference between the hardware filter and the filter build in *Simulink* is that the input data values of *Simulink* are infinite precision numbers (for all practical purposes) and all of the mathematical operations of the *Simulink* filter are floating-point operations. However, the performance of the hardware filter still may be compared to that of the *Simulink* filter.

Step Responses of Various Filters:

The impulse response of an FIR filter gives great insight into the filter's behavior. In fact, given an FIR filter's impulse response, the response to any other input signal is completely predictable, as stated in Chapter 2. Also, the step function is intimately related to the impulse function. The relationship of the impulse function an the step function is given by:

$$u(t) = \int_t \delta(t)dt\,, \tag{7.2}$$

where $u(t)$ is the unit step function, and $\delta(t)$ is the unit impulse function. From Equation 7.2, it is shown that the step function is the integral with respect to time of the impulse function. Therefore, if the step response of an FIR filter is given, the output of the filter is completely predictable. Given that the step response gives great insight into an FIR filters behavior, it is the input function that used to test the performance of the filter.

First, the coefficients of the filter must be determined. The filter coefficients used in the *Simulink* filter are the Kaiser window coefficients of a low-pass filter with a cutoff frequency of $\frac{\pi}{4}$. The Kaiser window coefficients are given in Table 7-1

Table 7-1 --- Kaiser Window Low-Pass Filter Coefficients Used in Performance Analysis

Coefficient Number	Coefficient Value
0	127
1	113
2	76
3	33
4	0
5	-16
6	-16
7	-8
8	0
9	4
10	3
11	1
12	0
13	0

The coefficients given in Table 7-1 are set to the gains of the *Simulink* model. The *Simulink* simulation is then run to give the step response of the low-pass filter. The frequency response of this filter is given in Figure 2-10, and the step response of the *Simulink* filter is given in Figure 7-2.

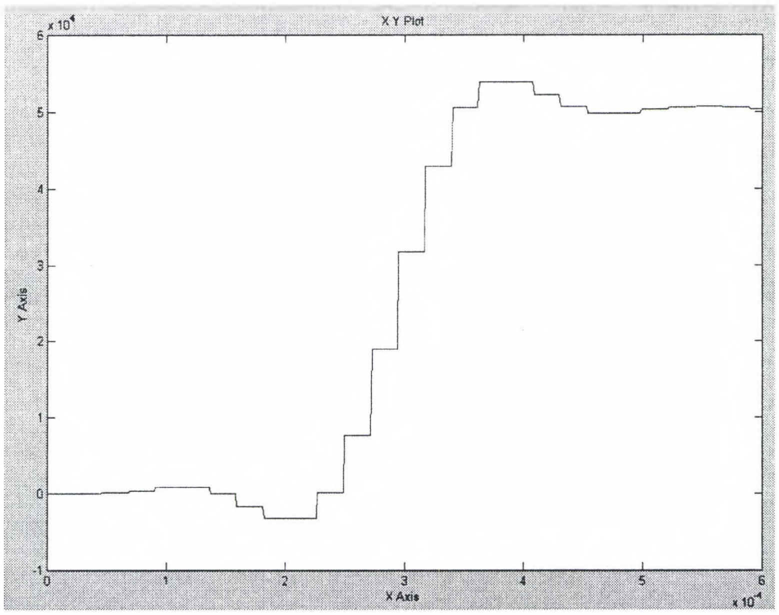


Figure 7-2 --- *Simulink* Step Response of an FIR Low-Pass Filter

As seen in Figure 7-2, the step response of the low-pass filter slowly rises to the final value of the input step. The output of the Altera simulator is also examined. The coefficients given in Table 7-1 are loaded into the Altera simulator, and the output is given. The Serial Loading of Filter coefficients in the Altera simulator is given in Figure 7-3.

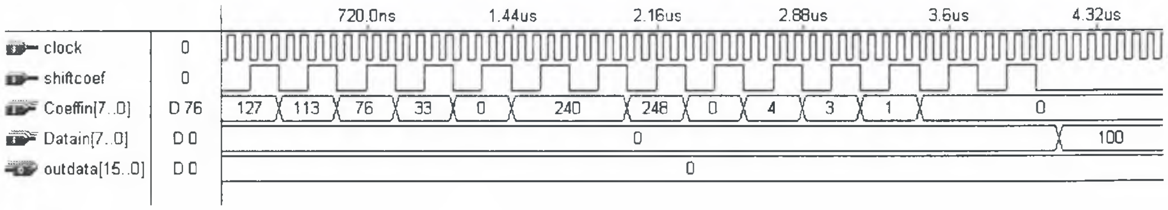


Figure 7-3 --- Altera Simulation Results of the Serial Loading of Filter Coefficients

The Altera simulator is run for approximately 600 μ s, and the input data transitions from 0 to 100 after the coefficients are loaded into the filter. The output response of the simulation is given in Figure 7-4.

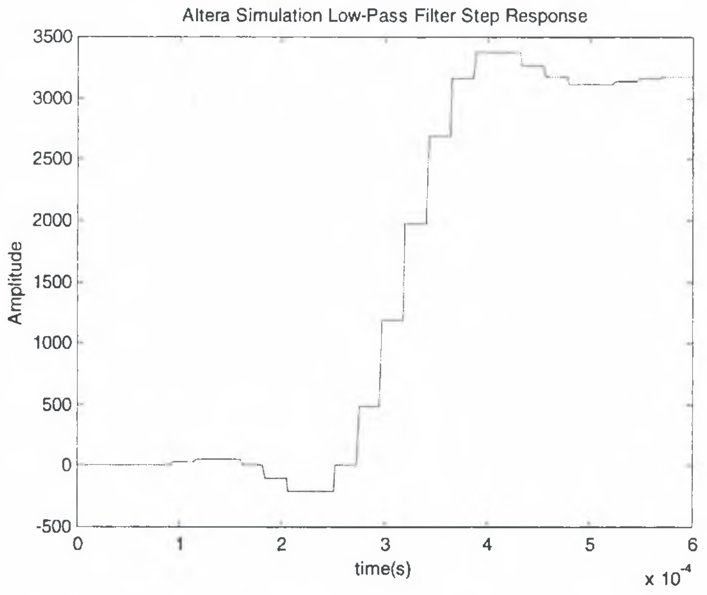


Figure 7-4 --- Altera Simulation Step Response of the Low-Pass Filter

As shown in Figure 7-4, the Altera simulation results slowly rise to the final step value of the input almost exactly as the *Simulink* filter step response. The two step responses are almost identical. However, the step response amplitudes are different. This is due to the truncation that takes place in the hardware FIR filter. Because the output data is restricted in 16 bits wide, the output step response amplitude is restricted.

The filter implemented in hardware is given the coefficient values of the low-pass filter as well, and a real-time performance analysis is made by inputting a low frequency square wave at the input and monitoring the output on an oscilloscope. The output of the FIR filter is given in Figure 7-5.

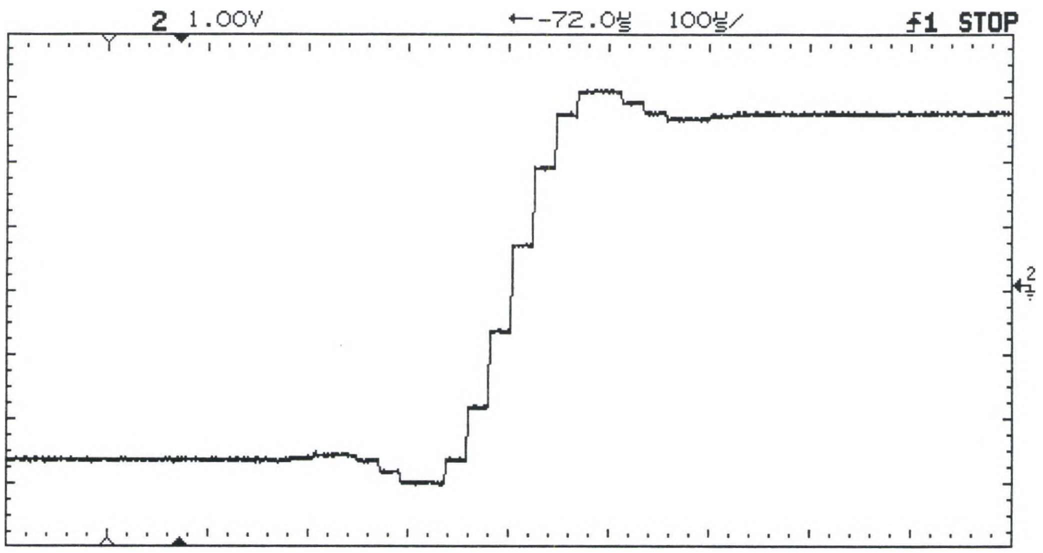


Figure 7-5 --- Hardware FIR Filter Step Response with Low-Pass Filter Coefficients

As seen in Figure 7-5, the response of the filter with low-pass coefficients exactly matches the form of the Altera simulation response given in Figure 7-4 and the *Simulink* results given in Figure 7-2.

High-pass filter coefficients are then generated and run through both the *Simulink* filter and the hardware filter simulation. The high-pass filter coefficients generated are a Kaiser window design with a cutoff frequency of $\frac{\pi}{4}$. The filter coefficients input into the filter are given in Table 7-2.

Table 7-2 --- Kaiser Window High-Pass Filter Coefficients Used in Performance Analysis

Coefficient Number	Coefficient Value
0	127
1	-38
2	-25
3	-11
4	0
5	5
6	5
7	3
8	0
9	-1
10	-1
11	0
12	0
13	0

The filter coefficients are input into the *Simulink* filter and the output step response is given in Figure 7-6.

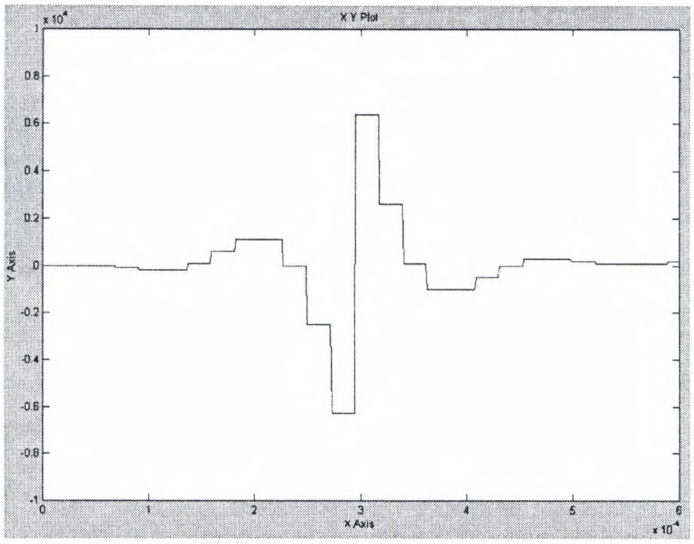


Figure 7-6 --- *Simulink* Step Response of an FIR High-Pass Filter

The high-pass coefficients are then input into the Altera simulator of the FIR filter, and the step response is given in Figure 7-7.

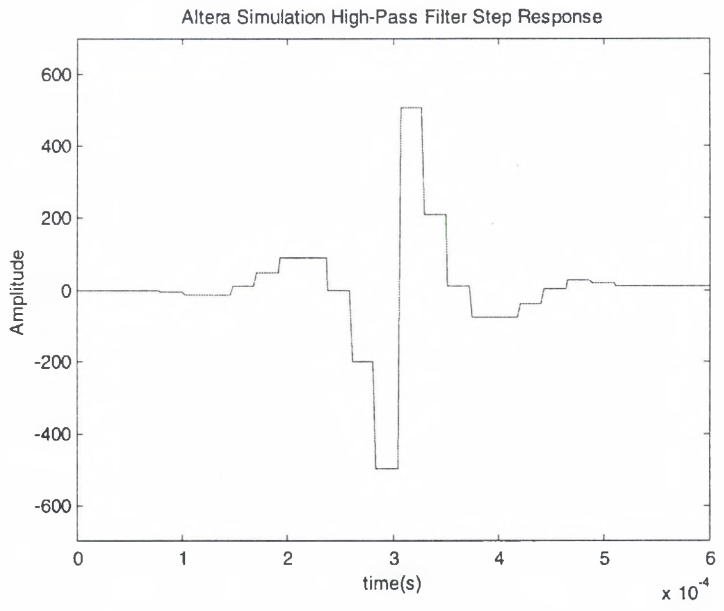


Figure 7-7 --- Altera Simulation Step Response of the High-Pass Filter

Again, as seen in Figure 7-7, the Altera simulation results for the step response of the high-pass filter coefficients exactly matches the step response of the *Simulink* simulation.

The hardware FIR filter is given the high-pass coefficients from Table 7-2 and a real-time performance test is given in the same was as in the low-pass filter test. The performance results of the hardware FIR filter with high-pass coefficients are given in Figure 7-8.

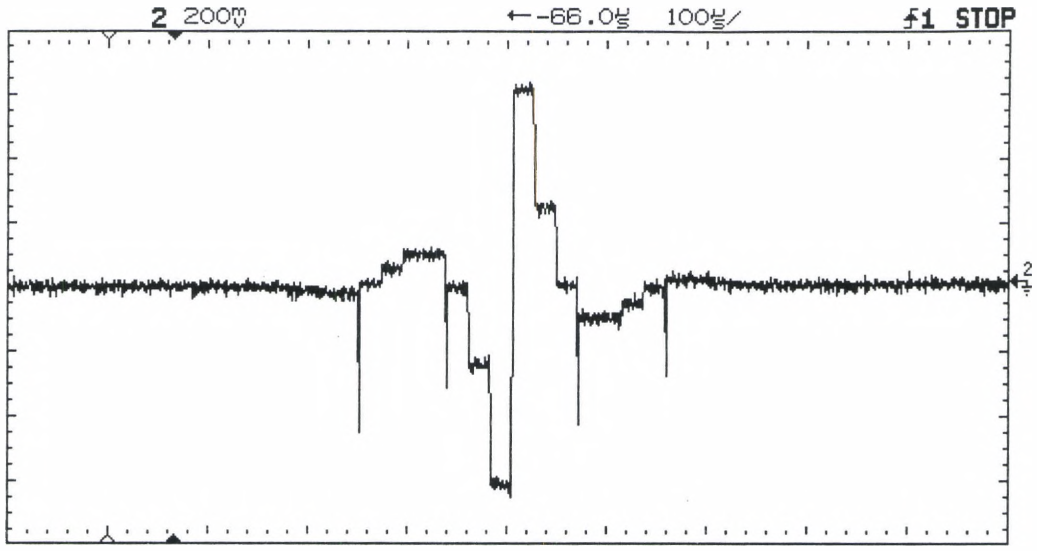


Figure 7-8 --- Hardware FIR Filter Step Response with High-Pass Filter Coefficients

As seen in Figure 7-8, the step response of the filter matches the step response of the Altera simulation given in Figure 7-6 and the *Simulink* results given in Figure 7-5.

The variable coefficient FIR filter designed and built in hardware seems to function exactly as the software simulation results. Because the step response of an FIR filter can mathematically predict the filter response due to any input, and the step responses of various FIR filter coefficients match the software simulations, the filter seems to perform as specified. However, to ensure proper functionality, some elementary signals are filtered.

Filtering of a Square wave:

One of the simplest waveforms to filter is the square wave. From Fourier analysis, it is known that a square wave is proved to be comprised of an infinite sum of sinusoids which exist at harmonic frequencies. These sinusoids that make up a square wave can then be extracted by filtering.

The low-pass coefficients given in Table 7-1 are input into the FIR filter. These coefficients are designed to cutoff at a digital frequency of $\frac{\pi}{4}$. However, the equivalent analog cutoff frequency must be determined. The analog cutoff frequency of the Filter is calculated by equation 1.6.

$$f = \frac{f_s \omega}{2\pi} = \frac{44100 \cdot \frac{\pi}{4}}{2\pi} \cong 5512.5Hz \tag{7.3}$$

A 3 KHz square wave is input into the filter with the set of coefficient given in Table 7-1. The output of the filter should be only the fundamental frequency of the square wave, or in other words a sine wave. The filter should effectively eliminate all other harmonic frequencies. Figure 7-9 gives the output of the FIR filter with low-pass coefficients.

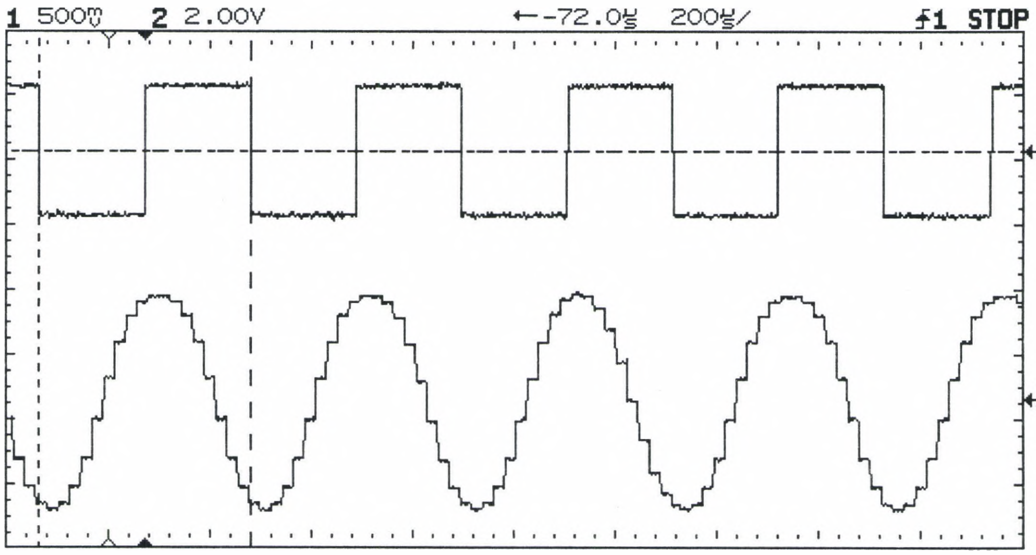


Figure 7-9 --- Low-pass Filtering of a Square Wave by the FIR Filter

As seen in Figure 7-9, the input waveform is shown in Trace 1 and the output waveform is shown in Trace 2. The output of the filter is indeed a sine wave. All other harmonic frequencies are effectively eliminated.

The same square wave is input into the hardware filter, but the coefficients are changed from low-pass to the high-pass coefficient given in Table 7-2. The digital cutoff frequency of the filter is the same as the lowpass filter. Therefore, this filter will eliminate the fundamental frequency of the square wave while passing the harmonic frequencies. The output of the FIR filter is given in Figure 7-10.

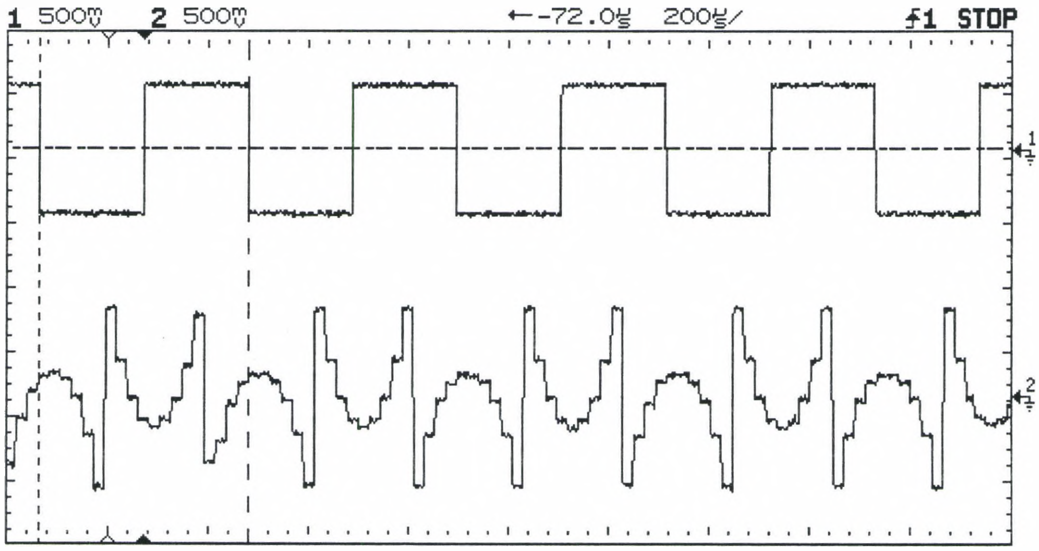


Figure 7-10 --- High-pass Filtering of a Square Wave by the FIR Filter

As seen in Figure 7-10, the output waveform is quite different than the output given in Figure 7-9 with the same input waveform. The output of the filter can be determined to be correct without much analysis. The output looks like a square wave with the fundamental sine wave "missing". The high-pass filter functions as specified.

From the performance analysis of the hardware FIR filter, the filter works as specified. From the simulation results it is verified that the hardware FIR filter performs exactly as specified by the input coefficients. Therefore, any filter designed can be loaded into the hardware through the serial port of the computer, and any filter designed can immediately be physically realized in real-time.

Frequency Response of an Audio Equalizer:

The FIR filter may be used as an audio enhancement device, such as an equalizer as given in Chapter 2. However, the hardware FIR filter has not been tested to verify that the equalizer coefficients developed in chapter 2 create the same results as the simulation.

The frequency response of the equalizer is given in Figure 2-13, and the hardware FIR filter should give the same frequency response with the same coefficient values. The coefficient values of the audio equalizer are given in Table 7-3.

Table 7-3 --- Audio Equalizer Coefficient Values

Coefficient Number	Coefficient Value
0	127
1	-2
2	1
3	4
4	6
5	7
6	6
7	4
8	2
9	1
10	0
11	0
12	0
13	0

The coefficient values of the equalizer are input into the Altera simulator, and a sinusoid of varying frequency called a sweep is input into the filter. The sweep signal input into the filter varies from 0 Hz. to 22500 Hz., thus giving an approximation of the frequency response. The simulated frequency response of the audio equalizer is given in Figure 7-11.

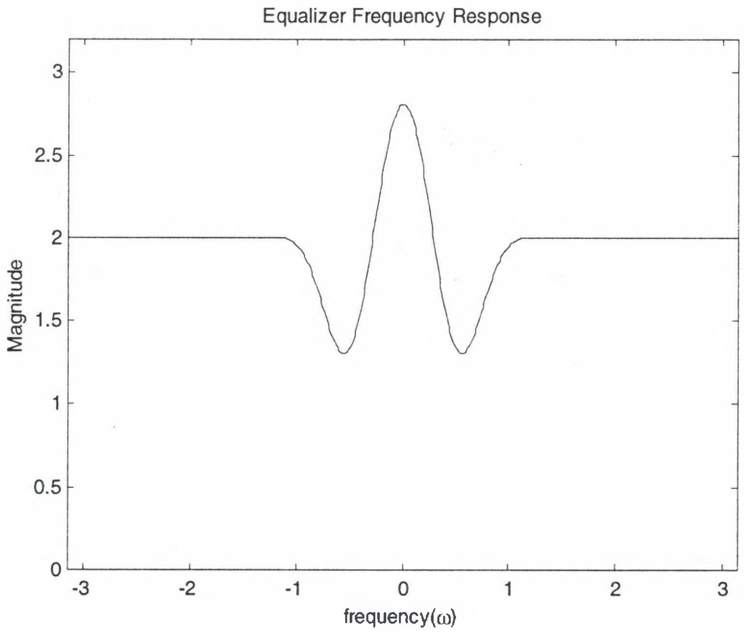


Figure 7-11 --- Equalizer Frequency Response

The sweep file is generated and run through the Altera simulator. The approximate frequency response of the hardware FIR filter is given in Figure 7-12.

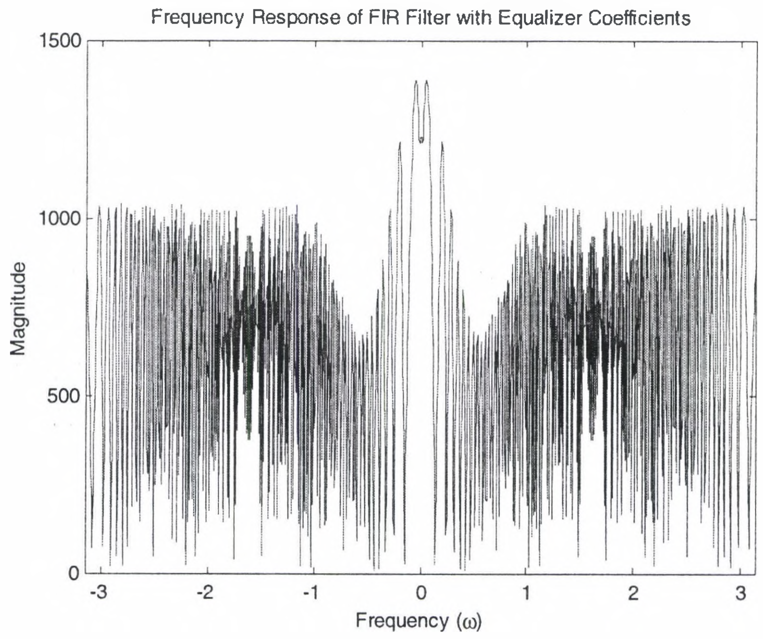


Figure 7-12 --- Altera Simulation of an FIR Filter Sweep Response With Equalizer Coefficients

Although the sweep response is not a true frequency response, Figure 7-12 does give the correct magnitude frequency response of the equalizer. The output given in Figure 7-12 is a sweeping sinusoid running through digital frequencies from $-\pi$ to π. Therefore, the resolution of Figure 7-12 is not as clear as the resolution of Figure 7-11.

The equalizer coefficients are input into the hardware filter for an actual frequency response. Sinusoids of varying frequencies are input into the filter, and the magnitude of the output sinusoids are recorded. In this way, the frequency response of the hardware FIR filter can be calculated. The frequency response of the hardware equalizer is given in Figure 7-13.

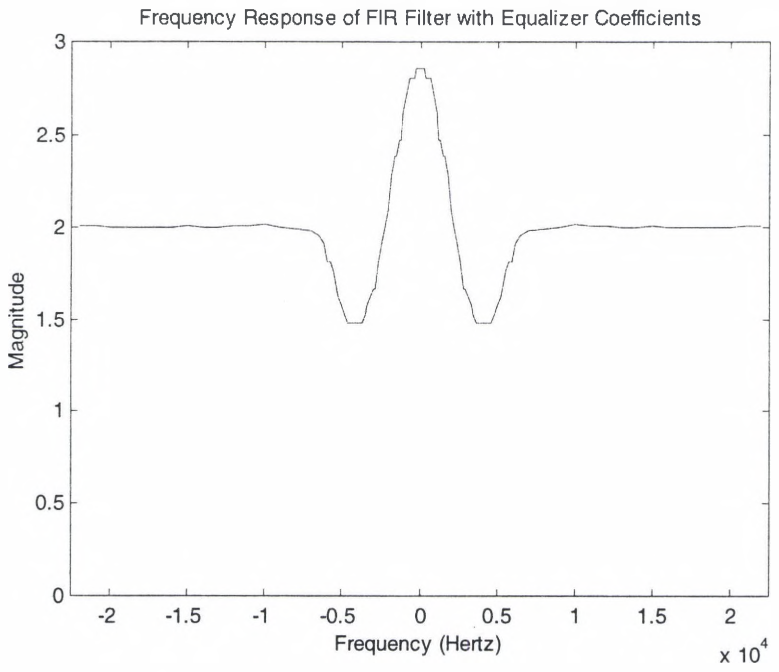


Figure 7-13 --- Hardware FIR Filter Equalizer Frequency Response

Figure 7-13 shows that the frequency response of the hardware FIR filter is the same as the frequency responses of the simulations. The correlation between the simulated responses and the real-time hardware responses further verifies that the hardware FIR filter functions properly.

CHAPTER 8

CONCLUSIONS AND FUTURE WORK

The variable coefficient FIR filter works as specified, and the enhanced audio is successful. Many equalizer designs have been developed, and all can be loaded into the filter for a frequency response desired by the music listener. The Matlab software developed makes it easy to obtain a desired frequency response with minimal effort, and as shown in Chapter 7, the frequency response calculated in the Matlab software becomes the frequency response characteristic of the filter when the coefficients are loaded.

Although the digital hardware equalizer performs well, there is much work that can be done to further enhance the quality of music, and for the design to become more appealing to industry.

First, the audio equalizer developed in this body of work is a monaural design. For a design that is appealing to the music industry, the design must be adapted to a stereo design. This is a fairly simple process, however. Two instances of the monaural design must work in parallel to independently filter the right and left audio components. The filter coefficients can be the same for both right and left audio filters; so some resource sharing is possible.

Second, other types of digital filters can be used for audio enhancement. An IIR (Infinite Impulse Response) filter not only uses previous input signal values, but also previous output signal values to calculate the next output value. The IIR filter can filter signals more efficiently than FIR filters. Also, IIR filters are able to produce echo and other types of audio processing that FIR filters cannot. However, the IIR filter is more difficult to design because it necessarily uses floating-point arithmetic. Therefore, floating-point adders and multipliers must be designed for this type of filter.

Lastly, for the filter design to become more appealing to industry, the dynamic loading of coefficients must also be accomplished in hardware. The software developed to load the filter coefficients into the filter is good for demonstrative purposes, but a stand-alone device is necessary for a commercially viable product. One way to accomplish this task is to have several filter design coefficient values “burned” into an EPROM (Electrically Programmable Read-Only Memory), and those coefficient values may then be loaded into the filter upon the user’s request. However, a more robust design may serve as a better solution.

Appendix

Appendix A: Derivation of Ideal Low-pass, High-pass, Band-pass, and Band-stop FIR Filter Coefficients:

Derivation of Ideal Low-pass Filter Tap Coefficients (Cutoff frequency of ω_c):

$$h(n) = \frac{1}{2\pi}\int_{-\pi}^{\pi} H(\omega)e^{j\omega n}d\omega \tag{A1}$$

$$h(n) = \frac{1}{2\pi}\int_{-\omega_c}^{\omega_c} e^{j\omega n}d\omega \tag{A2}$$

$$h(n) = \frac{1}{j2\pi n}[e^{j\omega_c n} - e^{-j\omega_c n}] \tag{A3}$$

$$h(n) = \frac{1}{\pi n}\sin(\omega_c n). \tag{A4}$$

Derivation of Ideal High-pass Filter Tap Coefficients (Cutoff frequency of ω_c):

$$h(n) = \int_{-\pi}^{\pi} H(\omega)e^{j\omega n}d\omega \tag{A5}$$

$$h(n) = \int_{-\pi}^{-\omega_c} e^{j\omega n}d\omega + \int_{\omega_c}^{\pi} e^{j\omega n}d\omega \tag{A6}$$

$$h(n) = \frac{1}{2\pi}[\frac{1}{jn}e^{j\omega n}]_{-\pi}^{-\omega_c} + \frac{1}{2\pi}[\frac{1}{jn}e^{j\omega n}]_{\omega_c}^{\pi} \tag{A7}$$

$$h(n) = \frac{1}{j2\pi n}(e^{j\pi n} - e^{-j\pi n}) - \frac{1}{j2\pi n}(e^{j\omega_c n} - e^{-j\omega_c n}) \tag{A8}$$

$$h(n) = \frac{1}{\pi n}\sin(\pi n) - \frac{1}{\pi n}\sin(\omega_c n) \tag{A9}$$

$$\lim_{n->0}\frac{1}{\pi n}\sin(\pi n) = 1 \tag{A10}$$

$$\sin(\pi n) = 0, n \neq 0 \tag{A11}$$

$$h(n) = \delta(n) - \frac{1}{\pi n}\sin(\omega_c n) \tag{A12}$$

Derivation of Ideal Band-pass Filter Tap Coefficients (Pass Frequencies ω_a to ω_b):

$$h(n) = \frac{1}{2\pi}\int_{-\pi}^{\pi} H(\omega)e^{j\omega n}d\omega \tag{A13}$$

$$h(n) = \frac{1}{2\pi}\int_{-\omega_b}^{-\omega_a} e^{j\omega n}d\omega + \frac{1}{2\pi}\int_{\omega_a}^{\omega_b} e^{j\omega n}d\omega \tag{A14}$$

$$h(n) = \frac{1}{2\pi}[\frac{1}{jn}e^{j\omega n}]_{-\omega_b}^{-\omega_s} z + \frac{1}{2\pi}[\frac{1}{2\pi}e^{j\omega n}]_{\omega_a}^{\omega_b} \tag{A15}$$

$$h(n) = \frac{1}{2j\pi n}[e^{j\omega_b n} - e^{-j\omega_b n}] - \frac{1}{2jn\pi}[e^{j\omega_a n} - e^{-j\omega_a n}] \tag{A16}$$

$$h(n) = \frac{1}{\pi n}[\sin(\omega_b n) - \sin(\omega_a n)] \tag{A17}$$

Derivation of Ideal Band-stop Filter Tap Coefficients (Attenuate Frequencies ω_a to ω_b):

$$h(n) = \int_{-\pi}^{\pi} H(\omega)e^{j\omega n}d\omega \tag{A18}$$

$$h(n) = \frac{1}{2\pi}\int_{-\pi}^{-\omega_b} e^{j\omega n}d\omega + \frac{1}{2\pi}\int_{-\omega_a}^{\omega_b} e^{j\omega n}d\omega + \frac{1}{2\pi}\int_{\omega_b}^{\pi} e^{j\omega n}d\omega \tag{A19}$$

$$h(n) = \frac{1}{2\pi}[\frac{1}{jn}e^{j\omega n}]_{-\pi}^{-\omega_b} + \frac{1}{2\pi}[\frac{1}{jn}e^{j\omega n}]_{-\omega_a}^{\omega_b} + \frac{1}{2\pi}[\frac{1}{jn}e^{j\omega n}]_{\omega_b}^{\pi} \tag{A20}$$

$$h(n) = \frac{1}{2j\pi n}[e^{j\pi n} - e^{-j\pi n}] - \frac{1}{2j\pi n}[e^{j\omega_b n} - e^{-j\omega_b n}] + \frac{1}{2j\pi n}[e^{j\omega_a n} - e^{-j\omega_a n}] \tag{A21}$$

$$h(n) = \frac{1}{\pi n}\sin(\pi n) + \frac{1}{\pi n}\sin(\omega_a n) - \frac{1}{\pi n}\sin(\omega_b n) \tag{A22}$$

$$h(n) = \delta(n) + \frac{1}{\pi n}(\sin(\omega_a n) - \sin(\omega_b n)) \qquad \text{(A23)}$$

Appendix B: Matlab Software to calculate FIR filter Coefficients:

```
% FIR .m
% creates FIR window filter coefficients for
% lowpass, highpass, bandpass, and bandstop filters

%clear all;
close all; clc;
d = 0;
h = 0;
k = 0;
N = 0;
M = 0;

disp('Type the number of Taps in the filter');
disp('Number of Taps must be odd');

% user inputs number of filter taps
N = input('Number of Taps>');

% user inputs type of filter
disp('Type the number coresponding to the type');
disp('of filter you wand to create');
disp('1) Lowpass');
disp('2) Highpass');
disp('3) Bandpass');
disp('4) Bandstop');

num = input('Type of Filter>');

% Calculation of half the filter length
M = (N-1) / 2;

if num == 1
   disp('Enter Digital Cutoff Frequency, w (between 0 and pi)');
   w = input ('Radian Cutoff Frequency>');
   % calculate lowpass filter impulse response
   d(1) = w/pi;
   for i=1:M,
      d(i+1) = (sin(w*i)/(pi*i));
   end

elseif num == 2
   disp('Enter Digital Cutoff Frequency, w (between 0 and pi)');
   w = input ('Radian Pass Frequency>');
   % calculate highpass filter impulse response
   d(1) = 1 - w/pi;
   for i=1:M,
      d(i+1) = -(sin(w*i)/(pi*i));
   end

elseif num == 3
   disp('Enter Low Cutoff frequency wa (between 0 and pi)');
   wa = input ('Radian Low Pass Frequency>');
   disp('Enter High Cutoff frequency wb (between 0 and pi)');
   wb = input ('Radian High Pass Frequency>');
```

```
    % calculate bandpass filter impulse response
    d(1) = (wb-wa)/pi;
    for i=1:M,
       d(i+1) = (sin(wb*i)-sin(wa*i))/(pi*i);
    end

 elseif num == 4
    disp('Enter Low Cutoff frequency wa (between 0 and pi)');
    wa = input ('Radian Low Pass Frequency>');
    disp('Enter High Cutoff frequency wb (between 0 and pi)');
    wb = input ('Radian High Pass Frequency>');
    % calculate bandstop filter impulse response
    d(1) = 1 - (wb-wa)/pi;
    for i=1:M,
       d(i+1) = -(sin(wb*i)-sin(wa*i))/(pi*i);
    end

 else
    disp('Illegal Value, Program Terminated');
    i = 1;
 end

 if i ~= 1
    disp('Enter Windowing type:');
    disp('1) Rectangular Window');
    disp('2) Hamming Window');
    disp('3) Kaiser Window');
    % user input which filter window to use
    win = input ('Window Type>');


    % clear window
    clc;
    disp('Coefficients are folded, therfore a filter with 21 taps');
    disp('Will only have 11 coefficients');
    disp('The first coefficient displayed is the coefficiens h(0)');
    disp('The next coefficient displayed is h(1) and h(-1), etc.');

    w = linspace(-pi,pi,512);

    if win == 1
       disp('Rectangular window coefficients:');
       % extend filter coefficients to entire impulse response
       for i=1:M,
          dplot(M+i) = d(i);
          dplot(M-i+1) = d(i+1);
       end
       dplot(N+1) = d(M+1);

       hw = fft(dplot,512);
       % plot frequency response of filter
       figure(1);plot(w,abs(fftshift(hw)));
       xlabel('frequency(\omega)');ylabel('Magnitude');
       title('Rectangular Window Frequency Response');
       d

    elseif win == 2
```

```
      disp('Hamming window coefficients:');
      h = hwind(d);
      % extend filter coefficients to entire impulse response
      for i=1:M,
         dplot(M+i) = h(i);
         dplot(M-i+1) = h(i+1);
      end
      dplot(N+1) = k(M+1);
      hw = fft(dplot,512);
      % plot frequency response of filter
      figure(1);plot(w,abs(fftshift(hw)));
      xlabel('frequency(\omega)');ylabel('Magnitude');
      title('Hamming Window Frequency Response');
      h

   elseif win == 3
      disp('Kaiser window coefficients:');
      k = kwind(d);
      % extend filter coefficients to entire impulse response
      for i=1:M,
         dplot(M+i) = k(i);
         dplot(M-i+1) = k(i+1);
      end
      dplot(N+1) = k(M+1);
      hw = fft(dplot,512);
      % plot frequency response of filter
      figure(1);plot(w,abs(fftshift(hw)));
      xlabel('frequency(\omega)');ylabel('Magnitude');
      title('Kaiser Window Frequency Response');
      k

   else
      disp('Illegal Value, Program Terminated');
   end
end




% hwind.m - Hamming window
%
% w(n) = 0.54 - 0.45 * cos(2*pi*M)/(N-1)
%
% 0.2 % overshoot

function w = hwind(d)

M = length(d);
N = 2*M-1;

for n = 0:M-1,
   w(n+1) = d(n+1).*(0.54-0.46.*cos((2*pi.*(n+M))/(N-1)));
end

% kwind.m - Kaiser window.
%
% w   = kwind(alpha, N) = row vector
```

```
%
% alpha = Kaiser window shape parameter
% N  = 2M+1 = window length (must be odd)

function w = kwind(d)

M = length(d)+1;
N = 2*M - 1;

alpha = 7;

den = I0(alpha);

for n = 0:M-2,
   num = I0(alpha * sqrt(1 - n^2/(M^2)));
   w(n+1) = d(n+1) .* (num / den);
end

% I0.m - modified Bessel function of 1st kind and 0th order.
%
% S = I0(x)
%
% defined only for scalar x >= 0
% based on I0.c

function S = I0(x)

eps = 10^(-9);
n = 1; S = 1; D = 1;

while D > (eps * S),
        T = x / (2*n);
        n = n+1;
        D = D * T^2;
        S = S + D;
end
```

Appendix C: C++ Software for Windows 95, 98, and NT for Writing Coefficient Value to FIR Filter

```
/* A Program to communicate with the serial port */
/* To be used with Variable Coefficient FIR Filter Hardware */
/* 23 Taps, (12 input Coefficients) */
/* Serial data rate is 28800 bps */


# include <stdio.h>
# include <windows.h>
# include <iostream.h>

void main()
{
        HANDLE  comHandle; /* handle to open serial port file */
        BOOL    success; /* when open or write is accomplished successfully */
        DCB             dcb; /* Data Control Block */
        char str[100];
        char writechar[12];
        DWORD numWrite;
        int coeff;
        int i;

        /* Open the comm port. Com1 */
        comHandle = CreateFile("COM1",
                                                GENERIC_READ|GENERIC_WRITE,
                                                0,     0, OPEN_EXISTING,
                                                FILE_ATTRIBUTE_NORMAL, 0);
        if (comHandle == INVALID_HANDLE_VALUE)
                MessageBox(NULL,"Comm Port Not Opened", "Error",MB_OK);

        /* Get the current settings of the COMM port */
        success = GetCommState(comHandle, &dcb);
        if (!success)
                MessageBox(NULL,"Cannot Get Com1 State", "Error",MB_OK);

        /* Modify the daud rate, etc. */
        dcb.BaudRate = 28800;                                   /* Transmit at 28.8 kbit/sec. */
        dcb.ByteSize = 8;                                       /* Eight Bit numbers */
        dcb.Parity = NOPARITY;                                  /* No Parity error detection */
        dcb.StopBits = ONESTOPBIT;                              /* use only one stop bit */
        dcb.fBinary = TRUE;                                             /* use binary mode */
        dcb.fParity = FALSE;                                    // do not report parity errors
        dcb.fOutxCtsFlow = FALSE;                       // do not wait for CTS to transmit
        dcb.fOutxDsrFlow = FALSE;                       // do not wait for DSR to transmit
        dcb.fDtrControl = DTR_CONTROL_DISABLE;     //Lowers the DTR line when the device is opened.
                                        //The application can adjust the state of the
                                        //line with EscapeCommFunction
        dcb.fDsrSensitivity = FALSE;                    //ignore bytes received

        dcb.fTXContinueOnXoff = TRUE;                   /* no flow control */
        dcb.fOutX = FALSE;
        dcb.fInX = FALSE;
        dcb.ErrorChar = (char)NULL;

        dcb.fNull = FALSE;                         //ignore null bytes.
        dcb.fRtsControl = RTS_CONTROL_DISABLE;     //can modify with EscapeCommFunction
        dcb.fAbortOnError = FALSE;
        dcb.wReserved = 0;                         // not used, must be set to zero

        /* Apply the new comm port settings */
        success = SetCommState(comHandle, &dcb);
        if (!success)
                MessageBox(NULL,"Cannot Apply Com1 Settings", "Error",MB_OK);

        /* Set the Data Terminal Ready line */
        EscapeCommFunction(comHandle, SETDTR);

        cout << "This Program will send numerical eight bit 2's complement data through" << endl;
        cout << "the serial port (COM1).  Because the data length is only eight bits wide" << endl;
        cout << "only values of -127 to 127 are valid.  Any data entered that is not" << endl;
        cout << "valid will result in an error and a value of zero will be sent to the serial" << endl;
        cout << "port." << endl;
        printf("\n\n");

        for(;;)
        {
                cout << "What type of Filter do you want to implement?" << endl;
                cout << "Type 1 for Lowpass (Cutoff at 2800 Hz)" << endl;
                cout << "Type 2 for Highpass (Cutoff at 2800 Hz)" << endl;
                cout << "Type 3 for Bandpass (Pass 2800 Hz. through 5500 Hz.)" << endl;
                cout << "Type 4 for Bandstop (Cut frequencies 2800 Hz through 5500 Hz.)" << endl;
                cout << "Type 0 to enter custom filter coefficients" << endl;

                gets(str);
```

```
		coeff = atoi(str);

		if(coeff == 1)
		{
			cout << "Lowpass Coefficients input into filter" << endl;
			writechar[1] = 127;
			writechar[2] = 122;
			writechar[3] = 108;
			writechar[4] = 87;
			writechar[5] = 64;
			writechar[6] = 41;
			writechar[7] = 22;
			writechar[8] = 8;
			writechar[9] = 0;
			writechar[10] = -3;
			writechar[11] = -4;
			writechar[12] = -4;
			writechar[13] = -2;
			writechar[14] = -1;
		}
		if(coeff == 2)
		{
			cout << "Highpass Coefficients input into filter" << endl;
			writechar[1] = 127;
			writechar[2] = -17;
			writechar[3] = -15;
			writechar[4] = -12;
			writechar[5] = -9;
			writechar[6] = -6;
			writechar[7] = -3;
			writechar[8] = -1;
			writechar[9] = 0;
			writechar[10] = 1;
			writechar[11] = 1;
			writechar[12] = 1;
			writechar[13] = 0;
			writechar[14] = 0;
		}
		if(coeff == 3)
		{
			cout << "Bandpass Coefficients input into filter" << endl;
			writechar[1] = 127;
			writechar[2] = 69;
			writechar[3] = 30;
			writechar[4] = -14;
			writechar[5] = -43;
			writechar[6] = -49;
			writechar[7] = -36;
			writechar[8] = -16;
			writechar[9] = 0;
			writechar[10] = 7;
			writechar[11] = 7;
			writechar[12] = 4;
			writechar[13] = 1;
			writechar[14] = 0;
		}
		if(coeff == 4)
		{
			cout << "Bandstop Coefficients input into filter" << endl;
			writechar[1] = 127;
			writechar[2] = -16;
			writechar[3] = -7;
			writechar[4] = 3;
			writechar[5] = 10;
			writechar[6] = 11;
			writechar[7] = 8;
			writechar[8] = 4;
			writechar[9] = 0;
			writechar[10] = -2;
			writechar[11] = -2;
			writechar[12] = -1;
			writechar[13] = 0;
			writechar[14] = 0;
		}
		if (coeff == 0)
		{
			cout << "Fourteen tap coefficients will need to be entered into the filter
starting with" << endl;
			cout << "d(0)" << endl;

			/* get coefficient values from user */

			for(i=0;i<14;i++)
			{
				if(i==0)
				{
					/* get d(0) */
					printf("type first coefficient to be entered into filter
(i.e. d[%2d])\n",i);

					gets(str);
					if (!(coeff = atoi(str)) || (coeff > 127) || (coeff < -127))
						printf("Zero Value Entered as Coefficient\n\n");
```

```
                                        writechar[1] = (char) coeff;
                                }
                                else
                                {
                                        /* get other coefficient values */
                                        printf("type next coefficient to be entered into filter
(i.e. d[%2d])\n",i);
                                        gets(str);
                                        if (!(coeff = atoi(str)) || (coeff > 127) || (coeff < -127))
                                                printf("Zero Value Entered as Coefficient\n\n");

                                        writechar[i+1] = (char) coeff;
                                }
                        }
                }
                        printf("Writing Coefficient values into filter\n\n");

                        for(i=0;i<14;i++)
                        {
                                success = WriteFile(comHandle, &writechar[i+1], 1,
                                        &numWrite, 0);
                                if (!success)
                                        MessageBox(NULL,"Cannot Write to Com1", "Error",MB_OK);
                        }

                        printf("Writing Complete\n\n");


                        gets(str);

                        for(i=0;i<50;i++)
                                printf("\n");

        }
                        /* Close the file */
                        CloseHandle(comHandle);


}
```

Appendix D: AHDL Code for Variable Coefficient FIR Filter

```
% 27-Tap Variable Coefficient Fir Filter %
% 8-bit input wide bus %
% Clock Frequency is 13.89 MHz %
% Clock Period is 72.00 ns, Grid size is 36.00 ns %
% Sampling occurs every 315 clock cycles %
% Sampling frequency is 44.1 kHz %
% Serial Multiplication takes 21 clock cycles %
% times 14 tap multiplications %
% plus one addition wait time %

include "DSPadd8.inc";
include "DSPadd9.inc";
include "DSPadd12.inc";
include "DSPadd13.inc";
include "DSPadd14.inc";
include "DSPadd15.inc";

Subdesign 'firfilt8AD'
(
clock                :input; % all registers and flip-flops are clocked by clock %
Datain[7..0]  :input; % input data is sampled at 44.1 kHz %
shiftcoef            :input; % input signal to shift in FIR coefficients %
Coeffin[7..0] :input; % FIR coefficients to be shifted in %
outdata[15..0]       :output; % Filtered output (16-bits wide) %

% A/D Converter Control I/O %
res                         :input;  % Reset for A/D Converter %
ChipSelect           :output; % Select output for A/D Converter %
Read                 :output; % Read From A/D Converter %
Write                :output; % Write To A/D Converter %
CK                          :output; % A/D Converter clock %
)

Variable

% A/D Controller Variables %
CK                                      :dff; % 2 MHz clock for A/D Converter%
adcount[5..0]               :dff; % counter for A/D Converter%
adreset                                 :dff; % counter reset %
ChipSelect                        :dff; % Select output %
Read                              :dff; % Read From A/D %
Write                             :dff; % Write To A/D %

% Filter Variables %
multcnt[3..0]               :dff; % Counts number of Multiplications that take place %
multcntreset                :dff; % Signal to reset multcnt[3..0] %
coeffdelay[1..0]            :dff; % Two-bit register used to create edge detection %
shiftcoefedge               :node; % Edge detected signal %
Zero, One, Two                    :node; % signal flags to indicate how many %
Three, Four, Five           :node; % multiplications have taken place %
Six, Seven, Eight           :node;
Nine, Ten, Eleven           :node;
Twelve, Thirteen            :node;
Tapdat[26..0][7..0]         :dff; % data input registers %
Tapcoef[13..0][7..0]:dff; % Coefficient Registers %
addin[13..0][11..0]         :dff;
Add1[12..0]                       :DSPadd8;  % Adders used to fold filter %
Add2[6..0]                        :DSPadd12; % Adders used after all multiplications have %
Add3[2..0]                        :DSPadd13; % taken place %
Add4[1..0]                        :DSPadd14; % Adder architecture is given in include files %
Add5                              :DSPadd15;
Outdata[15..0]                    :dff; % output Register %

% 16 bit multiplier Variables %
% ----------------------------- %
Mplierbus[8..0]                         :dff; % Register to hold the Multiplier %
Resbus[17..0]                     :dff; % Register to hold the Result %
count[4..0]                             :dff; % counter to count number of operations %
```

```
multDone                                        :dff; % Multiplication done flag %
multstart                                       :dff; % Multiplication start flag %
Np, Nc                                          :dff; % Negative Multiplier and Multiplicand flags %
Pos, Neg                                        :node; % Positive and Negative Product Flags %
Load, shift                         :node; % Multiplier Control Flags %
add                                                          :node;
Mplier[8..0], Mcand[8..0]  :node; % input multiplier and multiplicand nodes %
addMult                                                      :DSPadd9; % adder used in Multiplication
architecture %
% ----------------------------- %

begin

% Creation of the A/D Controller %

% Controller counter is clocked by clock %
adcount[5..0].clk = clock;

% all control outputs are clocked by clock %
adreset.clk = clock;
CK.clk = clock;
ChipSelect.clk = clock;
Read.clk = clock;
Write.clk = clock;

% A/D counter counts every seven clock cycles %
% giving the A/D converter controller a "clock" frequency of 2 MHz %
% 2Mhz is a design specification of the A/D converter %
adcount[5..0].d = (adcount[5..0] + 1 and CK and !adreset and res) or
                             (adcount[5..0] and !CK and res);

% CK becomes valid during the 0, 7, and 14 counts of the multiplication counter, count %
CK.d = ((!count[4] and !count[3] and count[2] and count[1] and count[0]) or (!count[4] and
                           count[3] and count[2] and count[1] and !count[0]) or (!count[4]
and !count[3] and
                           !count[2] and !count[1] and !count[0])) and res;

% adcount counts from 0 to 44, (clock freq/sampling freq/seven clock cycles = 45) %
adreset.d = adcount[5] and !adcount[4] and adcount[3] and adcount[2] and !adcount[1] and
!adcount[0];

if (res == gnd) then
      ChipSelect.d = Vcc;
      Read.d = Vcc;
      Write.d = Vcc;
end if;

if (adcount[5..0] == 0) then
      ChipSelect.d = gnd;
      Write.d = Vcc;
      Read.d = Vcc;
elsif (adcount[5..0] == 1) then
      ChipSelect.d = gnd;
      Write.d = gnd;
      Read.d = Vcc;
elsif (adcount[5..0] < 37) then
      ChipSelect.d = Vcc;
      Write.d = Vcc;
      Read.d = Vcc;
elsif (adcount[5..0] == 37) then
      ChipSelect.d = gnd;
      Write.d = Vcc;
      Read.d = Vcc;
else
      ChipSelect.d = gnd;
      Write.d = Vcc;
      Read.d = gnd;
end if;

% Creation of a serial Multiplier %
%----------------------------------------------------------------------------------------%
```

```
% Create timing generator for multiplication %
count[4..0].clk = clock;
multDone.clk = clock;
multstart.clk = clock;

Count[4..0].d = (Count[4..0].q + 1) and !multdone and res; % Count up until multdone flag %

% Start multiplication at count zero %
multstart.d = multDone.q;
% Multiplication is complete at count 17 %
multDone.d = count[4] and !count[3] and !count[2] and count[1] and count[0];

%Create two one bit registers to keep track of negative numbers for 2's comp. multiplication%
Np.clk = clock; % Multiplier register %
Nc.clk = clock; % Multiiplicand register %
Np.d = (Mplier[8] and multstart) or (Np.q and !multstart); % Load Multiplication flags %
Nc.d = (Mcand[8] and multstart) or (Nc.q and !multstart);  % at startup %

Neg = Np.q XOR Nc.q; % Negative product flag %
Pos = !Neg; % Positive product flag %

% Create input Register to hold Multiplier %
Mplierbus[8..0].clk = clock;
Mplierbus[8..0].d = (Mplier[8..0] and Load and !Np) or
                                    % Change Negative number to positive %
                                    ((!Mplier[8..0] + 1) and load and Np) or
                                    (Mplierbus[8..0] and !Load);

% Create Output Register to handle addend and shifting %
Resbus[17..0].clk = clock;

% Load, add, Shift and hold architecture for resultant bus %
Resbus[16..9].d = (AddMult.result[7..0] and Add) or (Resbus[17..10] and shift)
                                    or (gnd and Load) or (Resbus[16..9] and !add and !shift and
!load);
Resbus[17].d = (addMult.result[8] and Add) or (gnd and (shift or load))
                                    or (Resbus[17] and !add and !shift and !load);
% 9 least significant bits hold Multiplicand %
Resbus[8..0].d = (Resbus[8..0] and Add) or (Resbus[9..1] and shift)
                                   or (Mcand[8..0] and load and !Nc) or
                                   % Change Negative number to positive %
                                   ((!Mcand[8..0] + 1) and load and Nc) or
                                   (Resbus[8..0] and !add and !shift and !load);

% Connect Multiplier adder inputs %
addMult.dataa[8..0] = Mplierbus[8..0];
addMult.datab[8..0] = Resbus[17..9];

% Create Multiplier Controller %
% Load data when counter is equal to zero %
% Shift on even counts and add on odd counts when Resbus[0] is one %
if (Count[4..0] == H"01") then
        Load = Vcc;
        Shift = gnd;
        Add = Gnd;
elsif ((Count[0] == gnd) and (Resbus[0] == Gnd)) then
        Load = gnd;
        Shift = gnd;
        Add = Gnd;
elsif ((Count[0] == gnd) and (Resbus[0] == Vcc)) then
        Load = gnd;
        Shift = gnd;
        Add = Vcc;
Else
        Load = Gnd;
        Shift = Vcc;
        Add = gnd;
end if;
%-----------------------------------------------------------------------------------------------%

% Create counter to count multiplications ( Multiplies Sixteen sets of numbers ) %
Multcnt[3..0].clk = clock;
```

```
Multcntreset.clk = clock;
Multcnt[3..0].d = (((!Multcntreset) & (Multcnt[3..0]+1) & multdone) or
                              (Multcnt[3..0].q and !multdone)) and res;

Multcntreset.d = Multcnt[3] and Multcnt[2] and Multcnt[1] and !Multcnt[0];

% Create Timing generator to keep track of multiplications %

Zero = !Multcnt[3] and !Multcnt[2] and !Multcnt[1] and !Multcnt[0];
One = !Multcnt[3] and !Multcnt[2] and !Multcnt[1] and Multcnt[0];
Two = !Multcnt[3] and !Multcnt[2] and Multcnt[1] and !Multcnt[0];
Three = !Multcnt[3] and !Multcnt[2] and Multcnt[1] and Multcnt[0];
Four = !Multcnt[3] and Multcnt[2] and !Multcnt[1] and !Multcnt[0];
Five = !Multcnt[3] and Multcnt[2] and !Multcnt[1] and Multcnt[0];
Six = !Multcnt[3] and Multcnt[2] and Multcnt[1] and !Multcnt[0];
Seven = !Multcnt[3] and Multcnt[2] and Multcnt[1] and Multcnt[0];
Eight = Multcnt[3] and !Multcnt[2] and !Multcnt[1] and !Multcnt[0];
Nine = Multcnt[3] and !Multcnt[2] and !Multcnt[1] and Multcnt[0];
Ten = Multcnt[3] and !Multcnt[2] and Multcnt[1] and !Multcnt[0];
Eleven = Multcnt[3] and !Multcnt[2] and Multcnt[1] and Multcnt[0];
Twelve = Multcnt[3] and Multcnt[2] and !Multcnt[1] and !Multcnt[0];
Thirteen = Multcnt[3] and Multcnt[2] and !Multcnt[1] and Multcnt[0];


% Assign clock and nodes to Taps for data shift %
Tapdat[26..0][7..0].clk = clock;
% Shift data when all processes are finished and ready to %
% Start new processes %
Tapdat[26..1][7..0].d = (((Tapdat[25..0][7..0].q) and Thirteen and multdone) or
                                          ((Tapdat[26..1][7..0].q) and (!Thirteen or
!multdone)));
% Data input shifts into Tapdat[0] %
Tapdat[0][7..0].d = ((Datain[7..0] and Thirteen and multdone) or
                                    (Tapdat[0][7..0] and (!Thirteen or !multdone)));

% Create Receiver/Registers to shift in Coefficients %
% create edge detection for coefficient shift %
coeffdelay[1..0].clk = clock;
coeffdelay[0].d = shiftcoef;
coeffdelay[1].d = !coeffdelay[0].q;
shiftcoefedge = coeffdelay[0].q and coeffdelay[1].q;

% Tap coefficients are clocked by clock %
Tapcoef[13..0][7..0].clk = clock;
% Shift coefficients when input Shiftcoefedge signal is high %
Tapcoef[13..1][7..0].d = ((Tapcoef[12..0][7..0].q) and shiftcoefedge) or
                                           ((Tapcoef[13..1][7..0].q) and !shiftcoefedge);
Tapcoef[0][7..0].d = (coeffin[7..0] and shiftcoefedge) or
                                    (Tapcoef[0][7..0] and !shiftcoefedge);

% Fold filter to reduce number of multiplications %
% Semetric Coefficient filtering %
Add1[0].dataa[7..0] = Tapdat[26][7..0].q;
Add1[0].datab[7..0] = Tapdat[0][7..0].q;
Add1[1].dataa[7..0] = Tapdat[25][7..0].q;
Add1[1].datab[7..0] = Tapdat[1][7..0].q;
Add1[2].dataa[7..0] = Tapdat[24][7..0].q;
Add1[2].datab[7..0] = Tapdat[2][7..0].q;
Add1[3].dataa[7..0] = Tapdat[23][7..0].q;
Add1[3].datab[7..0] = Tapdat[3][7..0].q;
Add1[4].dataa[7..0] = Tapdat[22][7..0].q;
Add1[4].datab[7..0] = Tapdat[4][7..0].q;
Add1[5].dataa[7..0] = Tapdat[21][7..0].q;
Add1[5].datab[7..0] = Tapdat[5][7..0].q;
Add1[6].dataa[7..0] = Tapdat[20][7..0].q;
Add1[6].datab[7..0] = Tapdat[6][7..0].q;
Add1[7].dataa[7..0] = Tapdat[19][7..0].q;
Add1[7].datab[7..0] = Tapdat[7][7..0].q;
Add1[8].dataa[7..0] = Tapdat[18][7..0].q;
Add1[8].datab[7..0] = Tapdat[8][7..0].q;
Add1[9].dataa[7..0] = Tapdat[17][7..0].q;
Add1[9].datab[7..0] = Tapdat[9][7..0].q;
```

```
Add1[10].dataa[7..0] = Tapdat[16][7..0].q;
Add1[10].datab[7..0] = Tapdat[10][7..0].q;
Add1[11].dataa[7..0] = Tapdat[15][7..0].q;
Add1[11].datab[7..0] = Tapdat[11][7..0].q;
Add1[12].dataa[7..0] = Tapdat[14][7..0].q;
Add1[12].datab[7..0] = Tapdat[12][7..0].q;


% Create inputs into Serial Multiplier (Multiplicand and Multiplier) %
% Add1[0] * Tapcoef[0] results multiplied at time 0 to time 18%
% Add1[1] * Tapcoef[1] result multiplied at time 19 to time 37 %
% Add1[2] * Tapcoef[2] result multiplied at time 38 to time 56 %
% Add1[3] * Tapcoef[3] result multiplied at time 57 to time 75 %
% Add1[4] * Tapcoef[4] result multiplied at time 76 to time 94%
% Add1[5] * Tapcoef[5] result multiplied at time 95 to time 113%
% Add1[6] * Tapcoef[6] result multiplied at time 114 to time 132%
% Add1[7] * Tapcoef[7] result multiplied at time 133 to time 151%
% Add1[8] * Tapcoef[8] result multiplied at time 152 to time 170%
% Add1[9] * Tapcoef[9] result multiplied at time 171 to time 189%
% Add1[10] * Tapcoef[10] result multiplied at time 190 to time 208%
% Tapdat[11] * Tapcoef[11] result multiplied at time 209 to time 227%

% Multiplier inputs %
Mplier[7..0] =  (Add1[0].result[7..0] and Zero) or (Add1[1].result[7..0] and One) or
                           (Add1[2].result[7..0] and Two) or (Add1[3].result[7..0] and Three)
or
                           (Add1[4].result[7..0] and Four) or (Add1[5].result[7..0] and Five)
or
                           (Add1[6].result[7..0] and Six) or (Add1[7].result[7..0] and Seven)
or
                           (Add1[8].result[7..0] and Eight) or (Add1[9].result[7..0] and
Nine) or
                           (Add1[10].result[7..0] and Ten) or (Add1[11].result[7..0] and
Eleven) or
                           (Add1[12].result[7..0] and Twelve) or (Tapdat[13][7..0] and
Thirteen);

% Tapdat[11][7..0] needs to be lengthened one bit because %
% it was not added to another number in the folding process %
Mplier[8] =   (Add1[0].result[8] and Zero) or (Add1[1].result[8] and One) or
                           (Add1[2].result[8] and Two) or (Add1[3].result[8] and Three) or
                           (Add1[4].result[8] and Four) or (Add1[5].result[8] and Five) or
                           (Add1[6].result[8] and Six) or (Add1[7].result[8] and Seven) or
                           (Add1[8].result[8] and Eight) or (Add1[9].result[8] and Nine) or
                           (Add1[10].result[8] and Ten) or (Add1[11].result[8] and Eleven) or
                           (Add1[12].result[8] and Twelve) or (Tapdat[13][7] and Thirteen);

% Multiplicand inputs %
Mcand[7..0] =   (Tapcoef[0][7..0] and Zero) or (Tapcoef[1][7..0] and One) or
                           (Tapcoef[2][7..0] and Two) or (Tapcoef[3][7..0] and Three) or
                           (Tapcoef[4][7..0] and Four) or (Tapcoef[5][7..0] and Five) or
                           (Tapcoef[6][7..0] and Six) or (Tapcoef[7][7..0] and Seven) or
                           (Tapcoef[8][7..0] and Eight) or (Tapcoef[9][7..0] and Nine) or
                           (Tapcoef[10][7..0] and Ten) or (Tapcoef[11][7..0] and Eleven) or
                           (Tapcoef[12][7..0] and Twelve) or (Tapcoef[13][7..0] and
Thirteen);

Mcand[8] =          (Tapcoef[0][7] and Zero) or (Tapcoef[1][7] and One) or
                           (Tapcoef[2][7] and Two) or (Tapcoef[3][7] and Three) or
                           (Tapcoef[4][7] and Four) or (Tapcoef[5][7] and Five) or
                           (Tapcoef[6][7] and Six) or (Tapcoef[7][7] and Seven) or
                           (Tapcoef[8][7] and Eight) or (Tapcoef[9][7] and Nine) or
                           (Tapcoef[10][7] and Ten) or (Tapcoef[11][7] and Eleven) or
                           (Tapcoef[12][7] and Twelve) or (Tapcoef[13][7] and Thirteen);


% Create Resgisters to hold different products for addition %
% All addin registers are clocked by clock %
% Negative multiplication is preserved by using Np and Nc values %
% All addin registers are cleared by res %

addin[13..0][11..0].clk = clock;
```

```
addin[0][11..0].d = (Resbus[15..4] and multDone and Pos and Zero) or
                        ((!Resbus[15..4] + 1) and multDone and Neg and Zero) or
                        (addin[0][11..0] and (!multDone or !Zero));
addin[1][11..0].d = (Resbus[15..4] and multDone and Pos and One) or
                        ((!Resbus[15..4] + 1) and multDone and Neg and One) or
                        (addin[1][11..0] and (!multDone or !One));
addin[2][11..0].d = (Resbus[15..4] and multDone and Pos and Two) or
                        ((!Resbus[15..4] + 1) and multDone and Neg and Two) or
                        (addin[2][11..0] and (!multDone or !Two));
addin[3][11..0].d = (Resbus[15..4] and multDone and Pos and Three) or
                        ((!Resbus[15..4] + 1) and multDone and Neg and Three) or
                        (addin[3][11..0] and (!multDone or !Three));
addin[4][11..0].d = (Resbus[15..4] and multDone and Pos and Four) or
                        ((!Resbus[15..4] + 1) and multDone and Neg and Four) or
                        (addin[4][11..0] and (!multDone or !Four));
addin[5][11..0].d = (Resbus[15..4] and multDone and Pos and Five) or
                        ((!Resbus[15..4] + 1) and multDone and Neg and Five) or
                        (addin[5][11..0] and (!multDone or !Five));
addin[6][11..0].d = (Resbus[15..4] and multDone and Pos and Six) or
                        ((!Resbus[15..4] + 1) and multDone and Neg and Six) or
                        (addin[6][11..0] and (!multDone or !Six));
addin[7][11..0].d = (Resbus[15..4] and multDone and Pos and Seven) or
                        ((!Resbus[15..4] + 1) and multDone and Neg and Seven) or
                        (addin[7][11..0] and (!multDone or !Seven));
addin[8][11..0].d = (Resbus[15..4] and multDone and Pos and Eight) or
                        ((!Resbus[15..4] + 1) and multDone and Neg and Eight) or
                        (addin[8][11..0] and (!multDone or !Eight));
addin[9][11..0].d = (Resbus[15..4] and multDone and Pos and Nine) or
                        ((!Resbus[15..4] + 1) and multDone and Neg and Nine) or
                        (addin[9][11..0] and (!multDone or !Nine));
addin[10][11..0].d = (Resbus[15..4] and multDone and Pos and Ten) or
                        ((!Resbus[15..4] + 1) and multDone and Neg and Ten) or
                        (addin[10][11..0] and (!multDone or !Ten));
addin[11][11..0].d = (Resbus[15..4] and multDone and Pos and Eleven) or
                        ((!Resbus[15..4] + 1) and multDone and Neg and Eleven) or
                        (addin[11][11..0] and (!multDone or !Eleven));
addin[12][11..0].d = (Resbus[15..4] and multDone and Pos and Twelve) or
                        ((!Resbus[15..4] + 1) and multDone and Neg and Twelve) or
                        (addin[12][11..0] and (!multDone or !Twelve));
addin[13][11..0].d = (Resbus[15..4] and multDone and Pos and Thirteen) or
                        ((!Resbus[15..4] + 1) and multDone and Neg and Thirteen) or
                        (addin[13][11..0] and (!multDone or !Thirteen));


% Add all addin Registers after multiplication %
% Second level of addition %
Add2[0].dataa[11..0] = addin[0][11..0].q;
Add2[0].datab[11..0] = addin[1][11..0].q;
Add2[1].dataa[11..0] = addin[2][11..0].q;
Add2[1].datab[11..0] = addin[3][11..0].q;
Add2[2].dataa[11..0] = addin[4][11..0].q;
Add2[2].datab[11..0] = addin[5][11..0].q;
Add2[3].dataa[11..0] = addin[6][11..0].q;
Add2[3].datab[11..0] = addin[7][11..0].q;
Add2[4].dataa[11..0] = addin[8][11..0].q;
Add2[4].datab[11..0] = addin[9][11..0].q;
Add2[5].dataa[11..0] = addin[10][11..0].q;
Add2[5].datab[11..0] = addin[11][11..0].q;
Add2[6].dataa[11..0] = addin[12][11..0].q;
Add2[6].datab[11..0] = addin[13][11..0].q;

% Third Level of addition %
Add3[0].dataa[12..0] = add2[0].result[12..0];
Add3[0].datab[12..0] = add2[1].result[12..0];
Add3[1].dataa[12..0] = add2[2].result[12..0];
Add3[1].datab[12..0] = add2[3].result[12..0];
Add3[2].dataa[12..0] = add2[4].result[12..0];
Add3[2].datab[12..0] = add2[5].result[12..0];

% Fourth Level of addition %
```

```
Add4[0].dataa[13..0] = add3[0].result[13..0];
Add4[0].datab[13..0] = add3[1].result[13..0];
Add4[1].dataa[13..0] = add3[2].result[13..0];
Add4[1].datab[12..0] = add2[6].result[12..0];
Add4[1].datab[13] = add2[6].result[12];

% Fifth Level of addition %
Add5.dataa[14..0] = add4[0].result[14..0];
Add5.datab[14..0] = add4[1].result[14..0];

% Create Output Register%
% Output Register is clocked by clock %
Outdata[15..0].clk = clock;
% Output Register is updated when all processes are finished %
Outdata[15].d = (Add5.result[15] and (Multstart and Zero)) or
                              (Outdata[15].q and (!Multstart or !Zero));

Outdata[14..0].d = (!Add5.result[14..0] and (Multstart and Zero)) or
                              (Outdata[14..0].q and (!Multstart or !Zero));

end;
```

```
% 8-bit Adder %
include "adder.inc";
Subdesign 'dspadd8'
(
dataa[7..0], datab[7..0]                :Input;
result[8..0]                                   :Output;
)

Variable
Adder8[7..0]                          :Adder;

Begin
% ***** series of carry assignments ***** %
Adder8[0].Cin=Gnd;
Adder8[7..1].Cin=Adder8[6..0].co;

% ***** Addend assignments ***** %
Adder8[7..0].A=dataa[7..0];
Adder8[7..0].B=datab[7..0];

% ***** Sum assignments ***** %
result[7..0]=Adder8[7..0].s;

% ***** Create Automatic Sign Extension Bit ***** %
result[8] = (dataa[7] and datab[7] and Vcc) or
                      (!dataa[7] and !datab[7] and Gnd) or
                      ((dataa[7] xor datab[7]) and result[7]);

End;
```

```
% 9-bit Adder %
include "adder.inc";
Subdesign 'dspadd9'
(
dataa[8..0], datab[8..0]                :Input;
result[9..0]                                   :Output;
)

Variable
Adder9[8..0]                          :Adder;
```

```
Begin
% ***** series of carry assignments ***** %
Adder9[0].Cin=Gnd;
Adder9[8..1].Cin=Adder9[7..0].co;

% ***** Addend assignments ***** %
Adder9[8..0].A=dataa[8..0];
Adder9[8..0].B=datab[8..0];

% ***** Sum assignments ***** %
result[8..0]=Adder9[8..0].s;

% ***** Create Automatic Sign Extension Bit ***** %
result[9] = (dataa[8] and datab[8] and Vcc) or
                    (!dataa[8] and !datab[8] and Gnd) or
                    ((dataa[8] xor datab[8]) and result[8]);

End;




% 13-bit Adder %
include "adder.inc";
Subdesign 'dspadd12'
(
dataa[11..0], datab[11..0]                    :Input;
result[12..0]                                        :Output;
)

Variable
Adder12[11..0]                                :Adder;

Begin
% ***** series of carry assignments ***** %
Adder12[0].Cin=Gnd;
Adder12[11..1].Cin=Adder12[10..0].co;

% ***** Addend assignments ***** %
Adder12[11..0].A=dataa[11..0];
Adder12[11..0].B=datab[11..0];

% ***** Sum assignments ***** %
result[11..0]=Adder12[11..0].s;

% ***** Create Automatic Sign Extension Bit ***** %
result[12] = (dataa[11] and datab[11] and Vcc) or
                    (!dataa[11] and !datab[11] and Gnd) or
                    ((dataa[11] xor datab[11]) and result[11]);

End;




% 14-bit Adder %
include "adder.inc";
Subdesign 'dspadd13'
(
dataa[12..0], datab[12..0]                    :Input;
result[13..0]                                        :Output;
)

Variable
Adder13[12..0]                                :Adder;

Begin
% ***** series of carry assignments ***** %
Adder13[0].Cin=Gnd;
Adder13[12..1].Cin=Adder13[11..0].co;
```

```
% ***** Addend assignments ***** %
Adder13[12..0].A=dataa[12..0];
Adder13[12..0].B=datab[12..0];

% ***** Sum assignments ***** %
result[12..0]=Adder13[12..0].s;

% ***** Create Automatic Sign Extension Bit ***** %
result[13] = (dataa[12] and datab[12] and Vcc) or
                     (!dataa[12] and !datab[12] and Gnd) or
                     ((dataa[12] xor datab[12]) and result[12]);

End;




% 15-bit Adder %
include "adder.inc";
Subdesign 'dspadd14'
(
dataa[13..0], datab[13..0]                    :Input;
result[14..0]                                        :Output;
)

Variable
Adder14[13..0]                                :Adder;

Begin
% ***** series of carry assignments ***** %
Adder14[0].Cin=Gnd;
Adder14[13..1].Cin=Adder14[12..0].co;

% ***** Addend assignments ***** %
Adder14[13..0].A=dataa[13..0];
Adder14[13..0].B=datab[13..0];

% ***** Sum assignments ***** %
result[13..0]=Adder14[13..0].s;

% ***** Create Automatic Sign Extension Bit ***** %
result[14] = (dataa[13] and datab[13] and Vcc) or
                     (!dataa[13] and !datab[13] and Gnd) or
                     ((dataa[13] xor datab[13]) and result[13]);

End;




% 16-bit Adder %
include "adder.inc";
Subdesign 'dspadd15'
(
dataa[14..0], datab[14..0]                    :Input;
result[15..0]                                        :Output;
)

Variable
Adder15[14..0]                                :Adder;

Begin
% ***** series of carry assignments ***** %
Adder15[0].Cin=Gnd;
Adder15[14..1].Cin=Adder15[13..0].co;

% ***** Addend assignments ***** %
Adder15[14..0].A=dataa[14..0];
Adder15[14..0].B=datab[14..0];
```

```
% ***** Sum assignments ***** %
result[14..0]=Adder15[14..0].s;

% ***** Create Automatic Sign Extension Bit ***** %
result[15] = (dataa[14] and datab[14] and Vcc) or
                     (!dataa[14] and !datab[14] and Gnd) or
                     ((dataa[14] xor datab[14]) and result[14]);

End;



Subdesign 'adder'
(
Cin, A, B           :Input;
S, Co               :Output;
)
Variable
Temp                :Node;
Begin
Temp = A xor B;
% Create Sum Term %
S = Cin xor Temp;
% Create Carry Out Term %
Co = (A and B) or (Cin and Temp);
End;
```

BIBLIOGRAPHY

1. Brain, Marshall. *Win32 System Services, The Heart of Windows 95 and Windows NT.* Upper Saddle River, New Jersey: Prentice Hall, 1996.

2. Oppenheim, Alan V. and Ronald W. Schafer. *Discrete-Time Signal Processing.* Upper Saddle River, New Jersey: Prentice Hall, 1999.

3. Orfanidis, Sophocles J. *Introduction to Signal Processing.* Upper Saddle River, New Jersey: Prentice Hall, 1996.

4. Patterson, David A. and John L Hennessy. *Computer Organization and Design, The Hardware/Software Interface.* San Francisco, California: Morgan Kaufmann Publishers, Inc., 1994.

5. Scarpino, Frank A. *VHDL and AHDL, Digital System Implementation.* Upper Saddle River, New Jersey: Prentice Hall, 1998.