

Single Sideband Modulation Of Audio Signals Using the DSP



E.L Mabitsela

**HSR Switzerland
1999**

N. Shiburi

Word of thanks

We would like to start this by thanking our sponsors without whom the whole thing would only be a dream. First on the list are the co-ordinators from both institutions, Mr. Alyward from Technikon Pretoria and Mr. Schnider from Hochschule Rapperswil (HSR). Our financial sponsors Holderbank and Alpha Cement for providing the financial backing. We really hope that the lifeline thrown into this project by these sponsors is not cut and the project goes on for years to come.

Our appreciation goes to the way Technikon Pretoria showed faith in us by letting us become part of this Exchange Program. The 1999 DSP GROUP (or “DS-Labor Groupe5” as it stated on the Lab door) would like to thank the HSR for opening their facilities to us and providing us with a great learning experience. We enjoyed every bit of the time we had in the lab.

We would also like to thank the following people for showing us what Swiss Hospitality is all about:

Mr. Motzer for providing us with the Bikes. Thank you for helping us to settle into our new surroundings when we got here, and who would forget the lovely “Braai vleis” at your place. We would also like to thank your Daughter Nicole for the for the Skiing adventure in Bruni (We learned that its really not as easy as it looks).

To our Swiss Family, “The Wespes” we would like to say we really appreciated the way you made us feel welcomed. We really felt like we were part of your family. You guys really went out of your way to make our stay here as pleasant as it was and we really thank you for that.

To Mr. and Mrs. Schädler for a lovely trip to the Rheinfall. We really enjoyed the picnic and the dinner.

To Pete Wiser for taking a lot of his time to show us around. Thank u for another Skiing adventure at Zermatt. We really appreciate your taking Elias to the Doctor after he twisted his knee.

To Andre Ruegg for all the guidance throughout the project.

To Lucia (Lushia) and Mr. Graetzer for all they did for us. It was greatly appreciated.

*A special Thanks goes to Dr Schüeli, our project sponsor, for taking us under his wing and introducing us to a wonderful world of Digital Signal Processing.
Last but not least, to our fellow students at the HSR, thank u for making us feel at home throughout our stay here.*

E.L Mabitsela

N Shiburi

Table of Contents

Section A (Project Definition)

PROJECT	SA 1
Aims	SA 1
Tasks	SA 2
Systems Specifications	SA 2
Aid, Tools	SA 2
References	SA 2
Report	SA 3
Dates	SA 3
Organization	SA 3

Section B (Report)

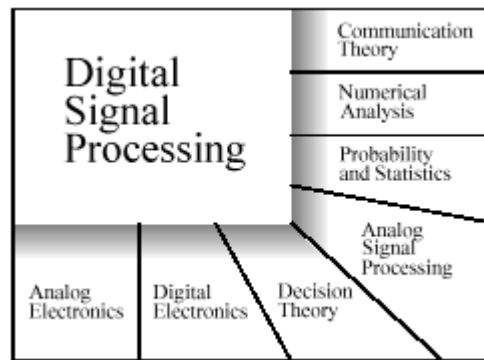
1 Digital Signal Processing	1
1.1 Introduction	1
1.2 Why Process Signals Digitally?	2
1.2.1 Programmability	3
1.2.2 Stability	3
1.2.3 Repeatability	3
1.2.4 Adaptability	4
2 Filtering	5
2.1 FIR Filters	6
2.1.1 Digital Filters	5
2.1.2 Digital Filter Equation	6

2.2 FIR Design	8
2.2.1 Digital Filter Specifications	8
2.2.2 Fourier Transforms	10
2.3 Design Strategies	11
2.3.1 Strategy of the Windowed-Sinc	11
2.3.2 Designing the Filter	16
3 ADSP 2106*(DSP Chip) Introductory Theory and Architecture	19
3.1 How DSPs are Different from Other Microprocessors	19
3.2 Architecture of the Digital Signal Processor	21
3.3 C versus Assembly	27
3.4 Circular Buffering	29
3.5 General Description of the Bittware board	32
4 Modulation	33
4.1 Introduction	33
4.2 Single Sideband Modulation	34
5 Project Implementation (Solution)	36
5.1 Specifications of the Systems	36
5.1.1 The Weaver Method	36
5.1.2 The Phase Method	37
5.2 Tasks and Solutions to them	37
5.3 Comparison of Both Methods	48
6. Hardware Setup	49
6.1 Software Setting up and Running	49

7. Conclusion	51
8. Bibliography	53
8.1 Reference books	53
8.2 User Manuals	53
8.3 Internet Resources	54
APPENDIX	55

SECTION A

SECTION B



1 Introduction to Digital Signal Processing (DSP)

Digital Signal Processing (DSP) is used in a variety of applications, and it is hard to find a good definition that is general.

Dictionary description of the words are :

Digital

Operating by the use of discrete signals to represent data in the form of numbers

Signal

A variable parameter by which information is conveyed through an electronic circuit

Processing

To perform operations on data according to programmed instructions

Which leads to a simple definition of

Digital Signal processing: changing or analyzing information, which is measured as discrete sequence of numbers.

Our environment is full of analog (continuous variable defined with infinite precision) signals, such as sound, temperature and light. In the case of sound the ear converts the sound into electrical impulses to the brain. The ear then analyzes the properties of the sound such as amplitude, frequency, and phase to categorize the sound and determine its direction.

Electronic sensors can be used to convert pressure, temperature, and sound to electrical signals in the same way, but have to be converted to digital form for processing. This conversion process is called analog-to-digital, or A/D, conversion. The signal is then processed to determine its properties. This processing is called Digital Signal processing.

The signal is still in digital form and maybe needed to be converted back into analog form before being passed to the loud speaker. The complete conversion and processing chain is shown in figure 1.1.

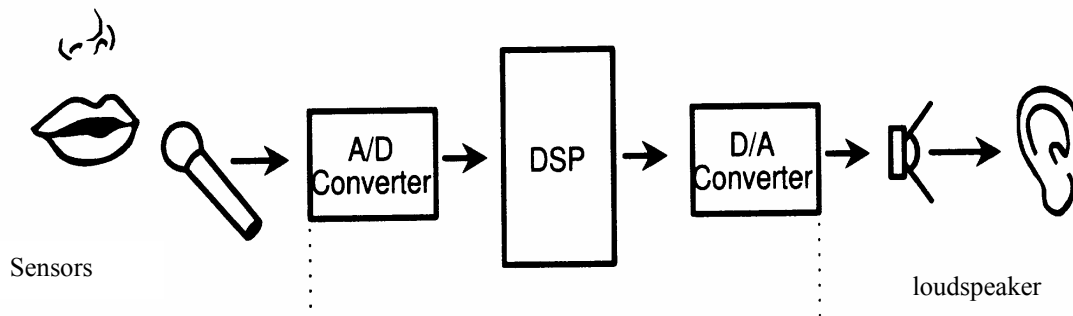


Figure 1.1 The digital signal processing system.

The DSP is a technology that dates back to the mid-1960's when computers and other digital circuitry became a fast enough to process large amounts of data efficiently. The implementation of DSP systems falls into two categories – hard ware and software.

Many systems can be implemented completely by software in general-purpose computers, especially for relatively low sampling rates or non-real time applications. On the other hand, many applications with high sampling rates, high production volumes, and/or low cost space requirements can only be satisfied by special-purpose hardware.

1.2 Why Process Signals Digitally ?

Signals are naturally Analog and need to be converted to a Digital signals for them to be processed digitally. FIGURE 1.2 illustrates this



process

FIGURE 1.2 DIGITAL PROCESING OF AN ANALOG SIGNAL

The question might be `why process signals digitaly` , since it may seem easier to process them as analog signals.

There are more advantages to Prosessing signals Digitally than there are disadvantages. A list of a few is given with here and discusscusion will be done on an individual basis. These are :

- **Programmability**
- **Stability**
- **Repeatability**
- **Adaptability**

1.2.1 Programmability

The most important reason why Digital Signal Processing is favoured over analog Signal processing is that it is possible to design one hardware configuration that can be programmed to perform a very wide variety of signal processing tasks, simply by loading in different software . For example , a digital filter may be reprogrammed from a low pass to a high pass with no change in the hardware, which in an analog system would result in a complete change of circuit components.

This programmability of these devices makes them more suitable because they can be easily upgraded by simply changing the software in the system.

1.2.2 Stability

Performance is one thing we look into very critically when it comes to the design of any system. In analog systems the individual components (resitors, capasitors etc.) change their characteristics with changes in temperature.

Digital circuits will show no variation with temperature throughout their guaranteed operating range. Another form of variability that affects analog circuits is component aging. DSP circuits can be prograded to detect and compensate for changes in the aging of analog and mechanical parts of the system.

1.2.3 Repeatability

This refers to the ability to produce the same output with different systems with the identical specifications . This is one great advantage of a digital system, because Analog circuit components have a tolerance specification which causes a spread of performance in analog systems . Resitors can have a tolerance of 5% of their value , depending on the prize of the component.

1.2.4 Adaptivity

This is the ability to change its parameters according to a change in the environment. An example of this is the noise cancellation system in a car. In this case the noise that is cancelled is originally caused by the engine and the resonances set up in the body panels by engine vibrations .

The noise cancellation system takes the engine speed as a reference and attempts to produce an ,“anti-noise“, signal to cancel the cockpit noise. There are microphones in each headrest that determine the success of the attempt. Based on the changes detected by the microphones, the system changes the characteristics of the anti-noise until the best noise reduction is achieved . When the engine speed changes , the systems adapts once more to the new engine speed.

This answers the question asked earlier about why convert from Analog to Digital. Only a few of the advantages of digital signal processing over analog are listed. The list goes on and can be found in books that cover the subject of Digital signal processing.

Since no system is perfect there are also some disadvantages of digital processing, some of them being: increased system complexity , available frequency range and the fact that active devices used results in electrical power consumption.

However the advantages outweigh the disadvantages and with the cost of digital processor hardware constantly decreasing , there is an increase in the use of DSPs.



2. Filtering

Filtering is a process of selecting, or suppressing, certain frequency components of a signal.

A coffee filter allows small particles to pass while trapping the larger grains. A digital filter does a similar thing, but with more subtlety. The digital filter allows passing certain frequency components of the signal. In this it is similar to the coffee filter, with frequency standing in for particle size. But the digital filter can be subtler than simply trapping or allowing through: it can attenuate, or suppress each frequency components by a desired amount. This allows a digital filter to shape the frequency spectrum of the signal.

Filtering is often, though not always, done to suppress noise. It depends on the signal's frequency spectrum being different from that of the noise:

2.1 DIGITAL FILTERS

For a long period digital filters have been the most common application of digital signal processors. Like stated before, there are two basic types of digital filters Infinite Impulse Response Filter (IIR) and the Finite Impulse Response Filter (FIR).

Digital filters are a very important part of Digital Signal Processing. It is a fact that their extraordinary performance is one of the key reasons that DSP has become so popular.

They have two important uses: signal *separation* and signal *restoration*. Signal separation is needed when a signal has been contaminated with interference, noise, or other signals. For example imagine a device for measuring the electrical activity of a baby's heart(EKG) while still in the womb. The breathing and heartbeat of the mother will likely corrupt the raw signal. A filter might be used in this case to separate these signals so that they can be individually analysed.

Signal restoration is used when a signal has been distorted in some way. For example, an audio recording made with poor equipment may be filtered to better represent the sound as it actually occurred. These problems can be attacked with either analog or digital filters. Which is better? 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.

The design of the IIR filters is similar to that of an analogue filter whereas the design of the FIR filters are unique to digital filtering. The order of a FIR to meet the desired filter specifications is much greater than that of an IIR filter. This may be so, but FIR filters possess characteristics unknown to IIR filters. The most important of these are linear phase and constant group delay. This makes FIR filters a necessity in applications which demand little phase distortion.

The following section is based on the FIR filters first the description and then later the design.

2.1.1 Digital filter equation

Output from a digital filter is made up from previous inputs and previous outputs, using the operation of **convolution** (More discussion about this topic to follow)

$$y(n) = \sum c[k] * x[n-k] + \sum d[j] * y[n-j]$$

EQUATION .1 The equation for a digital filter.

Where ,
y(n) : output value

$\Sigma c[k]$ and $\Sigma d[j]$: Coefficients
 $x[n-k]$: Previous input
 $y[n-j]$: Previous output

Two convolutions are involved: one with the previous inputs, and one with the previous outputs. In each case the convolving function is called the filter coefficients.

It is much easier to approach the problem of calculating filter coefficients if the filter equation simplified so that only the previous inputs have to dealt with (that is, excluding the possibilities of feedback). The filter equation is then simplified:

$$y(n) = \Sigma c[k] * x[n-k]$$

EQUATION.2 Simplified Filter equation.

If such a filter is subjected to an impulse (a signal consisting of one value followed by zeroes) then it's output must necessarily become zero after the impulse has run through the summation. So the impulse response of such a filter must necessarily be finite in duration. Such a filter is called a Finite Impulse Response filter or FIR filter.

Convolution

Convolution is the smoothing or averaging operation in which one of the functions is weighted by the other. If there are two functions $x(t)$ and $y(t)$, the convolution of the two functions depicted

$$z(t) = x(t) * y(t)$$

EQUATION.3 Convolution Sign Example.

can be replaced by the following integral

$$z(t) = \int x(\tau)y(t - \tau) d\tau$$

EQUATION.4 Convolution Integral.

This equation I known as the general convolution equation. The following shows how the algorithm is implemented on the DSP chip.

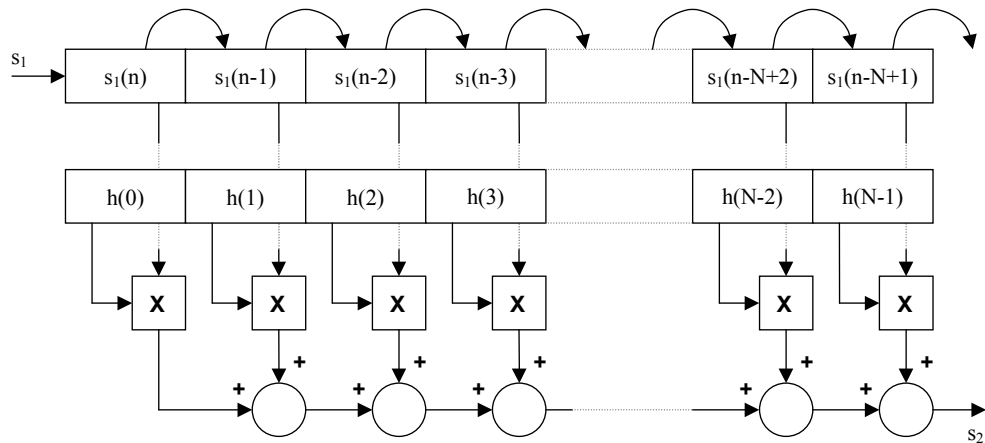


FIGURE 2.1 Convolution as performed by the DSP for linear buffers.

2.2 FIR Design.

2.2.1 Digital filter specifications

Digital filters can be more subtly specified than analogue filters, and so are specified in a different way:

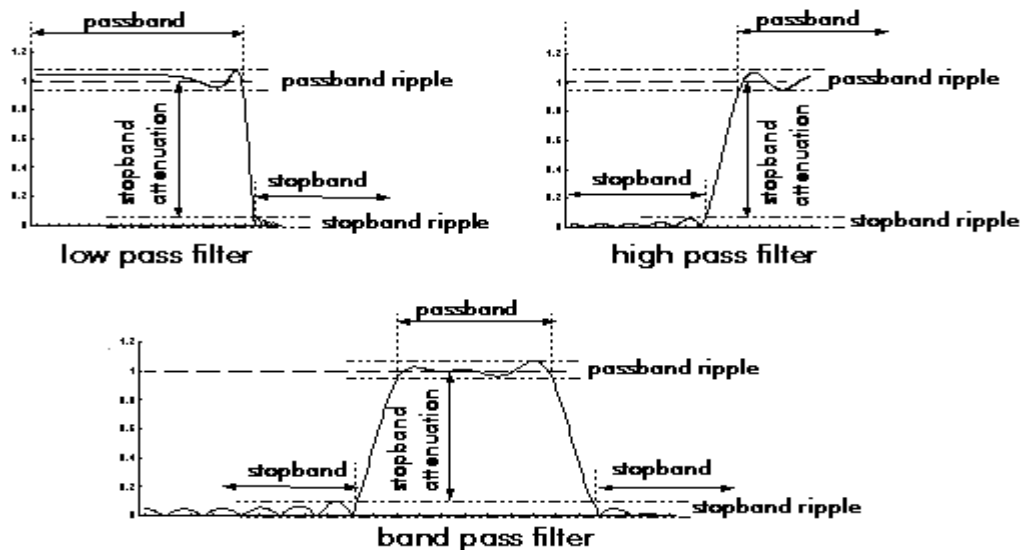


FIGURE 2.1 Different ways to specify FIR filters

Whereas analogue filters are specified in terms of their '3dB point' and their 'rolloff', digital filters are specified in terms of desired attenuation, and permitted deviations from the desired value in their frequency response:

passband : the band of frequency components that are allowed to pass.

stopband : the band of frequency components that are suppressed.

passband ripple : the maximum amount by which attenuation in the passband may deviate from nominal gain.

stopband attenuation: the minimum amount by which frequency components in the stopband are attenuated.

The passband need not necessarily extend to the 3 dB point: for example, if passband ripple is specified as -45 dB, then the passband only extends to a point at which attenuation has increased to -45 dB.

Between the passband and the stopband lies a transition band where the filter's shape may be unspecified.

Note that the stopband attenuation is formally specified as the attenuation to the top of the first sidelobe of the filter's **frequency response**.

Digital filters can also have an 'arbitrary response': meaning, the attenuation is specified at certain chosen frequencies, or for certain frequency bands. Digital filters are also characterised by their response to an impulse.

The impulse response is an indication of how long the filter takes to settle into a steady state: it is also an indication of the filter's stability - an impulse response that continues oscillating in the long term indicates the filter may be prone to instability.

The impulse response defines the filter just as well as does the frequency response.

Filter frequency response

Since filtering is a frequency selective process, the important thing about a digital filter is its frequency response.

The filter's frequency response can be calculated from its filter equation:

$$H(f) = \frac{\sum c[k] * \exp(-2\pi jk(f\Delta))}{1 - \sum d(j) * \exp(-2\pi jk(f\Delta))}$$

EQUATION.5 Filter Equation for the Frequency Response

Where,

j :is the square root of minus one (defined as a number whose sole property is that its square is minus one).

The frequency response $H(f)$ is a continuous function, even though the filter equation is a discrete summation.

Whilst it is appreciable to be able to calculate the frequency response given the filter coefficients when designing a digital filter, it was decided the inverse operation would be done: that is, to calculate the filter coefficients having first defined the desired frequency response. The challenge at this point was the inverse solution to the frequency response equation. Unfortunately, there was no general inverse solution.

Filters are usually designed with the idea that they will be implemented on hardware, the DSP chip in this case. This means the task was to design a filter that meets the requirement but which requires the least possible amount of computation: that is, using the smallest number of coefficients. So we are faced with an insoluble inverse problem, on which we wanted to impose additional constraints.

This is why digital filter design is considered more an art than a science: the art of finding an acceptable compromise between conflicting constraints.

2.2.2 Fourier transforms

Jean Baptiste Fourier showed that any signal or waveform could be made up just by adding together a series of pure tones (sine waves) with appropriate amplitude and phase.

This is a rather startling theory. It means, for instance, that by simply turning on a number of sine wave generators one could sit back and enjoy some lovely Melodies. Of course one would have to use a very large number of sine wave generators, and it would take a long time.

Fourier's theorem assumes we add sine waves of infinite duration.

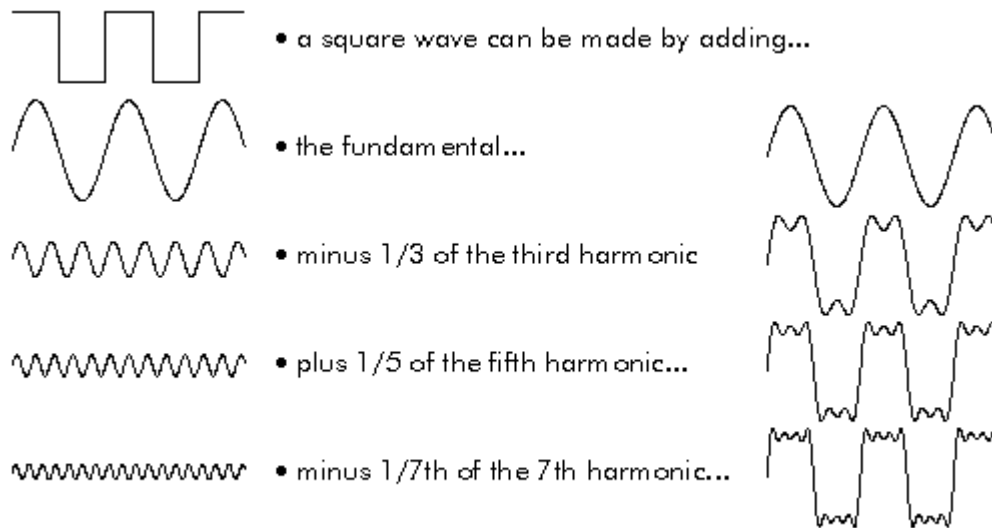


FIGURE 2.2 Different waves can be combined to form a square wave.

The diagram shows how a square wave can be made up by adding together pure sine waves at the harmonics of the fundamental frequency.

Any signal can be made up by adding together the correct sine waves with appropriate amplitude and phase. The *Fourier transform is an equation to calculate the frequency, amplitude and phase of each sine wave needed to make up any given signal.*

- The Fourier Transform (FT) is a mathematical formula using integrals
- The Discrete Fourier Transform (DFT) is a discrete numerical equivalent using sums instead of integrals
- The Fast Fourier Transform (FFT) is just a computationally fast way to calculate the DFT

The Discrete Fourier Transform involves a summation:

$$H(f) = \sum c[k] * \exp(-2\pi jk(f\Delta))$$

EQUATION.6 Summation for the Discrete Fourier Transform

Note that the DFT and the FFT involve a lot of multiplying and then accumulating the result - this is typical of DSP operations and is called a 'multiply/accumulate' operation. It is the reason that DSP processors can do multiplication and additions in parallel.

2.3 Design Strategies

2.3.1 Strategy of the Windowed-Sinc

Figure 2.3 (Page 14) illustrates the idea behind the windowed-sinc filter. In (a), the frequency response of the *ideal* low-pass filter is shown. All frequencies below the cutoff frequency, are passed with unity amplitude, while all higher f_c frequencies are blocked. The passband is perfectly flat, the attenuation in the stopband is infinite, and the transition between the two is infinitesimally small.

Taking the Inverse Fourier Transform of this ideal frequency response produces the ideal filter length (impulse response) shown in (b). This curve is of the general form, called $\sin(x)/x$ the **sinc function**, given by:

$$h[i] = \frac{\sin(2\pi f_c i)}{i\pi}$$

EQUATION.7 Sinc function

Convolving an input signal with this filter provides a *perfect low-pass* filter. The problem is, the sinc function continues to both negative and positive infinity without dropping to zero amplitude. While this infinite length is not a problem for *mathematics*, it is a showstopper for *computers*. To get around this problem, two modifications were made to the sinc function in (b), resulting in the waveform shown in (c).

First, it is truncated to points, symmetrically chosen around the main lobe, where N is an $N - 1$ even number. All samples outside these points are set to zero, or simply $N - 1$ ignored. Second, the entire sequence is shifted to the right so that it runs from 0 to N . This allows the filter length to be represented using only *positive* indexes.

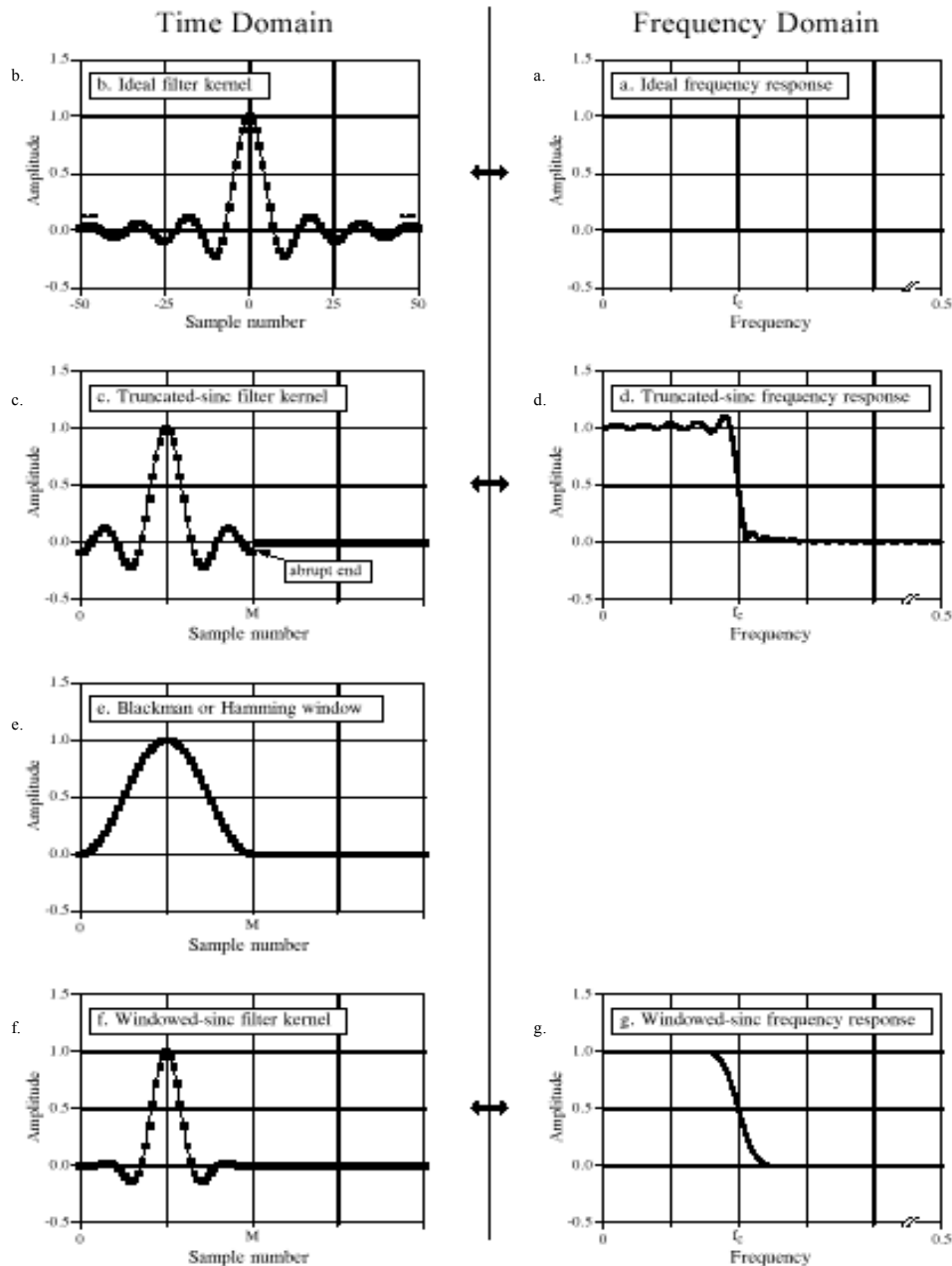


FIGURE 2.3 Derivation of the windowed-sinc filter length. The frequency response of the ideal low-pass filter is shown in (a), with the corresponding filter length in (b), a sinc function. Since the sinc is infinitely long, it must be truncated to be used in a computer, as shown in (c). However, this truncation results in undesirable changes in the frequency response, (d). The solution is to multiply the truncated-sinc with a smooth window, (e), resulting in the windowed-sinc filter length, (f). The frequency response of the windowed-sinc, (g), is smooth and well behaved. These figures are not to scale.

While many programming languages allow *negative* indexes, it was discovered they are a not very convenient to use. The sole effect of this shift in the filter length is to $N/2$ shift the output signal by the same amount.

Since the modified filter length is only an approximation to the ideal filter length, 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

These problems result from the abrupt discontinuity at the ends of the truncated sinc function. Increasing the length of the filter length does not reduce these problems; the discontinuity is significant no matter how long N is made.

Fortunately, there is a simple method of improving this situation. Figure (e) shows a smoothly tapered curve called a **Blackman window**. Multiplying the truncated-sinc, (c), by the Blackman window, (e), results in the **windowed-sinc** filter length 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 were considered in our project, the **Hamming window** and the **Blackman window**. These are given by:

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

$$w[i] = 0.42 - 0.5 \cos(2\pi i/M) + 0.08 \cos(4\pi i/M)$$

EQUATION.8 Formulae for both windows.

Figure 2.4 shows the shape of these two windows for (i.e., 51 total N ' 50 points in the curves). Which of these two windows to use? It's a trade-off between parameters. As shown in FIGURE. 2.4b, the Hamming window has about a 20% faster *roll-off* than the Blackman. However,

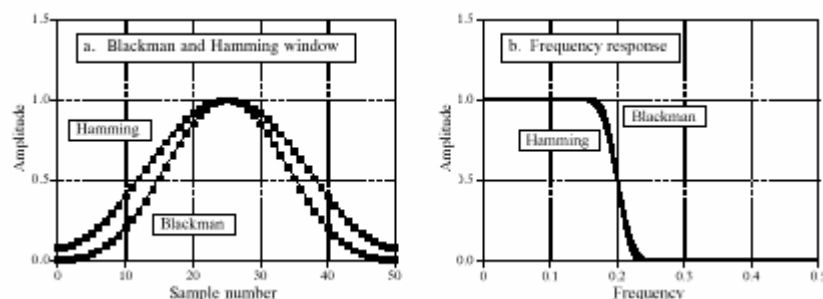
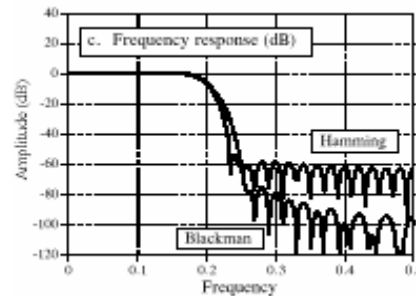


FIGURE 2.4

Characteristics of the Blackman and Hamming windows. The shapes of these two windows are shown in (a), and given by Eqs.6. As shown in (b), the Hamming window results in about 20% faster roll-off than the Blackman window. However, the Blackman window has better stop-band attenuation (Blackman: 0.02%, Hamming: 0.2%), and a lower passband ripple (Blackman:0.02% Hamming: 0.2%).

(c) shows that the Blackman has a better *stopband attenuation*. To be exact, the stopband attenuation for the Blackman is -74dB (-0.02%), while the Hamming is only -53dB (-0.2%). Although it cannot be seen in these graphs, the Blackman has a *passband ripple* of only about 0.02%, while the Hamming is typically 0.2%. In general, the Blackman should have been the first choice; a slow roll-off is easier to handle than poor stopband attenuation. With the given stopband attenuation of -45 dB for our project, the Hamming window seemed a natural starting point for experimentation.



There are other windows, although they fall short of the Blackman and Hamming. The **Bartlett window** is a triangle, using straight lines for the taper. The **Hanning window**, also called the **raised cosine window**. These two windows have $w[i] = 0.5 \& 0.5\cos(2\pi i / N)$ about the same roll-off speed as the Hamming, but worse stopband attenuation (Bartlett: -25dB or 5.6%, Hanning -44dB or 0.63%).

Also in the list is the **rectangular window**. This is the same as *no* window, just a truncation of the tails (such as in FIGURE2.4 c). While the roll-off is -2.5 times faster than the Blackman, the stopband attenuation is only -21dB (8.9%).

2.3.2 Designing the Filter

To design a windowed-sinc, two parameters were be selected: the cutoff frequency, and the length of the filter, N . The cutoff frequency f_C is expressed as a fraction of the sampling rate, and therefore must be between 0 and 0.5.

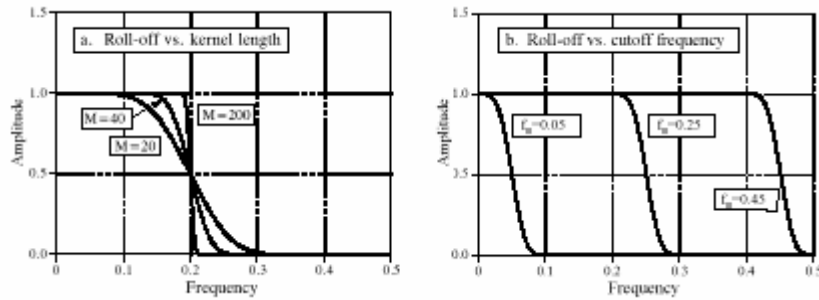


FIGURE 2.5 Filter length vs. roll-off of the windowed-sinc filter. As shown in (a), for $N = 20, 40,$ and 200 , the transition bandwidths are $BW = 0.2, 0.1,$ and 0.02 of the sampling rate, respectively. As shown in (b), the shape of the frequency response does not change with different cutoff frequencies. In (b), $N = 60$.

The value for N sets the *roll-off* according to the approximation:

$$N = 4/BW$$

EQUATION 9

Filter length vs. roll-off. The length of the filter length, N , determines the transition bandwidth of the filter, BW .

This is only an approximation since roll-off depends on the particular window being used.

BW = the width of the transition band

Measured from where the curve just leaves one, to where it almost reaches zero (say, 99% to 1% of the curve). The transition bandwidth is also expressed as a fraction of the sampling frequency, and must be between 0 and 0.5. Figure 2.5a shows an example of how this approximation is used. The three curves shown are generated from filter lengths with: From Eq. 7, the $N = 20, 40,$ and 200 transition bandwidths are: respectively. Figure (b) $BW = 0.2, 0.1,$ and 0.02 shows that the shape of the frequency response does not depend on the cutoff frequency selected.

Since the time required for a convolution is proportional to the length of the signals, Eq. 7 expresses a trade-off between *computation time* (depends on the value of N) and *filter sharpness* (the value of BW). For instance, the 20% slower roll-off of the Blackman window (as compared with the Hamming) can be compensated for by using a filter length 20% longer.

In other words, it could be said that the Blackman window is 20% slower to execute than an equivalent roll-off Hamming window. This is important

because the execution speed of windowed-sinc filters is already terribly slow. This led us to choosing the Hamming window eventually.

As also shown in FIGURE.2.5b, the cutoff frequency of the windowed-sinc filter is measured at the *one-half amplitude* point. Why use 0.5 instead of the standard 0.707 (-3dB) used in analog electronics and other digital filters? This is because the windowed-sinc's frequency response is *symmetrical* between the passband and the stopband. The Hamming window results in a passband ripple of 0.2%, and an *identical* stopband attenuation (i.e., ripple in the stopband) of 0.2%. Other filters do not show this symmetry, and therefore have no advantage in using the one-half amplitude point to mark the cutoff frequency.

As shown later in this chapter, this symmetry makes the windowed-sinc ideal for *spectral inversion*. After N has been selected, the filter length is calculated from the fC relation:

$$h[i] = K \frac{\sin(2\pi f_c (i - M/2))}{i - M/2} \left[0.42 - 0.5 \cos\left(\frac{2\pi i}{M}\right) + 0.08 \cos\left(\frac{4\pi i}{M}\right) \right]$$

EQUATION 10

The windowed-sinc filter length. The cutoff frequency, is expressed as a fC fraction of the sampling rate, a value between 0 and 0.5. N determines the length of the filter length, which must be an even integer. The sample number i , is an integer that runs from 0 to N , resulting in total points in the filter $N - 1$ length. The constant, K , is chosen to provide unity gain at zero frequency. To avoid a divide-by-zero error, for $i = N/2$, use $h[i] = 2BfC K$

The equation is not as complex as it looks. Based on the previous discussion, the *sinc function*, the $N/2$ shift, and the *window* are seen here. For the filter to have unity gain at DC, the constant K must be chosen such that the sum of all the samples is equal to one.

During implementation, K was ignored during the calculation of the filter length, and then *normalized* all of the samples as needed.

This equation may be long, but it is easy to use. To be specific about where the filter length described by Eq. 10, It is located in the computer array. As an example, N will be chosen to be 100.

Remember, N must be an even number. The first point in the filter length is in array location 0, while the last point is in array location 100. This means that the entire signal is 101 points long. The center of symmetry is at point 50, i.e. The 50 points to the left of point 50 are symmetrical with the 50 $N/2$ points to the right. Point 0 is the same value as point 100, and point 49 is the same as point 51. If you must have a specific number of samples in the filter length, such as to use the FFT, simply add zeros to one end or the other. For example, with $N = 100$, you could make samples 101 through 127 equal to $N - 100$ zero, resulting in a filter length 128 points long.

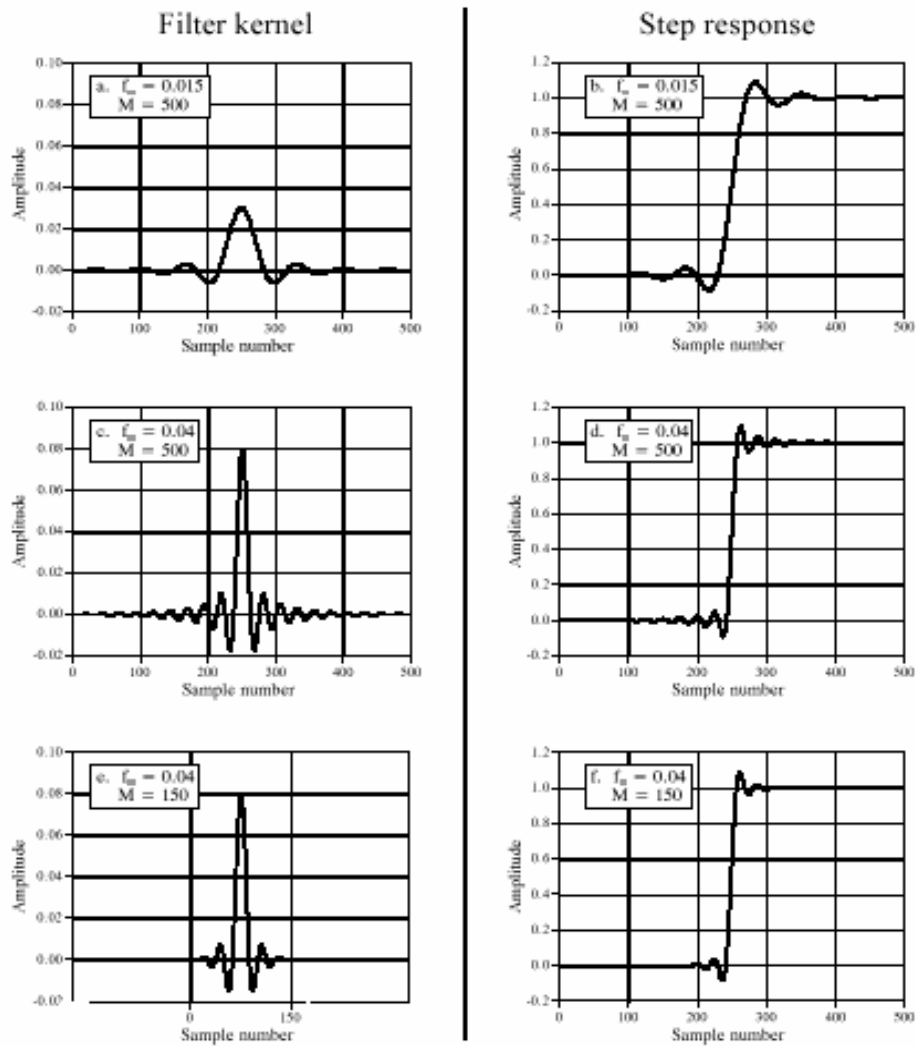
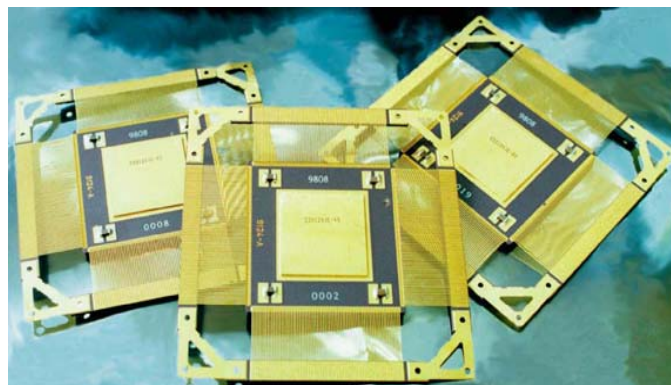


FIGURE 3.7 Examples of windowed sinc functions. The M used here is the N referred to in earlier sections.



3. ADSP 2106*(DSP Chip) Introductory Theory and Architecture

3.1 How DSPs are Different from Other Microprocessors

Computers are extremely capable in two broad areas, (1) **data manipulation**, such as word processing and database management, and (2) **mathematical calculation**, used in science, engineering, and Digital Signal Processing. All microprocessors can perform both tasks; however, it is difficult (expensive) to make a device that is *optimized* for both. There are technical tradeoffs in the hardware design, such as the size of the instruction set and how interrupts are handled. Even more important, there are *marketing* issues involved: development and manufacturing cost, competitive position, product lifetime, and so on.

As a broad generalization, these factors have made traditional microprocessors, such as the Pentium®, primarily directed at data manipulation. Similarly, DSPs are designed to perform the mathematical calculations needed in Digital Signal Processing.

	Data Manipulation	Math Calculation
Typical Applications	Word processing, database management, spread sheets, operating systems, etc.	Digital Signal Processing, motion control, scientific and engineering simulations, etc.
Main Operations	data movement ($A \rightarrow B$) value testing (<i>If $A=B$ then ...</i>)	addition ($A+B=C$) multiplication ($A \times B=C$)

FIGURE 3.1
Data manipulation versus mathematical calculation. Digital computers are useful for two general tasks: *data manipulation* and *mathematical calculation*. Data manipulation is based on moving data and testing inequalities, while mathematical calculation uses multiplication and addition.

Figure 3.1 lists the most important differences between these two categories. Data manipulation involves storing and sorting information. For instance, consider a word processing program. The basic task is to store the information (typed in by the operator), organize the information (cut and paste, spell checking, page layout, etc.), and then retrieve the information (such as saving the document on a floppy disk or printing it with a laser printer). These tasks are accomplished by *moving* data from one location to another, and

testing for inequalities ($A=B$, $A<B$, etc.). As an example, imagine sorting a list of words into alphabetical order. Each word is represented by an 8 bit number, the ASCII value of the first letter in the word. Alphabetizing involved rearranging the order of the words until the ASCII values continually increase from the beginning to the end of the list. This can be accomplished by repeating two steps over-and-over until the alphabetization is complete. First, test two adjacent entries for being in alphabetical order (IF $A>B$ THEN ...). Second, if the two entries are not in alphabetical order, switch them so that they are (AWB). This two step process is repeated many times on all adjacent pairs, the list will eventually become alphabetized.

In comparison, the execution speed of most DSP algorithms is limited almost completely by the number of multiplications and additions required.

Using the standard notation, the input signal is referred to by $x[]$, while the output signal is denoted by $y[]$. Our task is to calculate the sample at location n in the output signal, i.e.,. An FIR filter $y[n]$ performs this calculation by multiplying appropriate samples from the input signal by a group of coefficients, denoted by: a_0 , a_1 , a_2 , a_3 ,, and then adding the products. In equation form, is found by: $y[n]$

$$y(n) = a_0 x(n) + a_1 x(n-1) + a_2 x(n-2) + a_3 x(n-3) + \dots$$

EQUATION .11 Formula for DSP to calculate the FIR filters.

This is simply saying that the input signal has been *convolved* with a filter length (i.e., an impulse response) consisting of: a_0 , a_1 , a_2 , a_3 ,, Depending on the application, there may only be a few coefficients in the filter length, or many thousands. While there is some data transfer and inequality evaluation in this algorithm, such as to keep track of the intermediate results and control the loops, the math operations dominate the execution time.

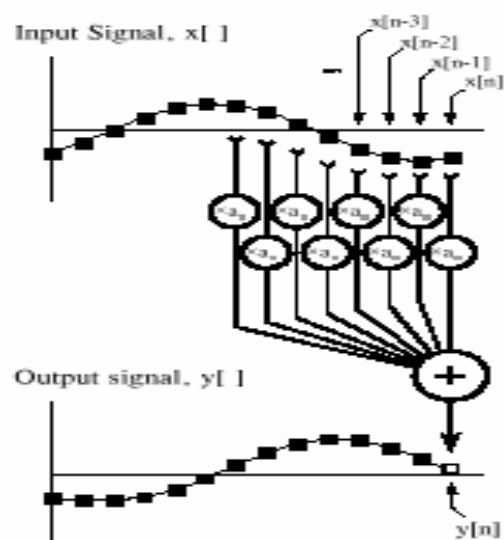


FIGURE 3.2 FIR digital filter. In FIR filtering, each sample in the output signal, $y[n]$, is found by multiplying samples from the input signal, $x[n]$, $x[n-1]$, $x[n-2]$,, by

the filter length coefficients, $a_0, a_1, a_2, a_3 \dots$, and summing the products.

In comparison, most DSPs are used in applications where the processing is *continuous*, not having a defined start or end. For instance, consider an engineer designing a DSP system for an audio signal, such as a hearing aid. If the digital signal is being received at 20,000 samples per second, the DSP must be able to maintain a sustained throughput of 20,000 samples per second. However, there are important reasons not to make it any faster than necessary. As the speed increases, so does the *cost*, the *power consumption*, the *design difficulty*, and so on. This makes an accurate knowledge of the execution time critical for selecting the proper device, as well as the algorithms that can be applied.

3.2 Architecture of the Digital Signal Processor

One of the biggest bottlenecks (That is Engineering Slang for Bugs) in executing DSP algorithms is transferring information to and from memory.

This includes *data*, such as samples from the input signal and the filter coefficients, as well as *program instructions*, the binary codes that go into the program sequencer. For example, suppose it is needed to multiply two numbers that reside somewhere in memory.

To do this, the computer must fetch three binary values from memory, the numbers to be multiplied, plus the program instruction describing what to do.

Figure 3.3a shows how this seemingly simple task is done in a traditional microprocessor. This is often called a **Von Neumann architecture**, after the brilliant American mathematician John Von Neumann.

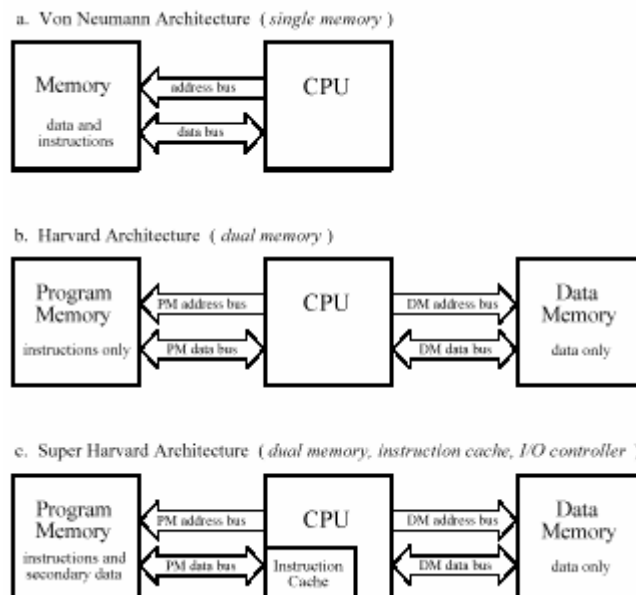
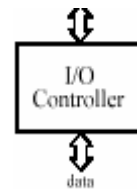


FIGURE 3.3 The Architectures

As shown in (a), a Von Neumann architecture contains a single memory and a single bus for transferring data into and out of the central processing unit (CPU). Multiplying two numbers requires at least three clock cycles, one to transfer each of the three numbers over the bus from the memory to the CPU.

We don't count the time to transfer the result back to memory, because we assume that it remains in the CPU for additional manipulation (such as the sum of products in an FIR filter).



The Von Neumann design is quite satisfactory when you are content to execute all of the required tasks in serial. In fact, most computers today are of the Von Neumann design. Other architectures are only needed when very fast processing is required, higher costs come with increased complexity.

This leads us to the **Harvard architecture**, shown in (b). This is named for the work done at Harvard University in the 1940s under the leadership of Howard Aiken (1900-1973). As shown in this illustration, Aiken insisted on separate memories for data and program instructions, with separate buses for each. Since the buses operate independently, program instructions and data can be fetched at the same time, improving the speed over the single bus design.

Most present day DSPs use this dual bus architecture. Figure (c) illustrates the next level of sophistication, the **Super Harvard Architecture**. This term was coined by Analog Devices to describe the internal operation of their ADSP-2106x and new ADSP-211xx families of Digital Signal Processors.

These are called **SHARC®** DSPs, a contraction of the longer term, Super Harvard ARChitecture. First, a look at how the instruction cache improves the performance of the Harvard architecture. A handicap of the basic Harvard design is that the datamemory bus is busier than the program memory bus. When two numbers are multiplied, two binary values (the numbers) must be passed over the data memory bus, while only one binary value (the program instruction) is passed over the program memory bus. To improve upon this situation, start by relocating part of the "data" to program memory. For instance, placing the filter coefficients in program memory, while keeping the

input signal in data memory. (This relocated data is called "secondary data" in the illustration).

The perception at this point might be that this doesn't seem to help the situation; now we must transfer one value over the data memory bus (the input signal sample), but two values over the program memory bus (the Input signal value and the coefficient). The fact of the matter is, if instructions were executed randomly, this situation would be no better at all.

However, DSP algorithms generally spend most of their execution time in loops. This means that the same set of program instructions will continually pass from program memory to the CPU. The Super Harvard architecture takes advantage of this situation by including an **instruction cache** in the CPU. This is a small memory that contains about 32 of the most recent program instructions.

The first time through a loop, the program instructions must be passed over the program memory bus. This results in slower operation because of the conflict with the coefficients that must also be fetched along this path. However, on additional executions of the loop, the program instructions can be pulled from the instruction cache.

This means that all of the memory to CPU information transfers can be accomplished in a single cycle: the sample from the input signal comes over the data memory bus, the coefficient comes over the program memory bus, and the program instruction comes from the instruction cache. In the jargon of the field, this efficient transfer of data is called a *high memory-access bandwidth*. Figure 3.4([Page 27](#)) presents a more detailed view of the SHARC architecture, showing the **I/O controller** connected to data memory. This is how the signals enter and exit the system. For instance, the SHARC DSPs provides both serial and parallel communications ports. These are extremely high-speed connections.

For example, at a 44 MHz clock speed, there are two serial ports that operate at 44 Mbits/second each, while six parallel ports each provide a 44 Mbytes/second data transfer. When all six parallel ports are used together, the data transfer rate is an incredible 240 Mbytes/second.

This is fast enough to transfer the entire text of this Documentation in only 2 milliseconds! Just as important, dedicated hardware allows these data streams to be transferred directly into memory (Direct Memory Access, or DMA), without having to pass through the CPU's registers.

The main buses (program memory bus and data memory bus) are also accessible from outside the chip, providing an additional interface to off-chip

memory and peripherals. This allows the SHARC DSPs to use a four Gigaword (16 Gbyte) memory, accessible at 40 Mwords/second (160 Mbytes/second), for 32 bit data. This type of high speed I/O is a key characteristic of DSPs. The overriding goal is to move the data in, perform the math, and move the data out before the next sample is available.

Everything else is secondary. Some DSPs have on-board analog-to-digital and digital-to-analog converters, a feature called **mixed signal**. However, all DSPs can interface with external converters through serial or parallel ports.

At the top of the diagram are two blocks labeled **Data Address Generator** (DAG), one for each of the two memories. These control the addresses sent to the program and data memories, specifying where the information is to be read from or written to.

DSPs are designed to operate with *circular buffers*, and benefit from the extra hardware to manage them efficiently. This avoids needing to use precious CPU clock cycles to keep track of how the data are stored. For instance, in the SHARC DSPs, each of the two DAGs can control *eight* circular buffers. This means that each DAG holds 32 variables (4 per buffer), plus the required logic.

Why so many circular buffers? Some DSP algorithms are best carried out in stages. We also had to do this on the implementation of our project after finding problems with the precision of our coefficients. More about this in the next chapter.

SHARC DSPs are also designed to efficiently carry out the *Fast Fourier transform*. In this mode, the DAGs are configured to generate **bit-reversed addresses** into the circular buffers, a necessary part of the FFT algorithm. In addition, an abundance of circular buffers greatly simplifies DSP code generation- both for the human programmer as well as high-level language compilers, such as C.

The data register section of the CPU is used in the same way as in traditional microprocessors. In the ADSP-2106x SHARC DSPs, there are 16 general-purpose registers of 40 bits each. These can hold intermediate calculations, prepare data for the math processor, serve as a buffer for data transfer, hold flags for program control, and so on. If needed, these registers can also be used to control loops and counters; however, the SHARC DSPs have extra hardware registers to carry out many of these functions.

The math processing is broken into three sections, a **multiplier**, an **arithmetic logic unit (ALU)**, and a **barrel shifter**.

The multiplier takes the values from two registers, multiplies them, and places the result into another register. The ALU performs addition, subtraction, absolute value, logical operations (AND, OR, XOR, NOT), conversion between fixed and floating-point formats, and similar functions. Elementary binary operations are carried out by the barrel shifter, such as shifting, rotating, extracting and depositing segments, and so on. A powerful feature of the SHARC family is that the multiplier and the ALU can be accessed in parallel. In a single clock cycle, data from registers 0-7 can be passed to the multiplier, data from registers 8-15 can be passed to the ALU, and the two results returned to any of the 16 registers.

There are also many important features of the SHARC family architecture that aren't shown in this simplified illustration. For instance, an 80-bit accumulator is built into the multiplier to reduce the round-off error associated with multiple fixed-point math operations.

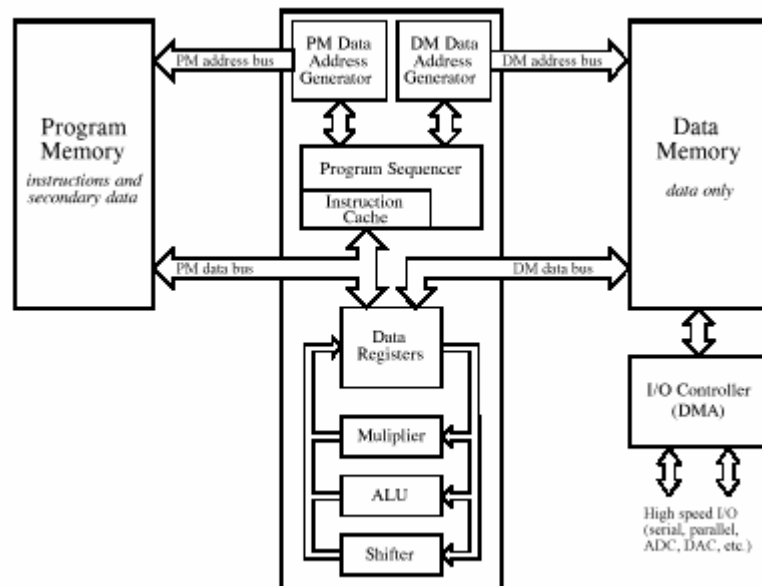


FIGURE 3.4 Typical DSP architecture. Digital Signal Processors are designed to implement tasks in parallel. This simplified diagram is of the Analog Devices SHARC DSP. Compare this architecture with the tasks needed to implement an FIR filter, as listed in Algorithm 3.1. All of the steps within the loop can be executed in a single clock cycle.

Another feature that was of interest is the use of **shadow registers** for all the CPU's key registers. These are duplicate registers that can be switched with their counterparts in a single clock cycle. They are used for *fast context switching*, the ability to handle interrupts quickly. When an interrupt occurs in traditional microprocessors, all the internal data must be saved before the interrupt can be handled. This usually involves pushing all of the occupied registers onto the stack, one at a time. In comparison, an interrupt in the SHARC family is handled by moving the internal data into the shadow registers in a *single clock cycle*. When the interrupt routine is completed, the registers are just as quickly restored.

1. Obtain a sample with the ADC; generate an interrupt
2. Detect and manage the interrupt
3. Move the sample into the input signal's circular buffer
4. Update the pointer for the input signal's circular buffer
5. Zero the accumulator
6. Control the loop through each of the coefficients
7. Fetch the coefficient from the coefficient's circular buffer
8. Update the pointer for the coefficient's circular buffer
9. Fetch the sample from the input signal's circular buffer
10. Update the pointer for the input signal's circular buffer
11. Multiply the coefficient by the sample
12. Add the product to the accumulator
13. Move the output sample (accumulator) to a holding buffer
14. Move the output sample from the holding buffer to the DAC

Algorithm 3.1 FIR filter steps.

We come to the critical performance of the architecture, how many of the operations within the loop can be carried out at the same time. Because of its highly parallel nature, the SHARC DSP can simultaneously carry out *all* of these tasks. Specifically, within a single clock cycle, it can perform a multiply (step 11), an addition (step 12), two data moves (steps 7 and 9), update two circular buffer pointers (steps 8 and 10), and control the loop (step 6).

There will be extra clock cycles associated with beginning and ending the loop (steps 3, 4, 5 and 13, plus moving initial values into place); however, these tasks are also handled very efficiently. If the loop is executed more than a few times, this overhead will be negligible. As an example, suppose you write an efficient FIR filter program using 100 coefficients. You can expect it to require about 105 to 110 clock cycles per sample to execute (i.e., 100 coefficient loops plus overhead). A traditional microprocessor requires many thousands of clock cycles for this algorithm.

3.3 C versus Assembly

DSPs are programmed in the same languages as other scientific and engineering applications, usually C. Programs written in assembly can execute faster, while programs written in C are easier to develop and maintain. In traditional applications, such as programs run on personal computers, C is Mostly the first choice.

If assembly is used at all, it is restricted to short subroutines that must run with the utmost speed. This is shown graphically in FIGURE. 3.5a; for every traditional programmer that works in assembly, there are approximately *ten* that use C.

However, DSP programs are different from traditional software tasks in two important respects. First, the programs are usually much shorter, say, one-hundred lines versus ten-thousand lines. Second, the execution speed is often a critical part of the application. These two factors motivate many software engineers to switch from C to assembly for programming Digital Signal Processors.

This is illustrated in (b); nearly as many DSP programmers use assembly as use C. Figure (c) takes this further by looking at the revenue produced by DSP products. For every dollar made with a DSP programmed in C, two dollars are made with a DSP programmed in assembly. The reason for this is simple; outperforming the competition makes money.

From a pure performance standpoint, such as execution speed and manufacturing cost, assembly almost always has the advantage over C. For instance, C code usually requires a larger memory than assembly, resulting in more expensive hardware.

However, the DSP market is continually changing. As the market grows, manufacturers are responding by designing DSPs that are *optimized* for programming in C. For instance, C is much more efficient when there is a large, general purpose register set and a unified memory space. These future improvements will minimize the difference in execution time between C and assembly, and allow C to be used in more applications.

To better understand this decision between C and assembly, look at a typical DSP task programmed in each language. The example used is the calculation of the *dot product* of the two arrays, and. $x [] y []$ This is a simple mathematical operation, each coefficient multiplied in one array by the corresponding coefficient in the other array, and sum the products. That is, each sample in the output signal is found by multiplying stored samples from the input signal (in one array) by the filter coefficients (in the other array), and summing the products.

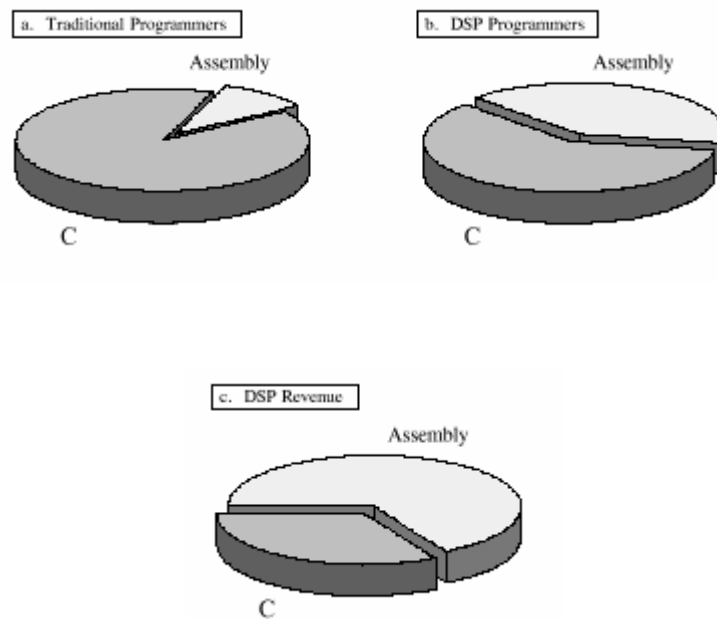


FIGURE 3.5 All the statistics (References to the statistics can be found in the references Bibliography)

3.4 Circular Buffering

Digital Signal Processors are designed to quickly carry out FIR filters and similar techniques. To understand the *hardware*, we went through the *algorithms* first. In this section a detailed list of the steps needed to implement an FIR filter is discussed.

As a starting point, a need to distinguish between **off-line processing** and **real-time processing** is apparent. In off-line processing, the *entire* input signal resides in the computer at the same time. For example, a geophysicist might use a seismometer to record the ground movement during an earthquake. After the shaking is over, the information may be read into a computer and analyzed in some way.

Another example of off-line processing is medical imaging, such as computed tomography and MRI. The data set is acquired while the patient is inside the machine, but the image reconstruction may be delayed until a later time. The key point is that *all* of the information is simultaneously available to the processing program.

In real-time processing, the output signal is produced at the same time that the input signal is being acquired. For example, this is needed in telephone communication, hearing aids, and radar. These applications must have the information immediately available, although it can be delayed by a short amount.

For instance, the speaker or listener cannot detect a 10-millisecond delay in a telephone call. Likewise, it makes no difference if a radar signal is delayed by a few seconds before being displayed to the operator. Real-time applications input a sample, perform the algorithm, and output a sample, over-and-over.

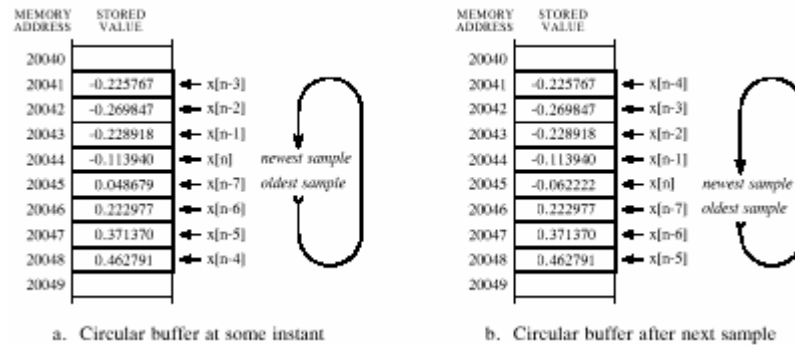


FIGURE 3.6 Circular buffer operation. Circular buffers are used to store the most recent values of a continually updated signal. This illustration shows how an eight sample circular buffer might appear at some instant in time (a), and how it would appear one sample later (b).

Alternatively, they may input a group of samples, perform the algorithm, and output a group of samples. This is the world of Digital Signal Processors.

Figure 4.6 illustrates an eight sample circular buffer. We have placed this circular buffer in eight consecutive memory locations, 20041 to 20048. Figure (a) shows how the eight samples from the input might be stored at one particular instant in time, while (b) shows the changes after the next sample is acquired. The idea of circular buffering is that the end of this linear array is connected to its beginning; memory location 20041 is viewed as being next to 20048, just as 20044 is next to 20045.

You keep track of the array by a **pointer** (a variable whose value is an *address*) that indicates where the most recent sample resides. For instance, in (a) the pointer contains the address 20044, while in (b) it contains 20045. When a new sample is acquired, it replaces the oldest sample in the array, and the pointer is moved one address ahead.

Circular buffers are efficient because only one value needs to be changed when a new sample is acquired.

Four parameters are needed to manage a circular buffer. First, there must be a pointer that indicates the start of the circular buffer in memory (in this example, 20041). Second, there must be a pointer indicating the end of the array (e.g., 20048), or a variable that holds its length (e.g., 8).

Third, the step size of the memory addressing must be specified. In FIGURE. 3.6 the step size is one, for example: address 20043 contains

one sample, address 20044 contains the next sample, and so on. This is frequently not the case. For example, the addressing may refer to bytes, and each sample may require two or four bytes to hold its value. In these cases, the step size would need to be two or four, respectively. These three values define the size and configuration of the circular buffer, and will not change during the program operation. The fourth value, the pointer to the most recent sample, must be modified as each new sample is acquired. In other words, there must be program logic that controls how this fourth value is updated based on the value of the first three values. While this logic is quite simple, it must be very fast. This is the whole point of this discussion; DSPs should be optimized at managing circular buffers to achieve the highest possible execution speed.

As an aside, circular buffering is also useful in *off-line* processing. Consider a program where both the input and the output signals are completely contained in memory. Circular buffering isn't needed for a convolution calculation, because every sample can be immediately accessed. However, many algorithms are implemented in *stages*, with an intermediate signal being created between each stage.

Circular buffering provides another option: store only those intermediate samples needed for the calculation at hand. This reduces the required amount of memory, at the expense of a more complicated algorithm. The important idea is that circular buffers are *useful* for off-line processing, but *critical* for real-time applications. Now we can look at the steps needed to implement an FIR filter using circular buffers for both the input signal and the coefficients.

This list (Page 28) may seem trivial and overexamined- it's not! The efficient handling of these individual tasks is what separates a DSP from a traditional microprocessor. For each new sample, all the following steps need to be taken: The goal is to make these steps execute quickly. Since steps 6-12 will be repeated many times (once for each coefficient in the filter), special attention must be given to these operations.

Traditional microprocessors must generally carry out these 14 steps in serial (one after another), while DSPs are designed to perform them in parallel. In some cases, all of the operations within the loop (steps 6-12) can be completed in a single clock cycle. Let's look at the internal architecture that allows this magnificent performance.

3.5 General Description of the Bittware Board

At this stage it would be appropriate to describe the Bittware board we used before we start discussing the implementation of the two methods of FIR filtering.

Single Processor ADSP-2106x Board for PCI

Features:

- ***One SHARC digital signal processor***
- ***Optimal external SRAM up to 512K * 48***
- ***EZ-ICE and ICEPAC in-circuit emulator compatible***
- ***SHARC-optimized high-performance***
- ***BITSI I/O mezzanine site***
- ***Two external link ports(up to 40 MB/s each)***
- ***Complete development tools available.***



4 Modulation

4.1 Introduction

It is no exaggeration to state that in the communication sciences, modulation holds a central place. The terms amplitude modulation and frequency modulation are used by every layperson at one time or another, even if their only associations with these words are popular music for the former and classical music for the latter. Nevertheless, the word modulation has a distinct technical meaning and was defined by an appropriate committee of the IRE [6-1] as the “process whereby some characteristic of a wave is varied in accordance with another wave.” In this case the *controlled* modulation, i.e. the desired and controlled shifting of the spectrum of a message wave which is usually baseband and contains information of interest to humans.

The term baseband refers generally to a low-pass wave such as simple speech or, as in more complicated systems, a multiplex wave consisting of many low- pass waves.

The main reason for modulating is the need for simultaneous transmission of different signals. The signals of interest to a human are primarily in a frequency band that spans from tens of hertz to several thousand hertz. If no modulation took place, only a one baseband signal in any locality at a given time; simultaneous transmission of more than one signal would cause the signals to overlap without hope of separation. Through use of modulation, however, messages can be transmitted over the same medium and still ensure their separability at the receiver. It is the multi-channel capability furnished by modulation, which enables to have many radio and television in the same locality at the same time.

There are three methods of modulation namely, Double-sideband (DSB), single-sideband (SSB), and the vestigial sideband (VSB). All of these

modulation methods are closely related; either both sidebands, or one sideband, or fractions of both, with or without a carrier signal.

The Single sideband modulation will be discussed since it was required for the project.

4.2 Single Sideband Modulation.

Since information in DSB is in fact duplicated in two sidebands, the transmission bandwidth of the modulated wave can be reduced by 50% by sending only one sideband rather than two. In SSB systems one of the sidebands of the DSB is removed, usually by direct filtering, and the remaining sideband is sent by itself or with a low-level carrier. Thus SSB is more “efficient” than DSB from the point of view of bandwidth conversation.

Single sideband in general practice, and as referred to in this book, actually is single sideband with suppressed carrier. Carrier-suppression techniques reduce the level of the carrier but do not eliminate it. This is a useful concept, since the carrier frequency is a reference around which the sidebands are produced in the transmitter and from which the nature of a single-sideband signal may be defined. To visualize carrier suppression, consider an SSB transmitter rated at 100 watts output. With 50 dB of carrier suppression, the output power contained in the carrier is 0.001 watt. This is a very small fraction of total output power, but it is finite.

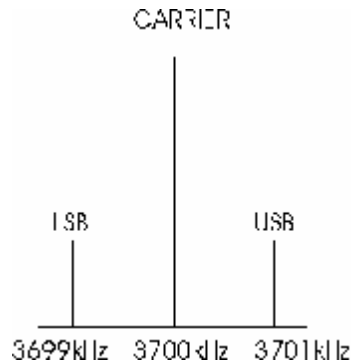


FIGURE 4.1

In essence, a single-sideband signal is an AM signal with the nonessential elements effectively removed. Figure 4.1 and Figure 4.2 illustrates the spectrum occupied by 3700 kHz AM and SSB signals, each modulated by a single 1 kHz tone. AM (Amplitude Modulation) is very easy to understand and is essentially a direct translation of the message spectrum.

The AM signal consists of a carrier and two identical sidebands which are spaced above and below the carrier by an amount equal to the frequency of the modulating tone.

It is the sidebands, which contain the intelligence of the signal.

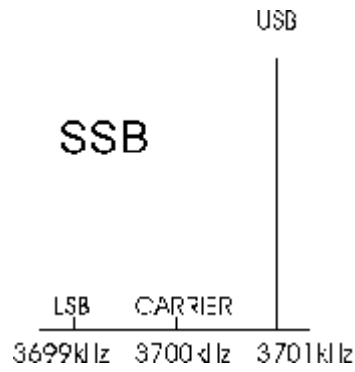


FIGURE 4.2

Removing one of the sidebands by filtering is not easy to do when there are significant signal components at low frequencies. In such a case, vestigial sideband (VSB) is preferred.

The type of signal spectrum that is conveniently filtered with a lowpass filter is shown in Fig 4.3.

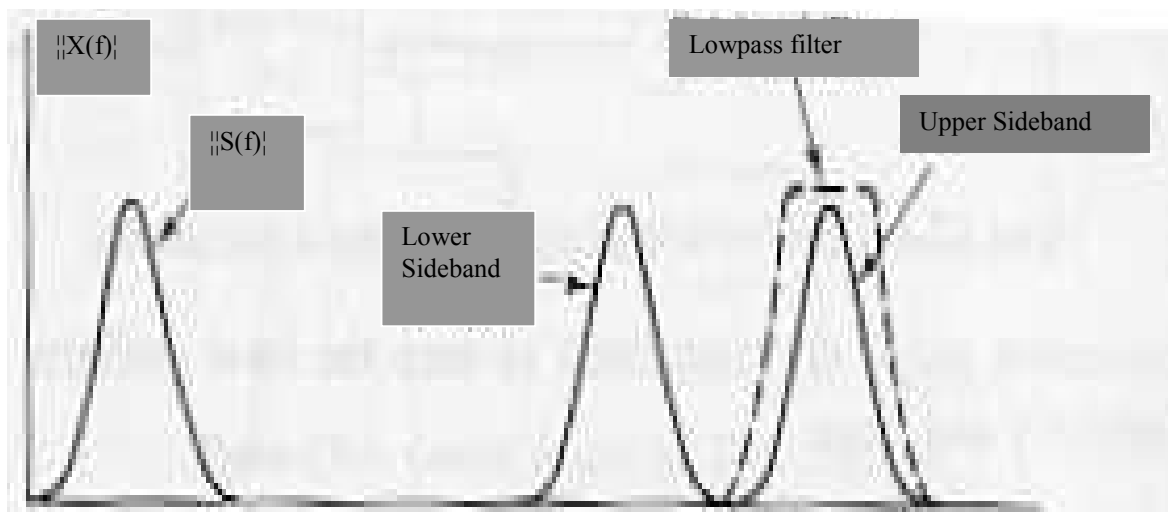


FIGURE 4.3 Sideband Filtering of Signals.



5. Project Implementation (Solution)

In the past three chapters we gave the necessary theory we had to learn before we came to realize the solution of the project. In this chapter we will give a detailed structure of the realized solution of our project. We will also show how the theory in the previous chapters was used to help make decisions.

We had two designs that were to be implemented and compared upon completion of both, thus it was decided to do the research and implementation of both of them concurrently so that the realized solutions can be compared immediately upon completion.

First to be mentioned before we start with the steps taken for implementation, is the specifications of the two realized methods.

5.1 Specifications of the Systems

5.1.1 The Weaver (Third) Method:

- ***Filter length (Number of coefficients): 171***
- ***LF bandwidth (audio signal): 20 . . . 20 000hz***
- ***Sideband suppression: -48 dB***
- ***Carrier frequency: 105 kHz(From the external Modulator Stage)***

5.1.2 The Phase Method:

- **Filter length (Number of coefficients): 251**
- **LF bandwidth (audio signal): 20 . . . 20 000hz**
- **Sideband suppression: -50 dB**
- **Carrier frequency: 105 kHz (From the external Modulator Stage)**

5.2 Tasks and Solutions to them

We will break the discussion of implementation down to tasks we had to do and solutions thereof. This is not in the order that the task list was given to us.

Task 1. Designing of Coefficients for the Lowpass

The MATLAB SIGNAL PROSSESING TOOLBOX contains a lot of techniques for FIR filter design. We had to familiarize ourselves with these techniques before we used them for design of coefficients for both the lowpass and hilbert transform.

After some simulation and practice with the TOOLBOX we used the window method that we explained in the past chapters. We used the fir1 to do the windowing of a simple sinc function. The command:

```
L = fir1 (n , Wn, `filtertype`, window);
```

returns a row vector L containing n+1 coefficients where:

n = the order of the filter

Wn = is the cutoff frequency, its number between 0 and 1 corresponds to half the sampling frequency which is The Nyquist frequency.

filtertype =lowpass is the default type. You can give the name if you want to do a bandpass or any other filter.

Window = Hamming window.

The cutoff frequency for our lowpass is 11kHz after we decided to use the sampling frequency of a normal CD player (44kHz). We designed our coefficients to work at 48 kHz so as to make a 4 kHz designing allowance. The final command for the generation of lowpass coefficients was:

```
L = fir1 (170 ,0.5, Hamming);
```

And it plotted the following impulse response curve.

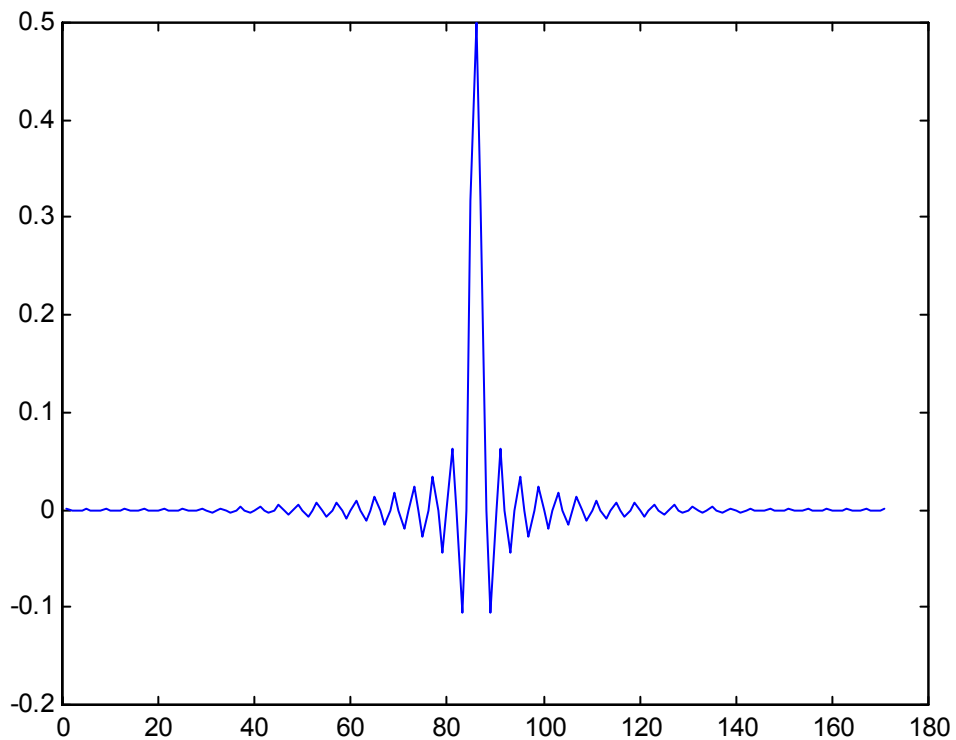
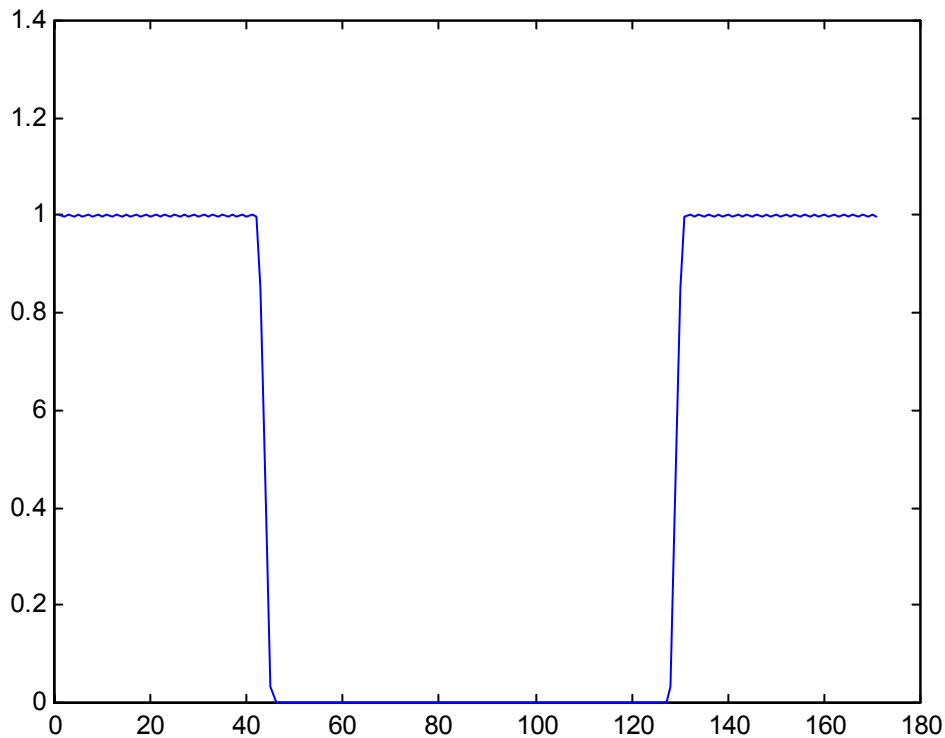


FIGURE 5.1 The Plot of the Impulse Response of the Lowpass

This is the sinc function after multiplication with the Hamming window. To see how accurate the approximation compares with a perfect step function we look at the plot of the function in the frequency domain.

The command for this involves taking the FFT of the function and plotting it.

```
L=FIR1(170,0.5,hamming(171));
Hb=fft(L);
plot(abs(Hb))
```



This is the best approximation to a perfect step function using 171 coefficients.

On the Weaver method we used two sets of lowpass coefficients. These were used concurrently with only the sine wave generated with a 90° phase shift to integrate the realized outputs. The two output signals are thus the convolution of the input signal with the coefficients plus the generated signal to give the two outputs the 90° phase difference needed.

Task 2. Designing of Coefficients for the Hilbert Transform for the Phase Method.

A Hilbert Transformer is a 90° phase shifter. It is related to the actual data by a 90° phase shift .

The remez function was used in this instance to realize the hilbert transformer. We deviated from the windowing method when we used this function.

REMEZ is a Parks-McClellan optimal equiripple FIR filter design tool.

`B=REMEZ(N,F,A)`

Returns a length $N+1$ linear phase (real, symmetric coefficients) FIR filter which has the best approximation to the desired frequency response described by F and A in the minimax sense. F is a vector of frequency band edges in pairs, in ascending order between 0 and 1. 1 corresponds to the Nyquist frequency or half the sampling frequency. A is a real vector the same size as F which specifies the desired amplitude of the frequency response of the resultant filter B .

The desired response is the line connecting the points $(F(k),A(k))$ and $(F(k+1),A(k+1))$ for odd k ; $REMEZ$ treats the bands between $F(k+1)$ and $F(k+2)$ for odd k as "transition bands" regions. Thus the desired amplitude is piecewise linear with transition bands. The maximum error is minimized.

`B=REMEZ(N,F,A,W,'Hilbert')`

Design filters that have odd symmetry, that is, $B(k) = -B(N+2-k)$ for $k = 1, \dots, N+1$. A special case is a Hilbert transformer, which has an approximate amplitude of 1 across the entire band.

The exact command for our final Hilbert Transformer was:

```
B = remez(250,[.05 .95],[1 1],'hilbert')
```

The impulse response plot is the following:

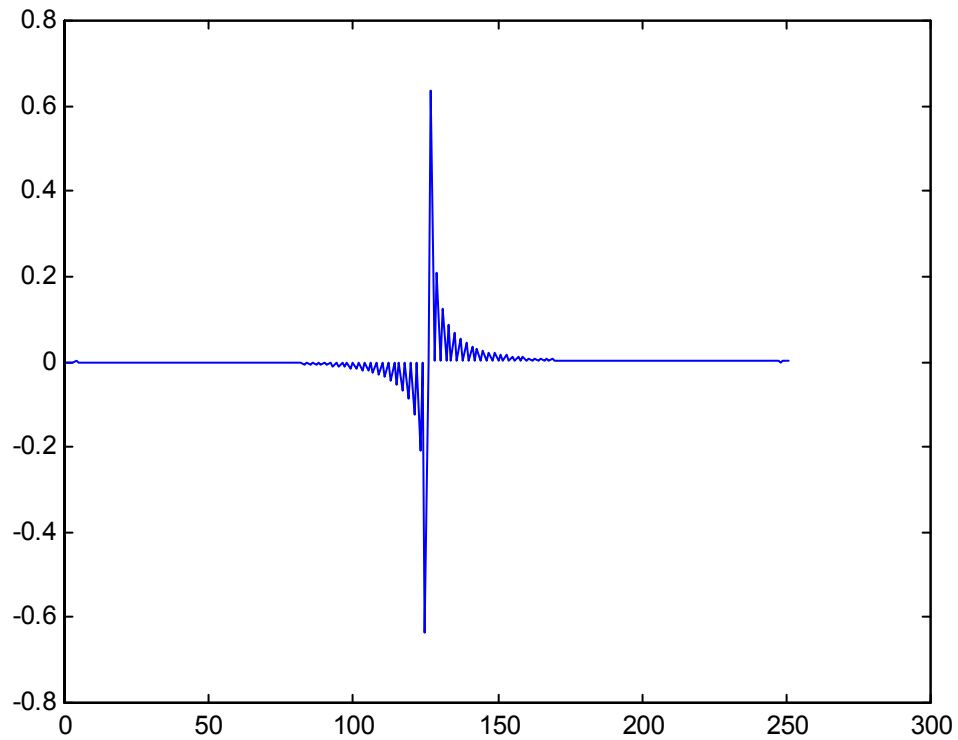


FIGURE 5.3 Impulse Response of the Hilbert Transformer

Again to check the accuracy of the coefficients against the ideal step function we plot the frequency domain. For this we need to transform this to the frequency domain and get the following.

```
B = remez(250, [.05 .95], [1 1], 'hilbert');
Hb=fft(B);
plot(abs(Hb)) %this is the Freq domain plot
```

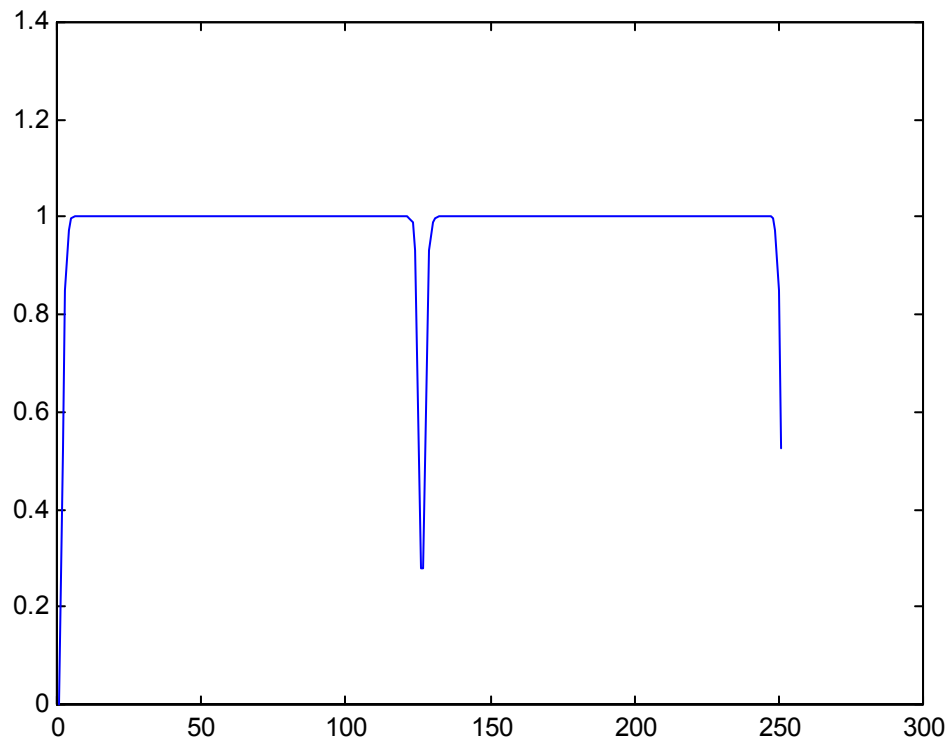


FIGURE 5.4 Frequency domain plot of the Hilbert transform.

Having the coefficients at hand prompted us to go on and do the implementation of what we were simulating on Matlab. We started the implementation with the Weaver method. The first thing to do was to get all the components of each method and then assemble them to form a system. We had the two lowpasses, we needed to generate a Sine wave and have the 90° shifted wave on the other side (refer to the Diagram on the project Definition). The 90° shift could be omitted by simply generating a sine and a cosine wave.

Task 3. Settling into the DSP

The first thing was to learn more about the Matlab interface to the DSP.

Architecture files

The Architecture file with the extension ACH determines the paging for the processor-own and the external MEMORY.

This means that we could pick out from this file, where the program memory, run time header and where the c-code is placed. Besides also the memory adjustment (32-bit or 48-bit) is determined by this file. The file in use for our project is SHARC6XC.ACH.

The G-21K Compiler

G21K is a driver program: it controls the operation of other programs (software development tools) during compilation. Other files involved with compilation are the runtime header and the architecture file. The runtime header controls initialization of the C runtime environment and interrupt handling.

Below is the G21K command line syntax:

```
g21k [-switches] filename [. ext ] [filename [.ext ]]
```

Some commonly used switch options are shown on the next list.

Switch	Effect
-E	Preprocess source files only
-S	Generate assembly source files only
-c	Generate object files only
-O	Optimize code using some optimizations
-O2	Optimize code using more optimizations
-O3	Optimize code using all optimizations
-v	Generate verbose output
-ansi	Disable all non-ANSI language extensions
-g	Produce debuggable code for use with CBUG
-h	Display list of switches
-a	filename Specify alternate architecture file
-Ipath	Specify additional paths to search for include files
-Dmacro[=value]	Define a macro for the C preprocessor
-Lpath	Specify an additional path to search for library files
-lxxx	Include library libxxx.a in link line
-map	Generate map file (default is 21k.map)
-o	filename Place output in filename
-nomem	Do not execute runtime memory initializer
-runhdr	filename Specify alternate runtime header file
-w	Inhibit all warning messages
-Wall	Combine all warnings in this list
-Wimplicit	Warn when a function is implicitly declared
-Wreturn-type	Warn when a function defaults to returning an int
-Wunused	Warn when an automatic variable is unused
-Wswitch	Warn when a switch does not use all enumeration types
-Wcomment	Warn when a comment contains a /* sequence
-Wfloat-convert	Warn when a float number is implicitly converted to a double , or a double is implicitly converted to a float

With reference to the make.bat file, the following command line is found in this file.

Task 4. Generating Sine and Cosine Waves for LF Mixers.

Though a sine and cosine waves were used in the weaver method, only one of the waves will be discussed. Both are generated using the same technique.

To generate the sine wave, the function has to be sampled since it represents a real signal in continuous-time (i.e. in analog form). Without getting into detail, sampling is when continuous-time signal is converted into a discrete-time signal.

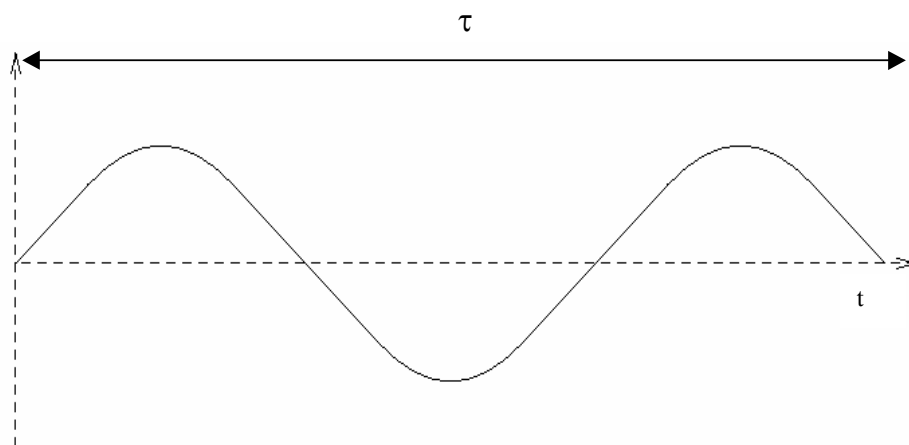


FIGURE 5.5 Analog sine wave

After the sampling, the same sine wave can be seen as

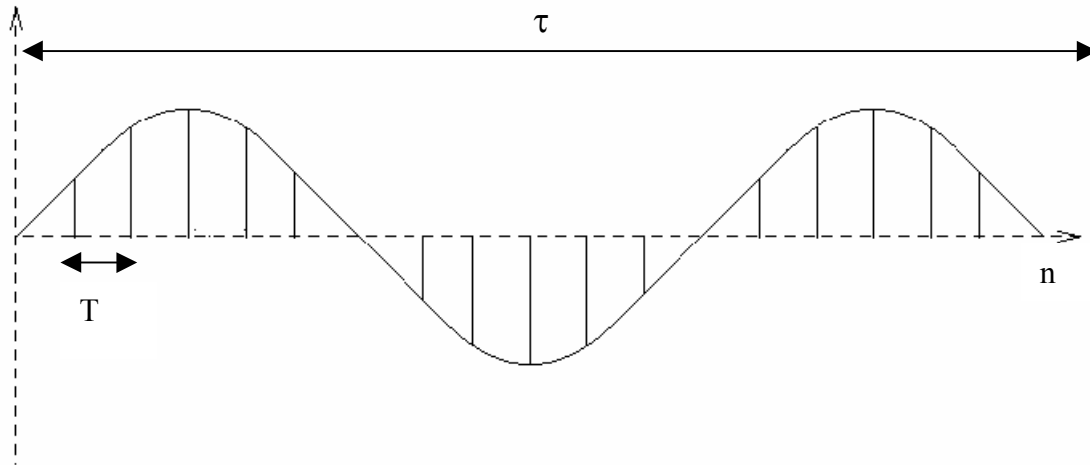


FIGURE 5.6 Sampled Sine wave

The function for fig 5.5, is

$$Y = \sin(x)$$

Where $x = 2\pi nT$;

and $f = 0.25 * f_c$ (sampling frequency)

For the implementation of this method f has to be a quarter of the sampling frequency.

For the function to be sampled, an array of the sine values was generated and stored as values of the sine wave in memory. Typically, one makes use of the symmetry properties of the sine wave to minimise the memory storage requirements.

The length of the function (func_length) was determined by dividing the sampling frequency with the frequency of the sine wave, since the signal is only needed for one period. This length can also determine the size of the array and also the number of discrete points.

$$\text{Func_length} = f_c/f$$

and

$$\text{Sinf}(x) = \sin (n 2*\pi fT) ;$$

where

$\tau/T = n$ (number of discrete points.) $\tau = 1/f$ and $T = 1/fC$

therefore ;

$\sin f(x) = \sin (n*2\pi f/fC)$

but $fC/f = \text{func_length}$

therefore

$\sin f(x) = \sin(n*(2*\pi)/(\text{func_length}))$

These are the formulae used in the generation of the sine wave. The c code can be seen on the Appendix.

LF mixers are used in the multiplication operations between the generated waves and the input signal. With all this at hand we had the nessesary tools to implement both Filters.

Task 5. The Weaver method

The weaver method was the first to be implemented. We used linear buffers to perform the convolution. This was fairly easy to implement but we were not utilising the full power of the DSP at this point. We already discussed the architecture of the DSP in the past chapter and it was clear that the fact is that DSPs come with the ability to do the multiplication and addition operations needed for the convolution algorithm. We were not using this advantage at this point. The linear buffer method performs these operations like the name suggests (linearly). It was also explained in the theory that DSPs are designed to accommodate the circular buffer algorithms because of mathematical capabilities witch are unique to them.

This is the reason why it was decided to try circular buffers when we could not get the accuracy we needed using linear buffers. The source code for linear and circular buffer algorithms can be found in the appendix. We could not accommodate more than 95 lowpass filter coefficients even after optimization. Double the amount was comfortably accommodated after the circular buffers were introduced into the fray of things.

After experimenting with different input signals and looking at the results on the Frequency spectrum analyser we discovered that after modulation we do get a single sideband modulation.

The problem came when we wanted to demodulate. We were not able to get an output signal similar to the input signal. After some investigation into the theory of this type of filter and after consultation with Prof. Shüeli we found out that we would need additional hardware for demodulation. We would need an extra frequency generator to generate some frequency that would compensate the frequency that is suppressed even when we are acting in the inverse side of the cut-off frequency.

Task 6. The Phase method

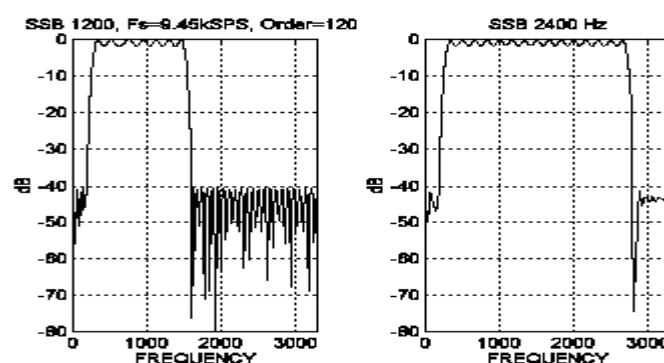
With the coefficients of the Hilbert transformer at hand, what we had to do was to create a delay routine. In this routine what happens is that every clock cycle the value from the coefficients is stored in an array, and then shifted.

This method was also first implemented using the linear buffer operation. Although it has a lesser amount of operations that it does in a complete cycle, it did not accept a satisfactory number of coefficients to make a accurate enough filter. It was also decided to implement this method using circular buffers.

After circular buffers were used, this method accepted 251 coefficients comfortably and the filter for this method was of a more accurate. With the phase method it is possible to demodulate after the modulation stage without employing the use of extra hardware. A comparison between the two methods is in the next section and it will show how this method compares to the former method

Reaction to increased frequency.

Both the weaver and the Phase method have the same reaction to frequency increase and decrease. The cut-off frequency of both is 11kHz and an increase over this cut-off



frequency causes the system to change polarity about the carrier frequency spectrum. An example of a suppressed signal, more specifically an example of single sideband modulation as seen of the frequency spectrum analyser is seen in the following diagram.

FIGURE 5.7 Single sideband Modulated signal

5.3 Comparison of Both Methods

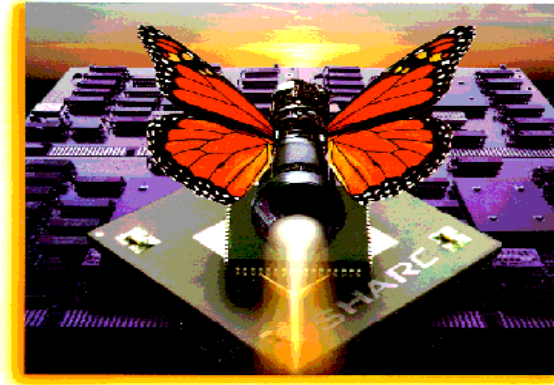
It is difficult to compare both methods because they both have advantages and disadvantages. We will use only the criterion given to us for comparison, i.e. the required computing power of the DSP for the required sideband suppression.

While the Phase method can take more coefficients because it has less tasks do perform in one cycle, this is at the expense of the sideband suppression after modulation. It is really a trade-off between the required sideband suppression and the accuracy of the filter because it takes more coefficients, thus doing a better approximation of the filters impulse response, but it requires more coefficients than the weaver method to the same amount of suppression.

The weaver method on the other hand has a mammoth of a task in a single cycle because it has a lost more operations to perform. This leads to it being able to accept less coefficients and thus not a good approximation to the impulse response.

The conclusion is that it really depends on what you want to use the filter for.

There are advantages to using both filters. It depends on what you want to trade-off.



6. Hardware Setup

The following equipments were needed to implement the Single sideband Modulation of Audio Signals using the DSP (refer to Fig 6.1 on next page).

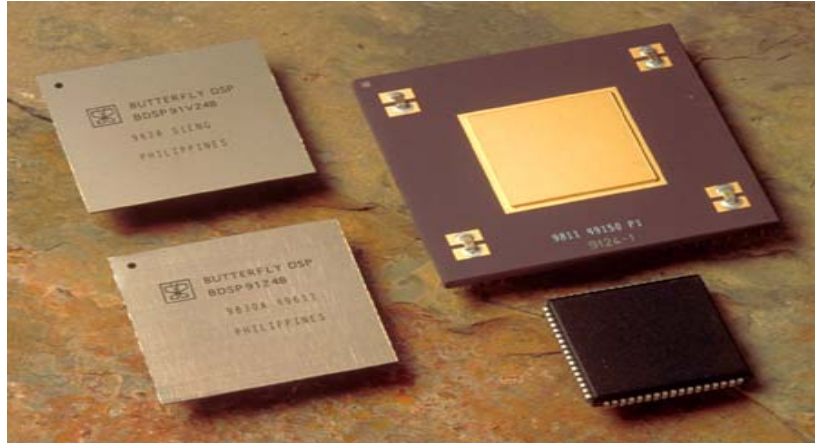
- **Pentium® II class computer (We used a 450 MHz Pentium® Processor)**
- **32 Meg of RAM**
- **A minimum space of 2 Mbytes of hard disk space**
- **Monitor**
- **A Bittware board with SHARC DSP must also be installed on the PC.**
- **Frequency generator (An audio source can also be used)**
- **Oscilloscope to view matched output signal**
- **Frequency Spectrum Analyser**
- **Modulator Hardware**

6.1 Software Setting up and Running

All the files needed to run both methods are in the disk provided with this report. The files should be copied into one directory as they are on the disk. Map the directories on the Matlab Path browser. The commands for compiling and running the project are:

MK_Weaver1 for the weaver method and

MK_Phase1 for the phase method.



7. Conclusion

After spending time on this project we learned a lot about the diverse field of digital signal Processing. Having started with no knowledge about this field, a lot of things were a blur to us when the project was handed to us. After consultation with Prof. Shüeli and doing our research in the field of filter design a lot seemed clearer to us.

We came to appreciate how valuable a tool the DSP is for signal processing. It is a known fact that technology in this field has the shortest half life and changes occur at a very high speed, but the fundamentals and the basic operations of them are constant. We also adopted interesting design techniques used in the design of filters in industry. We also learned about real-time applications of these devices.

It was intriguing to implement theoretical principles on hardware. We had the experience shared by a lot of people in industry, i.e. what works in principle and theory might not work in real-time. We also learned that there are many ways to implement the same thing. With each method having its own advantages, its up to the designer to decide which method to use. It also depends on the type of application you are working on and what trade-offs you are willing to do. It is our believe that the final working programs in this project were the best for this kind of operation and they are in accordance with the specifications as set by our project sponsor.

We also learned how powerful a tool Matlab is to an engineer. It took some “getting used to” since it was a totally new package for us to work with.

After some time on Matlab it was apparent that critical timing and interrupt control are important for the proper functioning of your filter. It was a big moment of joy to us when we saw the weaver method working for the first time. Though there was a lot that had to be done on it at that point, seeing it in practice made us see the light at the end of the tunnel. We owe a lot to Prof Shüeli and Andre Rüegg for their guidance throughout the project. The project Single sideband Modulation of Audio Signals using the DSP was a success!

8.1 Reference books

- **Title: Communication Systems**
Author: A.B Carlson
- **Title: Handbook for Digital Signal Processing.**
Authors: Mitra S.K and Kaiser J.F.
- **Title: A Simple Approach to Digital Signal Processing.**
Authors: Craig Marven and Gillian Ewers
- **Title: Advanced Digital Signal Processing.**
Authors: Glenn Zelnikcer and Fred J. Taylor
- **Title: Modern Electrical Communications.**
Authors: Henry Stark and Franz B. Tuteur

8.2 User Manuals

- **Title: ADSP-2106x SHARC Users Manual.**
Supplier: Analog Devices
- **Title: C-Tools Manual.**
Supplier: Analog Devices
- **Title: ADSP-2106x SHARC Compactor Manual.**
Supplier: Analog Devices

8.3 Internet Resources

- http://grove.circa.ufl.edu/matlab_help/toolbox/signal/remez.html
- <http://www.cs.york.ac.uk/~fisher/mkfilter/hilbert.html>
- <http://www.cs.york.ac.uk/~fisher/>

- <http://www.bores.com/courses.htm>
- <http://www.dsp.rice.edu/>
- http://icebear.cmsa.wmin.ac.uk/filter_details.html
- <http://www.peak.org/~forrerj/ASP/article.html>

The Weaver Method Using Circular Buffers

```

/*****
*
*   File Name:          Weaver1.c
*   Type :             C file
*   Description:       This file implements the weaver method using circular
*                   buffers
*   Authors:          N Shiburi & E.L Mabitsele
*   Date :            07/05/99, HSR Switzerland
*   Version:          1.1
*
*
\*****/

/*-----*/

/* ADSP-21060 System Register bit definitions */
#include <def21060.h>          /* register definitions*/
#include <21060.h>            /* architecture */
#include <signal.h>          /* interrupt */
#include <sport.h>           /* serial port */
#include <math.h>            /* fmod */
#include "bitsibb.h"         /* bitsi board */
#include "Wcoeff1.h"         /* include the coeffienncy file for the low pass*/
#include <macros.h>         /* for the CIRUCULAR_BUFFER*/

#define SetIOP(addr, val)    (* (int *) addr) = (val)
#define GetIOP(addr)        (* (int *) addr)
#define N 151
#define MAX_U_AD  2.75      /* maximum input voltage for A/D converter */
#define MAX_U_DA  3.0      /* maximum output voltage on D/A converter */
#define pi 3.1415          /* for the sin/cos functions */

/* Convert a 16 bit Integer value (the 16 right bits in a 32 bit word x)
   to a 32 bit Integer value */
#define convertleft16to32(result, x) \
asm("%0=fext %1 by 16:16 (se);" : "=d" (result) : "d" (x));

/* a DM pointer to a circular buffer #2/
CIRCULAR_BUFFER(float,1,Ptr1);
CIRCULAR_BUFFER(float,2,Ptr2);

/*-----*/
/* pointer for serial port (sport) transfer */
int *dms3;

/* set correction terms for input / output voltages */
float NORM = MAX_U_AD / 32767.0;      /* normalization of sampled input values */
float CORR = 32767.0 / MAX_U_DA;      /* denormalization for output values */

/* A/D, D/A converters */

```

```

unsigned long AD_IN;          /* 32-bit A/D input value (CH1IN | CH2IN) */
long temp;                   /* temporary variable to separate channels */

/* matlab interface */
float insample1;             /* input sample value1 */
float insample2;            /* input sample value2 */
float dm dly1[N];           /*init for N sample1 */
float dm dly2[N];           /*init for N sample2 */
float fc=44000.0;           /* sampling frequency */
register float outsample1 asm("r13"); /*output sample value1*/
register float outsample2 asm("r14"); /*output sample value2*/
unsigned outsample;         /*output unsigned sample */

/* Array for sin */
int f = 11000;               /* Frequency of the sin [Hz]*/
+int count1 = 0;            /* Actual Position in the 1st Array func */
int count2 = 0;            /* Actual Position in the 2nd Array func */
const int func_length = 4; /* Length of the Array func, func_length = fc/f*/
float func1[4];            /* Array with the sin-values */
float func2[4];            /* Array with the cos-values */

/*-----*/

/* sample transmit irq */
void spt0_asserted( int sig_num )
{
    float sin_insample1;    /*the values after the input is multiplied by sin
                             function*/
    float cos_insample2;    /*the values after the input is multiplied by cos
                             function*/
    int index;
    float buffer_value1;    /*the variable to read the values to read the
                             circular buffer/
    float buffer_value2;
    convertleft16to32(temp, GetIOP(RX0)); /* Convert 16 bit int to 32 bit int */
    insample1 = NORM * ((float)temp); /* normalize input signal from CH1IN */

    sin_insample1 = func1[count1] * insample1; /*the input signal is multiplied
with the sin wave*/
    cos_insample2 = func2[count1] * insample1; /*the input signal is multiplied
with the cos wave*/

    CIRC_WRITE(Ptr1,0,sin_insample1,dm); /*write into the circular buffer*/
    CIRC_WRITE(Ptr2,0,cos_insample2,dm); /*write into the circular buffer*/

    outsample1 = 0.0;      /*reset the values*/
    outsample2 = 0.0;      /*reset the values*/

    for (index = 0;index<N;index+=2)
    {
        CIRC_READ(Ptr1,2,buffer_value1,dm);
        CIRC_READ(Ptr2,2,buffer_value2,dm);
        outsample1 += h[index]*buffer_value1; /*perform convolution */
        outsample2 += h[index]*buffer_value2; /*perform convolution */
    }

    /*move the pointer to the value needed*/
    CIRC_MODIFY(Ptr1, (N-1)/2-1);
    CIRC_MODIFY(Ptr2, (N-1)/2-1);

    /* read the value, where the pointer is pointing */
    CIRC_READ(Ptr1,0,buffer_value1,dm);
    CIRC_READ(Ptr2,0,buffer_value2,dm);

```

```

/*move the pointer to the value needed*/
CIRC_MODIFY(Ptr1, (N-1)/2);
CIRC_MODIFY(Ptr2, (N-1)/2);

/*perform convolution for the middle value which is not zero*/
outsample1 += h[(N-1)/2]*buffer_value1;
outsample2 += h[(N-1)/2]*buffer_value2;

/* a loop to check if the array has reached the last point */
if(count1 == func_length-1)
    count1 = 0;
else
    count1++;
    /* mask the values and shift*/
    outsample = ((unsigned)(CORR * outsample1) << 16) & 0xffff0000;
    outsample = outsample | (((unsigned)(CORR * outsample2)) & 0xffff);

SetIOP(TX0,outsample);    /*give the value to the DAC*/
}
/*-----*/

void setup_sport0( void )
{
    /* Configure SHARC serial port */

    /* TRANSMIT CONTROL REGISTER */
    /* An alternate (and more efficient) way of doing this would be to */
    /* write the 32-bit register all at once with a statement like this: */
    /*     SetIOP(STCTL0, 0x001c00f2); */
    /* But the following is more descriptive... */

    sport0_iop.txc.mdf = 0;    /* multichannel frame delay (MFD) */
    sport0_iop.txc.schen = 0; /* Tx DMA chaining enable */
    sport0_iop.txc.sden = 0; /* Tx DMA enable */
    sport0_iop.txc.lafs = 0; /* Late TFS (alternate) */
    sport0_iop.txc.ltfs = 1; /* Active low TFS */
    sport0_iop.txc.ditfs = 0; /* Data independent TFS */
    sport0_iop.txc.itfs = 0; /* Internally generated TFS */
    sport0_iop.txc.tfsr = 1; /* TFS Required */

    sport0_iop.txc.ckre = 0; /* Data and FS on clock rising edge */
    sport0_iop.txc.gclk = 0; /* Enable clock only during transmission*/
    sport0_iop.txc.iclk = 0; /* Internally generated Tx clock */
    sport0_iop.txc.pack = 0; /* Unpack 32b words into two 16b tx's */

    sport0_iop.txc.slen = 31; /* Data word length minus one */
    sport0_iop.txc.sendn = 0; /* Data word endian 1 = LSB first */
    sport0_iop.txc.dtype = SPORT_DTYPE_RIGHT_JUSTIFY_SIGN_EXTEND;
    /* Data type specifier */
    sport0_iop.txc.spen = 1; /* Enable (clear for MC operation) */

    /* RECEIVE CONTROL REGISTER */
    sport0_iop.rxc.nch = 31; /* multichannel number of channels - 1 */
    sport0_iop.rxc.mce = 0; /* multichannel enable */
    sport0_iop.rxc.spl = 0; /* Loop back configure (test) */
    sport0_iop.rxc.d2dma = 0; /* Enable 2-dimensional DMA array */
    sport0_iop.rxc.schen = 0; /* Rx DMA chaining enable */
    sport0_iop.rxc.sden = 0; /* Rx DMA enable */
    sport0_iop.rxc.lafs = 0; /* Late RFS (alternate) */
    sport0_iop.rxc.ltfs = 1; /* Active low RFS */
    sport0_iop.rxc.irfs = 0; /* Internally generated RFS */
    sport0_iop.rxc.rfsr = 1; /* RFS Required */
    sport0_iop.rxc.ckre = 0; /* Data and FS on clock rising edge */
    sport0_iop.rxc.gclk = 0; /* Enable clock only during transmission*/
    sport0_iop.rxc.iclk = 0; /* Internally generated Rx clock */
    sport0_iop.rxc.pack = 0; /* Pack two 16b rx's into 32b word */

    sport0_iop.rxc.slen = 31; /* Data word length minus one */
    sport0_iop.rxc.sendn = 0; /* Data word endian 1 = LSB first */
    sport0_iop.rxc.dtype = SPORT_DTYPE_RIGHT_JUSTIFY_SIGN_EXTEND;
    /* Data type specifier */
    sport0_iop.rxc.spen = 1; /* Enable (clear for MC operation) */
}

```

```

    /* Enable sport0 xmit irqs */
    interruptf(SIG_SPT0I, spt0_asserted);
}
/*-----*/

void setup_bitsibb( void )
{
    BITSIBB_TIM0 = BB_TIM(fc); /* set sampling frequency */
    BITSIBB_CTL = 1; /* enable just one DAC */
    BITSIBB_CLKSEL = 0x0844; /* Ch 1-2 Tim 0, Ch 3-4 Tim 1 */
}
/*-----*/

void init_data( void )
{
    /* function to initialise the input samples */
    insample1 = 0.0;
    insample2 = 0.0;
}
/*-----*/
/* Generates a sine wave to be multiplied with the input signal */

int sine_wave()
{
    int n = 0;
    for(n= 0; n < func_length; n++){

        func1[n] = (sin((float) n*(2*pi)/(func_length))); //the funtion of f(x) goes here!
        func2[n] = (cos((float) n*(2*pi)/(func_length))); //the funtion of f(x) goes here!

    }

}
/*-----*/

void main ( void )
{
    int t, msize;

    /* get base location of dms3 (BITSI) */
    t = GetIOP(SYSCON);
    msize = (t & 0xF000) >> 12;
    dms3 = (int *) (0x400000 + (3 * (0x2000 << msize)));

    /* declarations for the first circular buffer for the filter */
    BASE(Ptr1)= dly1;
    LENGTH(Ptr1)= N;

    /* declarations for the first circular buffer for the filter */
    BASE(Ptr2)= dly2;
    LENGTH(Ptr2)= N;

    /* set IOP registers */
    SetIOP(WAIT,0x21A4E421);

    /* initialize data buffers */
    init_data();

    /* call up the sine_wave function */
    sine_wave();

    /* initialize hardware */
    setup_bitsibb();
}

```

```
setup_sport0();

/* do forever */
while(1)
{}
}
```

Summary

This is the final Solution to the implementation of the Weaver method. Circular buffers were used. To compile this file, type this command from the Matlab command window: MK_Weaver1 (this is the make file for this program). This file can be obtained from the disk containing all the project files. This program accepted 171 lowpass filter coefficients without being unstable. This was a more accurate filter compared to the earlier one used with less than 101 coefficients.

Header File For Weaver1 (Circular Buffer Solution)

```
/******
*
*   File Name:          Wcoeff1.h
*
******
```

```

*   Type :                Header file                                *
*   Description:          This file contains lowpass coefficients for the weaver *
*                           method using circular buffers                *
*   Author:              E.L Mabitsela                               *
*   Date :               26/04/99, HSR Switzerland                  *
*                                                                    *
*                                                                    *
*                                                                    *
*                                                                    *

```

```

\*****/
/*-----*/

```

```

#define N 171
const dm float h[N]= {

    2.994970971692274e-004,  5.969682170532792e-019,  -3.115299938741654e-004,
    5.039497039600437e-018,  3.340001847825552e-004,  -1.711365178113277e-018,
    -3.676176196053604e-004,  3.238466400552031e-018,  4.131099206142489e-004,
    -5.180304591591382e-018,  -4.712255295790682e-004,  4.943085977436705e-019,
    5.427376906950164e-004,  -2.573585641777258e-018,  -6.284494279917497e-004,
    5.306035247610863e-018,  7.291997106627280e-004,  2.310722132893812e-018,
    -8.458710441345732e-004,  4.447515773083571e-019,  9.793987814265138e-004,
    -4.125623586554306e-018,  -1.130782522359557e-003,  8.918030712607691e-018,
    1.301100062811466e-003,  -1.502326565091717e-017,  -1.491524479818111e-003,
    -2.161186910179802e-021,  1.703345101068307e-003,  -6.232775837302540e-018,
    -1.937993323470929e-003,  1.415983651999831e-017,  2.197074535144405e-003,
    9.161828106876178e-018,  -2.482407787237603e-003,  -1.294580160638788e-018,
    2.796075398236710e-003,  -8.712160203777609e-018,  -3.140485415208815e-003,
    2.117845637463755e-017,  3.518450895142319e-003,  -3.646447325176706e-017,
    -3.933291443272451e-003,  -4.076955734546335e-018,  4.388964566236889e-003,
    -1.134858803825008e-017,  -4.890237498007835e-003,  3.033651221415588e-017,
    5.442914765063811e-003,  -1.265417735111213e-017,  -6.054143736150291e-003,
    -9.390727217850135e-018,  6.732831187820820e-003,  -1.391324070807570e-017,
    -7.490220965849215e-003,  4.259298606514225e-017,  8.340710453905035e-003,
    -1.509830939677421e-017,  -9.303029613400017e-003,  -1.928094232260260e-017,
    1.040198556910602e-002,  -1.618352903048081e-017,  -1.167111691720245e-002,
    6.067568113847771e-017,  1.315686416521343e-002,  -1.714522360765993e-017,
    -1.492537337760950e-002,  1.757307417751629e-017,  1.707410146901768e-002,
    -1.796241204616114e-017,  -1.975270459745551e-002,  1.831111078516723e-017,

    2.320320675089511e-002,  -1.861726592294094e-017,  -2.784398685613123e-002,
    1.887920534628564e-017,  3.446574644658771e-002,  -1.909549843290416e-017,
    -4.476341888239694e-002,  1.926496386496171e-017,  6.314465631785622e-002,
    -1.938667608103415e-017,  -1.057722570251323e-001,  1.945997033120366e-017,
    3.181156975481291e-001,  4.998519988766039e-001,  3.181156975481291e-001,
    1.945997033120366e-017,  -1.057722570251323e-001,  -1.938667608103415e-017,
    6.314465631785622e-002,  1.926496386496171e-017,  -4.476341888239694e-002,

```

```

-1.909549843290416e-017, 3.446574644658771e-002, 1.887920534628564e-017,
-2.784398685613123e-002, -1.861726592294094e-017, 2.320320675089512e-002,
1.831111078516723e-017, -1.975270459745551e-002, -1.796241204616114e-017,
1.707410146901768e-002, 1.757307417751629e-017, -1.492537337760950e-002,
-1.714522360765993e-017, 1.315686416521343e-002, 6.067568113847771e-017,
-1.167111691720245e-002, -1.618352903048082e-017, 1.040198556910602e-002,
-1.928094232260260e-017, -9.303029613400017e-003, -1.509830939677421e-017,
8.340710453905038e-003, 4.259298606514225e-017, -7.490220965849215e-003,
-1.391324070807570e-017, 6.732831187820820e-003, -9.390727217850139e-018,
-6.054143736150293e-003, -1.265417735111213e-017, 5.442914765063813e-003,
3.033651221415590e-017, -4.890237498007839e-003, -1.134858803825009e-017,
4.388964566236890e-003, -4.076955734546335e-018, -3.933291443272452e-003,
-3.646447325176707e-017, 3.518450895142321e-003, 2.117845637463755e-017,
-3.140485415208815e-003, -8.712160203777608e-018, 2.796075398236710e-003,
-1.294580160638788e-018, -2.482407787237604e-003, 9.161828106876181e-018,
2.197074535144406e-003, 1.415983651999832e-017, -1.937993323470931e-003,
-6.232775837302544e-018, 1.703345101068307e-003, -2.161186910179802e-021,
-1.491524479818111e-003, -1.502326565091718e-017, 1.301100062811467e-003,
8.918030712607688e-018, -1.130782522359557e-003, -4.125623586554305e-018,
9.793987814265136e-004, 4.447515773083571e-019, -8.458710441345732e-004,
2.310722132893812e-018, 7.291997106627285e-004, 5.306035247610870e-018,
-6.284494279917504e-004, -2.573585641777261e-018, 5.427376906950161e-004,
4.943085977436705e-019, -4.712255295790682e-004, -5.180304591591385e-018,
4.131099206142492e-004, 3.238466400552033e-018, -3.676176196053607e-004,
-1.711365178113278e-018, 3.340001847825552e-004, 5.039497039600437e-018,
-3.115299938741654e-004, 5.969682170532792e-019, 2.994970971692274e-004
};

```

Summary

This header file contains all the lowpass coefficients. A closer look at these will show a characteristic common to an impulse response of the lowpass filter, i.e. every second number is a Zero, The middle number is not a zero and all numbers from outside inwards are identical. This file is also included in the disk provided with the project.

The Weaver Method Using Linear Buffers

```
/*
 *
 * File Name:          Weaver0.c
 * Type :             C file
 * Description:       This file implements the weaver method using linear buffers
 * Authors:          N Shiburi & E.L Mabitsele
 * Date :           22/04/99, HSR Switzerland
 * Version:         1.0
 *
 *
 */
/*-----*/
/* ADSP-21060 System Register bit definitions */
#include <def21060.h> /* register definitions */
#include <21060.h> /* architecture */
#include <signal.h> /* interrupt */
#include <sport.h> /* serial port */
#include <math.h> /* fmod */
#include "bitsibb.h" /* bitsi board */
#include "wcoeff0.h" /* include the coeffienncy file for the low pass */
```

```

#define SetIOP(addr, val)  (* (int *) addr) = (val)
#define GetIOP(addr)      (* (int *) addr)
#define N 95
#define MAX_U_AD  2.75      /* maximum input voltage for A/D converter */
#define MAX_U_DA  3.0      /* maximum output voltage on D/A converter */
#define pi 3.1415

/* Convert a 16 bit Integer value (the 16 right bits in a 32 bit word x)
   to a 32 bit Integer value */
#define convertleft16to32(result, x) \
asm("%0=fext %1 by 16:16 (se);" : "=d" (result) : "d" (x));

/*-----*/

/* pointer for serial port (sport) transfer */
int *dms3;

/* set correction terms for input / output voltages */
float NORM = MAX_U_AD / 32767.0; /* normalization of sampled input values */
float CORR = 32767.0 / MAX_U_DA; /* denormalization for output values */

/* A/D, D/A converters */
unsigned long AD_IN; /* 32-bit A/D input value (CH1IN | CH2IN) */
long temp; /* temporary variable to separate channels */

/* matlab interface */
float insample1; /* input sample value1 */
float insample2; /* input sample value2 */
float pm dly1[N]; /*init for N sample1 */
float dm dly2[N]; /*init for N sample2 */
float fc=44000.0; /* sampling frequency */
register float outsample1 asm("r13"); /*output sample value1*/
register float outsample2 asm("r14"); /*output sample value2*/
unsigned outsample; /*output unsigned sample value#1*/
int k = N-1-2; /* index variable for shifting the delay-line
globally declared */

/* Array for sin */
int f = 11000; /* Frequency of the sin [Hz]*/
int count1 = 0; /* Actual Position in the 1st Array func */
int count2 = 0; /* Actuap Position in the 2nd Array func */
const int func_length = 4; /* Length of the Array func, func_length = fc/f */
float func1[4]; /* Array with the sin-values */
float func2[4]; /* Array with the cos-values */

/*-----*/

/* sample transmit irq */
void spt0_asserted( int sig_num )
{
    float sin_insample1;
    float cos_insample2;
    int index;

    convertleft16to32(temp, GetIOP(RX0)); /* Convert 16 bit int to 32 bit int */

    insample1 = NORM * ((float)temp); /* normalize input signal from CH1IN */

    sin_insample1 = func1[count1] * insample1; /*the input signal is multiplied
with the sin wave*/
    cos_insample2 = func2[count1] * insample1; /*the input signal is multiplied
with the cos wave*/

    /*get new "multiplied" sample*/
    dly1[0] = sin_insample1;
    dly2[0] = cos_insample2;

    /*reset the values*/
    outsample1 = 0.0;
    outsample2 = 0.0;

```

```

/*perform convolution */
for (index = 0;index<N;index+=2)
{
    outsample1 += h[index]*dly1[index];
    outsample2 += h[index]*dly2[index];
}

/*perform convolution with the middle value which is not zero */
    outsample1 += h[(N-1)/2]*dly1[(N-1)/2];
    outsample2 += h[(N-1)/2]*dly2[(N-1)/2];

/* shift the values after being added*/
for (index=k;index>=0;index--)
{
    dly1[index+1]=dly1[index];
    dly2[index+1]=dly2[index];
}

/* a loop to check if the array has reached the last point */
if(count1 == func_length-1)
    count1 = 0;
else
    count1++;

    /* mask and shift the final values */
    outsample = ((unsigned)(CORR * outsample1) << 16) & 0xffff0000;
    outsample = outsample | (((unsigned)(CORR * outsample2)) & 0xffff);

SetIOP(TX0,outsample);    /*give the value to the DAC*/
}

/*-----*/

void setup_sport0( void )
{
    /* Configure SHARC serial port */

    /* TRANSMIT CONTROL REGISTER */
    /* An alternate (and more efficient) way of doing this would be to */
    /* write the 32-bit register all at once with a statement like this: */
    /*     SetIOP(STCTL0, 0x001c00f2); */
    /* But the following is more descriptive... */

    sport0_iop.txc.mdf = 0;    /* multichannel frame delay (MFD) */
    sport0_iop.txc.schen = 0; /* Tx DMA chaining enable */
    sport0_iop.txc.sden = 0;  /* Tx DMA enable */
    sport0_iop.txc.lafs = 0;  /* Late TFS (alternate) */
    sport0_iop.txc.ltfs = 1;  /* Active low TFS */
    sport0_iop.txc.ditfs = 0; /* Data independent TFS */
    sport0_iop.txc.itfs = 0;  /* Internally generated TFS */
    sport0_iop.txc.tfsr = 1;  /* TFS Required */

    sport0_iop.txc.ckre = 0;  /* Data and FS on clock rising edge */
    sport0_iop.txc.gclk = 0;  /* Enable clock only during transmission*/
    sport0_iop.txc.iclk = 0;  /* Internally generated Tx clock */
    sport0_iop.txc.pack = 0;  /* Unpack 32b words into two 16b tx's */

    sport0_iop.txc.slenn = 31; /* Data word length minus one */
    sport0_iop.txc.sendn = 0;  /* Data word endian 1 = LSB first */
    sport0_iop.txc.dtype = SPORT_DTYPE_RIGHT_JUSTIFY_SIGN_EXTEND;
    /* Data type specifier */
    sport0_iop.txc.spen = 1;  /* Enable (clear for MC operation)

    /* RECEIVE CONTROL REGISTER */
    sport0_iop.rxc.nch = 31;  /* multichannel number of channels - 1 */
    sport0_iop.rxc.mce = 0;  /* multichannel enable */
    sport0_iop.rxc.spl = 0;  /* Loop back configure (test) */
    sport0_iop.rxc.d2dma = 0; /* Enable 2-dimensional DMA array */
    sport0_iop.rxc.schen = 0; /* Rx DMA chaining enable */
    sport0_iop.rxc.sden = 0;  /* Rx DMA enable */
    sport0_iop.rxc.lafs = 0;  /* Late RFS (alternate) */
    sport0_iop.rxc.ltfs = 1;  /* Active low RFS

```

```

sport0_iop.rxc.irfs = 0; /* Internally generated RFS */
sport0_iop.rxc.rfsr = 1; /* RFS Required */
sport0_iop.rxc.ckre = 0; /* Data and FS on clock rising edge */
sport0_iop.rxc.gclk = 0; /* Enable clock only during transmission*/
sport0_iop.rxc.iclk = 0; /* Internally generated Rx clock */
sport0_iop.rxc.pack = 0; /* Pack two 16b rx's into 32b word */

sport0_iop.rxc.slen = 31; /* Data word length minus one */
sport0_iop.rxc.sendn = 0; /* Data word endian 1 = LSB first */
sport0_iop.rxc.dtype = SPORT_DTYPE_RIGHT_JUSTIFY_SIGN_EXTEND;
/* Data type specifier */
sport0_iop.rxc.spen = 1; /* Enable (clear for MC operation) */

/* Enable sport0 xmit irqs */
interruptf(SIG_SPT0I, spt0_asserted);
}
/*-----*/

void setup_bitsibb( void )
{
    BITSIBB_TIM0 = BB_TIM(fc); /* set sampling frequency */
    BITSIBB_CTL = 1; /* enable just one DAC */
    BITSIBB_CLKSEL = 0x0844; /* Ch 1-2 Tim 0, Ch 3-4 Tim 1 */
}
/*-----*/

void init_data( void )
{
    insample1 = 0.0;
    insample2 = 0.0;
}
/*-----*/
/* Generates a sine wave to be multiplied with the input signal */

int sine_wave()
{
    int a = 0;
    for(a = 0; a < func_length; a++){

        func1[a] = (sin((float) a*(2*pi)/(func_length))); /*the funtion of f(x) goes here!*/
        func2[a] = (cos((float) a*(2*pi)/(func_length))); /*the funtion of f(x) goes here!*/

    }

}
/*-----*/

void main ( void )
{
    int t, msize;

    /* get base location of dms3 (BITSI) */
    t = GetIOP(SYSICON);
    msize = (t & 0xF000) >> 12;
    dms3 = (int *) (0x400000 + (3 * (0x2000 << msize)));

    /* set IOP registers */
    SetIOP(WAIT,0x21A4E421);

    /* initialize data buffers */
    init_data();

    sine_wave();

    /* initialize hardware */
    setup_bitsibb();
    setup_sport0();

    /* do forever */
    while(1)
    {}
}

```

Summary

This is the first Solution to the implementation of the Weaver method. Linear buffers were used. It compiled without errors but the filter was not of the best quality and we could not work with more than 65 coefficients without optimization. We could only get to 95 after optimization and still it was not up to the standard we wanted. This file can be obtained from the disk containing all the project files. It was listed here to show the method and show the structure of the linear method compared to that of the circular buffer.

Header File For Weaver0 (Linear Buffer Solution)

```
/*
*
* File Name:          Wcoeff0.h
* Type :             Header file
* Description:       This file contains lowpass coefficients for the weaver
*                   method using Linear buffers
* Author:            E.L Mabitsele
* Date :             26/04/99, HSR Switzerland
*
*
*
*/
/*****
/*-----*/

#define N 95
const dm float h[N]= {  -3.140485415208815e-003,

    2.117845637463755e-017,   3.518450895142319e-003,  -3.646447325176706e-017,
    -3.933291443272451e-003,  -4.076955734546335e-018,   4.388964566236889e-003,
    -1.134858803825008e-017,  -4.890237498007835e-003,   3.033651221415588e-017,
    5.442914765063811e-003,  -1.265417735111213e-017,   -6.054143736150291e-003,
    -9.390727217850135e-018,  6.732831187820820e-003,   -1.391324070807570e-017,
    -7.490220965849215e-003,  4.259298606514225e-017,   8.340710453905035e-003,
    -1.509830939677421e-017,  -9.303029613400017e-003,  -1.928094232260260e-017,
    1.040198556910602e-002,  -1.618352903048081e-017,   -1.167111691720245e-002,
    6.067568113847771e-017,   1.315686416521343e-002,   -1.714522360765993e-017,
    -1.492537337760950e-002,  1.757307417751629e-017,   1.707410146901768e-002,
    -1.796241204616114e-017,  -1.975270459745551e-002,   1.831111078516723e-017,
    2.320320675089511e-002,  -1.861726592294094e-017,   -2.784398685613123e-002,
    1.887920534628564e-017,   3.446574644658771e-002,   -1.909549843290416e-017,
    -4.476341888239694e-002,  1.926496386496171e-017,   6.314465631785622e-002,
    -1.938667608103415e-017,  -1.057722570251323e-001,   1.945997033120366e-017,
```

```
3.181156975481291e-001, 4.998519988766039e-001, 3.181156975481291e-001,  
1.945997033120366e-017, -1.057722570251323e-001, -1.938667608103415e-017,  
6.3144465631785622e-002, 1.926496386496171e-017, -4.476341888239694e-002,  
-1.909549843290416e-017, 3.446574644658771e-002, 1.887920534628564e-017,  
-2.784398685613123e-002, -1.861726592294094e-017, 2.320320675089512e-002,  
1.831111078516723e-017, -1.975270459745551e-002, -1.796241204616114e-017,  
1.707410146901768e-002, 1.757307417751629e-017, -1.492537337760950e-002,  
-1.714522360765993e-017, 1.315686416521343e-002, 6.067568113847771e-017,  
-1.167111691720245e-002, -1.618352903048082e-017, 1.040198556910602e-002,  
  
-1.928094232260260e-017, -9.303029613400017e-003, -1.509830939677421e-017,  
8.340710453905038e-003, 4.259298606514225e-017, -7.490220965849215e-003,  
-1.391324070807570e-017, 6.732831187820820e-003, -9.390727217850139e-018,  
-6.054143736150293e-003, -1.265417735111213e-017, 5.442914765063813e-003,  
3.033651221415590e-017, -4.890237498007839e-003, -1.134858803825009e-017,  
4.388964566236890e-003, -4.076955734546335e-018, -3.933291443272452e-003,  
-3.646447325176707e-017, 3.518450895142321e-003, 2.117845637463755e-017,  
-3.140485415208815e-003  
};
```

Summary

This header file contains all the lowpass coefficients. These were the ones used after optimization of the linear buffer method. A closer look at these too will show a characteristic common to an impulse response of the lowpass filter, i.e. every second number is a Zero, The middle number is not a zero and all numbers from outside inwards are identical. This file is also included in the disk provided with the project.

The Phase Method Using Circular Buffers

```
/*-----*/
*
* File Name:          Phase1.c
* Type :             C file
* Description:       This file implements the Phase method using linear
*                   buffers
* Authors:          N Shiburi & E.L Mabitsele
* Date :           07/05/99, HSR Switzerland
* Version:         1.1
*
*-----*/
\*****/

/*-----*/

/* ADSP-21060 System Register bit definitions */
#include <def21060.h> /* register definitions */
#include <21060.h> /* architecture */
#include <signal.h> /* interrupt */
#include <sport.h> /* serial port */
#include <math.h> /* fmod */
#include <macros.h> /* for the circular buffer */
#include "bitsibb.h" /* bitsi board */
#include "Pcoeff1.h" /* includes the coefficient file of the Hilbert */
#include <macros.h> /* for the CIRUCULAR_BUFFER */

#define SetIOP(addr, val) (* (int *) addr) = (val)
#define GetIOP(addr) (* (int *) addr)
#define N 251
#define MAX_U_AD 2.75 /* maximum input voltage for A/D converter */
#define MAX_U_DA 3.0 /* maximum output voltage on D/A converter */
/* Convert a 16 bit Integer value (the 16 right bits in a 32 bit word x)
to a 32 bit Integer value */
#define convertleft16to32(result, x) \
asm("%0=fext %1 by 16:16 (se);" : "=d" (result) : "d" (x));
/*-----*/

/* pointer for serial port (sport) transfer */
int *dms3;

/* a DM pointer to a circular buffers*/
CIRCULAR_BUFFER(float,2,Ptr);
CIRCULAR_BUFFER(float,1,Ptr2);

/* set correction terms for input / output voltages */
float NORM = MAX_U_AD / 32767.0; /* normalization of sampled input values */
float CORR = 32767.0 / MAX_U_DA; /* denormalization for output values */

/* A/D, D/A converters */
long temp; /* temporary variable to separate channels */

/* matlab interface */
float insample; /* input sample value */
dm float dly1[N]; /*init for 2*N-1 sample */
dm float data_buff[N];
float fc=44000.0; /* sampling frequency */
unsigned outsample; /*output on both channels*/
```

```

register float outsampl1 asm("r13");          /*output sample value on
channell*/
float outsampl2;          /*output sample value on channel2*/

/*-----*/

/* sample transmit irq */
void spt0_asserted( int sig_num )
{
    int index=0;          /*index variables */
    float buffer_value;  /* buffer value */
    convertleft16to32(temp, GetIOP(RX0)); /* Convert 16 bit int to 32 bit int */
    insample = NORM * ((float)temp);      /* normalize input signal from CH1IN */

    CIRC_WRITE(Ptr2,0,insample,dm);

    outsampl1 = 0.0;

    for (index = 0;index<N;index+=2)
    {
        CIRC_READ(Ptr2,2,buffer_value,dm);
        outsampl1 += h[index]*buffer_value; /*perform convolution */
    }

    /* Move the pointer where the value is needed */
    CIRC_MODIFY(Ptr2,-2);

    /* Write into the buffer the new value */
    CIRC_WRITE(Ptr,1,insample,dm);

    /*Read the outsampl*/
    CIRC_READ(Ptr,0,outsampl2,dm);

    outsampl = ((unsigned)(CORR * outsampl1 ) << 16) & 0xffff0000;
    outsampl = outsampl | (((unsigned)(CORR * outsampl2*1.52)) & 0xffff);

    SetIOP(TX0,outsampl); /*give the value to the DAC*/
}
/*-----*/

void setup_sport0( void )
{
    /* Configure SHARC serial port */

    /* TRANSMIT CONTROL REGISTER */
    /* An alternate (and more efficient) way of doing this would be to
    /* write the 32-bit register all at once with a statement like this:
    /*      SetIOP(STCTL0, 0x001c00f2);
    /* But the following is more descriptive...

    sport0_iop.txc.mdf = 0; /* multichannel frame delay (MFD)
    sport0_iop.txc.schen = 0; /* Tx DMA chaining enable
    sport0_iop.txc.sden = 0; /* Tx DMA enable
    sport0_iop.txc.lafs = 0; /* Late TFS (alternate)
    sport0_iop.txc.ltfs = 1; /* Active low TFS
    sport0_iop.txc.ditfs = 0; /* Data independent TFS
    sport0_iop.txc.itfs = 0; /* Internally generated TFS
    sport0_iop.txc.tfsr = 1; /* TFS Required

    sport0_iop.txc.ckre = 0; /* Data and FS on clock rising edge
    sport0_iop.txc.gclk = 0; /* Enable clock only during transmission
    sport0_iop.txc.iclk = 0; /* Internally generated Tx clock
    sport0_iop.txc.pack = 0; /* Unpack 32b words into two 16b tx's

    sport0_iop.txc.slenn = 31; /* Data word length minus one
    sport0_iop.txc.sendn = 0; /* Data word endian 1 = LSB first
    sport0_iop.txc.dtype = SPORT_DTYPE_RIGHT_JUSTIFY_SIGN_EXTEND;
    /* Data type specifier
    sport0_iop.txc.spen = 1; /* Enable (clear for MC operation)

```

```

/* RECEIVE CONTROL REGISTER */
sport0_iop.rxc.nch = 31; /* multichannel number of channels - 1 */
sport0_iop.rxc.mce = 0; /* multichannel enable */
sport0_iop.rxc.spl = 0; /* Loop back configure (test) */
sport0_iop.rxc.d2dma = 0; /* Enable 2-dimensional DMA array */
sport0_iop.rxc.schen = 0; /* Rx DMA chaining enable */
sport0_iop.rxc.sden = 0; /* Rx DMA enable */
sport0_iop.rxc.lafs = 0; /* Late RFS (alternate) */
sport0_iop.rxc.ltfs = 1; /* Active low RFS */
sport0_iop.rxc.irfs = 0; /* Internally generated RFS */
sport0_iop.rxc.rfsr = 1; /* RFS Required */
sport0_iop.rxc.ckre = 0; /* Data and FS on clock rising edge */
sport0_iop.rxc.gclk = 0; /* Enable clock only during transmission*/
sport0_iop.rxc.iclk = 0; /* Internally generated Rx clock */
sport0_iop.rxc.pack = 0; /* Pack two 16b rx's into 32b word */

sport0_iop.rxc.slen = 31; /* Data word length minus one */
sport0_iop.rxc.sendn = 0; /* Data word endian 1 = LSB first */
sport0_iop.rxc.dtype = SPORT_DTYPE_RIGHT_JUSTIFY_SIGN_EXTEND;
/* Data type specifier */
sport0_iop.rxc.spen = 1; /* Enable (clear for MC operation) */

/* Enable sport0 xmit irqs */
interruptf(SIG_SPT0I, spt0_asserted);
}
/*-----*/

void setup_bitsibb( void )
{
    BITSIBB_TIM0 = BB_TIM(fc); /* set sampling frequency */
    BITSIBB_CTL = 1; /* enable just one DAC */
    BITSIBB_CLKSEL = 0x0844; /* Ch 1-2 Tim 0, Ch 3-4 Tim 1 */
}
/*-----*/

void init_data( void )
{
    insample = 0.0;
}

/*-----*/
void main ( void )
{
    int t, msize;

    /* get base location of dms3 (BITSI) */
    t = GetIOP(SYSCON);
    msize = (t & 0xF000) >> 12;
    dms3 = (int *) (0x400000 + (3 * (0x2000 << msize)));

    /* Declarations for the first circular buffer for the filter part */
    BASE(Ptr2) = dly1; /*Loads b1 and i1 with buff start address */
    LENGTH(Ptr2) = N; /*Loads L1 with the length of the buffer*/

    /*Declarations for the second circular buffer for the delay */
    BASE(Ptr) = data_buff ; /*Loads b1 and i1 with buff start address */
    LENGTH(Ptr) = (N-1)/2+1 ; /*Loads L1 with the length of the buffer*/

    /* set IOP registers */
    SetIOP(WAIT,0x21A4E421);

    /* initialize data buffers */
    init_data();

    /* initialize hardware */
    setup_bitsibb();
    setup_sport0();

    /* do forever */
    while(1)

```

```
} {}
```

Summary

This is the final Solution to the implementation of the Phase method. Circular buffers were used and it was also optimized. To compile this file, type this command from the Matlab command window: MK_Phase1 (this is the make file for this program). You have to copy all the files into the appropriate directories before doing this. This file can be obtained from the disk containing all the project files. This program accepted 251 Hilbert Transformer coefficients without being unstable. This was a more accurate filter compared to the earlier one used with less than 97 coefficients.

Note: The multiplication of the 1.52 constant with the second outsample (outsample2) was increase the amplitude of outsample2 so that is merges well with outsample1.

Header File For Phase1 (Circular Buffer Solution)

```

/*****
*
*   File Name:          Pcoeff1.h
*   Type :             Header file
*   Description:       This file contains lowpass coefficients for the phase
*                     method using circular buffers
*   Author:            E.L Mabitsele
*   Date :             07/05/99, HSR Switzerland
*
*
\*****/
/*-----*/
#define N 251
const pm float h[N]= { +0.0006400000, +0.0000000000, +0.0006551301, +0.0000000000,
+0.0006803515, +0.0000000000, +0.0007161361, +0.0000000000,
+0.0007629630, +0.0000000000, +0.0008213195, +0.0000000000,
+0.0008917019, +0.0000000000, +0.0009746164, +0.0000000000,
+0.0010705812, +0.0000000000, +0.0011801270, +0.0000000000,
+0.0013037993, +0.0000000000, +0.0014421606, +0.0000000000,
+0.0015957921, +0.0000000000, +0.0017652971, +0.0000000000,
+0.0019513036, +0.0000000000, +0.0021544677, +0.0000000000,
+0.0023754783, +0.0000000000, +0.0026150610, +0.0000000000,
+0.0028739840, +0.0000000000, +0.0031530639, +0.0000000000,
+0.0034531726, +0.0000000000, +0.0037752461, +0.0000000000,
+0.0041202928, +0.0000000000, +0.0044894056, +0.0000000000,
+0.0048837732, +0.0000000000, +0.0053046958, +0.0000000000,
+0.0057536015, +0.0000000000, +0.0062320666, +0.0000000000,
+0.0067418392, +0.0000000000, +0.0072848666, +0.0000000000,
+0.0078633286, +0.0000000000, +0.0084796765, +0.0000000000,
+0.0091366801, +0.0000000000, +0.0098374839, +0.0000000000,
+0.0105856760, +0.0000000000, +0.0113853710, +0.0000000000,
+0.0122413120, +0.0000000000, +0.0131589973, +0.0000000000,
+0.0141448365, +0.0000000000, +0.0152063486, +0.0000000000,
+0.0163524105, +0.0000000000, +0.0175935769, +0.0000000000,
+0.0189424923, +0.0000000000, +0.0204144303, +0.0000000000,
+0.0220280079, +0.0000000000, +0.0238061439, +0.0000000000,
+0.0257773664, +0.0000000000, +0.0279776262, +0.0000000000,
+0.0304528595, +0.0000000000, +0.0332626911, +0.0000000000,
+0.0364859127, +0.0000000000, +0.0402288217, +0.0000000000,
+0.0446383225, +0.0000000000, +0.0499233035, +0.0000000000,
+0.0563911027, +0.0000000000, +0.0645131456, +0.0000000000,
+0.0750511708, +0.0000000000, +0.0893211572, +0.0000000000,
+0.1098091513, +0.0000000000, +0.1418427991, +0.0000000000,

```

+0.1992745525, +0.0000000000, +0.3328976980, +0.0000000000,
+0.9998547271, +0.0000000000, -0.9998547271, +0.0000000000,
-0.3328976980, +0.0000000000, -0.1992745525, +0.0000000000,
-0.1418427991, +0.0000000000, -0.1098091513, +0.0000000000,
-0.0893211572, +0.0000000000, -0.0750511708, +0.0000000000,
-0.0645131456, +0.0000000000, -0.0563911027, +0.0000000000,
-0.0499233035, +0.0000000000, -0.0446383225, +0.0000000000,
-0.0402288217, +0.0000000000, -0.0364859127, +0.0000000000,
-0.0332626911, +0.0000000000, -0.0304528595, +0.0000000000,
-0.0279776262, +0.0000000000, -0.0257773664, +0.0000000000,
-0.0238061439, +0.0000000000, -0.0220280079, +0.0000000000,
-0.0204144303, +0.0000000000, -0.0189424923, +0.0000000000,
-0.0175935769, +0.0000000000, -0.0163524105, +0.0000000000,
-0.0152063486, +0.0000000000, -0.0141448365, +0.0000000000,
-0.0131589973, +0.0000000000, -0.0122413120, +0.0000000000,
-0.0113853710, +0.0000000000, -0.0105856760, +0.0000000000,
-0.0098374839, +0.0000000000, -0.0091366801, +0.0000000000,
-0.0084796765, +0.0000000000, -0.0078633286, +0.0000000000,
-0.0072848666, +0.0000000000, -0.0067418392, +0.0000000000,
-0.0062320666, +0.0000000000, -0.0057536015, +0.0000000000,
-0.0053046958, +0.0000000000, -0.0048837732, +0.0000000000,
-0.0044894056, +0.0000000000, -0.0041202928, +0.0000000000,
-0.0037752461, +0.0000000000, -0.0034531726, +0.0000000000,
-0.0031530639, +0.0000000000, -0.0028739840, +0.0000000000,
-0.0026150610, +0.0000000000, -0.0023754783, +0.0000000000,
-0.0021544677, +0.0000000000, -0.0019513036, +0.0000000000,
-0.0017652971, +0.0000000000, -0.0015957921, +0.0000000000,
-0.0014421606, +0.0000000000, -0.0013037993, +0.0000000000,
-0.0011801270, +0.0000000000, -0.0010705812, +0.0000000000,

-0.0009746164, +0.0000000000, -0.0008917019, +0.0000000000,
-0.0008213195, +0.0000000000, -0.0007629630, +0.0000000000,
-0.0007161361, +0.0000000000, -0.0006803515, +0.0000000000,
-0.0006551301, +0.0000000000, -0.0006400000
};

Summary

This header file contains all the Hilbert transformer coefficients. A closer look at these will show a characteristic common to an impulse response of the Hilbert Transformer, i.e. the 90° phase shift, every second number being Zero, The middle number is also a zero and all numbers from outside inwards are inversely identical. This file is also included in the disk provided with the project.

The Phase Method Using Linear Buffers

```

/*****
*
*   File Name:          Phase0.c
*   Type :             C file
*   Description:       This file implements the Phase method using linear buffers
*   Authors:          N Shiburi & E.L Mabitsele
*   Date :            07/05/99, HSR Switzerland
*   Version:         1.0
*
*
*
\*****
/*-----*/

/* ADSP-21060 System Register bit definitions */
#include <def21060.h>      /* register definitions */
#include <21060.h>        /* architecture */
#include <signal.h>       /* interrupt */
#include <sport.h>        /* serial port */
#include <math.h>         /* fmod */
#include <macros.h>       /* for the circular buffer */
#include "bitsibb.h"      /* bitsi board */
#include "Pcoeff0.h"      /* includes the coefficient file of the Hilbert */
#define SetIOP(addr, val) (* (int *) addr) = (val)
#define GetIOP(addr)      (* (int *) addr)
#define N 95

```

```

#define MAX_U_AD 2.75 /* maximum input voltage for A/D converter */
#define MAX_U_DA 3.0 /* maximum output voltage on D/A converter */
/* Convert a 16 bit Integer value (the 16 right bits in a 32 bit word x)
to a 32 bit Integer value */
#define convertleft16to32(result, x) \
asm("%0=fext %1 by 16:16 (se);" : "=d" (result) : "d" (x));
/*-----*/
/* pointer for serial port (sport) transfer */
int *dms3;

/* loop counters */
int index;

/* index variable for shifting the delay-line globally declared */
int k = N-1-1;

/* set correction terms for input / output voltages */
float NORM = MAX_U_AD / 32767.0; /* normalization of sampled input values */
float CORR = 32767.0 / MAX_U_DA; /* denormalization for output values */

/* A/D, D/A converters */
long temp; /* temporary variable to separate channels */

/* matlab interface */
float insample; /* input sample value */
float dly1[N]; /*init for 2*N-1 sample */
float dly2[(N-1)/2]; /*init for 2*N-1 sample */
float fc=44000.0; /* sampling frequency */
unsigned outsample; /*output on both channels*/

register float outsample1 asm("r13"); /*output sample value on channel1*/
float outsample2; /*output sample value on channel2*/
/*-----*/

/* sample transmit irq */
void spt0_asserted( int sig_num )
{
    int index; /*index variables */

    convertleft16to32(temp, GetIOP(RX0)); /* Convert 16 bit int to 32 bit int */

    insample = NORM * ((float)temp); /* normalize input signal from CH1IN */

    dly1[0] = insample; /*get new sample*/

    outsample1 = 0.0;

    /*perform convolution */
    for (index = 0;index<N;index+=2)
        outsample1 += h[index]*dly1[index];

    /*take the middle value of the delay */
    outsample2 = dly1[(N-1)/2];
    for (index=k;index>=0;index--)
        dly1[index+1]=dly1[index];

    /* Mask and shift values */
    outsample = ((unsigned)(CORR * outsample1 ) << 16) & 0xffff0000;
    outsample = outsample | (((unsigned)(CORR * outsample2*1.52)) & 0xffff);

    SetIOP(TX0,outsample); /*give the value to the DAC*/
}
/*-----*/

void setup_sport0( void )
{
    /* Configure SHARC serial port */

```

```

/* TRANSMIT CONTROL REGISTER */
/* An alternate (and more efficient) way of doing this would be to */
/* write the 32-bit register all at once with a statement like this: */
/*      SetIOP(STCTL0, 0x001c00f2); */
/* But the following is more descriptive... */

sport0_iop.txc.mdf = 0; /* multichannel frame delay (MFD) */
sport0_iop.txc.schen = 0; /* Tx DMA chaining enable */
sport0_iop.txc.sden = 0; /* Tx DMA enable */
sport0_iop.txc.lafs = 0; /* Late TFS (alternate) */
sport0_iop.txc.ltfs = 1; /* Active low TFS */
sport0_iop.txc.ditfs = 0; /* Data independent TFS */
sport0_iop.txc.itfs = 0; /* Internally generated TFS */
sport0_iop.txc.tfsr = 1; /* TFS Required

sport0_iop.txc.ckre = 0; /* Data and FS on clock rising edge */
sport0_iop.txc.gclk = 0; /* Enable clock only during transmission*/
sport0_iop.txc.iclk = 0; /* Internally generated Tx clock */
sport0_iop.txc.pack = 0; /* Unpack 32b words into two 16b tx's

sport0_iop.txc.slen = 31; /* Data word length minus one */
sport0_iop.txc.sendn = 0; /* Data word endian 1 = LSB first */
sport0_iop.txc.dtype = SPORT_DTYPE_RIGHT_JUSTIFY_SIGN_EXTEND;
/* Data type specifier
sport0_iop.txc.spen = 1; /* Enable (clear for MC operation)

/* RECEIVE CONTROL REGISTER */
sport0_iop.rxc.nch = 31; /* multichannel number of channels - 1 */
sport0_iop.rxc.mce = 0; /* multichannel enable */
sport0_iop.rxc.spl = 0; /* Loop back configure (test) */
sport0_iop.rxc.d2dma = 0; /* Enable 2-dimensional DMA array */
sport0_iop.rxc.schen = 0; /* Rx DMA chaining enable */
sport0_iop.rxc.sden = 0; /* Rx DMA enable */
sport0_iop.rxc.lafs = 0; /* Late RFS (alternate) */
sport0_iop.rxc.ltfs = 1; /* Active low RFS */
sport0_iop.rxc.irfs = 0; /* Internally generated RFS */
sport0_iop.rxc.rfsr = 1; /* RFS Required
sport0_iop.rxc.ckre = 0; /* Data and FS on clock rising edge */
sport0_iop.rxc.gclk = 0; /* Enable clock only during transmission*/
sport0_iop.rxc.iclk = 0; /* Internally generated Rx clock */
sport0_iop.rxc.pack = 0; /* Pack two 16b rx's into 32b word

sport0_iop.rxc.slen = 31; /* Data word length minus one */
sport0_iop.rxc.sendn = 0; /* Data word endian 1 = LSB first */
sport0_iop.rxc.dtype = SPORT_DTYPE_RIGHT_JUSTIFY_SIGN_EXTEND;
/* Data type specifier
sport0_iop.rxc.spen = 1; /* Enable (clear for MC operation)

/* Enable sport0 xmit irqs */
interruptf(SIG_SPT0I, spt0_asserted);
}
/*-----*/

void setup_bitsibb( void )
{
    BITSIBB_TIM0 = BB_TIM(fc); /* set sampling frequency */
    BITSIBB_CTL = 1; /* enable just one DAC */
    BITSIBB_CLKSEL = 0x0844; /* Ch 1-2 Tim 0, Ch 3-4 Tim 1 */
}
/*-----*/

void init_data( void )
{
    insample = 0.0;
}

/*-----*/
void main ( void )

```

```

{
    int t, msize;

    /* get base location of dms3 (BITSI) */
    t = GetIOP(SYSCON);
    msize = (t & 0xF000) >> 12;
    dms3 = (int *) (0x400000 + (3 * (0x2000 << msize)));

    /* set IOP registers */
    SetIOP(WAIT, 0x21A4E421);

    /* initialize data buffers */
    init_data();

    /* initialize hardware */
    setup_bitsibb();
    setup_sport0();

    /* do forever */
    while(1)
    {}
}

```

Summary

This is the first Solution to the implementation of the Phase method. Linear buffers were used. It compiled without errors but the filter was not of the best quality and we could not work with more than 75 coefficients without optimization. We could only get to 97 after optimization and still it was not up to the standard we wanted. It was five times improved after the use of circular buffer. This file can be obtained from the disk containing all the project files. It was listed here to show the method and show the structure of the linear method compared to that of the circular buffer.

Note: The multiplication of the 1.52 constant with the second outsample (outsample2) was increase the amplitude of outsample2 so that is merges well with outsample1.

Header File For Phase0 (Linear Buffer Solution)

```

/*****
*
*   File Name:      Pcoeff0.h
*   Type :         Header file
*   Description:    This file contains lowpass coefficients for the phase
*                  method using linear buffers
*
*   Author:        E.L Mabitsele
*   Date :         07/05/99, HSR Switzerland
*
*
* *****/
\*****/

/*-----*/
-----*/

#define N 95
const pm float h[N]= { +0.0017230829, +0.0000000000,
+0.0019741949,
+0.0000000000, +0.0024281893, +0.0000000000, +0.0031082570,
+0.0000000000, +0.0040390764, +0.0000000000, +0.0052473970,
+0.0000000000, +0.0067628836, +0.0000000000, +0.0086193240,
+0.0000000000, +0.0108563614, +0.0000000000, +0.0135219947,
+0.0000000000, +0.0166762390, +0.0000000000, +0.0203965824,
+0.0000000000, +0.0247863235, +0.0000000000, +0.0299876928,
+0.0000000000, +0.0362032713, +0.0000000000, +0.0437325175,
+0.0000000000, +0.0530374871, +0.0000000000, +0.0648691596,
+0.0000000000, +0.0805314829, +0.0000000000, +0.1024973357,
```

+0.0000000000, +0.1360802087, +0.0000000000, +0.1951175719,
+0.0000000000, +0.3303870763, +0.0000000000, +0.9990151047,
+0.0000000000, -0.9990151047, +0.0000000000, -0.3303870763,
+0.0000000000, -0.1951175719, +0.0000000000, -0.1360802087,
+0.0000000000, -0.1024973357, +0.0000000000, -0.0805314829,

+0.0000000000, -0.0648691596, +0.0000000000, -0.0530374871,
+0.0000000000, -0.0437325175, +0.0000000000, -0.0362032713,
+0.0000000000, -0.0299876928, +0.0000000000, -0.0247863235,
+0.0000000000, -0.0203965824, +0.0000000000, -0.0166762390,
+0.0000000000, -0.0135219947, +0.0000000000, -0.0108563614,
+0.0000000000, -0.0086193240, +0.0000000000, -0.0067628836,
+0.0000000000, -0.0052473970, +0.0000000000, -0.0040390764,
+0.0000000000, -0.0031082570, +0.0000000000, -0.0024281893,
+0.0000000000, -0.0019741949, +0.0000000000, -0.0017230829

};

Summary

This header file contains the Hilbert transformer coefficients. These were the ones used after optimization of the linear buffer method. A closer look at these too will show a characteristic common to an impulse response of the Hilbert transformer. This file is also included in the disk provided with the project.