BMS Evening College of Engineering

CERTIFICATE

This is to certify that the following students

Bhaskar. G
Umapathy. J
Raghavendra. M

have successfully completed the Project Work Entitled

"ANGLE ENCODER NOISE REDUCTION USING DIGITAL FILTERS"

at

Raman Research Institute, Bangalore

in partial fulfillment of the requirements for the award of the degree of Bachelor of Engineering in Electronics and Communication Engineering by the Bangalore University during the academic year 1999 - 2000

Dr. M Murugesh Mudaliar
Principal,
BMSECE, Bangalore

Prof. B S Chandrashekar
HOD, Dept. of E & C,
BMSECE, Bangalore.

Head of the Department
Electronics & Communications
B.M.S. Evening College of Engineering
Bangalore - 560019.

RAMAN RESEARCH INSTITUTE
C.V. Raman Avenue, Sadashivanagar, Bangalore-560 080 India

August 28, 2000

Certificate

This is to certify that the project work titled

"Angle Encoder Noise Reduction using Digital Filter"

has been successfully completed by

Bhaskar G

Raghavendra M

Umapathy J

Students of the final year,
Bachelor of Engineering (Electronics and Communication)
B.M.S. Evening College of Engineering, Bangalore
under the guidance of *Mr. M. Selvamani.*

This project is in partial fulfillment of the requirement for the award of Bachelor's Degree in Electronics & Communication Engineering during the academic year 1999-2000.

M. Selvamani
Head
General Electronics & Instrumentation

Phone : (80) 334 0122 Fax : (80) 334 0492 Telegrams : RAMANINST e-mail : root@rri.ernet.in

ACKNOWLEDGEMENTS

A lot of thought, efforts, hard work was involved in shaping this project ***"ANGLE ENCODER NOISE REDUCTION USING DIGITAL FILTER"*** *and making it a fruitful success. In our endeavor to accomplish our project, we have received counsel, guidance assistance from many people.*

We take it as a privilege to express our heartfelt gratitude to ***Dr. A Krishnan*** *Visiting Professor, Raman Research Institute, Bangalore and* ***Mr. M Selvamani****, Raman Research Institute, for their encouragement and support which has gone a long way not only in the successful completion of our project but all through our project stay in this organization and has been our guiding spirit.*

We express our gratitude to ***Dr. M. Murugesh Mudaliar*** *(Principal, BMSECE) and* ***Prof. B S Chandra Shekar*** *(HOD, Dept of Electronics & communication) for giving us permission to conduct our project at Raman Research Institute, Bangalore.*

We thank from the depth of our hearts to ***Mr. K Ramesh,*** *and* ***Mr. K Gurukiran*** *who were instrumental in providing continuous assistance, encouragement and guidance while developing our project.*

We would also like to extend our cordial thanks to all the teaching and non-teaching staff and to every one who in some way or the other have been involved in carrying out the project.

Bhaskar G
Umapathy J
Raghavendra M

SYNOPSYS

Almost every field of Science and Engineering deal with signals. There is always the requirement of using filters to modify the frequency spectrum of such filters. Digital filters are increasingly being used because of their characteristics. Other advantages of digital filters are high reliability, high accuracy, and almost no effect of component drift and component tolerance on filter characteristics.

Absolute encoders are used for measuring angular positions of telescope fairly accurately. The ten-meter radio telescope at RRI uses a 21 bit encoders for both the azimuth and elevation axes. There is considerable jitter on the output of encoders because of vibrations and other noisy environments. It is planned to design a real-time digital filter for cleaning up the encoder outputs.

This will be implemented on a DSP based computer.

The Scope includes -

1. Understanding Digital Filters.
2. Defining the Specifications of the filter for this application.
3. Designing the real-time filter for implementation.
4. Actual realization of the filter on the DSP Board.
5. Testing the filter with the Encoder Data

CONTENTS

CHAPTER 1

Introduction

INTRODUCTION

Overview

The RRI 10.4m Radio Telescope as an alt-azimuth mount, with the elevation axis over the azimuth axis. The weight rotating about the Elevation axis including counterweights in 18 tons approximately. The weight rotating about the azimuth axis is approximately 30 tons inclusive of elevation. Tracking a star with this system requires simultaneous movements of both the elevation and the azimuth axes.

Telescope Drive Hardware

Each axis of the primary reflector has a pair of 1.6 HP high performance permanent magnet DC Servo motors. In both the axes the two motors are connected through a gearbox-bullgear system in an electromechanically tensioned mode (backtorque) to increase the stiffness of the drive transmission system. This helps reduce the backlash in the transmission system, an absolute necessity for high performance servo operation and accuracy of control. The elevation axis a 0-92° sector gear in place of the bull gear. The motors aid each other for 'slewing' providing nearly double the torque of each motor. While tracking a star a differential torque (backtorque) is maintained between the two motors.

Each motor has a four-quadrant phase controlled drive controller. The motors have built-in tachometers and brakes. The controllers have independent speed and torque loops in cascade. It is

possible to adjust the 'back torque' electrically to vary the tensioning force between the two motors.

Telescope Control Scheme

Telescope position information in each axis is obtained from a 21-bit synchro type encoder. Both the encoders are connected to a 'data converter' unit. The position values are available from the data converter in binary form. The data converter also has a strobe control line to latch the last read information of each axis position. The data converter updates the encoder values 500 times a second.

An Intel 486 computer is used to control the telescope for tracking and data acquisition. An astronomical clock of high accuracy interrupts the computer every 100 milliseconds. Both the azimuth and elevation are read into ISA data acquisition card. The position errors are computed and then a suitable algorithm updates the values of the drive voltages of the corresponding motor control units through suitable interface circuits and Digital to Analog converters of the computer. Two control algorithms are available to update the servo.

1. Every 100 milliseconds.
2. Every 1 second.

The secondary (sub) reflector is presently driven by a motor – gearbox arrangement in open loop. The speed of movement can be controlled from the observer desk only. The total movement possible for the secondary reflector is 20 millimeters. The position is measured through a precision 10-turn potentiometer. Provision has also been made to drive this one-axis secondary reflector through a D/A in a position control loop from the computer.

The computer also monitors the safety interlock signals to take appropriate corrective action in case of fault either in drive or receiver system. Observer interaction is available through a CRT

monitor of the computer. All the relevant information regarding the observation is available on the screen.

The entire telescope control system has been designed with high reliability. The following were also the major consideration in the design of the telescope control system.

a. High static accuracy under varying loads like wind gusts, static and dynamic frictional forces etc.
b. High resolution (Better than 1Arc second)
c. Good tracking accuracy at varying speeds.
d. Smooth acceleration and deceleration capabilities to avoid stress on the telescope structure and drive system.

Following are the important specification of the telescope control system

TELESCOPE	:	alt azimuth mount parabolic dish
SIZE	:	12.4m diameter
		84 hexagonal panel surface
SURFACE AREA	:	250 microns(rms)
FREQUENCY OF		
OPERATION	:	85 to 115 Ghz in mm band
		1.4Ghz and 6.7Ghz in cm band
TYPE OF OBSERVATIONS	:	Line observations like Methanol maser line observation
		21-cm neutral hydrogen emission lines
		S10 maser lines
		12co observations
BACK-ENDS AVAILABLE	:	Multi – channel filter banks
		Auto – correlators
		Acousto – optic spectrometers
SITE SPECIFICATIONS	:	Altitude – 930m
		Longitude – 13degrees North (13° 00' 44.6"N)
		Latitude – 77.5degrees East (77° 34' 59.67"E)

CHAPTER 2

Filters

FILTERS

Definition

Filtering is a process by which the frequency spectrum of a signal can be modified, reshaped, or manipulated according to some desired specification. It may entail amplifying or attenuating a range of frequency components, rejecting or isolating one specific frequency component, etc. The uses of filtering are manifold, e.g., to eliminate signal contamination such as noise, to remove signal distortion brought about by an imperfect transmission channel or by inaccuracies in measurement, to separate two or more distinct signals which were purposely mixed in order to maximize channel utilization, to resolve signals into their frequency components, to demodulate signals, to convert discrete-time signals into continuous-time signals, and to band limit signals.

Classifications:

Based on Implementation

Analog filters

Digital filters

Based on Characteristics

Low Pass Filter

High Pass Filter

Band Stop Filter

Band Pass Filter

Characteristics of Filter

The important characteristics of a filter are

Cut-off Frequency sharpness

Passband Ripple

Roll-off

Step Response

Impulse Response

Frequency Response

Phase Response

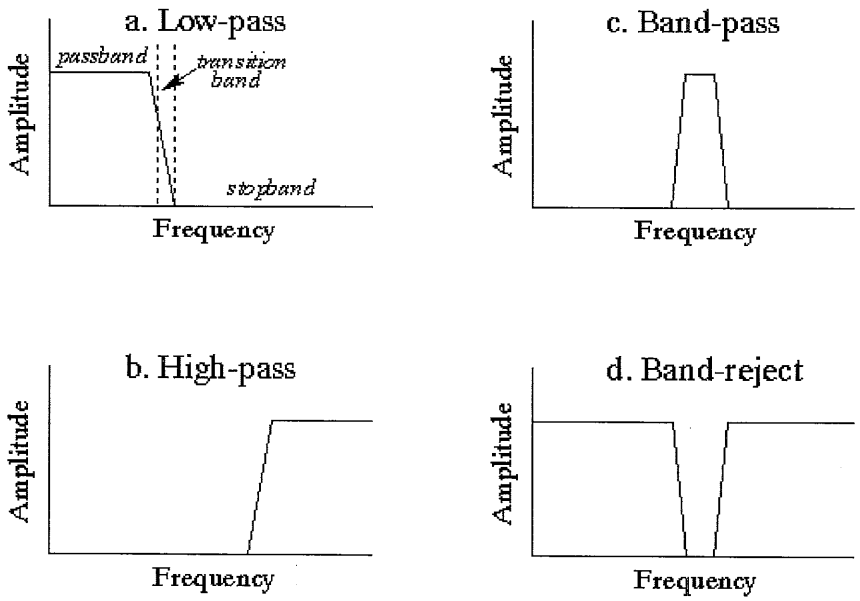
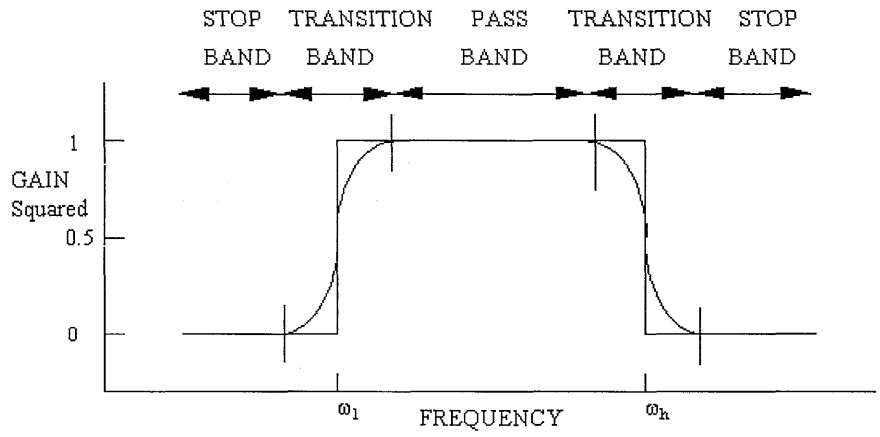


Figure shows the four basic filters response **a**.low-pass, **b**.high-pass, **c**.band-pass, and **d**.band-reject.

An Ideal filter would have a rectangular magnitude response as shown is Fig. The desired frequencies are passed with no attenuation, while the undesired frequencies are completely blocked.

Unfortunately, ideal filters are non-causal and therefore not realizable. However, there are practical filter designs that approximate the ideal filter characteristics and which are realizable.

The magnitude response of a practical filter is as shown in the fig. The filter characteristics divide the spectrum into three general regions as shown.



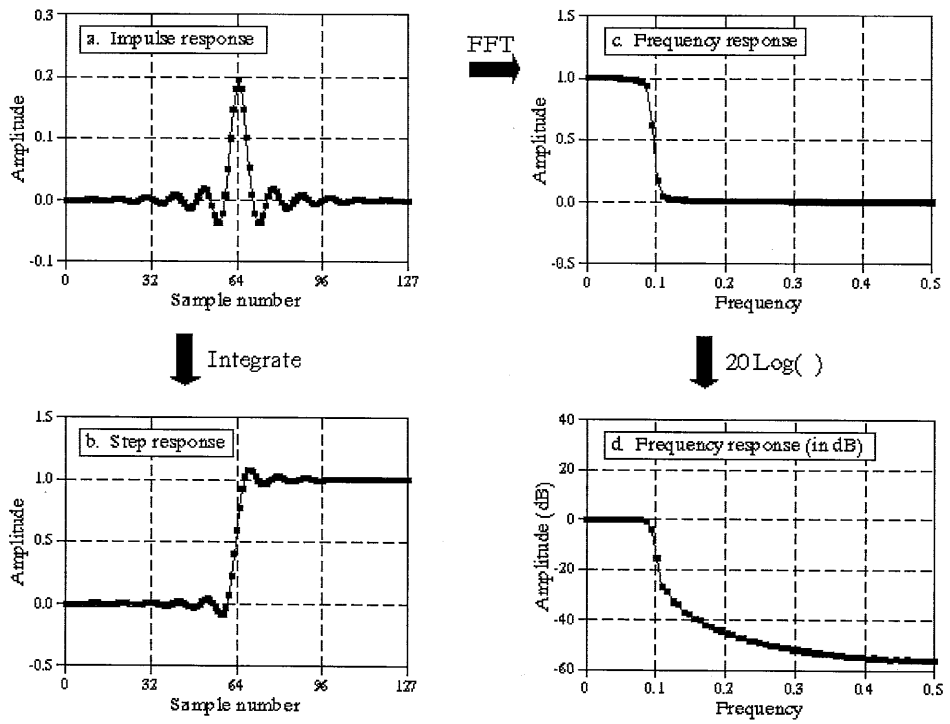
Referring to figure, the pass band of the filter refers to the range of frequencies that the filter will accept and transmit without severe degradation in amplitude; the stop band conversely describes those frequencies which are significantly attenuated. The transition band represents the frequencies over which the filter response changes (i.e., the frequencies between the stop and pass bands); for an ideal filter, this occurs instantaneously. A **fast roll-off** means that the transition band is very narrow. The division between the passband and transition band is called the **cutoff frequency.** The cut-off frequency, or break point, of an ideal filter refers to the frequency defining the edge of the pass band, although for real (non-ideal) filters it gives the transition point between stop and pass bands. This is taken to be the frequency at which the power of the output signal is half that of the power of the most efficiently passed signal. Power ratios are typically described in terms of

decibels, defined as: 10log10 (P1/P2). The cut off frequency is thus frequently referred to as the 3dB point, since

$$10\log_{10}\left(P_{out}/P_{in}\right) = 10\log_{10} 0.5 \approx -3\text{dB}$$

Every linear filter has an impulse response, a step response, and a frequency response. The step response can be found by discrete integration of the impulse response. The frequency response can be found from the impulse response by using the Fast Fourier Transform (FFT), and can be displayed either on a linear scale or in decibels.

Every linear filter has an **impulse response**, a **step response** and a **frequency response.** Each of these responses contains complete information about the filter, but in a different form. If one of the three is specified, the other two are fixed and can be directly calculated.



Filter parameter (a). Impulse response, (b). Step response, (c) The frequency response of the (c). impulse response (d). step response in dB.

Analog Vs Digital Filters

Most digital signals originate in analog electronics, the signal needs to be filtered in two ways either use a analog filter before digitization or a digital filter after digitization.

Analog filters are cheap, fast, and have a large dynamic range in both amplitude and frequency. Digital filters, in comparison, are vastly superior in the level of performance that can be achieved. For example, a low-pass digital filter having a gain of 1 +/- 0.0002 from DC to 1000 hertz, and a gain of less than 0.0002 for frequencies above 1001 hertz. The entire transition occurs within only 1 hertz, which cannot be expected from an op-amp circuit! Digital filters can achieve *thousands* of times better performance than analog filters. This makes a dramatic difference in how filtering problems are approached. With analog filters, the emphasis is on handling limitations of the electronics, such as the accuracy and stability of the resistors and capacitors. In comparison, digital filters are so good that the performance of the filter is frequently ignored. The emphasis shifts to the limitations of the *signals*, and the *theoretical* issues regarding their processing. Digital filters have better performance in many areas such as **pass band ripple, roll –off, stop band attenuation, step response symmetry.**

Introduction to Digital Filters

The **digital filter** is a digital system that can be used to filter discrete-time signals. It can be implemented by means of software (computer programs), dedicated hardware, or a combination of software and hardware. Software digital filters may be implemented using a low-level language on a general-purpose DSP chip or in terms of a high level language on a personal computer or workstation. At the other extreme, hardware digital filters can be designed using a number of

highly specialized inter connected VLSI chips. Both software and hardware digital filters can be used to process real-time and non-real-time (recorded) signals, except that the latter are usually much faster and can process signals whose frequency spectra extend too much higher frequencies. Digital filters are used for two general purposes:

a) separation of signals that have been combined, and

b) restoration of signals that have been distorted in some way.

Analog (electronic) filters can be used for these same tasks; however, digital filters can achieve far superior results.

Filter Basics

Digital filters are very useful for their extraordinary performance and is one of the key reasons that DSP has become so popular. Signal *separation* and signal *restoration*. Signal separation is needed when a signal has been contaminated with interference, noise, or other signals. Signal restoration is used when a signal has been distorted in some way.

The most straightforward way to implement a digital filter is by convolving the input signal with the digital filter's impulse response. There is also another way to make digital filters, called **recursion**. When a filter is implemented by convolution, each sample in the output is calculated by *weighting* the samples in the input, and adding them together. Recursive filters are an extension of this, using previously calculated values from the *output*, besides points from the *input*. Instead of using a filter kernel, recursive filters are defined by a set of **recursion coefficients**. The impulse responses of recursive filters are composed of sinusoids that exponentially decay in amplitude, which is infinitely long. However, the amplitude eventually drops below the round-off noise of the

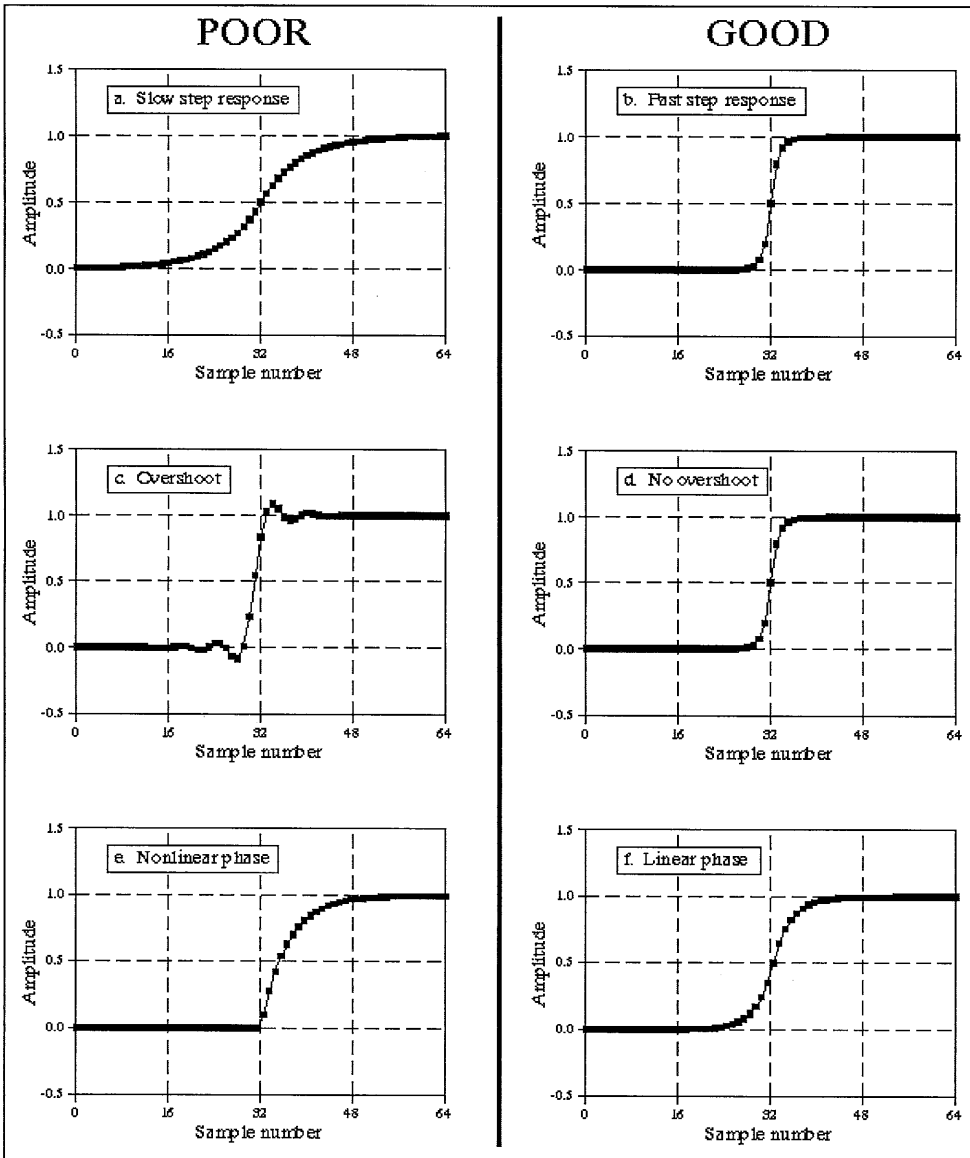
system, and the remaining samples can be ignored. Because of this characteristic, recursive filters are also called **Infinite Impulse Response or IIR** filters. In comparison, filters carried out by convolution are called **Finite Impulse Response or FIR** filters.

Time Domain Parameters

The step response parameters that are important in filter design are shown in Fig. To distinguish events in a signal, the duration of the step response must be shorter than the spacing of the events. This dictates that the step response should be as fast as possible. This is shown in Figs. (a) & (b). The most common way to specify the **risetime** is to quote the number of samples between the 10% and 90% amplitude levels.

Figures (c) and (d) shows the next parameter that is important: **overshoot** in the step response. Overshoot must generally be eliminated because it changes the amplitude of samples in the signal; this is a basic distortion of the information contained in the time domain.

Finally, it is often desired that the upper half of the step response be **symmetrical** with the lower half, as illustrated in (e) and (f). This symmetry is needed to make the rising edges look the same as the falling edges. This symmetry is called **linear phase**, because the frequency response has a phase that is a straight line.

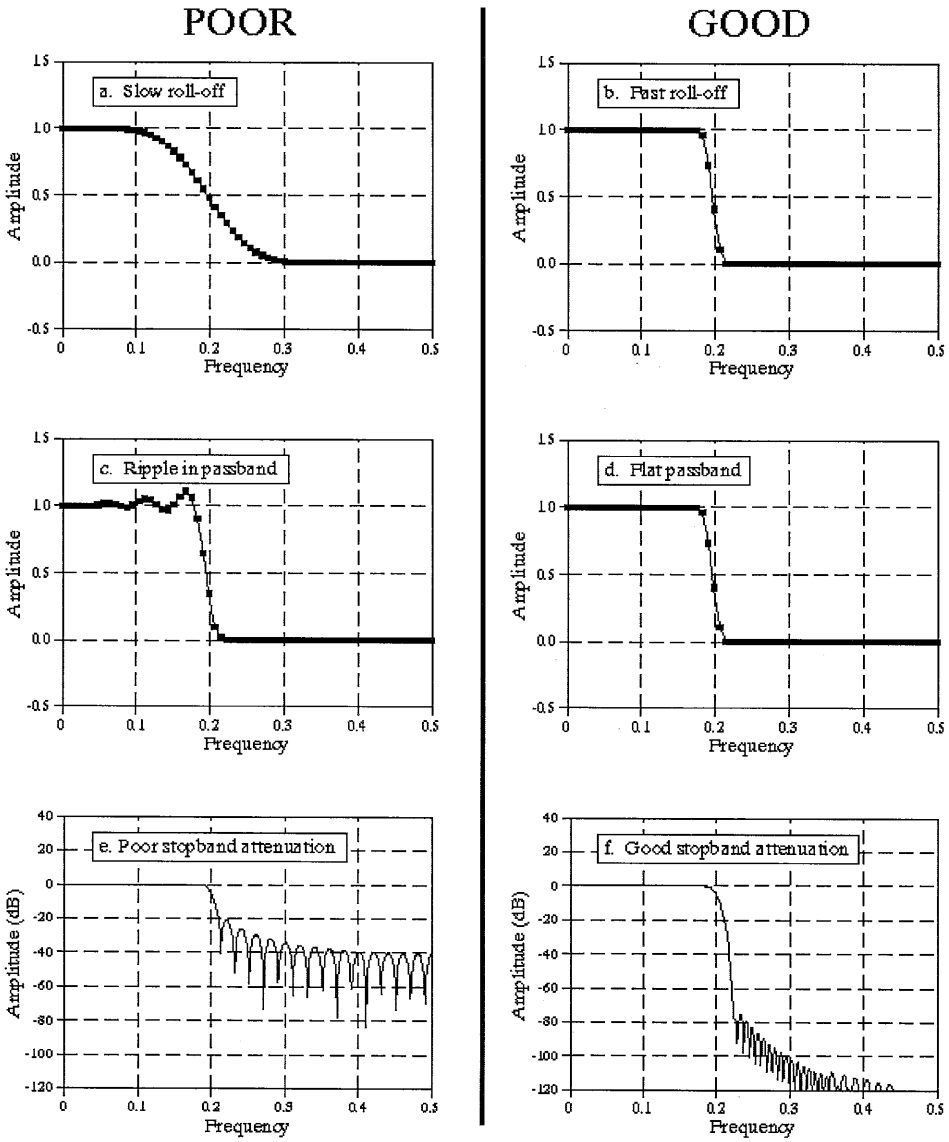


Parameters for evaluating time domain performance. The step response is used to measure how well a filter performs in the time domain. Three parameters are important: (1) transition speed (risetime), shown in (a) and (b), (2) overshoot, shown in (c) and (d), and (3) phase linearity (symmetry between the top and bottom halves of the step), shown in (e) and (f).

Frequency Domain parameters

Frequency domain filters are generally used to pass certain frequencies (the passband), while blocking others (the stopband). Four responses are the most common: low-pass, high-pass, band-pass, and band-reject.

The following figure shows three parameters that measure how well a filter performs in the frequency domain. To separate closely spaced frequencies, the filter must have a fast roll-off, as illustrated in (a) and (b). For the passband frequencies to move through the filter unaltered, there must be no passband ripple, as shown in (c) and (d). Lastly, to adequately block the stopband frequencies, it is necessary to have good stopband attenuation, displayed in (e) and (f).



Parameters for evaluating frequency domain performance. The frequency responses shown are for low-pass filters. Three parameters are important: (1) roll-off sharpness, shown in (a) and (b), (2) passband ripple, shown in (c) and (d), and (3) stopband attenuation, shown in (e) and (f).

Filter Classification

Table below summarizes how digital filters are classified by their use and by their implementation. The use of a digital filter can be broken into three categories: **time domain, frequency domain and custom**. Time domain filters are used when the information is encoded in the shape of the signal's waveform. Time domain filtering is used for such actions as: smoothing, DC removal, waveform shaping etc. Frequency domain filters are used when the information is contained in the amplitude, frequency, and phase of the component sinusoids. The goal of these filters is to separate one band of frequencies from another. Custom filters are used when a special action is required by the filter, something more elaborate than the four basic responses (high-pass, low-pass, band- pass and band-reject).

Digital filters can be implemented in two ways, by convolution (also called finite impulse response or FIR) and by recursion (also called infinite impulse response or IIR). Filters carried out by convolution can have far better performance than filters using recursion, but execute much more slowly.

	Convolution Finite Impulse Response (FIR)	Recursion Infinite Impulse Response (IIR)
Time Domain (smoothing DC removed)	Moving Average	Single Pole
Frequency Domain (separating frequencies)	Windowed –sinc	Chebyshev
Custom (Deconvolution)	FIR custom	Iterative design

Moving Average Filters

The moving average is the most common filter in DSP, mainly because it is the easiest digital filter to understand and use. In spite of its simplicity, the moving average filter is *optimal* for a common task: reducing random noise while retaining a sharp step response. This makes it the premier filter for time domain encoded signals. However, the moving average is the *worst* filter for frequency domain encoded signals, with little ability to separate one band of frequencies from another. Relatives of the moving average filter include the Gaussian, Blackman, and multiple- pass moving average. These have slightly better performance in the frequency domain, at the expense of increased computation time.

Implementation by Convolution

The moving average filter operates by averaging a number of points from the input signal to produce each point in the output signal. In equation form, this is written:

$$y[i] = \frac{1}{M} \sum_{k=0}^{M-1} x[i+k]$$

Where

x [ ] is the input signal,

y [ ] is the output signal, and

M is the number of points in the average.

As an alternative, the group of 5 points from the input signal can be chosen *symmetrically* around the output point:

$$y[i] = \frac{x[i-2] + x[i-1] + x[i] + x[i+1] + x[i+2]}{5}$$

Noise Reduction vs. Step Response

The moving average filter is very good for many applications, it is *optimal* for a common problem, reducing random white noise while keeping the sharpest step response. The smoothing action of the moving average filter decreases the amplitude of the random noise (good), but also reduces the sharpness of the edges (bad). Of all the possible linear filters that could be used, the moving average produces the lowest noise for a given edge sharpness. The amount of noise reduction is equal to the square-root of the number of points in the average. For example, a 100 point moving average filter reduces the noise by a factor of 10. The lowest noise is obtained when all the input samples are treated equally, i.e., the moving average filter.

Frequency Response

The roll-off is very slow and the stopband attenuation is ghastly. Clearly, the moving average filter cannot separate one band of frequencies from another. In short, the moving average is an exceptionally good *smoothing filter* (the action in the time domain), but an exceptionally bad *low-pass filter* (the action in the frequency domain).

Multiple-pass moving average filters involves passing the input signal through a moving average filter two or more times. Two passes are equivalent to using a *triangular* filter kernel (a rectangular filter kernel convolved with itself). After four or more passes, the equivalent filter kernel looks like a *Gaussian*. The exact shape of the Blackman window looks much like a Gaussian.

The advantages of Multiple-pass moving average filters are

i. These filters have better *stopband attenuation* than the moving average filter.

ii. The filter kernels *taper* to a smaller amplitude near the ends.

iii. The step responses are *smooth* curves, rather than the abrupt straight line of the moving average.

The biggest difference in these filters is *execution speed*. Using a recursive algorithm (described next), the moving average filter will run like lightning in your computer. In fact, it is the *fastest* digital filter available. Multiple passes of the moving average will be correspondingly slower, but still very quick. In comparison, the Gaussian and Blackman filters are excruciatingly slow, because they must use convolution.

Recursive Implementation

A tremendous advantage of the moving average filter is that it can be implemented with an algorithm that is very fast. If y[i+1] has been found using y[i], then y[i+2] can be calculated from sample y[i+1], and so on. After the first point is calculated in y[ ], all of the other points can be found with only a single addition and subtraction per point. This can be expressed in the equation:

$$y[i] = y[i - 1] + x[i + p] - x[i - q]$$

where

$$p = (M - 1)/2$$

$$q = p + 1$$

Notice that this equation uses two sources of data to calculate each point in the output: points from the input *and* previously calculated points from the output. This is called a **recursive** equation, meaning that the result of one calculation is used in *future* calculations. Most recursive filters have an infinitely long impulse response (IIR), composed of sinusoids and exponentials. The impulse response of the moving average is a rectangular pulse (finite impulse response, or FIR).

This algorithm is faster than other digital filters for several reasons.

i. Only two computations per point, regardless of the length of the filter kernel.

ii. Addition and subtraction are the only math operations needed.

iii. The indexing scheme is very simple.

iv. The entire algorithm can be carried out with integer representation.

Windowed-Sinc Filters

Windowed-sinc filters are used to separate one band of frequencies from another. They are very stable. These exceptional frequency domain characteristics are obtained at the expense of poor performance in the time domain, including excessive ripple and overshoot in the step response. When carried out by standard convolution, windowed-sinc filters are easy to program, but slow to execute. FFT can be used to dramatically improve the computational speed of these filters.

Strategy of the Windowed-Sinc

All frequencies below the cutoff frequency, f_c, are passed with unity amplitude, while all higher frequencies are blocked. The passband is perfectly flat, the attenuation in the stopband is infinite, and the transition between the two is infinitesimally small. The **sinc functio**n, given by:

$h[i] = \sin(2*pi*f_c*i)/i*pi$

Convolving an input signal with this filter kernel provides a *perfect* low-pass filter. The problem is, the sinc function continues to both negative and positive infinity without dropping to zero amplitude.

To get around this problem, we will make two modifications to the sinc function in (b), resulting in the waveform shown in (c). First, it is truncated to $M + 1$ points, symmetrically chosen around the main lobe, where M is an even number. All samples outside these $M + 1$ points are set to zero, or simply ignored. Second, the entire sequence is shifted to the right so that it runs from 0 to M. This allows the filter kernel to be represented using only *positive* indexes. While many programming languages allow *negative* indexes, they are a nuisance to use. The sole effect of this M/2shift in the filter kernel is to shift the output signal by the same amount.

Since the modified filter kernel is only an approximation to the ideal filter kernel, it will not have an ideal frequency response. To find the frequency response that is obtained, the Fourier transform can be taken of the signal in (c), resulting in the curve in (d). It's a mess! There is excessive ripple in the passband and poor attenuation in the stopband (recall the Gibbs effect discussed earlier). These problems result from the abrupt discontinuity at the ends of the truncated sinc function. Increasing the length of the filter kernel does not reduce these problems; the discontinuity is significant no matter how long M is made

Fortunately, there is a simple method of improving this situation. Figure (e) shows a smoothly tapered curve called a **Blackman windo**w. Multiplying the truncated-sinc, (c), by the Blackman window, (e), results in the **windowed-sinc** filter kernel shown in (f). The idea is to reduce the abruptness of the truncated ends and thereby improve the frequency response. Figure (g) shows this improvement. The passband is now flat, and the stopband attenuation is so good it cannot be seen in this graph.

Several different windows are available, most of them named after their original developers in the 1950s. Only two are worth using, the **Hamming window** and the **Blackman window** These are given by:

$w[i] = 0.54 - 0.46 \cos(2*pi*i/M)$

$w[i] = 0.42 - 0.5 \cos(3*pi*i/M) + 0.08 \cos(4*pi*i/M)$

CHAPTER 3

DESIGN OF FIR FILTER

DESIGN

Objective

The objective of the project was to simulate the Digital Filter Using ADSP 21061 card to process the signal at the output of the Twenty One bit Angle Encoder of the Radio Telescope and to have a noiseless control signal fed the Control System.

FIR filter was chosen for this purpose and following were the major milestones of the project

- Design specifications
- Arriving at the number of taps required for the filter based on the specified filter parameters
- Simulation of the Filter using Matlab for different Number of Taps
- Writing the C-Code for to simulate the filter
- Assembly code generation for ADSP 21061
- Compiling and linking of the assembly code
- Simulation of the Filter code using EZ-Lite Kit Simulator
- Plotting and analyzing the result at each stage

Design Specification

The Design specification for the FIR filter is as below

Cut-off Frequency - 1 Hz

Stop Band Attenuation of –50db at 2Hz

Sampling Frequency - 10 samples/second

Design Equations

The Design equation for calculating the number of taps N for the FIR filter using rectangular window is

$$N \geq 4\pi/(\omega 2 - \omega 1)$$

Where $\omega 1 = \Omega 1 * T$

$\Omega 1$ being the Cut-off frequency and T sampling interval

$\omega 2 = \Omega 2 * T$

$\Omega 2$ being the Stop band frequency.

Therefore we get

$$N \geq 4\pi / (4\pi/10 - 2\pi/10)$$

$$N \geq 20$$

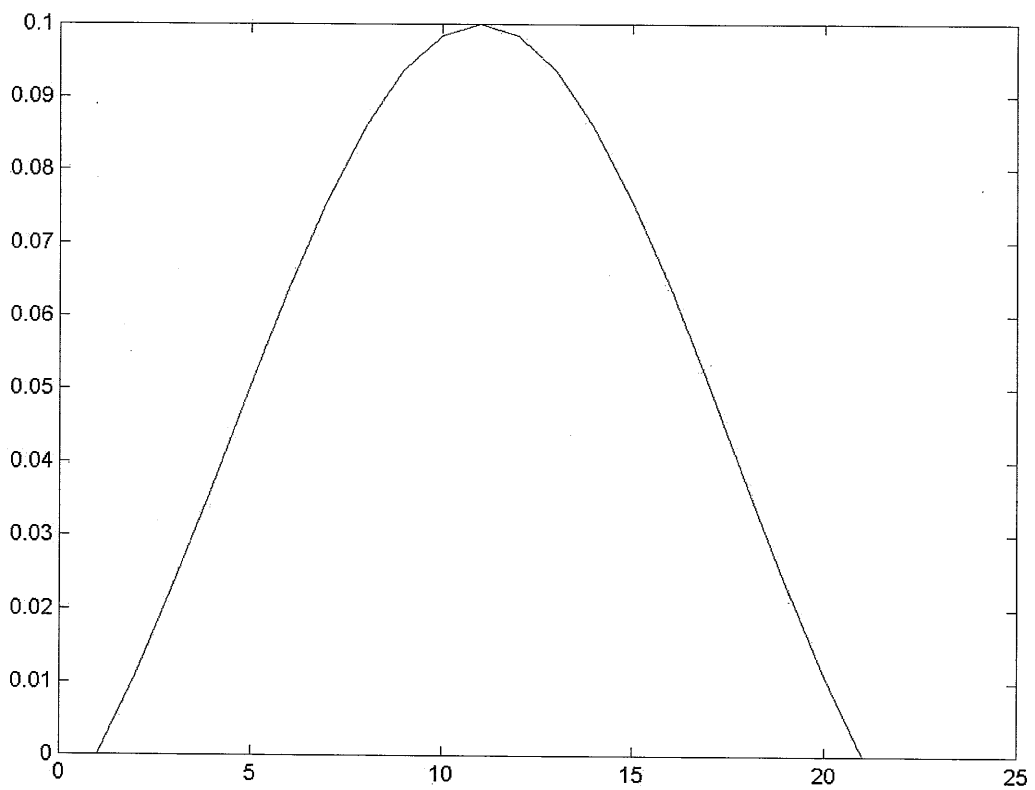
Thus the number of taps required for the realization of the FIR filter is $N \geq 20$

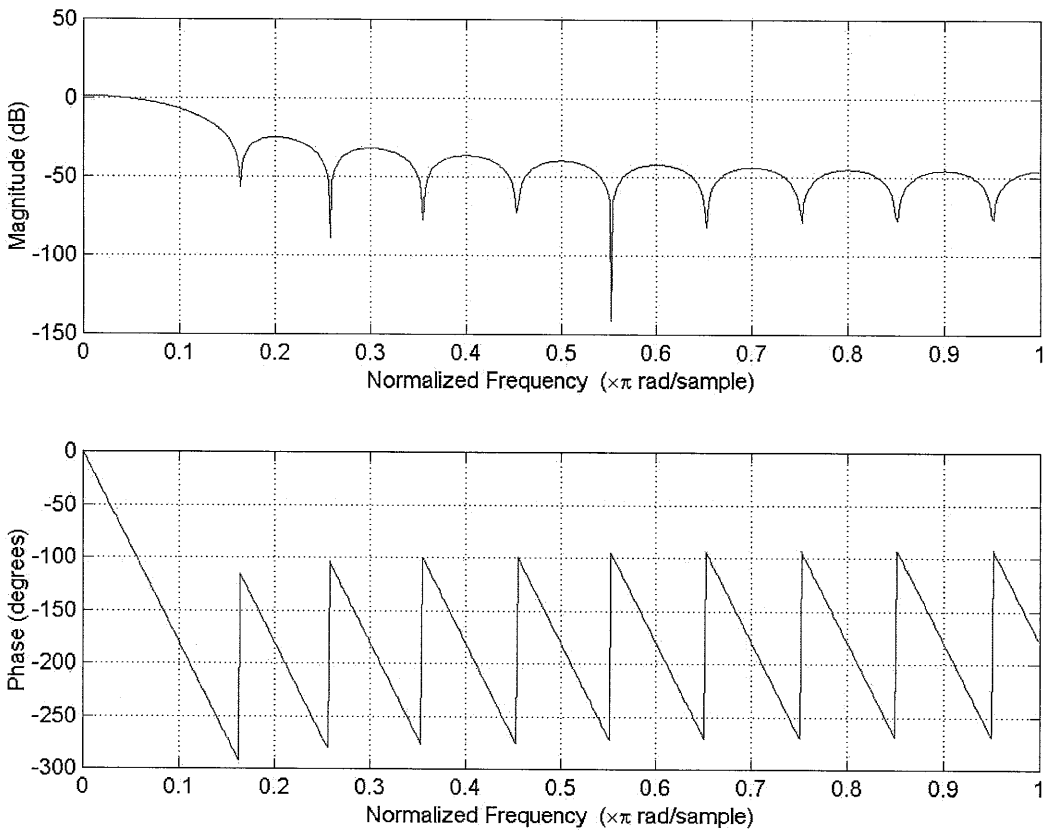
For FIR filters with rectangular window the number taps must be odd therefore the number of taps required should be $N \geq 21$

Matlab Code and Test Results

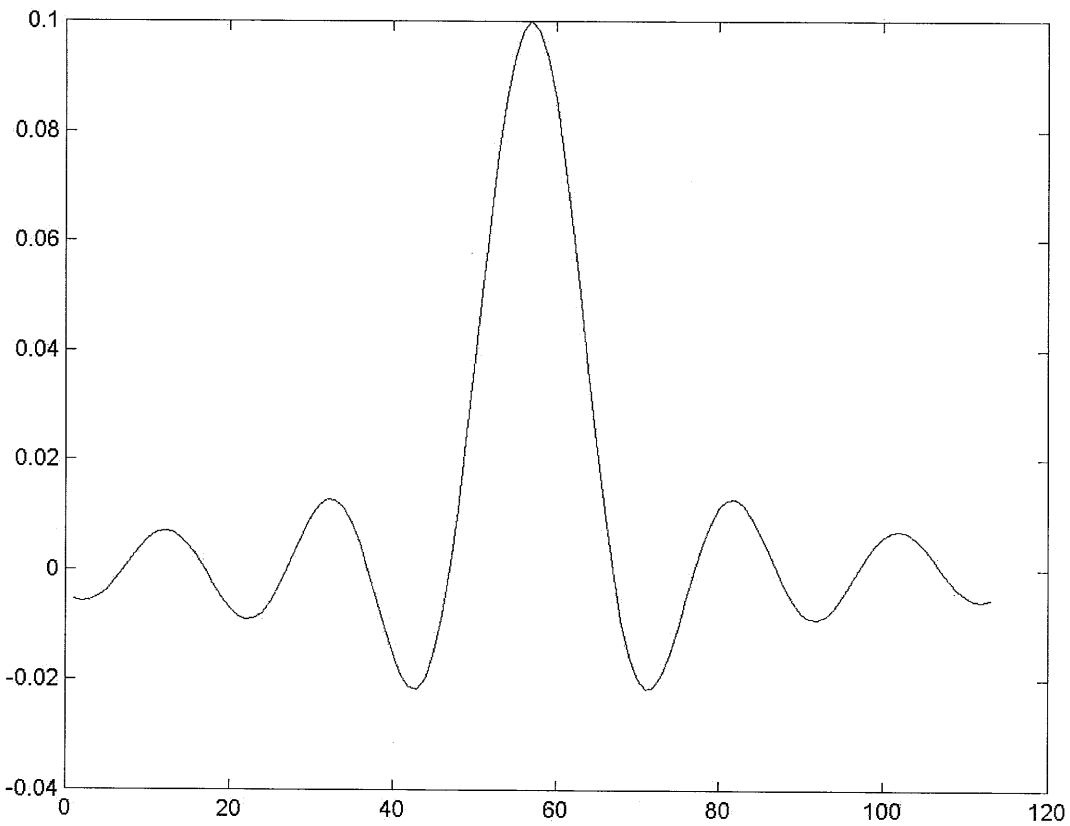
An FIR filter was designed in MATLAB using the number of taps derived. The MATLAB code and response of the filter is as given below

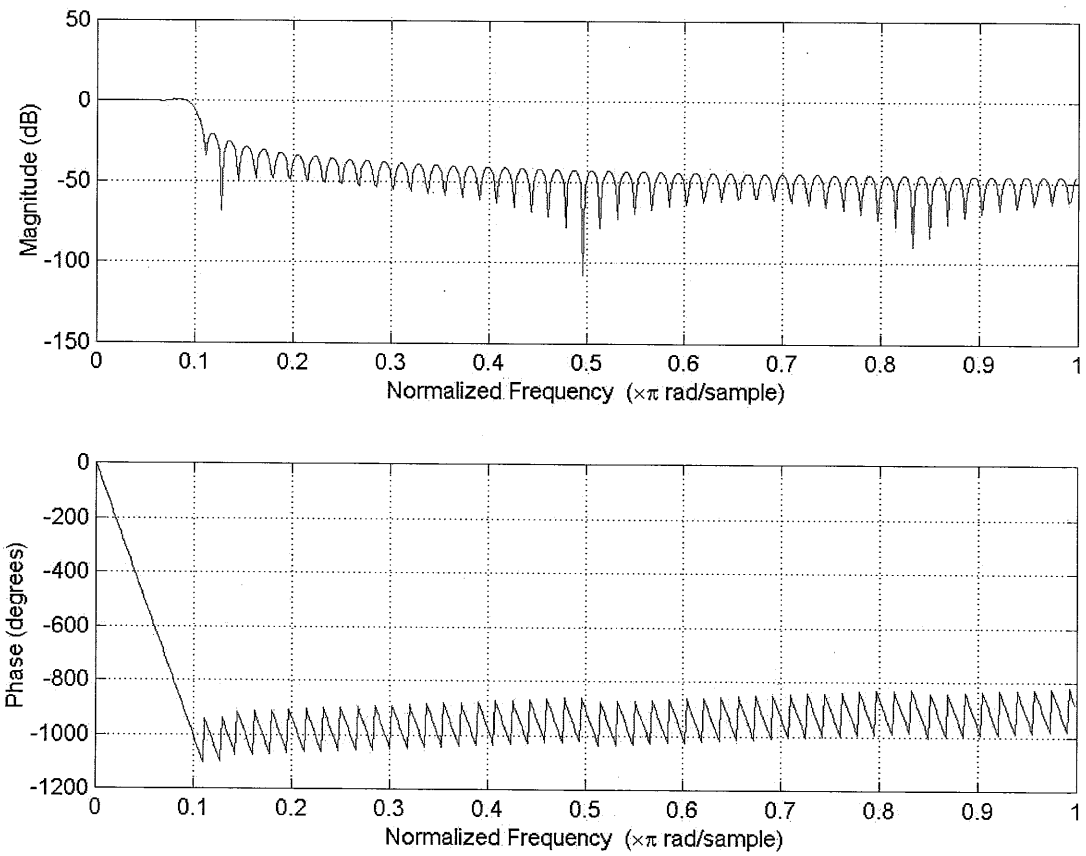
```
N=21;Wn=1/10;
B=fir1((N-1),Wn,boxcar(N),'noscale');
plot (B)
freqz
(B)
```





From the above plot we find that the filter response in the pass-band is not flat, which implies that the number of taps N must be increased. The taps were increased in the filter and it was found the at N=113 the filter response was flat in the pass-band. The response for the filter with taps N=113 is as given below


0.1
0.08
0.06
0.04
0.02
0
-0.02
-0.04
0
20
40
60
80
100
120


Magnitude (dB)
50
0
-50
-100
-150
0
0.1
0.2
0.3
0.4
0.5
0.6
0.7
0.8
0.9
1
Normalized Frequency (×π rad/sample)
Phase (degrees)
0
-200
-400
-600
-800
-1000
-1200
0
0.1
0.2
0.3
0.4
0.5
0.6
0.7
0.8
0.9
1
Normalized Frequency (×π rad/sample)

The code for the testing of the data on MATLAB and the plots of the input and output is given below

```
N=113;Wn=.1; % Define the Number of Taps &
B=fir1((N-1),Wn,boxcar(N),'noscale');
load data.txt; % Load the data
sum=0;  % Initialize sum
product=0;   % Initialize sum
%Calculate the output Y
for i=1:1:8000-N
   for j=1:1:N
      product=B(j)*data(i+j);
      sum=sum+product;
   end
   Y(i)=sum;
   sum=0;
end
%Plot the input and output
figure(1);subplot(2,1,1);plot(data);
figure(1);subplot(2,1,2);plot(Y);
```

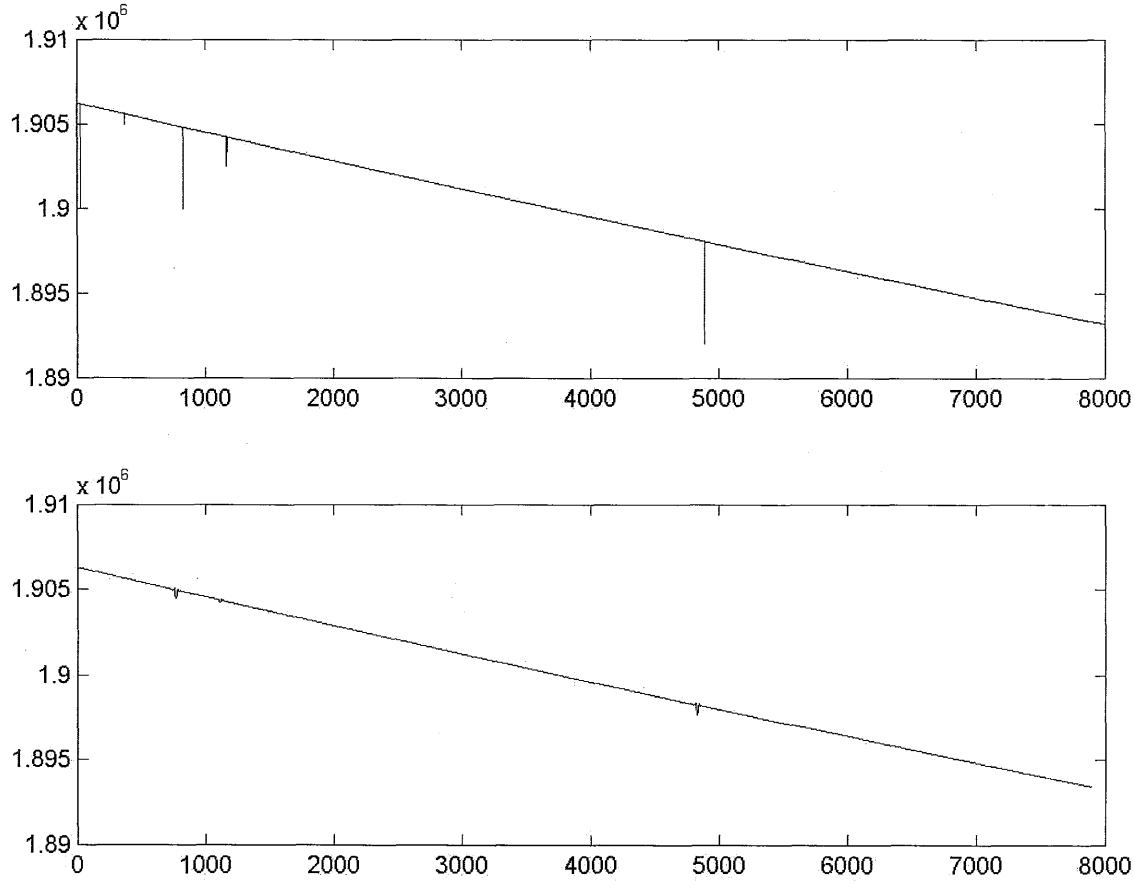


Fig: Input and output signals of the filter

The Fourier method of FIR filter design is based on the fact that the frequency response of a digital filter is periodic and is therefore representable as a Fourier series. A desired target frequency response is selected and expanded as Fourier series. This expansion is truncated into finite number of terms that are then used as the filter coefficients or tap weights. The resulting filter has a frequency response that approximates the original desired target response.

Design Steps :

Define the desired frequency response $H_d(\lambda)$

Specify the number of taps N

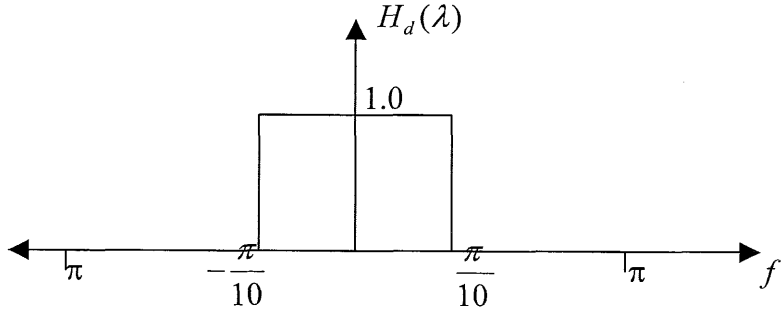
Compute the filter coefficients h[n] for n=0,1,2,3,4.....N-1 using

$$h[n]=\frac{1}{2\pi}\int_{2\pi}H_d(\lambda)[\cos(m\lambda)+j\sin(m\lambda)]d\lambda$$

where $m=\frac{N-1}{2}$

For 113 tap filter and 1Hz Cut-off and 10Hz Sampling rate and the desired frequency response

$H_d(\lambda)$



$$h[n]=\frac{1}{2\pi}\int_{-\pi/10}^{\pi/10}\cos(m\lambda)d\lambda+j\frac{1}{2\pi}\int_{-\pi/10}^{\pi/10}\sin(m\lambda)d\lambda$$

$$h[n] = \frac{\sin(m\pi/10)}{m\pi} \quad \text{where n=1,2,3.....(N-1) and } m = \frac{N-1}{2}$$

L'Hospital rule can be used to evaluate the above equation for the case of m=0 (that is n=56);

$$h[56] = \frac{(d/dm)\sin(m\pi/10)}{(d/dm)m\pi} \quad \text{for m=0}$$

The above method is used to calculate the 113 coefficients for the filter using C code

C Code and Test Results

```
#include<stdio.h>
#include<conio.h>
#include<math.h>
#include<io.h>
#define TAPS 113

void main()
{
   FILE *fir_out;
   FILE *filter_input;
   FILE *filter_output;
   float fir_output=0,product=0;
   long int i=0;
   long int k_process=0;
   long int j_output=0;
   float  input_data[8000];
   float n,m,cut_off_freq=1,samp_freq=10;
   float response[113],encoder_input;
  clrscr();
// Calculate the co-efficients
   fir_out=fopen("firout.dat","w");
   for(n=0;n<=(TAPS-1);n++)
   {
      m=n-(TAPS-1)/2;
      if(m==0)
      response[n]=(cut_off_freq/samp_freq);
       else
      response[n]=sin(m*M_PI*cut_off_freq/samp_freq)/(M_PI*m);
      fprintf(fir_out,"%f\n",response[n]);
     }
fclose(fir_out);

//Read the input data

      filter_input=fopen("qq1.txt","r");
      for(i=0;i<8000;i++)
      {
        fscanf(filter_input,"%f",&input_data[i]);
        // printf("%f\n",input_data[i]);
      }
fclose(filter_input);
```

```
// Processing Now

     filter_output=fopen("fir_out.txt", "w");
     for(k_process=0;k_process<8113;k_process++)
     {
          product=0;
          fir_output=0;
          for(i=0;i<(TAPS-1);i++)
          {
          product=(response[i]*input_data[i+k_process]);
          fir_output=fir_output+product;
          }
          fprintf(filter_output,"%f\n",fir_output);
     }
fclose(filter_output);
}
```

Result : The coefficients generated using the C-Code was found to be same as that of Co-efficients generated by the Matlab FIR function.

CHAPTER 4

ADSP 2106x AND EZ-KIT Lite

ADSP 2106x AND EZ-KIT Lite

Why DSP?

Digital signal processors are a special class of microprocessors that are optimized for computing the real- time calculations used in signal processing. Although it is possible to use some fast general- purpose microprocessors for signal processing, they are not optimized for that task. The resulting design can be hard to implement and costly to manufacture. In contrast, DSPs have an architecture that simplifies application designs and makes low- cost signal processing a reality.

The kinds of algorithms used in signal processing can be optimized if they are supported by a computer architecture specifically designed for them. In order to handle digital signal processing tasks efficiently, a microprocessor must have the following characteristics:

- fast, flexible computation units
- unconstrained data flow to and from the computation units
- extended precision and dynamic range in the computation units
- dual address generators
- efficient program sequencing and looping mechanisms

Why ADSP- 21000 Family?

The ADSP- 21020 and ADSP- 21060 are the first members of Analog Devices' ADSP- 21000 family of floating point digital signal processors (DSPs). The ADSP- 21000 family architecture meets the five central requirements for DSPs:

- Fast, flexible arithmetic computation units
- Unconstrained data flow to and from the computation units

- Extended precision and dynamic range in the computation units
- Dual address generators
- Efficient program sequencing and Fast & Flexible Arithmetic

The ADSP- 210xx can execute all instructions in a single cycle. It provides one of the fastest cycle times available and the most complete set of arithmetic operations, including Seed 1/ X, Seed 1/ R(x), Min, Max, Clip, Shift and Rotate, in addition to the traditional multiplication, addition, subtraction and combined addition/ subtraction. It is IEEE floating- point compatible and allows either interrupt on arithmetic exception or latched status exception handling.

Unconstrained Data Flow

The ADSP- 210xx has a Harvard architecture combined with a 10- port, 16 word data register file. In every cycle, all of these operations can be executed:

- the register file can read or write two operands off- chip
- the ALU can receive two operands
- the multiplier can receive two operands
- the ALU and multiplier can produce two results (three, if the ALU operation is a combined addition/ subtraction) The processors' 48- bit orthogonal instruction word supports parallel data transfer and arithmetic operations in the same instruction.

Extended IEEE- Floating- Point Support

All members of the ADSP- 21000 family handle 32- bit IEEE floating- point format, 32- bit integer and fractional formats (twos- complement and unsigned), and an extended- precision 40- bit IEEE floating- point format. These processors carry extended precision throughout their computation units, limiting intermediate data truncation errors. The fixed- point formats have an 80- bit accumulator for true 32- bit fixed- point computations.

Dual Address Generators

The ADSP- 210xx has two data address generators (DAGs) that provide immediate or indirect (pre- and post- modify) addressing. Modulus and bit- reverse operations are supported, without constraints on buffer placement.

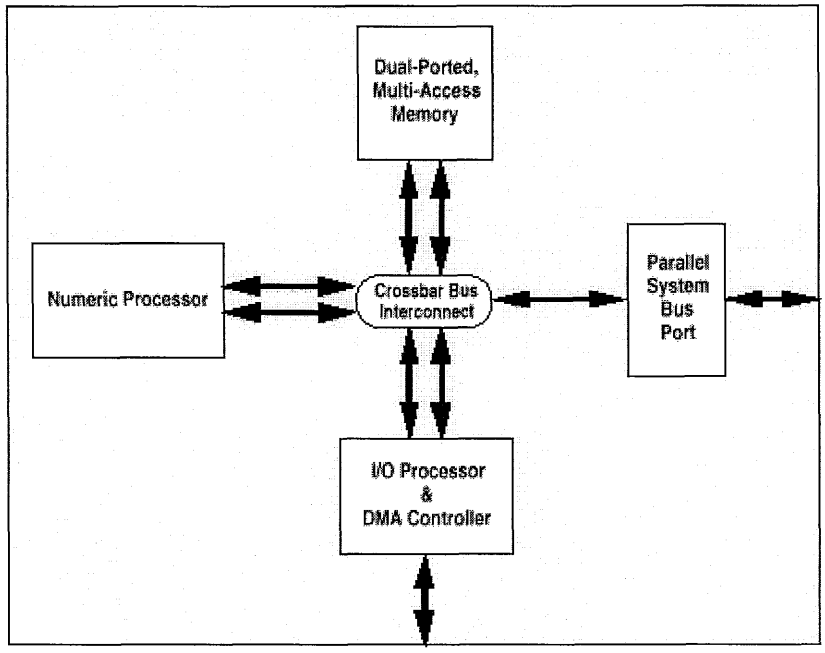
Efficient Program Sequencing

In addition to zero- overhead loops, the ADSP- 210xx supports single- cycle setup and exit for loops. Loops are nestable (six levels in hardware) and interruptable. The processor also supports delayed and non- delayed branches.

ADSP Architecture Details

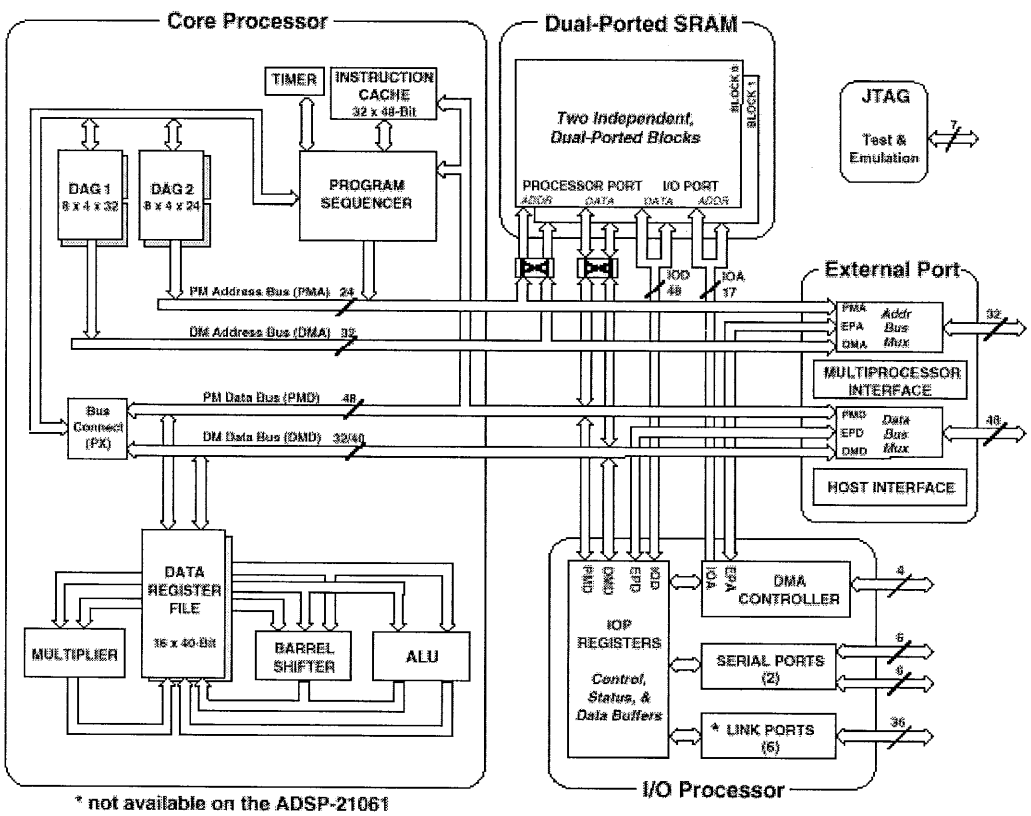
The ADSP- 2106x SHARC— Super Harvard Architecture Computer— is a high- performance 32- bit digital signal processor for speech, sound, graphics, and imaging applications. The SHARC builds on the ADSP- 21000 Family DSP core to form a complete system- on- a- chip, adding a dual- ported on- chip SRAM and integrated I/ O peripherals supported by a dedicated I/ O bus. With its on- chip instruction cache, the processor can execute every instruction in a single cycle. Four independent buses for dual data, instructions, and I/ O, plus crossbar switch memory connections, comprise the Super Harvard Architecture of the ADSP- 2106x.

The ADSP- 2106x SHARC represents a new standard of integration for digital signal processors, combining a high- performance floating- point DSP core with integrated, on- chip features including a host processor interface, DMA controller, serial ports, and link port and shared bus connectivity for glueless DSP multiprocessing.



Super Harvard Architecture

Figure illustrates the Super Harvard Architecture of the ADSP- 2106x: a crossbar bus switch connecting the core numeric processor to an independent I/ O processor, dual- ported memory, and parallel system bus port.



ADSP-2106x SHARC Block Diagram

Figure shows a detailed block diagram of the processor, illustrating the following architectural features:

32- Bit IEEE Floating- Point Computation Units— Multiplier, ALU, and Shifter

- Data Register File
- Data Address Generators (DAG1, DAG2)
- Program Sequencer with Instruction Cache
- Interval Timer ▒ Dual- Ported SRAM
- External Port for Interfacing to Off- Chip Memory & Peripherals
- Host Port & Multiprocessor Interface
- DMA Controller
- Serial Ports
- Link Ports
- JTAG Test Access Port

Figure also shows the three on- chip buses of the ADSP- 2106x: the PM bus (program memory), DM bus (data memory), and I/ O bus. The PM bus is used to access either instructions or data. During a single cycle the processor can access two data operands, one over the PM bus and one over the DM bus, an instruction (from the cache), and perform a DMA transfer.

The ADSP- 2106x's external port provides the processor's interface to external memory, memory-mapped I/ O, a host processor, and additional multiprocessing ADSP- 2106xs. The external port performs internal and external bus arbitration as well as supplying control signals to shared, global memory and I/ O devices.

The following sections summarize the features of the ADSP- 2106x SHARC architecture. These features are described in greater detail in succeeding sections.

Core Processor

The core processor of the ADSP- 2106x consists of three computation units, a program sequencer, two data address generators, timer, instruction cache, and data register file.

Computation Units

The ADSP- 2106x core processor contains three independent computation units: an ALU, a multiplier with a fixed- point accumulator, and a shifter. For meeting a wide variety of processing needs, the computation units process data in three formats: 32- bit fixed- point, 32- bit floating-point and 40- bit floating- point. The floatingpoint operations are single- precision IEEE-compatible. The 32- bit floating- point format is the standard IEEE format, whereas the 40- bit IEEE extended- precision format has eight additional LSBs of mantissa for greater accuracy.

The ALU performs a standard set of arithmetic and logic operations in both fixed- point and floating- point formats. The multiplier performs floating- point and fixed- point multiplication as well as fixed- point multiply/ add and multiply/ subtract operations. The shifter performs logical and arithmetic shifts, bit manipulation, field deposit and extraction and exponent derivation operations on 32- bit operands.

The computation units perform single- cycle operations; there is no computation pipeline. The units are connected in parallel rather than serially. The output of any unit may be the input of any unit on the next cycle. In a multifunction computation, the ALU and multiplier perform independent, simultaneous operations.

Data Register File

A general- purpose data register file is used for transferring data between the computation units and the data buses, and for storing intermediate results. The register file has two sets (primary and alternate) of sixteen registers each, for fast context switching. All of the registers are 40 bits wide. The register file, combined with the core processor's Harvard architecture, allows unconstrained data flow between computation units and internal memory.

Two dedicated address generators and a program sequencer supply addresses for memory accesses. Together the sequencer and data address generators allow computational operations to execute with maximum efficiency since the computation units can be devoted exclusively to processing data. With its instruction cache, the ADSP- 2106x can simultaneously fetch an instruction (from the cache) and access two data operands (from memory). The data address generators implement circular data buffers in hardware.

The program sequencer supplies instruction addresses to program memory. It controls loop iterations and evaluates conditional instructions. With an internal loop counter and loop stack, the ADSP- 2106x executes looped code with zero overhead. No explicit jump instructions are required to loop or to decrement and test the counter.

The ADSP- 2106x achieves its fast execution rate by means of pipelined fetch, decode and execute cycles. If external memories are used, they are allowed more time to complete an access than if there were no decode cycle.

The data address generators (DAGs) provide memory addresses when data is transferred between memory and registers. Dual data address generators enable the processor to output simultaneous addresses for two operand reads or writes. DAG1 supplies 32- bit addresses to data memory. DAG2 supplies 24- bit addresses to program memory for program memory data accesses.

Each DAG keeps track of up to eight address pointers, eight modifiers and eight length values. A pointer used for indirect addressing can be modified by a value in a specified register, either before (pre- modify) or after (post- modify) the access. A length value may be associated with each pointer to perform automatic modulo addressing for circular data buffers; the circular buffers can be located at arbitrary boundaries in memory. Each DAG register has an alternate register that can be activated for fast context switching.

Circular buffers allow efficient implementation of delay lines and other data structures required in digital signal processing, and are commonly used in digital filters and Fourier transforms. The DAGs automatically handle address pointer wraparound, reducing overhead, increasing performance, and simplifying implementation.

Instruction Cache

The program sequencer includes a 32- word instruction cache that enables three- bus operation for fetching an instruction and two data values. The cache is selective— only instructions whose fetches conflict with program memory data accesses are cached. This allows full- speed execution of core, looped operations such as digital filter multiply- accumulates and FFT butterfly processing.

Interrupts

The ADSP- 2106x has four external hardware interrupts: three general- purpose interrupts, IRQ2- 0 , and a special interrupt for reset. The processor also has internally generated interrupts for the timer, DMA controller operations, circular buffer overflow, stack overflows, arithmetic exceptions, multiprocessor vector interrupts, and user- defined software interrupts.

For the general- purpose external interrupts and the internal timer interrupt, the ADSP- 2106x automatically stacks the arithmetic status and mode (MODE1) registers in parallel with the interrupt servicing, allowing four nesting levels of very fast service for these interrupts.

Timer

The programmable interval timer provides periodic interrupt generation. When enabled, the timer decrements a 32- bit count register every cycle. When this count register reaches zero, the ADSP- 2106x generates an interrupt and asserts its TIMEXP output. The count register is automatically reloaded from a 32- bit period register and the count resumes immediately.

Core Processor Buses: The processor core has four buses: Program Memory Address, Data Memory Address, Program Memory Data, and Data Memory Data. On the ADSP- 2106x processors, data memory stores data operands while program memory is used to store both instructions and data (filter coefficients, for example)— this allows dual data fetches, when the instruction is supplied by the cache.

The PM Address bus and DM Address bus are used to transfer the addresses for instructions and data. The PM Data bus and DM Data bus are used to transfer the data or instructions stored in each type of memory. The PM Address bus is 24 bits wide allowing access of up to 16M words of mixed instructions and data. The PM Data bus is 48 bits wide to accommodate the 48- bit instruction width. Fixed- point and single- precision floating- point data is aligned to the upper 32 bits of the PM Data bus.

The DM Address bus is 32 bits wide allowing direct access of up to 4G words of data. The DM Data bus is 40 bits wide. Fixed- point and single- precision floating- point data is aligned to the upper 32 bits of the DM Data bus. The DM Data bus provides a path for the contents of any register in the processor to be transferred to any other register or to any data memory location in

a single cycle. The data memory address comes from one of two sources: an absolute value specified in the instruction code (direct addressing) or the output of a data address generator (indirect addressing).

Internal Data Transfers Nearly every register in the core processor of the ADSP- 2106x is classified as a universal register. Instructions are provided for transferring data between any two universal registers or between a universal register and memory. This includes control registers and status registers, as well as the data registers in the register file.

The PX bus connect registers permit data to be passed between the 48- bit PM Data bus and the 40- bit DM Data bus or between the 40- bit register file and the PM Data bus. These registers contain hardware to handle the 8- bit width difference.

Context Switching: Many of the processor's registers have alternate registers that can be activated during interrupt servicing to facilitate a fast context switch. The data registers in the register file, the DAG registers, and the multiplier result register all have alternates. Registers active at reset are called primary registers, while the others are called alternate (or secondary) registers. Control bits in a mode control register determine which set of registers is active at any particular time.

Instruction Set The ADSP- 21000 Family instruction set provides a wide variety of programming capabilities. Multifunction instructions enable computations in parallel with data transfers, as well as simultaneous multiplier and ALU operations. The addressing power of the ADSP- 2106x gives you flexibility in moving data both internally and externally. Every instruction can be executed in a single processor cycle. The ADSP- 21000 Family assembly language uses an algebraic syntax for ease of coding and readability. A comprehensive set of development tools supports program development.

Dual- Ported Internal Memory

The ADSP- 21060 contains 4 megabits of on- chip SRAM, organized as two blocks of 2 Mbits each, which can be configured for different combinations of code and data storage. The ADSP- 21062 includes a 2 Mbit SRAM, organized as two 1 Mbit blocks. Each memory block is dual- ported for single- cycle, independent accesses by the core processor and I/ O processor or DMA controller. The dual- ported memory and separate on- chip buses allow two data transfers from the core and one from I/ O, all in a single cycle.

All of the memory can be accessed as 16- bit, 32- bit, or 48- bit words. On the ADSP-21060, the memory can be configured as a maximum of 128K words of 32- bit data, 256K words of 16- bit data, 80K words of 48- bit instructions (and 40- bit data), or combinations of different word sizes up to 4 megabits. On the ADSP- 21062, the memory can be configured as a maximum of 64K words of 32- bit data, 128K words of 16- bit data, 40K words of 48- bit instructions (and 40- bit data), or combinations of different word sizes up to 2 megabits. On the ADSP21061, the memory can be configured as a maximum of 32K words of 32- bit data, 64K words of 16- bit data, 16K words of 48- bit instructions (and 40- bit data), or combinations of different word sizes up to 1 megabit.

A 16- bit floating- point storage format is supported which effectively doubles the amount of data that may be stored on chip. Conversion between the 32- bit floating- point and 16- bit floating- point formats is done in a single instruction.

While each memory block can store combinations of code and data, accesses are most efficient when one block stores data, using the DM bus for transfers, and the other block stores instructions and data, using the PM bus for transfers. Using the DM bus and PM bus in this way, with one dedicated to each memory block, assures single- cycle execution with two data transfers.

In this case, the instruction must be available in the cache. Single- cycle execution is also maintained when one of the data operands is transferred to or from off- chip, via the ADSP-2106x's external port.

External Memory & Peripherals Interface

The ADSP- 2106x's external port provides the processor's interface to off- chip memory and peripherals. The 4- gigaword off- chip address space is included in the ADSP- 2106x's unified address space. The separate on- chip buses— for PM addresses, PM data, DM addresses, DM data, I/ O addresses, and I/ O data— are multiplexed at the external port to create an external system bus with a single 32- bit address bus and a single 48- bit data bus. External SRAM can be either 16, 32, or 48 bits wide; the ADSP- 2106x's on- chip DMA controller automatically packs external data into the appropriate word width, either 48- bit instructions or 32- bit data.

Addressing of external memory devices is facilitated by on- chip decoding of high- order address lines to generate memory bank select signals. Separate control lines are also generated for simplified addressing of page- mode DRAM. The ADSP- 2106x provides programmable memory wait states and external memory acknowledge controls to allow interfacing to DRAM and peripherals with variable access, hold, and disable time requirements.

Host Processor Interface

The ADSP- 2106x's host interface allows easy connection to standard microprocessor buses, both 16- bit and 32- bit, with little additional hardware required. Asynchronous transfers at speeds up to the full clock rate of the ADSP- 2106x are supported. The host interface is accessed through the ADSP- 2106x's external port and is memorymapped into the unified address space. Four channels of DMA are available for the host interface; code and data transfers are accomplished with low software overhead. The host can directly read and write the internal

memory of the ADSP- 2106x, and can access the DMA channel setup and mailbox registers. Vector interrupt support is provided for efficient execution of host commands.

Multiprocessing

The ADSP- 2106x offers powerful features tailored to multiprocessing DSP systems. The unified address space allows direct interprocessor accesses of each ADSP- 2106x's internal memory. Distributed bus arbitration logic is included on- chip for simple, glueless connection of systems containing up to six ADSP- 2106xs and a host processor. Master processor changeover incurs only one cycle of overhead. Bus arbitration is selectable as either fixed or rotating priority. Processor bus lock allows indivisible read- modify- write sequences for semaphores. A vector interrupt capability is provided for interprocessor commands. Maximum throughput for interprocessor data transfer is 240 Mbytes/ sec over the link ports or external port. Broadcast writes allow simultaneous transmission of data to all ADSP- 2106xs and can be used to implement reflective semaphores.

Serial Ports

The ADSP- 2106x features two synchronous serial ports that provide an inexpensive interface to a wide variety of digital and mixed- signal peripheral devices. The serial ports can operate at the full clock rate of the processor, providing each with a maximum data rate of 40 Mbit/ s. Independent transmit and receive functions provide greater flexibility for serial communications. Serial port data can be automatically transferred to and from on- chip memory via DMA. Each of the serial ports offers a TDM multichannel mode.

The serial ports can operate with little- endian or big- endian transmission formats, with word lengths selectable from 3 to 32 bits. They offer selectable synchronization and transmit modes as well as optional m- law or A- law companding. Serial port clocks and frame syncs can be internally or externally generated.

The ADSP- 21062 and ADSP- 21060 feature six 4- bit link ports that provide additional I/ O capabilities. The link ports can be clocked twice per cycle, allowing each to transfer 8 bits per cycle. Link port I/ O is especially useful for point- to- point interprocessor communication in multiprocessing systems.

The link ports can operate independently and simultaneously, with a maximum data throughput of 240 Mbytes/ s. Link port data is packed into 32- bit or 48- bit words, and can be directly read by the core processor or DMA- transferred to on- chip memory. Each link port has its own double- buffered input and output registers. Clock/ acknowledge handshaking controls link port transfers. Transfers are programmable as either transmit or receive.

There are no link ports on the ADSP- 21061.

DMA Controller

The ADSP- 2106x's on- chip DMA controller allows zero- overhead data transfers without processor intervention. The DMA controller operates independently and invisibly to the processor core, allowing DMA operations to occur while the core is simultaneously executing its program. Both code and data can be downloaded to the ADSP- 2106x using DMA transfers.

DMA transfers can occur between the ADSP- 2106x's internal memory and external memory, external peripherals, or a host processor. DMA transfers can also occur between the ADSP- 2106x's internal memory and its serial ports or link ports. DMA transfers between external memory and external peripheral devices are another option. External bus packing to 16, 32, or 48- bit words is automatically performed during DMA transfers.

Ten channels of DMA are available on the ADSP- 21060 and ADSP- 21062— two via the link ports, four via the serial ports, and four via the processor's external port (for either host processor, other ADSP- 2106xs, memory or I/ O transfers). Four additional link port DMA channels are shared with serial port 1 and the external port. There are six channels of DMA

available on the ADSP- 21061— four via the serial ports and two via the external port. Asynchronous off- chip peripherals can control two DMA channels using DMA Request/ Grant lines (DMAR1- 2 , DMAG1- 2). Other DMA features include interrupt generation upon completion of DMA transfers and DMA chaining for automatic linked DMA transfers.

SHARC EZ- KIT Lite

SHARC EZ- KIT Lite™ is one of the best values in development systems available today. The Analog Devices ADSP- 21061 processor used in the SHARC EZ- KIT Lite has many features integrated onto a single chip:

- The industry's fastest general- purpose 32- bit single- precision (or 40- bit extended precision) IEEE floating- point and 32- bit fixed- point DSP core with three independent, parallel computational units: ALU, multiplier, and shifter (40- MIPS, with 120 MFLOPS peak, 80 MFLOPS sustained)
- On- chip, configurable memory banks: dual- ported 1- megabit internal SRAM for fast, independent local memory access for DSP core, DMA controller and I/ O processor
- Two 40 Mbit/ s synchronous serial ports
- A sophisticated DMA controller (6 simultaneous channels with zero impact on performance of DSP core)

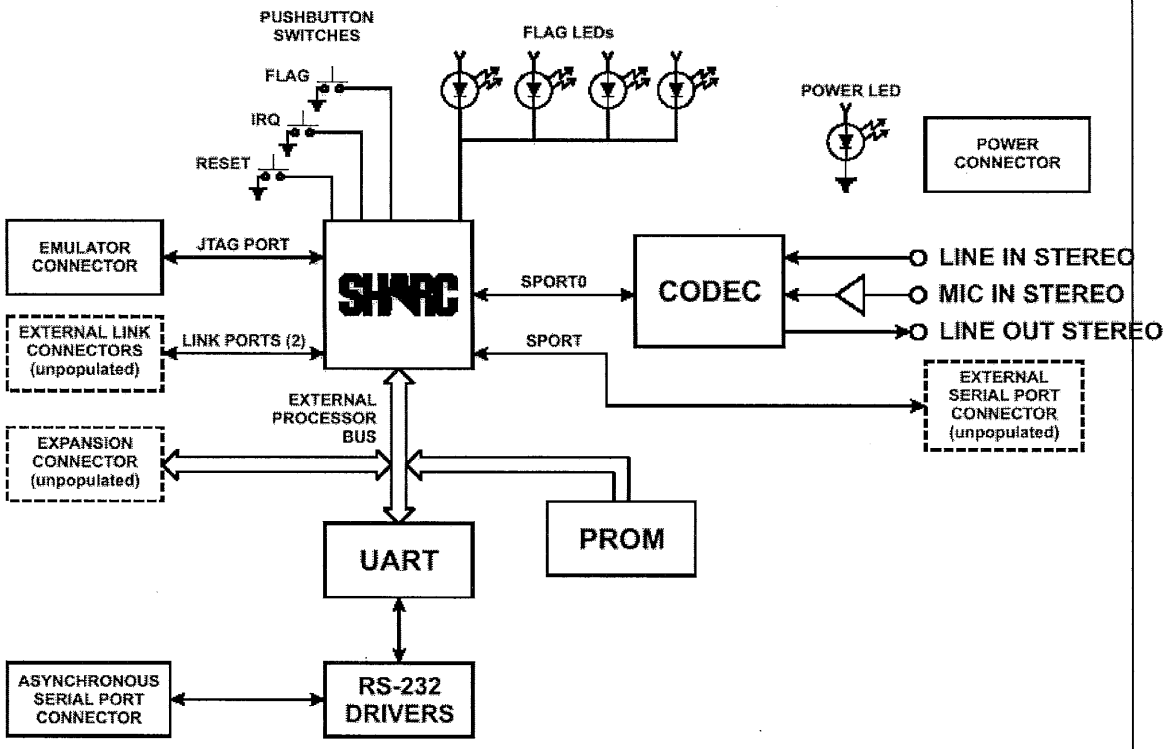
The SHARC EZ- KIT Lite provides an easy way for you to investigate the power of the SHARC family of processors and develop your own applications based on these high- performance DSPs. It is a complete development system package with a price that makes it ideal for getting started in DSP. The SHARC EZ- KIT Lite was designed to help you accomplish these goals:

- Evaluate Analog Devices' floating- point DSPs
- Learn about DSP applications
- Develop DSP applications

- Simulate and debug your application
- Prototype new applications

The SHARC EZ- KIT Lite consist of a small ADSP- 21061 based development / demonstration board with full 16- bit stereo audio I/ O capabilities. The board's features include:

- Analog Devices ADSP- 21061 DSP running at 40 MHz
- Analog Devices AD1847 16- bit Stereo SoundPort® Codec
- RS- 232 interface
- Socketed EPROM
- User push- buttons
- User programmable LEDs
- Power supply regulation
- Expansion connectors: The board can run standalone or can simply connect to the RS- 232 port of your PC. A monitor program running on the DSP in conjunction with a host program running on the PC lets you interactively download programs as well as interrogate the ADSP- 21061. The board comes with a socketed EPROM so that you can run the monitor program and demonstrations provided or you can plug in an EPROM containing you own code.



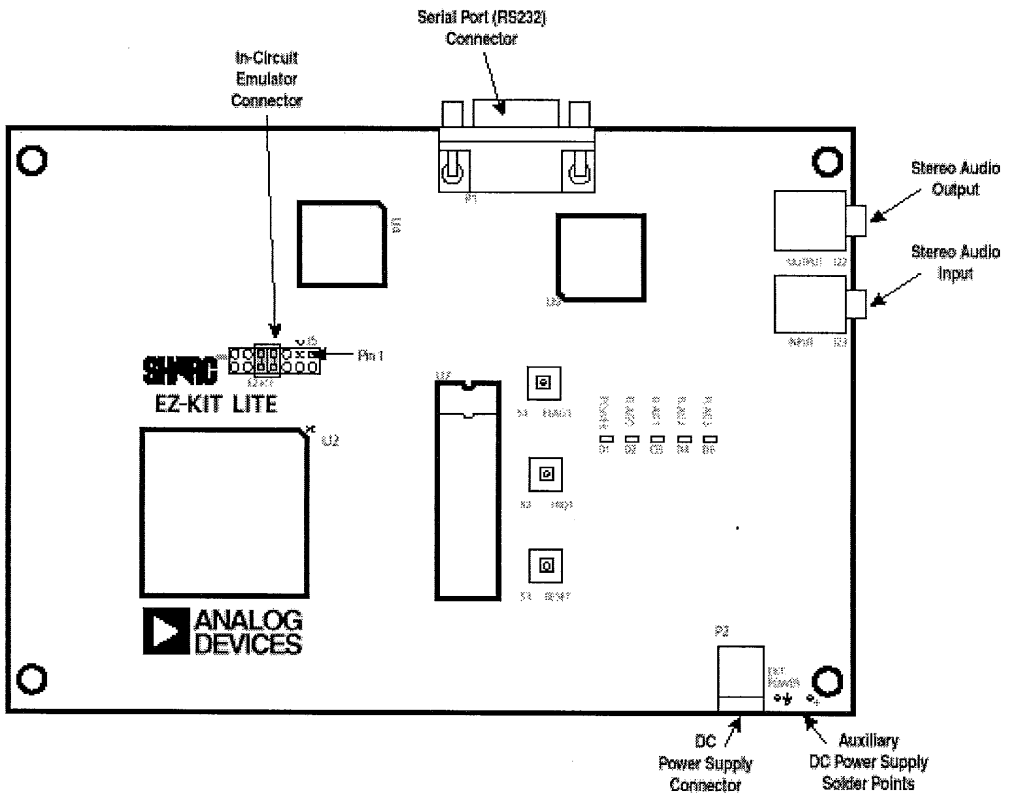
SHARC EZ-KIT Lite System Block Diagram

Figure is a block diagram of the SHARC EZ- KIT Lite system. One can access the ADSP 21061 SHARC processor from the PC through the RS- 232 interface. The boot PROM provides code for execution when the SHARC EZ- KIT Lite is operating in standalone mode and loads a kernel that manages the RS- 232 interface.

The SHARC EZ- KIT Lite board has several sites for connectors that allow you to expand the board's capabilities. These sites are not populated with connectors when you receive the board from Analog Devices.

HARDWARE CONNECTIONS

Figure highlights the external hardware connections to the SHARC EZ- KIT Lite. The following sections describe each of the connections.



Hardware Connections to the SHARC EZ-KIT Lite

Serial Port (RS- 232) Connector

P1 is female 9- pin D- Sub connector. It is used to communicate with a host computer using RS-232 signal levels and asynchronous serial protocols. You can connect this to your PC using the supplied cable. The cable provides a straight through connection from the DCE port on the EZ-KIT Lite to the DTE port on your PC. The DCD, DTR, and DSR signals are connected on the EZ-KIT Lite circuit board.

1 DCD 2 Transmit Data (output) 3 Receive Data (input) 4 DTR 5 Signal Ground 6 DSR 7 Request to Send (input) 8 Clear to Send (output) 9 Not Connected

Stereo Audio Output

The Stereo Audio Output jack connects to the left (L) and right ® LINE OUTPUT pins of the AD1847 codec. You can use standard audio cables with 1/ 8 inch (3.5mm) stereo plugs to connect these signals to a set of amplified speakers.

Stereo Audio Input

The Stereo Audio Input jack connects directly to the left (L) and right ® LINE 1 INPUT pins of the AD1847 codec. You can use standard audio cables with 1/ 8 inch (3.5mm) stereo plugs to supply these inputs with line- level signals. You can also connect a microphone level signal by changing the Input Source Selector jumpers (see 5.3.1).

DC Power Supply Connector

The power supply connector is used to supply DC voltages to the SHARC EZKIT Lite board. The DC power supply included with your board should mate directly to this connector.

DEVELOPING APPLICATION

If one likes to develop the own DSP programs, one can use the software development tools provided with the SHARC EZ- KIT Lite. If one have limited experience in developing code for a DSP- based system, one should review the steps in this chapter.

The following development steps serve as a guideline for creating the own programs. Keep in mind that the development process varies depending upon the style of the particular developer. Follow this guideline as a starting point and feel free to modify it to suit the own work style. In many cases one will be able to skip a step because of the components shipped with EZ- KIT Lite. For example, the hardware architecture description process is described below, yet it is not necessary to follow this step since an architecture file for the ADSP- 21061 development board is included in the EZ- KIT Lite package. The following sections explain these topics in more detail. All commands that are mentioned throughout the following sections are to be typed at the DOS prompt (C:\). One should check that the Analog Devices software development tools are installed and that the executable binaries directory (usually c:\ ez- kit\ bin) is listed in the PATH variable.

DEFINING THE SYSTEM

Before one can start developing the own application, one must define what it will do and how one can best use the available hardware resources. The following sections will take one through these steps.

Step 1: System Requirements

Step 2: System Design

Step 3: Architecture Description File

Architecture File

```
!-------------------------------------------------------------
.SYSTEM SHARC_ EZKIT_ Lite;
! ! This architecture file is required for use with the SHARC EZ- KIT
! Lite development software. It is structured for use with the C compiler but also can be used with
!assembly code.
! This architecture file allocates: Internal 133 words of 48- bit run- time header in memory block 0
! 16 words of 48- bit initialization code in memory block 0 619 words of 48- bit kernel !code in
!memory block 0 ! 7424 words of 48- bit C code space in memory block 0
! 4K words of 32- bit PM C data space in memory block 0
! ! 8K words of 32- bit C DM data space in memory block 1
! 4K words of 32- bit C heap space in memory block 1 ! 3712 words of 32- bit C stack space in
memory block 1
! 384 words of 32- bit kernel data in memory block 1
.PROCESSOR = ADSP21061; ! ----------------------------------------------------------
! Internal memory Block 0
 ! ----------------------------------------------------------
.SEGMENT/ RAM/ BEGIN= 0x00020000 /END= 0x00020084 /PM/ WIDTH= 48 seg_ rth;
.SEGMENT/ RAM/ BEGIN= 0x00020085 /END= 0x00020094 /PM/ WIDTH= 48 seg_ init;
.SEGMENT/ RAM/ BEGIN= 0x00020095 /END= 0x000202ff /PM/ WIDTH= 48 seg_ knlc;
.SEGMENT/ RAM/ BEGIN= 0x00020300 /END= 0x00021fff /PM/ WIDTH= 48 seg_ pmco;
.SEGMENT/ RAM/ BEGIN= 0x00023000 /END= 0x00023fff /PM/ WIDTH= 32 seg_ pmda;
! ----------------------------------------------------------
! Internal memory Block 1
! ----------------------------------------------------------
.SEGMENT/ RAM/ BEGIN= 0x00024000 /END= 0x00025fff /DM/ WIDTH= 32 seg_ dmda;
.SEGMENT/ RAM/ BEGIN= 0x00026000 /END= 0x00026fff /DM/ WIDTH= 32 /cheap seg_
heap;
.SEGMENT/ RAM/ BEGIN= 0x00027000 /END= 0x00027e7f /DM/ WIDTH= 32 seg_ stak;
.SEGMENT/ RAM/ BEGIN= 0x00027e80 /END= 0x00027fff /DM/ WIDTH= 32 seg_ knld;
! ----------------------------------------------------------
! External Memory Select 1 is reserved for the UART.
! ----------------------------------------------------------
.ENDSYS;
```

This architecture file assumes that one will be using the EZ- KIT Lite's built in kernel to download and start the program. The kernel uses certain regions of memory (seg_ krnlc and seg_ krnld) that the program should avoid. The remaining memory segments are available to the program. The development tools included with the SHARC EZ- KIT Lite will always use this architecture file.

One must buy the complete version of the ADSP- 21000 Family Software Development Tools to specify a different architecture file.

Step 4: Writing Source Code

Step 5: Running The Compiler or Assembler

Step 6: Running The Linker

Step 7: Running The Simulator

Step 8: Running on the SHARC EZ- KIT Lite Board

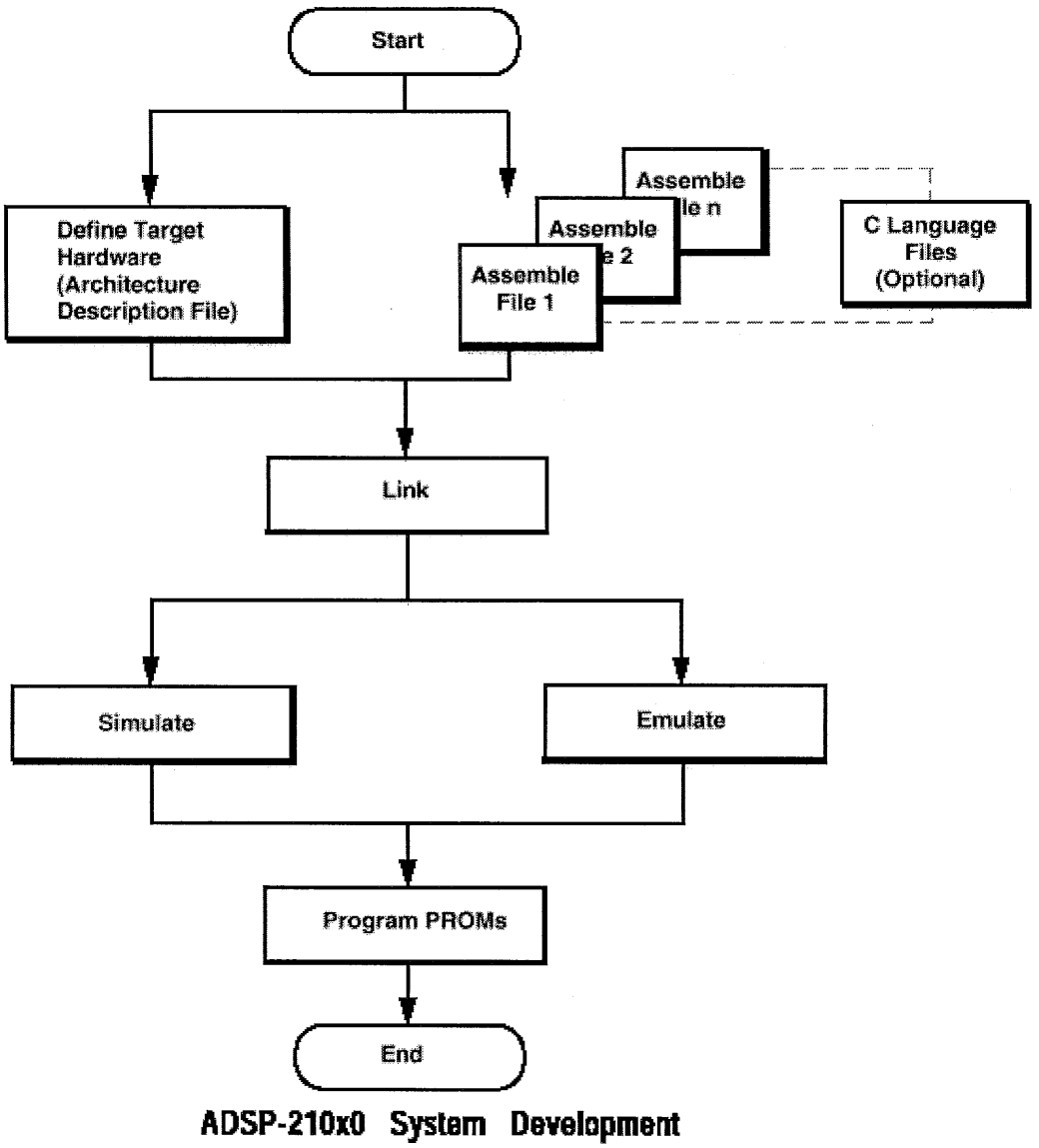
Once one install the host software onto the PC one should be able to run the host program under Windows. One can then download an executable file (. exe file produced by the linker). Programs that one write must conform to certain restrictions to ensure that the program does not interfere with the operation of the kernel.

If one prefer, one can program the own EPROM and insert it into the EPROM socket on the EZ- KIT Lite board to run the own program in stand alone mode.

Once one have verified that the software works one can format the executable so that it can be programmed into an EPROM. The EPROM can then be inserted into the EPROM socket on the board to run the program. One can invoke the boot loader tool with the following command:

ldr21k -bprom -o demo. ihx -id1exe= demo. exe

This will take the executable file demo. exe and create a PROM file called demo. ihx. The PROM format is the Intel Hex record format. This file can be downloaded to a PROM programmer to program an EPROM which can be inserted into the EPROM socket on the SHARC EZ- KIT Lite board. Upon power up reset or when one hit the reset button on the board, the contents of the EPROM are automatically loaded into the internal program and data memories of the ADSP-21061 and coded execution begins.



ADSP-210x0 System Development

CHAPTER 5

ADSP Programming

and

Simulation on

EZ-KIT Lite

ADSP Programming and Simulation on EZ-KIT Lite

This chapter gives the technical details that is neeed for writing programs for the SHARC EZ- KIT Lite. One section focuses on memory models, addresses, and other resources for programs that run on the SHARC processor.

Memory Map

The SHARC memory model defines three main memory spaces. Internal memory space addresses a SHARC processor's own on- chip dual- ported SRAM.

Multiprocessor memory space addresses the on- chip SRAM of other SHARC processors in the same cluster (i. e., SHARC processors that share a common processor bus). External memory space addresses other devices on the shared bus such as SRAM or DRAM.

Internal Memory Space

Since the SHARC EZ- KIT Lite has no external memory, one must store all code instructions and data in the built- in SRAM. The ADSP- 21061 processor has one megabit of internal dual- ported SRAM that is divided into two 512- kilobit blocks. The blocks are designed so that one can configure regions of memory to be either 32 or 48 bits wide.

If one is writing programs that will be loaded by the built- in kernel, one should be aware of how it uses memory. The architecture file for the EZ- KIT Lite (ezkit. ach) defines memory segments that are compatible with programs one write for the C compiler and the assembler. Table 6- 2 lists the internal memory regions that are defined in the architecture file.

Table : Internal Memory Map

Segment Name	Description	Memory Block	Length	Width
Start Address	End Address			
seg_ rth	Run- time header	0	133	48
0x00020000	0x00020084			
seg_ init	Initialization data	0	16	48
0x00020085	0x00020094			
seg_ knlc	Kernel Code	0	619	48
0x00020095	0x000202FF			
seg_ pmco	User Code	0	7424	48
0x00020300	0x00021FFF			
seg_ pmda	User PM Data	0	4096	32
0x00023000	0x00023FFF			
seg_ dmda	User DM Data	1	8192	32
0x00024000	0x00025FFF			
seg_ heap	User C Heap	1	4096	32
0x00026000	0x00026FFF			
seg_ stak	User C Stack	1	3712	32
0x00027000	0x00027E7F			
seg_ knld	Kernel Data	1	384	32
0x00027E80	0x00027FFF			

Multiprocessor Memory Space

The multiprocessor memory space (MMS) is consumed by any and all SHARC processors that are connected to the external processor bus (up to six as defined by the maximum cluster size). SHARC processors appear in specific portions of the MMS according to their multiprocessor ID. The default multiprocessor ID is one (001) for the EZ- KIT Lite processor. One can change the ID to zero (000) with a jumper setting (see 5.3.2).

Since the SHARC EZ- KIT Lite is a single- processor board, the only way to view other SHARC processors through the MMS is through the expansion connectors to the custom hardware.

External Memory Space

External memory space is further divided into four banked sections of memory and an non- banked section of memory. Each of the four banked sections (numbered 0 to 3) are sized the same and are defined by the MSIZE bits of the SHARC processor's SYSCON register (refer to the ADSP- 2106x SHARC User's Manual for more details). Bank 0 starts at 0x40 0000 and the bank size determines the starting address of each of the other banks. Non- banked memory space starts after Bank 3 and covers the remainder of external memory space (up to 0xFFFF FFFF).

Kernel Compatibility

When one write programs that run on the EZ- KIT Lite board's SHARC processor, there are certain programming restrictions one should consider to ensure that the on- board kernel program will continue to operate. If one violate any of these restrictions, the kernel may become disabled or unstable. If this happens, one must reset the EZ- KIT Lite to reload the kernel from the boot EPROM.

Avoid using kernel memory regions. The kernel uses two regions of the SHARC's internal SRAM.

These regions are defined as separate segments (seg_ knlc and seg_ knld) in the EZ- KIT Lite architecture description file (see Table 6- 2). If the program is written entirely in C, the linker will automatically use the appropriate memory segments.

Avoid the UART. Since the kernel uses the RS- 232 interface to communicate with the host computer, it must manage the UART chip. The UART is accessed in external memory space within external memory bank 1 (see Table 6- 4). The program can freely change the MSIZE setting in the SYSCON register (potentially changing the base address of external memory bank 1) since the kernel recalculates the UART base address every time it needs to access the chip.

Don't interfere with the UART interrupt. The kernel depends on the hardware interrupt signal IRQ2 to communicate with the UART. There are several ways that the program can affect the kernel's ability to receive interrupts from the UART:

Do not disable interrupts globally. Bit 12 (IRPTEN) in the MODE1 register must remain set.
Do not mask IRQ2. The IMASK register allows the program to enable or disable individual interrupts. Bit 6 (IRQ2I) in the IMASK register must remain set.
Do not change IRQ2's sensitivity. The UART uses the interrupt signal to indicate multiple conditions. If the kernel satisfies one condition, there may be other conditions requiring attention; however, the IRQ2 signal will not transition to the inactive state and then back to the active state. Therefore, the SHARC's interrupt controller must treat IRQ2 as a levelsensitive signal. Bit 2 (IRQ2E) in the MODE2 register must remain cleared.

Implementation

An FIR filter is a weighted sum of a finite set of inputs. The equation for an FIR filter is

$$y(n) = \sum_{k=0}^{m-1} akx(n-k)$$

where

x(n – k) is a previous history of inputs

y(n) is the filter output at time n ak is a vector of filter coefficients

The FIR code is a software implementation of this equation. FIR filtering is a convolution in time. The FIR filter equation is similar to the convolution equation:

$$y(n) = \sum h(k)\, x(n-k) \quad k = 0$$

The FIR filtering code uses predefined input samples stored in a buffer and coefficients that are stored in a data file. The code executes a 113- tap filter on predefined samples. After the program processes the last sample it enters an infinite loop.

The code for the FIR filter consists of two assembly- language modules,

FIRMAIN. asm and fir. asm . The FIRMAIN. asm module sets up buffers and pointers and calls fir. asm , the module that performs the multiply- accumulate operations. Two other files are needed: an architecture file and a data file. The file generic. ach is the architecture file, which defines the hardware in terms of memory spaces and peripherals. The fircoefs.dat file contains a list of filter coefficients for the filter.

The code in FIRMAIN. asm is executed first. The FIRMAIN.asm code starts by defining the constants TAPS and SAMPLES .

- TAPS is the number of taps of the filter. To change the number of taps, change the value of TAPS and add or delete coefficients to the file FIRCOEFS.DAT.

- SAMPLES is the constant that represents the number of samples that are input into the filter. If one need to change the number of samples, change SAMPLES and add or delete the number of predefined samples to the input buffer.

The next part of FIRMAIN.asm defines four buffers, COEFS , DLINE , INBUF , and OUTBUF :

- COEFS , a circular buffer in program memory, is filled with the filter coefficients data from the fircoefs.dat file. Note: The coefficients in fircoefs.dat are ordered so the k= max coefficient comes first, because the FIR loop processes the oldest input sample first and then the next oldest and so on. This means that the coefficient buffer must store the coefficient associated with the oldest value in the delay line in the first position in the coefficient buffer.
- The buffer DLINE , in data memory, contains the delay line, a circular buffer of past samples. The COEFS and DLINE buffers are used in the fir.asm module and hold the values that are multiplied and accumulated.
- INBUF , a non- circular buffer, contains the predefined input samples and is initialized when it is defined.
- OUTBUF , a non- circular buffer, holds the output samples. The executable code starts at the label INITIAL_ SETUP . It is placed at program memory location 0x8, which is the address of the reset interrupt service routine. The first instruction executed is a delayed branch jump to the label BEGIN . In the two cycles that it takes to branch, the memory wait states for program and data memory are set. At the BEGIN label, the data address generator registers are assigned to data buffers. The following four registers are defined to point to buffers:

Register	Buffer Name	Length	Description
I0——	dline	L0= TAPS	delay line
I1——	inbuf	L1= 0	input buffer
I2——	outbuf	L2= 0	outbuffer
I8——	coefs	L8= TAPS	coefficients

If the length of a buffer is not zero, the buffer is circular.

After the buffer is associated with data address registers, the code initializes the delay line. At the label FIR_ INIT a loop that puts zeros in the delay line buffer starts. This initialization loop is necessary because without it the contents of the buffer would be undefined until five samples are accessed.

The actual filtering algorithm is implemented as a loop. The loop is executed SAMPLE number of times (once for each input sample) and ends at the label FILTERING . Iterating through this loop is equivalent to varying n in the FIR filter equation. This loop (in FIRMAIN. ASM) calls FIR (in FIR. ASM) with a delayed branch. In the two cycles that it takes to branch, two transfers happen:

- A value from the input buffer is placed in F0. This is how the input sample is passed to the FIR. ASM module.
- R1 is set to a value that is the amount of times that the loop in the FIR module has to repeat. As shown by the FIR equation, R1 is one less than the number of multiplies that has to be executed for a given value of n. After the code in FIR. ASM completes, control returns to FIRMAIN. ASM

The instruction at the line labeled FILTERING writes a result to the output buffer. After the FILTERING loop completes, the execution goes into an infinite loop.

The sum of products operations, which comprise the core loop of the FIR filter, are executed by FIR.ASM . The code starts by zeroing some registers and loading data from buffers into other registers. At the line labeled FIR, a trick is used to zero the R12 register and fetch an input sample to the delay line buffer in a single cycle: use an XOR to zero R12 instead of using an immediate move. The XOR is an ALU compute operation. The ADSP- 21000 family allows for a computation and a data movement to be executed in the same instruction. If an immediate move instruction was used to set R12 to zero, the data move could not be executed in the same instruction.

Note that the pointer is post modified when the input sample moves to the delay line buffer. Because of this, the coefficients array must be arranged so that the coefficient associated with K = max is first in the array. The modify instruction moves the delay line pointer to the oldest value in the delay line. The input sample just written to the delay line buffer is not accessed until the rest of the buffer is accessed. See Figure

The MACS loop, which computes the sum of products and executes TAPS – 1 number of times, efficiently uses the multifunction capabilities of the ADSP- 21000 architecture. A multifunction instruction can fetch two operands and perform a multiplication and an addition operation in a single cycle.

- The multiplication works on the operands fetched in the previous cycle. Therefore, the coefficients and data must be loaded outside the sum of products loop.
- The operands for the addition are the results of the multiplication; valid operands for the addition are not generated until the loop executes twice. For the first two iterations of the loop, the code uses "dummy" operands of zero. The third time through the loop and after, the multiplication generates two valid operands for the addition.

The code and memory usage for this algorithm is determined by the number of taps the filter has. The code in FIR. ASM is common to FIR filter implementations. The code in FIRMAIN. ASM varies depending on how coefficients and data are input into the filter, and therefore is not included in the benchmark.

The code in FIR.ASM takes seven memory locations for instructions. In addition to the instruction memory requirement, there is also a requirement for the delay line and coefficient buffers. The delay line buffer is stored in data memory and the coefficient buffer is stored in program memory. The length of these buffers is equal to the filter's number of taps.

The number of cycles needed to execute the code is 7+ (number of TAPS).

This number is derived by the following calculation: three instructions plus one cache miss for the first three lines of code; (TAPS – 1) plus one cache miss for the MACS loop; then three instructions to finish off the code. This formula for the number of cycles is:

cycles = 3+ (TAPS – 1) + 3 + 2 cache misses = 7 + number of taps

A cache miss occurs when an instruction that requires two program memory fetches and a fetch from data memory is executed for the first time. When executing this instruction for the first time, the opcode is loaded in to the cache. The next time that instruction is executed, the opcode can be fetched from the cache, and the program memory bus is free to access data.

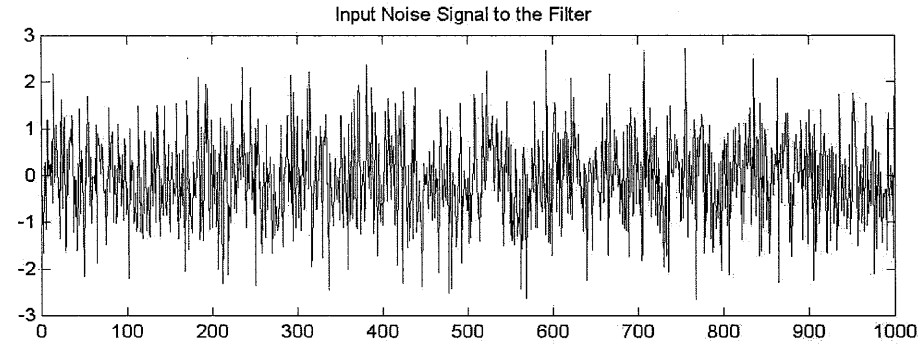
```
/*************************************************************************
FIRMAIN. ASM
*************************************************************************/
#define SAMPLES 0x3e8              /* number of input samples to be filtered*/
#define TAPS 113

 .EXTERN fir;
.SEGMENT /PM pm_ data;
.VAR coefs[ TAPS] = "fircoefs. dat";          /* FIR coefficient stored in file */
.ENDSEG;
.SEGMENT /DM dm_ data;
.VAR dline[ TAPS];                 /* buffer that holds the delay line */
.ENDSEG;
.SEGMENT /DM dm_ data;
.VAR inbuf[ SAMPLES]= "samples.dat";
.VAR outbuf[ SAMPLES]= "output.dat";
.ENDSEG;

.SEGMENT /PM pm_ code;
begin: l0= TAPS;                   /* delay line buffer pointer initialization*/
b0= dline;
m0= 1;
l1= 0; /* input buffer pointer initalization*/
b1= inbuf;
l2= 0;                    /* output buffer pointer initalization*/
b2= outbuf;
l8= TAPS;                  /* coefficent buffer pointer initalization*/
b8= coefs;
call fir_ init (db);       /* initalize delay line buffer to zero */
m8= 1;
r0= TAPS;
lcntr= SAMPLES, do filtering until lce;
        call fir (db);             /* input sample passed in F0, output returned in F0*/
r1= TAPS- 1;
 f0= dm( i1,1);
filtering:
        dm( i2,1)= f0;             /* result is stored in outbuf*/
done:
         jump done;
fir_ init:
lcntr= r0, do zero until lce;
zero:
dm( i0, m0)= 0;          /* initalize the delay line to 0*/
rts;
.ENDSEG;
```

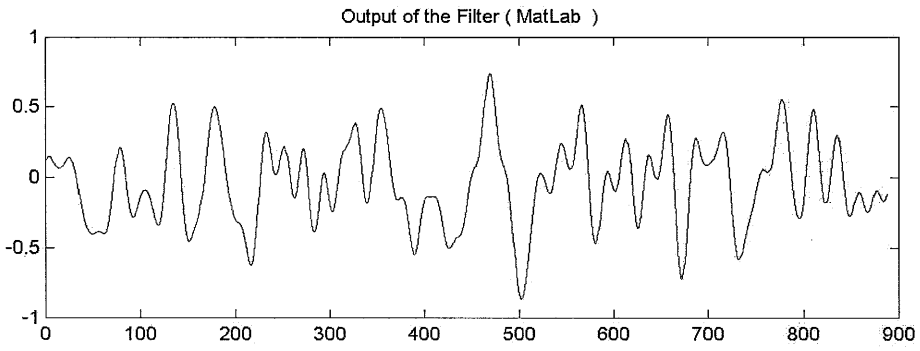
```
/*****************************************************************************
Filter Code
/*****************************************************************************
FIR. ASM
This is a subroutine that implements FIR filter code given coefficents and samples.
Equation Implemented y( n)= Summation from k= 0 to M of H sub k times x( n- k)
Calling Parameters f0 = input sample x( n) r1 = number of taps in the filter minus
 1 b0 = address of the delay line buffer m0 = modify value for the delay line buffer l0 = length
of the delay line buffer b8 = address of the coefficent buffer m8 = modify value for the coefficent
buffer l8 = length of the coefficent buffer
Return Values f0 = output sample y( n)
Registers Affected f0, f4, f8, f12 i0, i8
Cycle Count 6 + (Number of taps -1) + 2 cache misses
# PM Locations 7 instruction words Number of taps locations for coefficents
# DM Locations Number of taps of samples in the delay line
*****************************************************************************/
.GLOBAL fir;
.EXTERN coefs, dline;
.SEGMENT /PM pm_ code;
fir:
        r12= r12 xor r12, dm( i0, m0)= f0;   /* set r12 =0 and store input sample in dline*/
        r8= r8 xor r8, f0= dm( i0, m0),
        f4= pm( i8, m8);                      /* set r8= 0 and grab data from dline and coef*/
        lcntr= r1, do macs until lce;         /* set loop to iterate taps- 1 times*/
macs:
        f12= f0* f4, f8= f8+ f12, f0= dm( i0, m0), f4= pm( i8, m8);
                                              /* perform mult. accumlate and fetch data*/
        rts (db); f12= f0* f4, f8= f8+ f12;   /* perform mult on last pieces of data and 2nd to last
add*/
        f0= f8+ f12;                          /* perform last add and store result in f0*/
.ENDSEG;
```

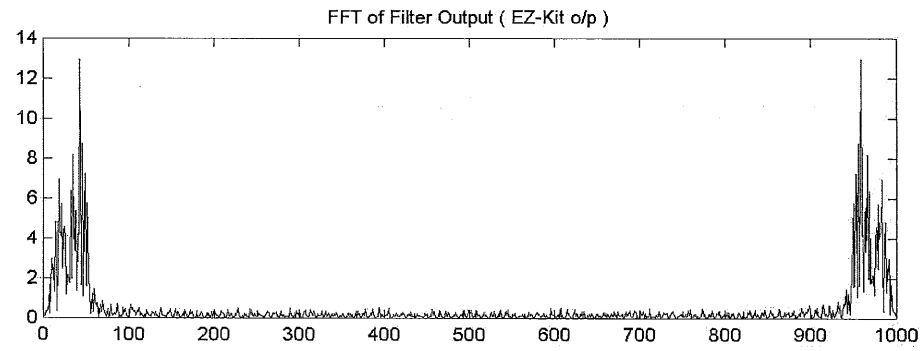
CHAPTER 6

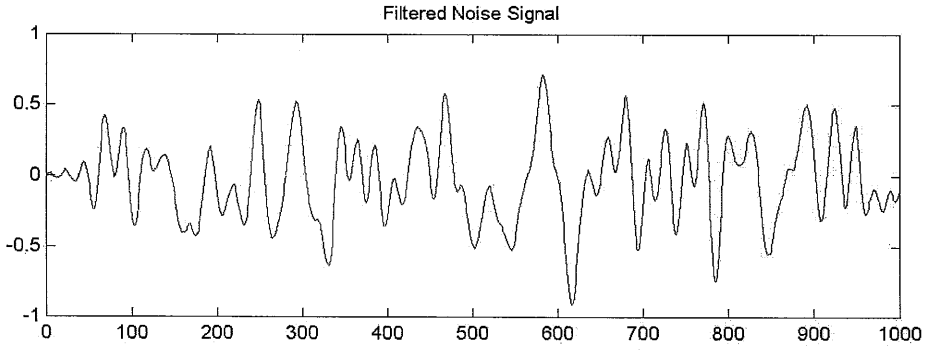
Results and Conclusions

Results of the Simulation of the Filter on the EZ-KIT Lite









Conclusions

The filter was designed with the 21 TAPs first and we found that the filter response in the pass-band is not flat, which implies that the number of taps N must be increased. The taps were increased in the filter and it was found the at N=113 the filter response was flat in the pass-band.

Using 113 TAPS the FIR filter was used to process the encoder data and the results obtained using MATLAB were found to be as expected.

Then we implemented the filter using the ADSP Card. In this process , we obtained the 113 coefficients using the C- Code (Given in the Design Chapter). Then the assembly code for FIR filter was written for ADSP compiled which takes the Encoder data stored in a file and the Coefficients , generated in the previous step from a file. The filter executable was loaded on to the ADSP card for simulation with the test Noise data and the filter output is as shown in the above figure.

The result was found to be same as that of the output from the filter program written in the Matlab. Both the input and output signals are as shown in the figure.

REFERENCES

Books :

Digital Filter Designers Hand Book	-	C Britton Rorabaugh
Digital Signal Processing	-	Richard Lyons
Digital Control Using Microprocessors	-	Paul Katz
ADSP Programmers Manual	-	Analog Devices
ADSP Users Manual	-	Analog Devices
Sharc EZ-KIT Lite Reference Manual	-	Analog Devices

Online Books :

http://www.dspguide.com

http://www.dspguru.com