



SATHYABAMA

INSTITUTE OF SCIENCE AND TECHNOLOGY

(DEEMED TO BE UNIVERSITY)

Accredited "A" Grade by NAAC | 12B Status by UGC | Approved by AICTE

www.sathyabama.ac.in

SCHOOL OF ELECTRICAL AND ELECTRONICS

DEPARTMENT OF ELECTRONICS AND COMMUNICATION
ENGINEERING

UNIT - I

SECA1506 - Digital Signal Processing

I. DISCRETE FOURIER TRANSFORM (DFT) AND FAST FOURIER TRANSFORM (FFT)

Review of Signals and Systems

Continuous Time signal – If the signal is defined over continuous-time, then the signal is a continuous-time signal.

Ex: Sinusoidal signal, Voice signal, Rectangular pulse function

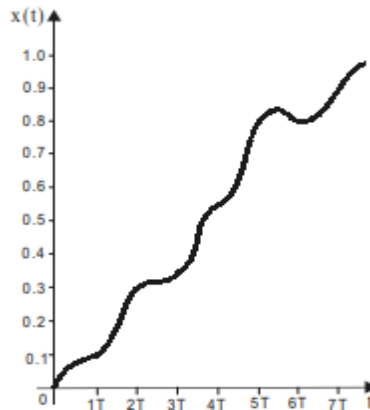


Fig 1 Continuous Time signal

Discrete Signal and Discrete Time Signal:

The discrete signal is a function of a discrete independent variable. The independent variable is divided into uniform intervals and each interval is represented by an integer. The letter "n" is used to denote the independent variable. The discrete or digital signal is denoted by $x(n)$.

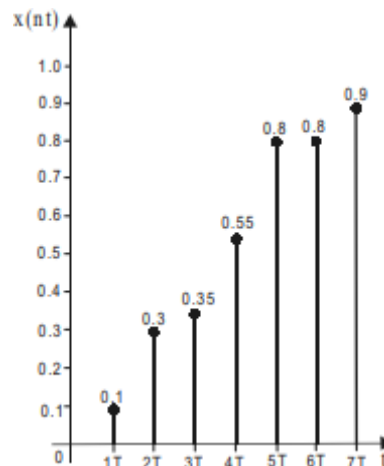


Fig 2: Discrete Time Signal

Digital Signal: The signals that are discrete in time and quantized in amplitude are called digital signal. The term "digital signal" applies to the transmission of a sequence of values of a discrete-time signal in the form of some digits in the encoded form.

Representation of Discrete Time Signals

1. Functional representation

In functional representation, the signal is represented as a mathematical equation, as shown in the following example.

$x(n) = -0.5$;	$n = -2$
$= 1.0$;	$n = -1$
$= -1.0$;	$n = 0$
$= 0.6$;	$n = 1$
$= 1.2$;	$n = 2$
$= 1.5$;	$n = 3$
$= 0$;	other n

2. Graphical representation

In graphical representation, the signal is represented in a two-dimensional plane. The independent variable is represented in the horizontal axis and the value of the signal is represented in the vertical axis as shown below

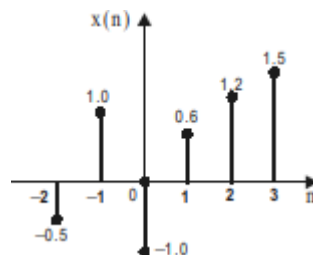


Fig 3: Discrete Time Signal

3. Tabular representation

In tabular representation, two rows of a table are used to represent a discrete time signal. In the first row, the independent variable "n" is tabulated and in the second row the value of the signal for each value of "n" are tabulated as shown in the following table I.

Table 1. Tabular representation

n	-2	-1	0	1	2	3
x(n)	-0.5	1.0	-1.0	0.6	1.2	1.5

4. Sequence representation

In sequence representation, the discrete time signal is represented as a one-dimensional array as shown in the following examples.

An infinite duration discrete time signal with the time origin, $n = 0$, indicated by the symbol - is represented as, $x(n) = \{ \dots - 0.5, 1.0, -1.0, 0.6, 1.2, 1.5, \dots \}$

An infinite duration discrete time signal that satisfies the condition $x(n) = 0$ for $n < 0$ is represented as,

$$x(n) = \{-1.0, 0.6, 1.2, 1.5, \dots\} \text{ or } x(n) = \{-1.0, 0.6, 1.2, 1.5, \dots\}$$

A finite duration discrete time signal with the time origin, $n = 0$, indicated by the symbol - is represented as, $x(n) = \{-0.5, 1.0, -1.0, 0.6, 1.2, 1.5\}$

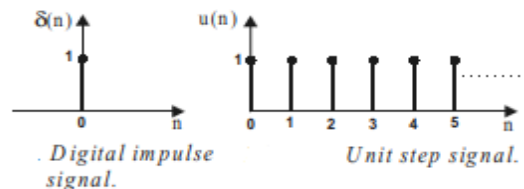
A finite duration discrete time signal that satisfies the condition $x(n) = 0$ for $n < 0$ is represented as,

$$x(n) = \{-1.0, -0.6, 1.2, 1.5\} \text{ or } x(n) = \{-1.0, 0.6, 1.2, 1.5\}$$

Standard Discrete Time Signals

1. Digital impulse signal or unit sample sequence

$$\begin{aligned} \text{Impulse signal, } \delta(n) &= 1 ; n = 0 \\ &= 0 ; n \neq 0 \end{aligned}$$

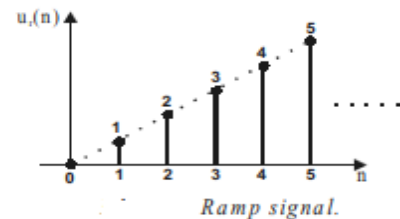


2. Unit step signal

$$\begin{aligned} \text{Unit step signal, } u(n) &= 1 ; n \geq 0 \\ &= 0 ; n < 0 \end{aligned}$$

3. Ramp signal

$$\begin{aligned} \text{Ramp signal, } u_r(n) &= n ; n \geq 0 \\ &= 0 ; n < 0 \end{aligned}$$



4. Exponential signal

$$\begin{aligned} \text{Exponential signal, } g(n) &= a^n ; n \geq 0 \\ &= 0 ; n < 0 \end{aligned}$$

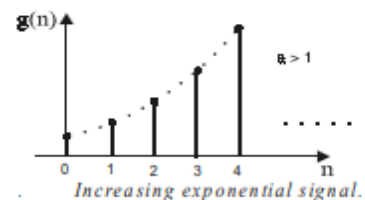
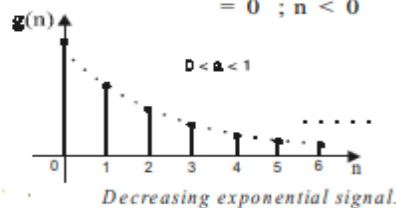


Fig 4: Standard Discrete Time Signals

Classification of Discrete Time Signals

The discrete time signals are classified depending on their characteristics. Some ways of classifying discrete time signals are,

1. Deterministic and nondeterministic signals
2. Periodic and aperiodic signals
3. Symmetric and antisymmetric signals
4. Energy and power signals
5. Causal and noncausal signals

Deterministic and Nondeterministic Signals

The signals that can be completely specified by mathematical equations are called deterministic signals. The step, ramp, exponential and sinusoidal signals are examples of deterministic signals. The signals whose characteristics are random in nature are called nondeterministic signals. The noise signals from various sources are best examples of nondeterministic signals.

Periodic and Aperiodic Signals

When a discrete time signal $x(n)$, satisfies the condition $x(n + N) = x(n)$ for integer values of N , then the discrete time signal $x(n)$ is called periodic signal. Here N is the number of samples of a period.

i.e, if, $x(n + N) = x(n)$, for all n , then $x(n)$ is periodic

The smallest value of N for which the above equation is true is called fundamental period. If there is no value of N that satisfies the above equation, then $x(n)$ is called aperiodic or nonperiodic signal. When N is the fundamental period, the periodic signals will also satisfy the condition $x(n + kN) = x(n)$, where k is an integer. The periodic signals are power signals. The discrete time sinusoidal and complex exponential signals are periodic signals when their fundamental frequency, f_0 is a rational number.

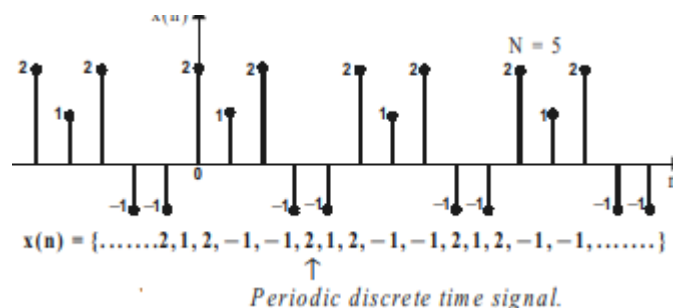


Fig 5. Periodic Discrete Time Signals

Symmetric (Even) and Antisymmetric (Odd) Signals

The discrete time signals may exhibit symmetry or antisymmetry with respect to $n = 0$. When a discrete time signal exhibits symmetry with respect to $n = 0$ then it is called an even signal. Therefore, the even signal satisfies the condition,

$$x(-n) = x(n)$$

When a discrete time signal exhibits antisymmetry with respect to $n = 0$, then it is called an odd signal. Therefore the odd signal satisfies the condition,

$$x(-n) = -x(n)$$

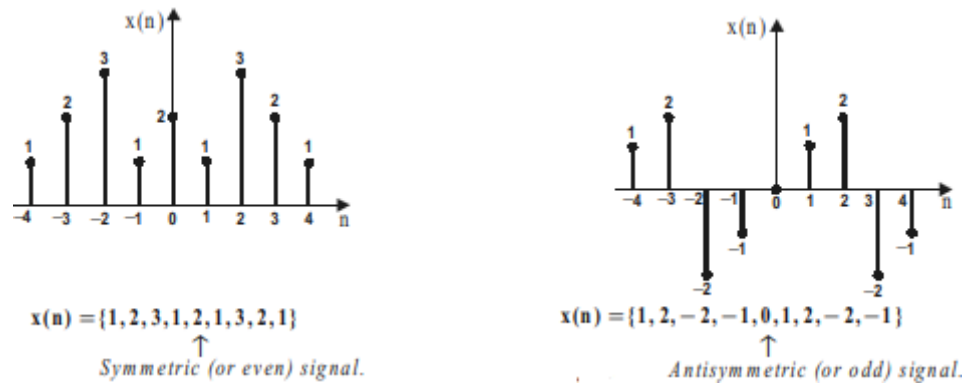


Fig 6. Symmetric and antisymmetric Discrete Time Signals

Energy and Power Signals

The energy E of a discrete time signal $x(n)$ is defined as,

$$\text{Energy, } E = \sum_{n=-\infty}^{\infty} |x(n)|^2$$

The energy of a signal may be finite or infinite, and can be applied to complex valued and real valued signals. If energy E of a discrete time signal is finite and nonzero, then the discrete time signal is called an energy signal. The exponential signals are examples of energy signals. The average power of a discrete time signal $x(n)$ is defined as,

$$\text{Power, } P = \lim_{N \rightarrow \infty} \frac{1}{2N+1} \sum_{n=-N}^N |x(n)|^2$$

If power P of a discrete time signal is finite and nonzero, then the discrete time signal is called a power signal. The periodic signals are examples of power signals. For energy signals, the energy will be finite and average power will be zero. For power signals the average power is finite and energy will be infinite.

<p>For energy signal, $0 < E < \infty$ and $P = 0$</p> <p>For power signal, $0 < P < \infty$ and $E = \infty$</p>

Causal, Noncausal and Anticausal signals

A discrete time signal is said to be causal, if it is defined for $n \geq 0$. Therefore if $x(n)$ is causal, then $x(n) = 0$ for $n < 0$. A discrete time signal is said to be noncausal, if it is defined for either $n \leq 0$, or for both $n \leq 0$ and $n > 0$. Therefore if $x(n)$ is noncausal, then $x(n) \neq 0$ for $n < 0$. A noncausal signal can be converted to causal signal by multiplying the noncausal signal by a

unit step signal, $u(n)$. When a noncausal discrete time signal is defined only for $n \leq 0$, it is called an anticausal signal.

Discrete-time Fourier transform (DTFT)

The Discrete Time Fourier Transform (DTFT) is the member of the Fourier transform family that operates on aperiodic, discrete signals. The best way to understand the DTFT is how it relates to the DFT. To start, imagine that you acquire an N sample signal, and want to find its frequency spectrum. By using the DFT, the signal can be decomposed into sine and cosine waves, with frequencies equally spaced between zero and one-half of the sampling rate. As discussed in the last chapter, padding the time domain signal with zeros makes the period of the time domain longer, as well as making the spacing between samples in the frequency domain narrower. As N approaches infinity, the time domain becomes aperiodic, and the frequency domain becomes a continuous signal. This is the DTFT, the Fourier transform that relates an aperiodic, discrete signal, with a periodic, continuous frequency spectrum.

The mathematics of the DTFT can be understood by starting with the synthesis and analysis equations

$$\begin{aligned}
 x[n] &= \frac{1}{2\pi} \int_{-\pi}^{\pi} X(\Omega) e^{j\Omega n} d\Omega && \text{synthesis} \\
 X(\Omega) &= \sum_{n=-\infty}^{+\infty} x[n] e^{-j\Omega n} && \text{analysis} \\
 x[n] &\xleftrightarrow{\mathcal{F}} X(\Omega) \\
 X(\Omega) &= \operatorname{Re} \{ X(\Omega) \} + j \operatorname{Im} \{ X(\Omega) \} \\
 &= |X(\Omega)| e^{j\angle X(\Omega)}
 \end{aligned}$$

The spectrum of the DTFT is continuous, so either f or ω can be used. The common choice is ω , because it makes the equations shorter by eliminating the always present factor of 2π . Remember, when ω is used, the frequency spectrum extends from 0 to π , which corresponds to DC to one-half of the sampling rate. To make things even more complicated, many authors use Ω (an upper case omega) to represent this frequency in the DTFT, rather than ω (a lower case omega).

PROPERTIES OF THE FOURIER TRANSFORM

$$x[n] \xleftrightarrow{\mathcal{F}} X(\Omega)$$

Periodic:

$$X(\Omega) = X(\Omega + 2\pi m)$$

Symmetry:

$$x[n] \text{ real} \Rightarrow X(-\Omega) = X^*(\Omega)$$

$$\left. \begin{array}{l} \operatorname{Re}\{X(\Omega)\} \\ |X(\Omega)| \end{array} \right\} \text{ even}$$

$$\left. \begin{array}{l} \operatorname{Im}\{X(\Omega)\} \\ -X(\Omega) \end{array} \right\} \text{ odd}$$

Time shifting:

$$x[n - n_0] \xleftrightarrow{\mathcal{F}} e^{-j\Omega n_0} X(\Omega)$$

Frequency shifting:

$$e^{j\Omega_0 n} x[n] \xleftrightarrow{\mathcal{F}} X(\Omega - \Omega_0)$$

Linearity:

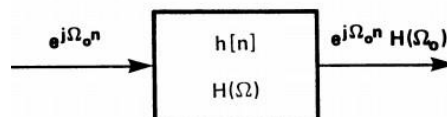
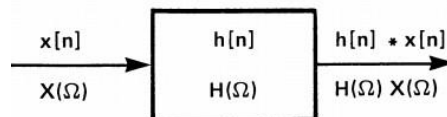
$$ax_1[n] + bx_2[n] \xleftrightarrow{\mathcal{F}} aX_1(\Omega) + bX_2(\Omega)$$

Parseval's relation:

$$\sum_{n=-\infty}^{+\infty} |x[n]|^2 = \frac{1}{2\pi} \int_{2\pi} |X(\Omega)|^2 d\Omega$$

CONVOLUTION PROPERTY

$$h[n] * x[n] \xleftrightarrow{\mathcal{F}} H(\Omega) X(\Omega)$$



Discrete Fourier Transform (DFT):

Definition (Discrete Fourier Transform): Given a finite sequence

$$x = [x(0), x(1), \dots, x(N-1)]$$

its Discrete Fourier Transform (DFT) is a finite sequence

$$X = DFT(x) = [X(0), X(1), \dots, X(N-1)]$$

where

$$X(k) = \sum_{n=0}^{N-1} x(n) w_N^{kn}, \quad w_N = e^{-j2\pi/N}$$

Inverse Discrete Fourier Transform (IDFT):

The inverse discrete Fourier transform of $X(k)$ is defined as

$$x(n) = \frac{1}{N} \sum_{k=0}^{N-1} X(k) e^{j2\pi kn/N} \quad 0 \leq n \leq N-1$$

For notation purpose discrete Fourier transform and inverse Fourier transform can be represented by

$$\begin{aligned} X(k) &= DFT[x(n)] \\ x(n) &= IDFT[X(k)] \end{aligned}$$

Formula:

$$\begin{aligned} X(k) &= \sum_{n=0}^{N-1} x(n) e^{-j2\pi \frac{kn}{N}} \\ x(n) &= \frac{1}{N} \sum_{k=0}^{N-1} X(k) e^{j2\pi \frac{kn}{N}} \end{aligned}$$

Where K and n are in the range of $0, 1, 2, \dots, N-1$ For example, if $N=4$, $K=0, 1, 2, 3$; $N=0, 1, 2, 3$

Alternative Formula:

$$X(k) = \sum_{n=0}^{N-1} x(n)W^{kn} \quad \leftarrow W = e^{-j\frac{2\pi}{N}}$$

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

Properties of DFT:

Periodicity property:

If $X(k)$ is the N-point DFT of $x(n)$, then

$$X(k+N)=X(k)$$

Linearity property:

If $X_1(k)=\text{DFT}[x_1(n)]$ & $X_2(k)=\text{DFT}[x_2(n)]$, then

$$\text{DFT}[a_1x_1(n)+a_2x_2(n)]=a_1X_1(k)+a_2X_2(k)$$

Convolution property:

If $X_1(k) = \text{DFT}[x_1(n)]$ & $X_2(k) = \text{DFT}[x_2(n)]$, then

$$\text{DFT}[x(n) \textcircled{N} x_2(n)] = X_1(k)X_2(k)$$

Where \textcircled{N} indicates N-point circular convolution.

Multiplication property:

If $X_1(k) = \text{DFT}[x_1(n)]$ & $X_2(k) = \text{DFT}[x_2(n)]$, then

$$\text{DFT}[x_1(n)x_2(n)] = (1/N)[X_1(k) \textcircled{N} X_2(k)]$$

Where \textcircled{N} Indicates N-point circular convolution.

Time reversal property:

If $X(k)$ is the N-point DFT of $x(n)$, then $\text{DFT}[x(N-n)] = X(N-k)$

Time shift property:

If $X(k)$ is the N-point DFT of $x(n)$, then

$$\mathcal{DFT}\{x((n-m))_N\} = X(k) e^{-j\frac{2\pi km}{N}}$$

Symmetry properties:

If $x(n)=x_R(n)+jx_I(n)$ is N-point complex sequence and $X(k)=X_R(k)+jX_I(k)$ is the N- point DFT of $x(n)$ where $x_R(n)$ & $x_I(n)$ are the real & imaginary parts of $x(n)$ and $X_R(k)$ & $X_I(k)$ are the those of $X(k)$, then

- (i) $\text{DFT}[x^*(n)] = X^*(N-k)$
- (ii) $\text{DFT}[x^*(N-n)] = X^*(k)$
- (iii) $\text{DFT}[x_R(n)] = (1/2)[X(k) + X^*(N-k)]$
- (iv) $\text{DFT}[x_I(n)] = (1/2j)[X(k) - X^*(N-k)]$
- (v) $\text{DFT}[x_{ce}(n)] = X_R(k)$ where $x_{ce}(n) = (1/2)[x(n) + x^*(N-n)]$
- (vi) $\text{DFT}[x_{co}(n)] = jX_I(k)$ where $x_{co}(n) = (1/2)[x(n) - x^*(N-n)]$

If $x(n)$ is real, then

- (i) If $x(n)$ is real, then
 - a. $X(k) = X^*(N-k)$
 - b. $X_R(k) = X_R(N-k)$
- (ii) If $x(n)$ is real, then

- a) $X(k) = X^*(N-k)$
- b) $X_R(k) = X_R(N-k)$
- c) $X_I(k) = -X_I(N-k)$
- d) $|X(k)| = |X(N-k)|$
- e) $|X(k)| = |X(N-k)|$
- f) $\angle X(k) = -\angle X(N-k)$

- (i) $\text{DFT}[x_{ce}(n)] = X_R(k)$ where $x_{ce}(n) = (1/2)[x(n) + x(N-n)]$
- (ii) $\text{DFT}[x_{co}(n)] = jX_I(k)$ where $x_{co}(n) = (1/2)[x(n) - x(N-n)]$

Problem

Compute 4-point DFT and 8-point DFT of causal three sample sequence given by

$$x(n) = \frac{1}{3} ; 0 \leq n \leq 2$$
$$= 0 ; \text{ else}$$

Solution

By the definition of N-point DFT, the k^{th} complex coefficient of $X(k)$, for $0 \leq k \leq N-1$, is given by,

$$X(k) = \sum_{n=0}^{N-1} x(n) e^{-j\frac{2\pi kn}{N}}$$

a) 4-point DFT (\ N = 4)

$$X(k) = \sum_{n=0}^{4-1} x(n) e^{-j\frac{2\pi kn}{4}} = \sum_{n=0}^2 x(n) e^{-j\frac{\pi kn}{2}} = x(0) e^0 + x(1) e^{-j\frac{\pi k}{2}} + x(2) e^{-j\pi k}$$

$e^{j\theta} = \cos\theta + j\sin\theta$

$$= \frac{1}{3} + \frac{1}{3} e^{-j\frac{\pi k}{2}} + \frac{1}{3} e^{-j\pi k} = \frac{1}{3} \left[1 + \cos\frac{\pi k}{2} - j\sin\frac{\pi k}{2} + \cos\pi k - j\sin\pi k \right]$$

For 4-point DFT, $X(k)$ has to be evaluated for $k = 0, 1, 2, 3$.

$$\text{When } k = 0 ; X(0) = \frac{1}{3} [1 + \cos 0 - j\sin 0 + \cos 0 - j\sin 0]$$
$$= \frac{1}{3} (1 + 1 - j0 + 1 - j0) = 1 = 1\angle 0$$

$$\text{When } k = 1 ; X(1) = \frac{1}{3} \left[1 + \cos\frac{\pi}{2} - j\sin\frac{\pi}{2} + \cos\pi - j\sin\pi \right]$$
$$= \frac{1}{3} (1 + 0 - j - 1 - j0) = -j\frac{1}{3} = \frac{1}{3} \angle -\pi/2 = 0.333\angle -0.5\pi$$

$$\text{When } k = 2 ; X(2) = \frac{1}{3} [1 + \cos\pi - j\sin\pi + \cos 2\pi - j\sin 2\pi]$$
$$= \frac{1}{3} (1 - 1 - j0 + 1 - j0) = \frac{1}{3} = 0.333\angle 0$$

$$\text{When } k = 3 ; X(3) = \frac{1}{3} \left[1 + \cos\frac{3\pi}{2} - j\sin\frac{3\pi}{2} + \cos 3\pi - j\sin 3\pi \right]$$
$$= \frac{1}{3} (1 + 0 + j - 1 - j0) = j\frac{1}{3} = \frac{1}{3} \angle \pi/2 = 0.333\angle 0.5\pi$$

\ The 4-point DFT sequence $X(k)$ is given by,

$$X(k) = \{ 1\angle 0, 0.333\angle -0.5\pi, 0.333\angle 0, 0.333\angle 0.5\pi \}$$

$$\therefore \text{ Magnitude Function, } |X(k)| = \{ 1, 0.333, 0.333, 0.333 \}$$

$$\text{Phase Function, } \angle X(k) = \{ 0, -0.5\pi, 0, 0.5\pi \}$$

Phase angles
are in radians.

b) 8-point DFT (\ N = 8)

$$X(k) = \sum_{n=0}^{8-1} x(n) e^{-j2\pi kn/8} = \sum_{n=0}^2 x(n) e^{-j\pi kn/4} = x(0) e^0 + x(1) e^{-j\pi k/4} + x(2) e^{-j\pi k/2}$$

$e^{+jq} = \cos q + j \sin q$

$$= \frac{1}{3} + \frac{1}{3} e^{-j\pi k/4} + \frac{1}{3} e^{-j\pi k/2} = \frac{1}{3} \left[1 + \cos \frac{\pi k}{4} - j \sin \frac{\pi k}{4} + \cos \frac{\pi k}{2} - j \sin \frac{\pi k}{2} \right]$$

For 8-point DFT, $X(k)$ has to be evaluated for $k = 0, 1, 2, 3, 4, 5, 6, 7$.

When $k = 0$; $X(0) = \frac{1}{3} [1 + \cos 0 - j \sin 0 + \cos 0 - j \sin 0]$

$$= \frac{1}{3} (1 + 1 - j0 + 1 - j0) = 1 = 1 \angle 0$$

When $k = 1$; $X(1) = \frac{1}{3} \left[1 + \cos \frac{\pi}{4} - j \sin \frac{\pi}{4} + \cos \frac{\pi}{2} - j \sin \frac{\pi}{2} \right]$

$$= 0.333 (1 + 0.707 - j0.707 + 0 - j1)$$

$$= 0.568 - j0.568 = 0.803 \angle -0.785 = 0.803 \angle -0.25\pi$$

$\frac{0.785}{\pi} \times \pi = 0.25\pi$

When $k = 2$; $X(2) = \frac{1}{3} \left[1 + \cos \frac{2\pi}{4} - j \sin \frac{2\pi}{4} + \cos \frac{2\pi}{2} - j \sin \frac{2\pi}{2} \right]$

$$= 0.333 (1 + 0 - j1 - 1 - j0)$$

$$= -j0.333 = 0.333 \angle -\pi/2 = 0.333 \angle -0.5\pi$$

When $k = 3$; $X(3) = \frac{1}{3} \left[1 + \cos \frac{3\pi}{4} - j \sin \frac{3\pi}{4} + \cos \frac{3\pi}{2} - j \sin \frac{3\pi}{2} \right]$

$$= 0.333 (1 - 0.707 - j0.707 + 0 + j1)$$

$$= 0.098 + j0.098 = 0.139 \angle 0.785 = 0.139 \angle 0.25\pi$$

When $k = 4$; $X(4) = \frac{1}{3} \left[1 + \cos \frac{4\pi}{4} - j \sin \frac{4\pi}{4} + \cos \frac{4\pi}{2} - j \sin \frac{4\pi}{2} \right]$

$$= 0.333 (1 - 1 - j0 + 1 - j0) = 0.333 = 0.333 \angle 0$$

When $k = 5$; $X(5) = \frac{1}{3} \left[1 + \cos \frac{5\pi}{4} - j \sin \frac{5\pi}{4} + \cos \frac{5\pi}{2} - j \sin \frac{5\pi}{2} \right]$

$$= 0.333 (1 - 0.707 + j0.707 + 0 - j1)$$

$$= 0.098 - j0.098 = 0.139 \angle -0.785 = 0.139 \angle -0.25\pi$$

When $k = 6$; $X(6) = \frac{1}{3} \left[1 + \cos \frac{6\pi}{4} - j \sin \frac{6\pi}{4} + \cos \frac{6\pi}{2} - j \sin \frac{6\pi}{2} \right]$

$$= 0.333 (1 + 0 + j1 - 1 - j0)$$

$$= j0.333 = 0.333 \angle \pi/2 = 0.333 \angle 0.5\pi$$

When $k = 7$; $X(7) = \frac{1}{3} \left[1 + \cos \frac{7\pi}{4} - j \sin \frac{7\pi}{4} + \cos \frac{7\pi}{2} - j \sin \frac{7\pi}{2} \right]$

$$= 0.333 (1 + 0.707 + j0.707 + 0 + j1)$$

$$= 0.568 + j0.568 = 0.803 \angle 0.785 = 0.803 \angle 0.25\pi$$

Phase angles
are in radians.

\ The 8-point DFT sequence $X(k)$ is given by,

$$X(k) = \{1 \angle 0, 0.803 \angle -0.25\pi, 0.333 \angle -0.5\pi, 0.139 \angle 0.25\pi, 0.333 \angle 0, 0.139 \angle -0.25\pi, 0.333 \angle 0.5\pi, 0.803 \angle 0.25\pi\}$$

\ \therefore Magnitude Function, $|X(k)| = \{1, 0.803, 0.333, 0.139, 0.333, 0.139, 0.333, 0.803\}$

Phase Function, $\angle X(k) = \{0, -0.25\pi, -0.5\pi, 0.25\pi, 0, -0.25\pi, 0.5\pi, 0.25\pi\}$

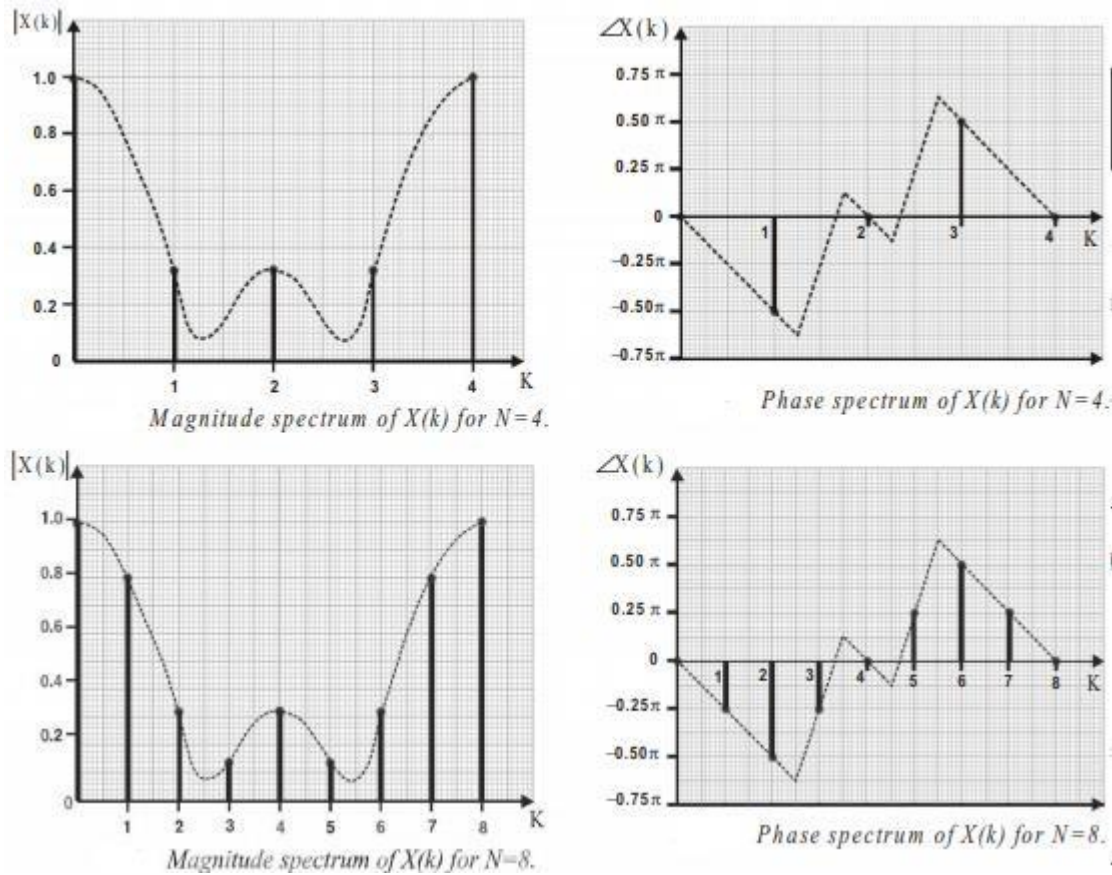


Fig 7. Magnitude and phasor representation of $N=4,8$ pont DFT Time Signals

[courtesy: DSP by Nagoorkani]

Fast Fourier Transform (FFT)

The Fast Fourier Transform (FFT) is a method (or algorithm) for computing the discrete Fourier transform (DFT) with reduced number of calculations. The computational efficiency is achieved if we adopt a divide and conquer approach. This approach is based on the decomposition of an N -point DFT into successively smaller DFTs. This basic approach leads to a family of an efficient computational algorithms known collectively as FFT algorithms. **Radix- r FFT** In an N -point sequence, if N can be expressed as $N = r^m$, then the sequence can be decimated into r -point sequences. For each r -point sequence, r -point DFT can be computed. From the results of r -point DFT, the r^2 -point DFTs are computed. From the results of r^2 -point DFTs, the r^3 -point DFTs are computed and so on, until we get r^m point DFT. This FFT algorithm is called radix- r FFT. In computing N -point DFT by this method the number of stages of computation will be m times.

Radix-2 FFT For radix-2 FFT, the value of N should be such that, $N = 2^m$, so that the N -point sequence is decimated into 2-point sequences and the 2-point DFT for each decimated sequence is computed. From the results of 2-point DFTs, the 4-point DFTs can be

computed. From the results of 4-point DFTs, the 8-point DFTs can be computed and so on, until we get N-point DFT.

Number of Calculations in N-point DFT

N^2 number of complex multiplications and $N(N - 1)$ number of complex additions

Number of Calculations in Radix-2 FFT

$N/2 \log_2 N$ complex multiplications and $N \log_2 N$ complex additions.

Radix-2 FFT algorithms:

Decimation-In-Time (DIT) FFT algorithm:

The algorithm in which the decimation is based on splitting the sequence $x(n)$ into successively smaller sequences is called the decimation-in-time algorithm.

The N-point DFT of a sequence $x(n)$ is given by

$$X(k) = \sum_{n=0}^{N-1} x(n) W_N^{nk}, \quad 0 \leq k \leq N-1 \quad (1)$$

where $W_N = e^{-j(2\pi/N)}$. $X(k)$ is periodic with period N i.e., $X(k+N) = X(k)$.

Splitting Equ(1) into two, one for even-indexed samples of $x(n)$ and the other for odd-indexed samples of $x(n)$, we have

$$X(k) = \sum_{\substack{n \text{ even}}} x(n) W_N^{nk} + \sum_{\substack{n \text{ odd}}} x(n) W_N^{nk} \quad (2)$$

Substituting $n=2n$ for n even and $n=2n+1$ for n odd, we have

$$X(k) = \sum_{n=0}^{N/2-1} x(2n) W_N^{2nk} + \sum_{n=0}^{N/2-1} x(2n+1) W_N^{(2n+1)k}$$

8-Point DFT Using Radix-2 DIT FFT

The input sequence is 8-point sequence. Therefore, $N = 8 = 2^3 = r^m$. Here, $r = 2$ and $m = 3$. Therefore, the computation of 8-point DFT using radix-2 FFT, involves three stages of computation. The given 8-point sequence is decimated to 2-point sequences. For each 2-point sequence, the 2-point DFT is computed. From the results of 2-point DFT, the 4-point DFT can be computed. From the results of 4-point DFT, the 8-point DFT can be computed.

Let the given sequence be $x(0)$, $x(1)$, $x(2)$, $x(3)$, $x(4)$, $x(5)$, $x(6)$, $x(7)$, which consists of 8 samples. The 8-samples should be decimated into sequences of 2-samples. Before decimation they are arranged in bit reversed order, as shown in table

Normal order		Bit reversed order	
$x(0)$	$x(000)$	$x(0)$	$x(000)$
$x(1)$	$x(001)$	$x(4)$	$x(100)$
$x(2)$	$x(010)$	$x(2)$	$x(010)$
$x(3)$	$x(011)$	$x(6)$	$x(110)$
$x(4)$	$x(100)$	$x(1)$	$x(001)$
$x(5)$	$x(101)$	$x(5)$	$x(101)$
$x(6)$	$x(110)$	$x(3)$	$x(011)$
$x(7)$	$x(111)$	$x(7)$	$x(111)$

Fig 8. Bit reversal order of 2 point DFT

Using the decimated sequences as input the 8-point DFT is computed. The fig shows the three stages of computation of an 8-point DFT.

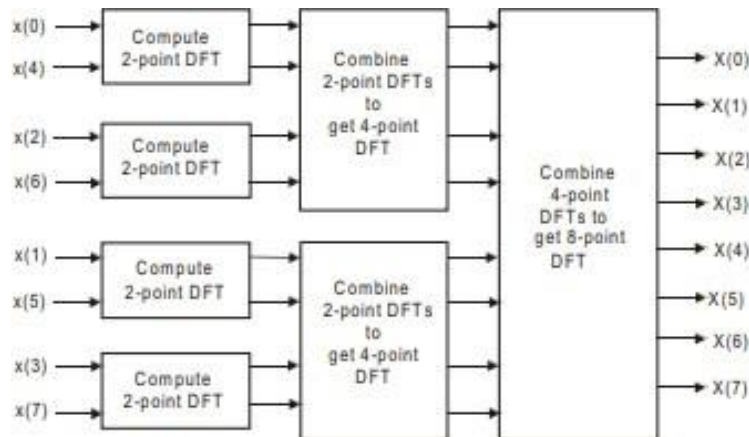


Fig 9. Block diagram representation of 8 pt DFT

Flow Graph for 8-Point DFT using Radix-2 DIT FFT

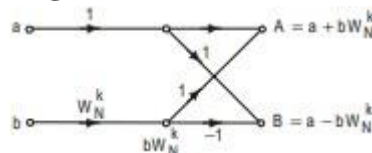


Fig 10. Basic butterfly or flow graph of DIT radix-2 FFT.

The signal flow graph is also called butterfly diagram since it resembles a butterfly

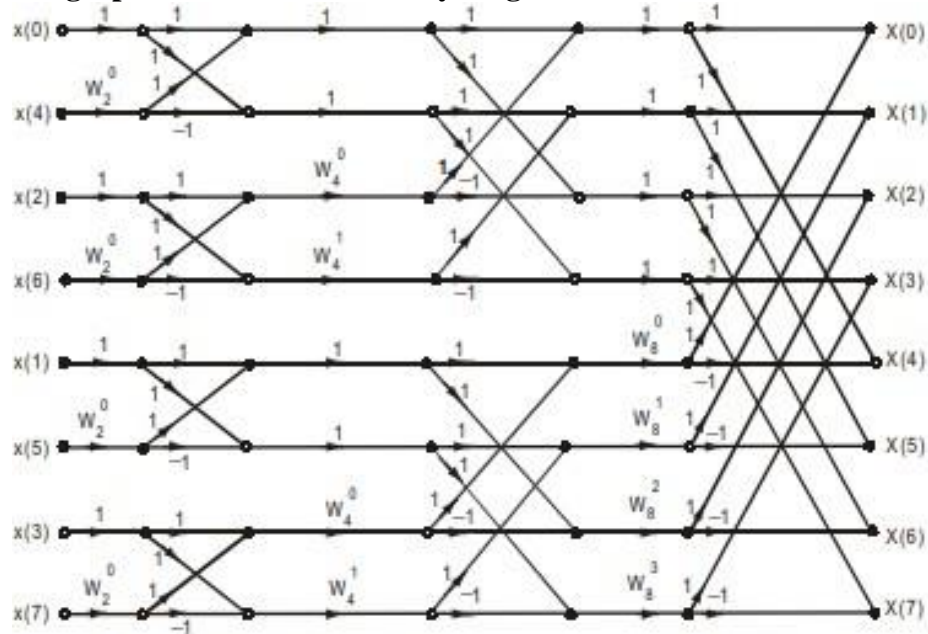


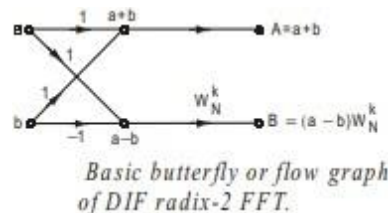
Fig 11. The flow graph (or butterfly diagram) for 8-point DFT via radix-2 DIT FFT.

8-point DFT Using Radix-2 DIF FFT

The DIF computation for an eight sequence is discussed in detail in this section. Let $x(n)$ be an 8-point sequence. Therefore $N = 8 = 2^3 = r^m$. Here, $r = 2$ and $m = 3$. Therefore, the computation of 8-point DFT using radix-2 FFT involves three stages of computation. The samples of $x(n)$ are, $x(0), x(1), x(2), x(3), x(4), x(5), x(6), x(7)$.

Flow Graph For 8-point DFT using Radix-2 DIF FFT

The above basic computation can be expressed by a signal flow graph shown in Fig



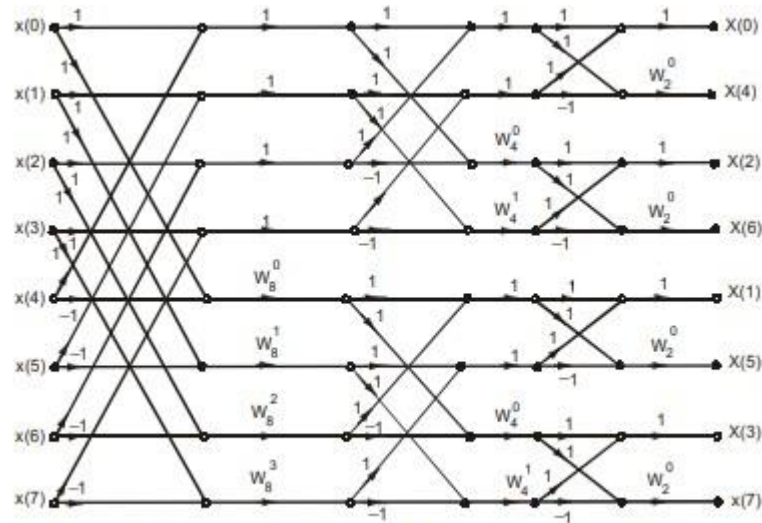


Fig 12. The flow graph (or butterfly diagram) for 8-point DFT via radix-2 DIF FFT.

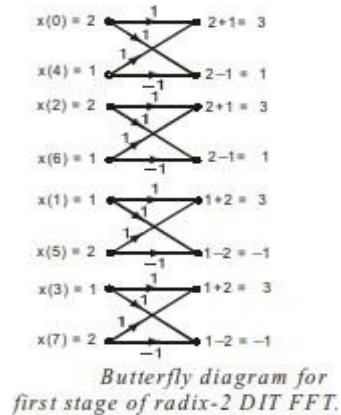
Problem:

An 8-point sequence is given by $x(n) = \{2, 1, 2, 1, 1, 2, 1, 2\}$. Compute 8-point DFT of $x(n)$ by
a) radix-2 DIT-FFT and b) radix-2 DIF-FFT. Also sketch the magnitude and phase spectrum.

a) 8-point DFT by Radix-2 DIT-FFT

The given sequence is first arranged in the bit reversed order

The sequence $x(n)$ in normal order	The sequence $x(n)$ in bit reversed order
$x(0) = 2$	$x(0) = 2$
$x(1) = 1$	$x(4) = 1$
$x(2) = 2$	$x(2) = 2$
$x(3) = 1$	$x(6) = 1$
$x(4) = 1$	$x(1) = 1$
$x(5) = 2$	$x(5) = 2$
$x(6) = 1$	$x(3) = 1$
$x(7) = 2$	$x(7) = 2$

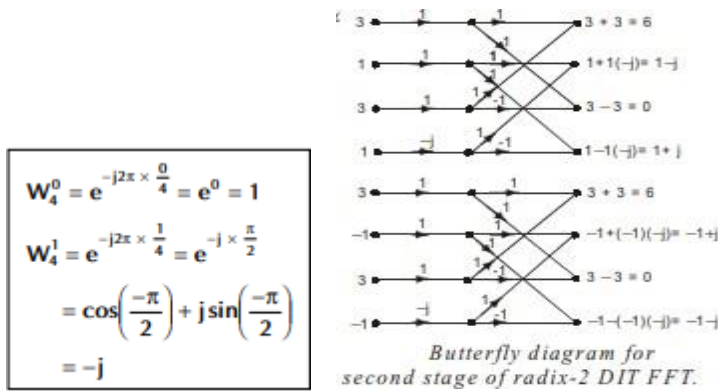


The 8-point DFT by radix-2 FFT involve 3 stages of computation with 4-butterfly computations in each stage. The sequence rearranged in the bit reversed order forms the input to the first stage. For other stages of computation the output of previous stage will be the input for current stage.

Second stage computation

The input sequence to second stage computation = $\{3, 1, 3, 1, 3, 1, 3, 1\}$

The phase factors involved in second stage computation are W_4^0 and W_4^1



Third stage computation The input sequence to third stage computation = {6, 1j, 0, 1+j, 6, 1+j, 0, 1j} The phase factors involved in third stage computation are W_8^0 , W_8^1 , W_8^2 and W_8^3

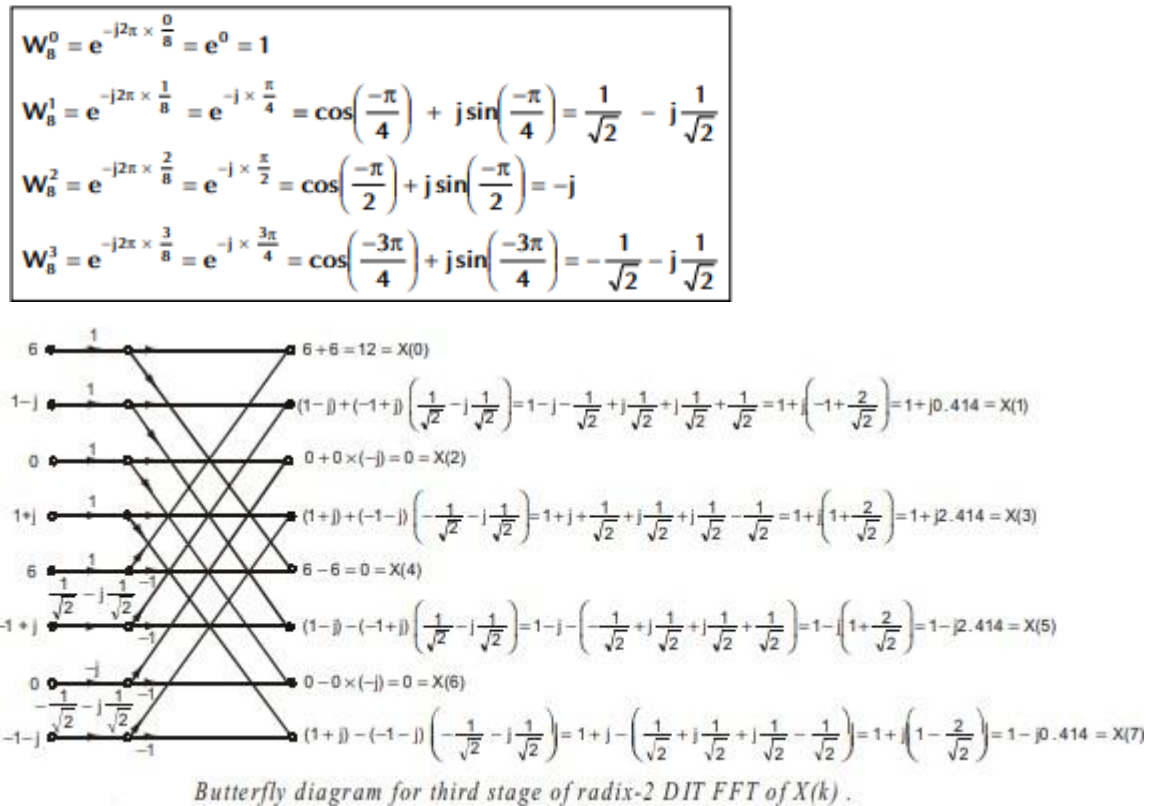


Fig 13. Butterfly diagram for third stage of radix-2 DIT FFT

b) 8-point DFT by Radix-2 DIF-FFT

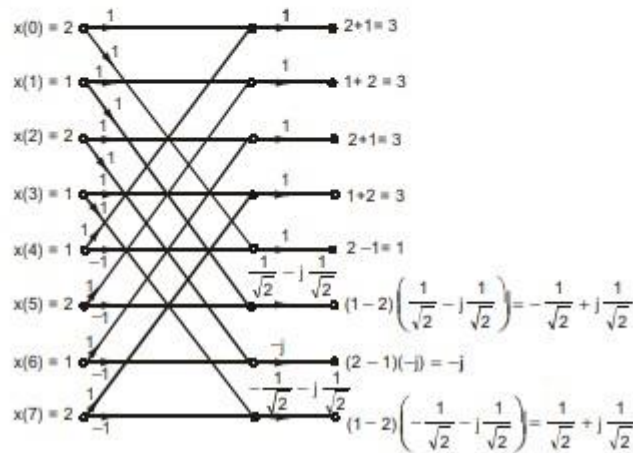
For 8-point DFT by radix-2 FFT we require 3-stages of computation with 4-butterfly computation in each stage. The given sequence is the input to first stage. For other stages of computations, the output of previous stage will be the input for current stage.

First stage computation

The input sequence for first stage of computation = { 2, 1, 2, 1, 1, 2, 1, 2 }

The phase factors involved in first stage computation are W_8^0 , W_8^1 , W_8^2 and W_8^3

$$\begin{aligned}
 W_8^0 &= e^{-j2\pi \times \frac{0}{8}} = 1 \\
 W_8^1 &= e^{-j2\pi \times \frac{1}{8}} = e^{-j\frac{\pi}{4}} = \cos\left(-\frac{\pi}{4}\right) + j\sin\left(-\frac{\pi}{4}\right) = \frac{1}{\sqrt{2}} - j\frac{1}{\sqrt{2}} \\
 W_8^2 &= e^{-j2\pi \times \frac{2}{8}} = e^{-j\frac{\pi}{2}} = \cos\left(-\frac{\pi}{2}\right) + j\sin\left(-\frac{\pi}{2}\right) = -j \\
 W_8^3 &= e^{-j2\pi \times \frac{3}{8}} = e^{-j\frac{3\pi}{4}} = \cos\left(-\frac{3\pi}{4}\right) + j\sin\left(-\frac{3\pi}{4}\right) = -\frac{1}{\sqrt{2}} - j\frac{1}{\sqrt{2}}
 \end{aligned}$$



Butterfly diagram for first stage of radix-2 DIF FFT.

Fig 14. Butterfly diagram for first stage of radix-2 DIT FFT

The output sequence of first

$$\text{stage of computation} = \left\{ 3, 3, 3, 3, 1, -\frac{1}{\sqrt{2}} + j\frac{1}{\sqrt{2}}, -j, \frac{1}{\sqrt{2}} + j\frac{1}{\sqrt{2}} \right\}$$

Second stage computation

The input sequence for second stage of computation =

$$\left\{ 3, 3, 3, 3, 1, -\frac{1}{\sqrt{2}} + j\frac{1}{\sqrt{2}}, -j, \frac{1}{\sqrt{2}} + j\frac{1}{\sqrt{2}} \right\}$$

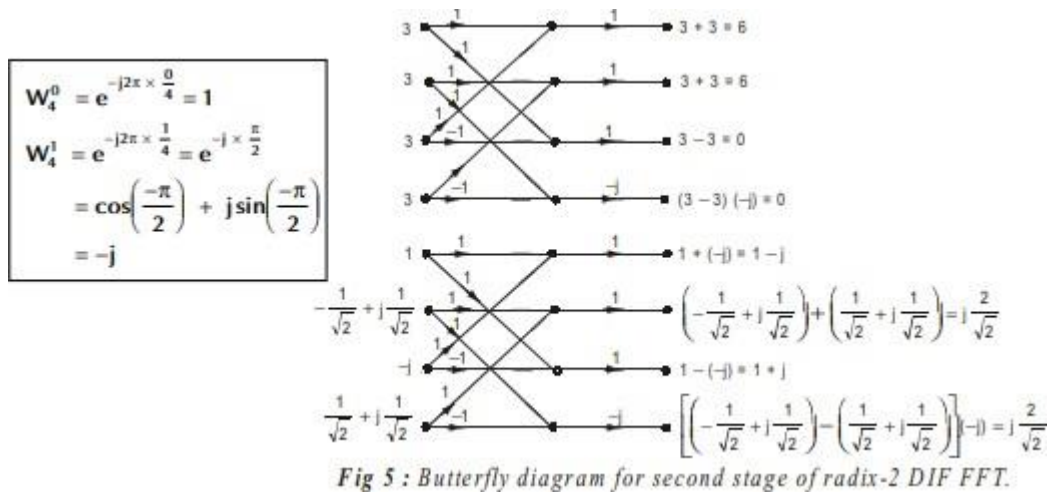


Fig 15. Butterfly diagram for second stage of radix-2 DIT FFT

The output sequence of second stage of computation = $\left\{6, 6, 0, 0, 1-j, j\frac{2}{\sqrt{2}}, 1+j, j\frac{2}{\sqrt{2}}\right\}$

Third stage computation

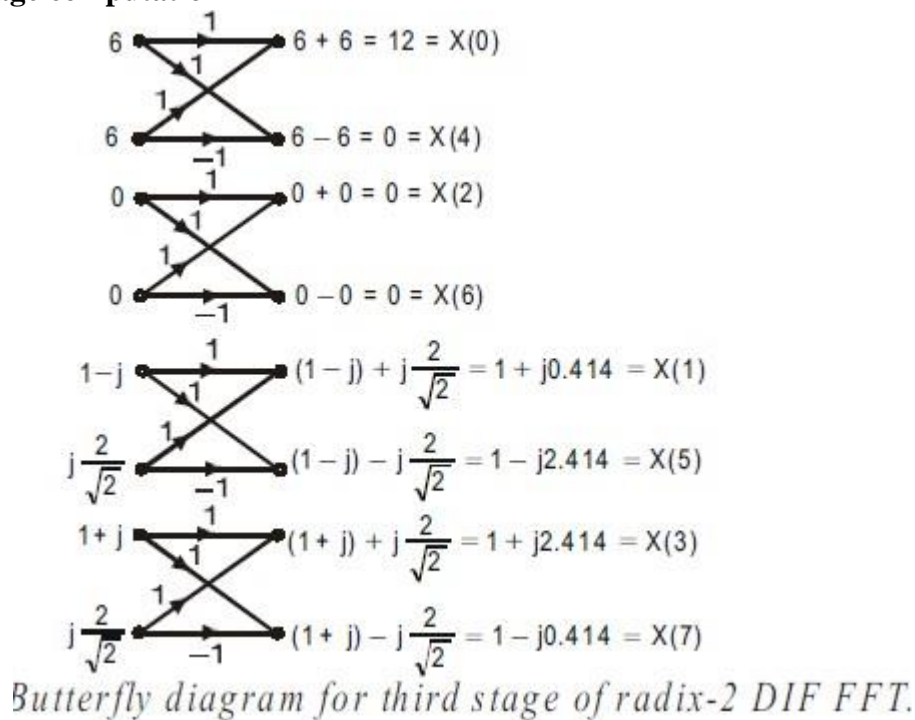


Fig 16. Butterfly diagram for third stage of radix-2 DIT FFT

Correlation

Correlation is a measure of similarity between two signals. The general formula for correlation is $\int_{-\infty}^{\infty} x_1(t) x_2(t - \tau) dt$

There are two types of correlation:

1. Auto correlation
2. Cross correlation

Auto Correlation Function

It is defined as correlation of a signal with itself. Auto correlation function is a measure of similarity between a signal & its time delayed version. It is represented with $R(\tau)$. Consider a signals $x(t)$. The auto correlation function of $x(t)$ with its time delayed version is given by

$$R_{11}(\tau) = R(\tau) = \int_{-\infty}^{\infty} x(t)x(t - \tau)dt \quad [+ve \text{ shift}]$$

$$= \int_{-\infty}^{\infty} x(t)x(t + \tau)dt \quad [-ve \text{ shift}]$$

Where τ = searching or scanning or delay parameter.

Properties

Auto correlation of power signal exhibits conjugate symmetry i.e. $R(-r) = (-T)$

Auto correlation function of power signal at $r = 0$ (at origin) is equal to total power of that signal. i.e. $R(0) = P$

Auto correlation function of power signal $R(0) \propto 1/T$. Auto correlation function of power signal is maximum at $r = 0$ i.e., $|R(r)| \leq R(0) \forall r$

Auto correlation function and power spectral densities are Fourier transform pairs. i.e., $F.T[R(T)] = S(\omega)$

$$S(\omega) = \int_{-\infty}^{\infty} R(\tau)e^{-j\omega\tau} d\tau$$

$$R(\tau) = x(\tau) * x(-\tau)$$

Cross Correlation Function

Cross correlation is the measure of similarity between two different signals. Consider two signals $x_1(t)$ and $x_2(t)$. The cross correlation of these two signals $R_{12}(\tau)$ is given by

$$R_{12}(\tau) = \int_{-\infty}^{\infty} x_1(t)x_2(t - \tau) dt \quad [+ve \text{ shift}]$$

$$= \int_{-\infty}^{\infty} x_1(t + \tau)x_2(t) dt \quad [-ve \text{ shift}]$$

Properties of Cross Correlation Function

Auto correlation exhibits conjugate symmetry i.e. $R_{12}(\tau) = R_{21}^*(-\tau)$.

Cross correlation is not commutative like convolution i.e. $R_{12}(\tau) \neq R_{21}(-\tau)$

If $R_{12}(0) = 0$ means, if $\int_{-\infty}^{\infty} x_1(t)x_2^*(t)dt = 0$, then the two signals are said to be orthogonal.

For power signal $\lim_{T \rightarrow \infty} \frac{1}{T} \int_{-\frac{T}{2}}^{\frac{T}{2}} x(t)x^*(t)dt$ then two signals are said to be orthogonal. Cross correlation function corresponds to the multiplication of spectrums of one signal to the complex conjugate of spectrum of another signal. i.e.

$$R_{12}(\tau) \longleftrightarrow X_1(\omega)X_2^*(\omega)$$

This also called as correlation theorem.

Realization of Discrete Time System:

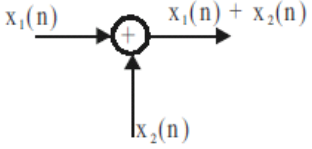
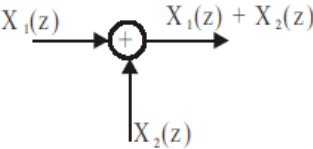
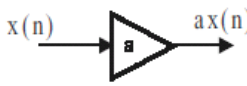
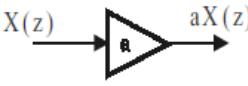
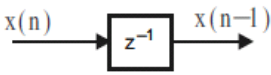
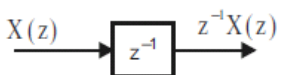
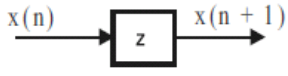
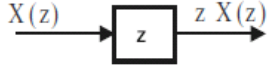
Discrete Time IIR System

Let, $H(z)$ = Transfer function of discrete time IIR system.

The general form of transfer function of IIR system is,

$$H(z) = \frac{b_0 + b_1 z^{-1} + b_2 z^{-2} + \dots + b_M z^{-M}}{1 + a_1 z^{-1} + a_2 z^{-2} + \dots + a_N z^{-N}}$$

Basic Elements of Block Diagram

Elements of block diagram	Time domain representation	z-domain representation
Adder		
Constant multiplier		
Unit delay element		
Unit advance element		

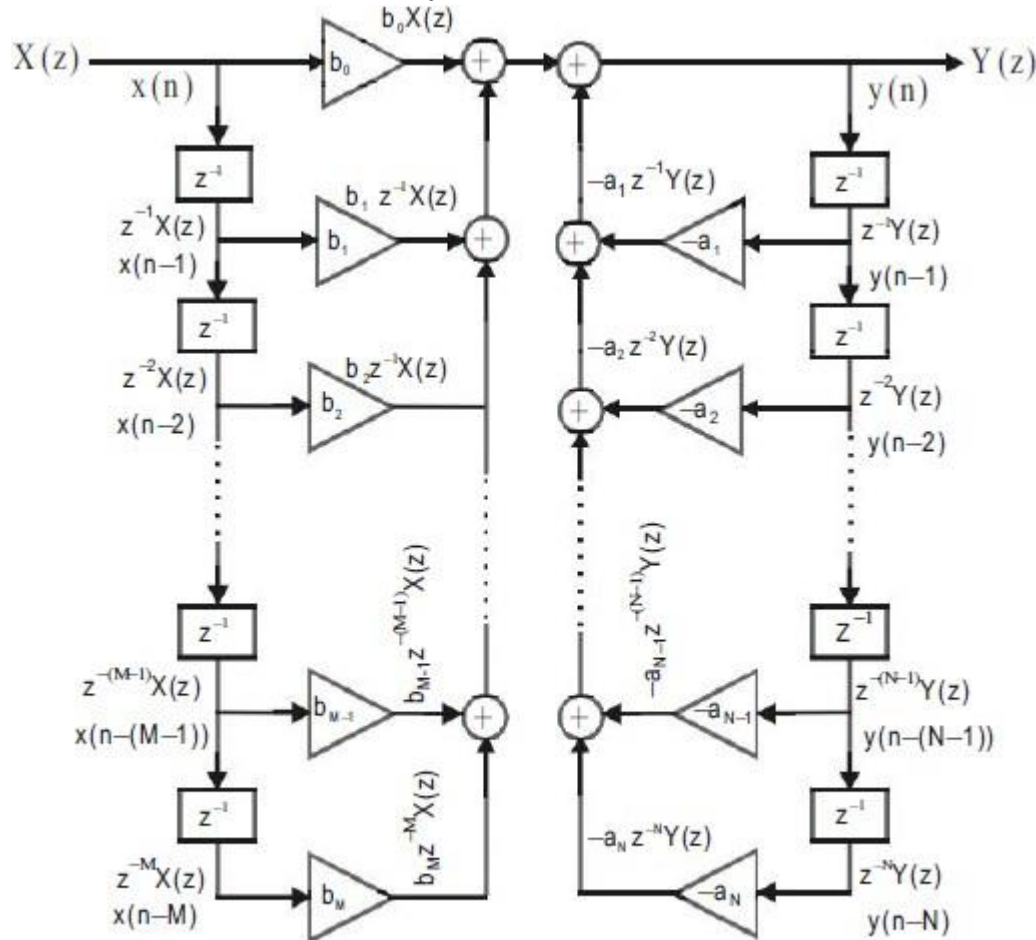
The different types of structures for realizing the IIR systems are,

1. Direct form-I structure
2. Direct form-II structure

3. Cascade form structure

4. Parallel form structure

Direct Form-I Structure of IIR System



Direct form -I structure of IIR system.

Fig.17. Direct form I structure of IIR system

From the direct form-I structure it is observed that the realization of an Nth order discrete time system with M number of zeros and N number of poles, involves $M+N+1$ number of multiplications and $M+N$ number of additions. Also this structure involves $M+N$ delays and so $M+N$ memory locations are required to store the delayed signals.

When the number of delays in a structure is equal to the order of the system, the structure is called canonic structure. In direct form-I structure the number of delays is not equal to order of the system and so direct form-I structure is noncanonic structure.

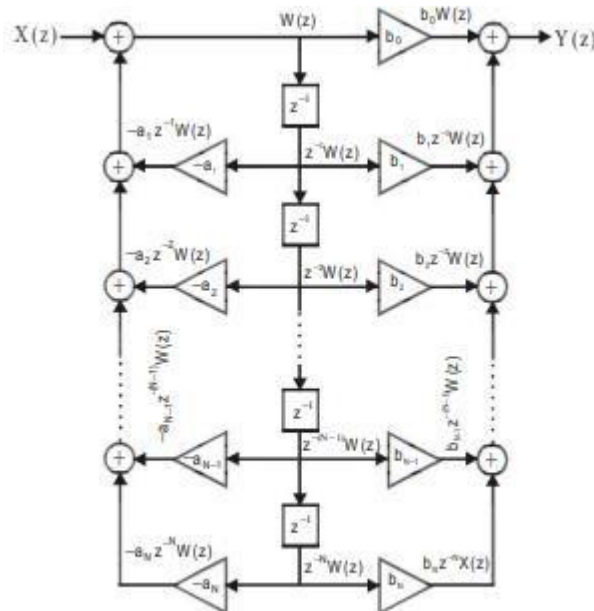
Direct Form-II Structure of IIR System

An alternative structure called direct form-II structure can be realized which uses less number of delay elements than the direct form-I structure.

$$\text{Let, } \frac{Y(z)}{X(z)} = \frac{W(z)}{X(z)} \times \frac{Y(z)}{W(z)}$$

$$\text{where, } \frac{W(z)}{X(z)} = \frac{1}{1 + a_1 z^{-1} + a_2 z^{-2} + \dots + a_N z^{-N}}$$

$$\frac{Y(z)}{W(z)} = b_0 + b_1 z^{-1} + b_2 z^{-2} + \dots + b_M z^{-M}$$



Direct form-II structure of IIR system for $N = M$.

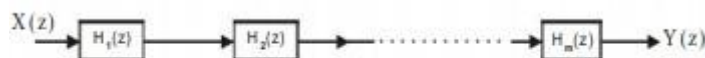
Fig.18.Direct form II structure of IIR system

Cascade Form Realization of IIR System The transfer function $H(z)$ can be expressed as a product of a number of second-order or first-order sections

$$H(z) = \frac{Y(z)}{X(z)} = H_1(z) \times H_2(z) \times H_3(z) \dots H_m(z) = \prod_{i=1}^m H_i(z)$$

$$\text{where, } H_i(z) = \frac{c_{0i} + c_{1i} z^{-1} + c_{2i} z^{-2}}{d_{0i} + d_{1i} z^{-1} + d_{2i} z^{-2}}$$

$$\text{or, } H_i(z) = \frac{c_{0i} + c_{1i} z^{-1}}{d_{0i} + d_{1i} z^{-1}}$$



Cascade form realization of IIR system.

Fig.19.cascade form realization structure of IIR system

Parallel Form Realization of IIR System The transfer function $H(z)$ of a discrete time system can be expressed as a sum of first and second-order sections, using partial fraction expansion technique

Fig.19.cascade form realization structure of IIR system

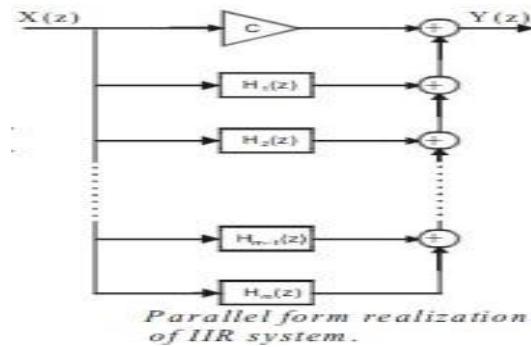


Fig.20.Parallel form realization structure of IIR system

TEXT / REFERENCE BOOKS:

1. John G. Proakis & Dimitris G.Manolakis, “Digital Signal Processing - Principles, Algorithms & Applications”, Fourth Edition, Pearson education / Prentice Hall, 2009
2. Sanjit K. Mitra , Digital Signal Processing: A Computer - Based Approach, McGrawHill Education, 4th Edition,2013
3. B.P.Lathi, “Signal Processing & Linear systems”, Oxford University Press, 2nd edition, 2009
4. Lyons, “Understanding Digital Signal Processing”, Prentice Hall, 3rd edition, 2010
5. Johnny R. Johnson, “Introduction to Digital Signal Processing”, PHI, 2006
6. Alan V. Oppenheim, Ronald W. Schaffer, Discrete-Time Signal Processing, Pearson, 3rd Edition,2010
7. Salivahanan, “Digital Signal Processing, 2nd Edition, TMH, 2010.
8. A.Nagoor Kani, “Digital Signal Processing”, Tata McGrawHill Education, 4th Edition, 2012



SATHYABAMA

INSTITUTE OF SCIENCE AND TECHNOLOGY
(DEEMED TO BE UNIVERSITY)

Accredited "A" Grade by NAAC | 12B Status by UGC | Approved by AICTE

www.sathyabama.ac.in

SCHOOL OF ELECTRICAL AND ELECTRONICS

**DEPARTMENT OF ELECTRONICS AND COMMUNICATION
ENGINEERING**

UNIT - II

Digital Signal Processing

SECA1506

II. FINITE IMPULSE RESPONSE DIGITAL FILTERS

2.1 Symmetric and Antisymmetric FIR filters

FIR filters are digital filters with finite impulse response. They are also known as non-recursive digital filters as they do not have the feedback (a recursive part of a filter), even though recursive algorithms can be used for FIR filter realization. FIR filters can be designed using different methods, but most of them are based on ideal filter approximation. The objective is not to achieve ideal characteristics, as it is impossible anyway, but to achieve sufficiently good characteristics of a filter. The transfer function of FIR filter approaches the ideal as the filter order increases, thus increasing the complexity and amount of time needed for processing input samples of a signal being filtered. The resulting frequency response can be a monotone function or an oscillatory function within a certain frequency range. The waveform of frequency response depends on the method used in design process as well as on its parameters.

This book describes the most popular method for FIR filter design that uses window functions. The characteristics of the transfer function as well as its deviation from the ideal frequency response depend on the filter order and window function in use.

Each filter category has both advantages and disadvantages. This is the reason why it is so important to carefully choose category and type of a filter during design process.

FIR filters can have linear phase characteristic, which is not like IIR filters that will be discussed in Chapter 3. Obviously, in such cases when it is necessary to have a linear phase characteristic, FIR filters are the only option available. If the linear phase characteristic is not necessary, as is the case with processing speech signals, FIR filters are not good solution at all.

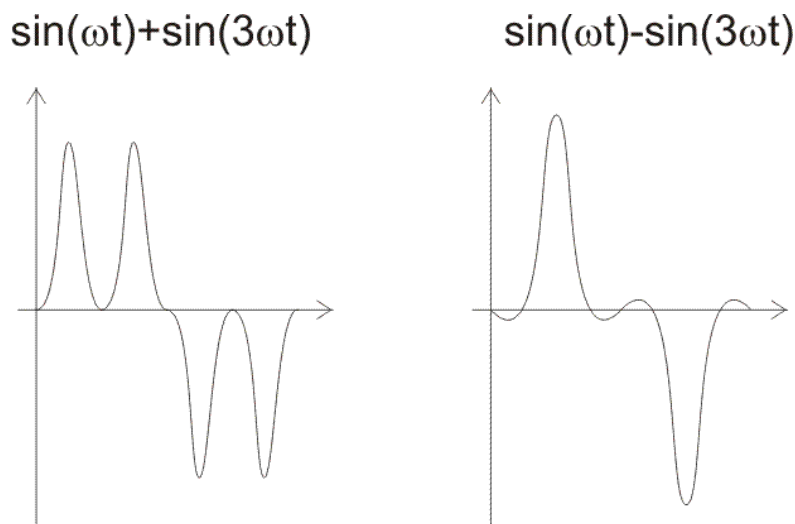


Fig.2.1. Illustration of input and output signals of non-linear phase systems.

The system introduces a phase shift of 0 radians at the frequency of ω , and π radians at three times that frequency. Input signal consists of natural frequency ω and one harmonic with the same amplitude at three times that frequency. Figure 2-1. shows the block diagram of input signal (left) and output signal (right). It is obvious that these two signals have different waveforms. The power of signals is not changed, nor the amplitudes of harmonics, only the phase of the second harmonic is changed.

If we assume that the input is a speech signal whose phase characteristic is not of the essence, such distortion in the phase of the signal would be unimportant. In this case, the system satisfies all necessary requirements. However, if the phase characteristic is of importance, such a great distortion mustn't be allowed.

In order that the phase characteristic of a FIR filter is linear, the impulse response must be symmetric or anti-symmetric, which is expressed in the following way:

$h[n] = h[N-n-1]$; symmetric impulse response (about its middle element)

$h[n] = -h[N-n-1]$; anti-symmetric impulse response (about its middle element)

One of the drawbacks of FIR filters is a high order of designed filter. The order of FIR filter is remarkably higher compared to an IIR filter with the same frequency response. This is the reason why it is so important to use FIR filters only when the linear phase characteristic is very important.

A number of delay lines contained in a filter, i.e. a number of input samples that should be saved for the purpose of computing the output sample, determines the order of a filter. For example, if the filter is assumed to be of order 10, it means that it is necessary to save 10 input samples preceeding the current sample. All eleven samples will affect the output sample of FIR filter.

The transform function of a typical FIR filter can be expressed as a polynomial of a complex variable z^{-1} . All the poles of the transfer function are located at the origin. For this reason, FIR filters are guaranteed to be stable, whereas IIR filters have potential to become unstable.

Finite impulse response (FIR) filter design methods

Most FIR filter design methods are based on ideal filter approximation. The resulting filter approximates the ideal characteristic as the filter order increases, thus making the filter and its implementation more complex.

The filter design process starts with specifications and requirements of the desirable FIR filter. Which method is to be used in the filter design process depends on the filter specifications and implementation. This chapter discusses the FIR filter design method using window functions.

Each of the given methods has its advantages and disadvantages. Thus, it is very important to carefully choose the right method for FIR filter design. Due to its simplicity and efficiency, the window method is most commonly used method for designing filters. The sampling frequency method is easy to use, but filters designed this way have small attenuation in the stopband.

As we have mentioned above, the design process starts with the specification of desirable FIR filter.

Basic concepts and FIR filter specification

First of all, it is necessary to learn the basic concepts that will be used further in this book. You should be aware that without being familiar with these concepts, it is not possible to understand analyses and synthesis of digital filters.

Figure 2.2 illustrates a low-pass digital filter specification. The word specification actually refers to the frequency response specification.

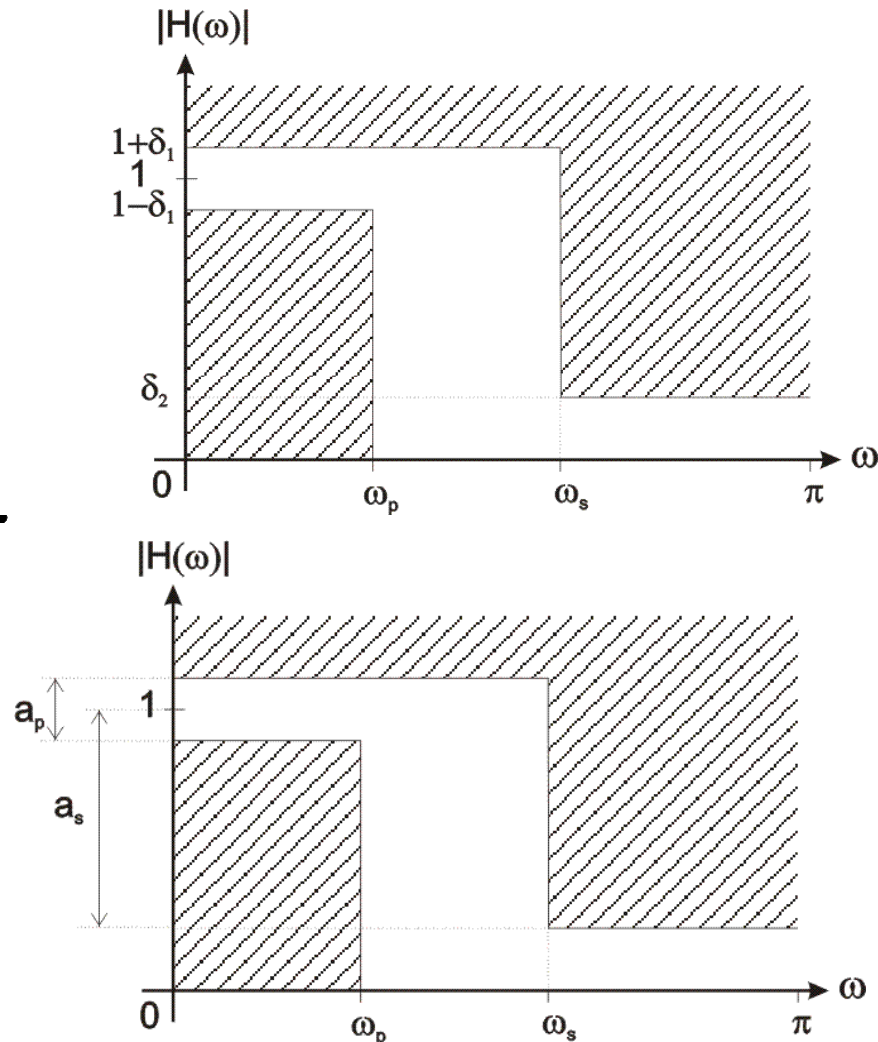


Fig.2.2. A low-pass digital filter specification

- ω_p – normalized cut-off frequency in the passband;
- ω_s – normalized cut-off frequency in the stopband;
- δ_1 – maximum ripples in the passband;
- δ_2 – minimum attenuation in the stopband [dB];

- a_p – maximum ripples in the passband; and
- a_s – minimum attenuation in the stopband [dB].

$$a_p = 20 \log_{10} \left(\frac{1 + \delta_1}{1 - \delta_1} \right)$$

$$a_s = -20 \log_{10} \delta_2$$

Frequency normalization can be expressed as follows:

$$\omega = \frac{2\pi f}{f_s}$$

where:

- f_s is a sampling frequency;
- f is a frequency to normalize; and
- ω is normalized frequency.

Table.3.1.Filters

Type of filter	Frequency response $h_d[n]$
low-pass filter	$h_d[n] = \begin{cases} \frac{\sin[\omega_c(n-M)]}{\pi(n-M)}; & n \neq M \\ \frac{\omega_c}{\pi}; & n = M \end{cases}$
high-pass filter	$h_d[n] = \begin{cases} 1 - \frac{\omega_c}{\pi}; & n \neq M \\ -\frac{\sin(\omega_c(n-M))}{\pi(n-M)}; & n = M \end{cases}$
band-pass filter	$h_d[n] = \begin{cases} \frac{\sin(\omega_{c2}(n-M))}{\pi(n-M)} - \frac{\sin(\omega_{c1}(n-M))}{\pi(n-M)}; & n \neq M \\ \frac{\omega_{c2} - \omega_{c1}}{\pi}; & n = M \end{cases}$
band-stop filter	$h_d[n] = \begin{cases} \frac{\sin(\omega_{c1}(n-M))}{\pi(n-M)} - \frac{\sin(\omega_{c2}(n-M))}{\pi(n-M)}; & n \neq M \\ 1 - \frac{\omega_{c2} - \omega_{c1}}{\pi}; & n = M \end{cases}$

The value of variable n ranges between 0 and N , where N is the filter order. A constant M can be expressed as $M = N / 2$. Equivalently, N can be expressed as $N = 2M$.

The constant M is an integer if the filter order N is even, which is not the case with odd order filters. If M is an integer (even filter order), the ideal filter frequency response is symmetric about its M th sample which is found via expression shown in the table 2-2-1 above. If M is not an integer, the ideal filter frequency response is still symmetric, but not about some frequency response sample.

Since the variable n ranges between 0 and N , the ideal filter frequency response has $N+1$ sample.

If it is needed to find frequency response of a non-standard ideal filter, the expression for inverse Fourier transform must be used:

$$h_d[n] = \frac{1}{\pi} \int_0^{\pi} e^{j\omega(n-M)} d\omega$$

Non-standard filters are rarely used. However, if there is a need to use some of them, the integral above must be computed via various numerical methods.

FIR filter design using window functions

The FIR filter design process via window functions can be split into several steps:

1. Defining filter specifications;
2. Specifying a window function according to the filter specifications;
3. Computing the filter order required for a given set of specifications;
4. Computing the window function coefficients;
5. Computing the ideal filter coefficients according to the filter order;
6. Computing FIR filter coefficients according to the obtained window function and ideal filter coefficients;
7. If the resulting filter has too wide or too narrow transition region, it is necessary to change the filter order by increasing or decreasing it according to needs, and after that steps 4, 5 and 6 are iterated as many times as needed.

The final objective of defining filter specifications is to find the desired normalized frequencies (ω_c , ω_{c1} , ω_{c2}), transition width and stopband attenuation. The window function and filter order are both specified according to these parameters.

Accordingly, the selected window function must satisfy the given specifications. After this step, that is, when the window function is known, we can compute the filter order required for a given set of specifications. When both the window function and filter order are known, it is possible to calculate the window function coefficients $w[n]$ using the formula for the specified window function.

1. Rectangular Window The rectangular window is what you would obtain if you were to simply segment a finite portion of the impulse response without any shaping in the time domain:

$$w(n) = 1 \quad 0 \leq n \leq M, \\ = 0 \text{ otherwise}$$

2. Bartlett (or triangular) window

The Bartlett window is triangularly shaped:

$$w(n) = 1 - \frac{2n}{M} \quad 0 \leq n \leq M, \\ = 0 \text{ otherwise}$$

3. Hanning window

The Hanning window (or more properly, the von Hann window) is nothing more than a raised cosine:

$$w(n) = 0.5 - 0.5 \cos \left(\frac{2\pi n}{M} \right) \quad 0 \leq n \leq M, \\ = 0 \text{ otherwise}$$

4. Hamming window

$$w(n) = 0.54 - 0.46 \cos \left(\frac{2\pi n}{M} \right) \quad 0 \leq n \leq M, \\ = 0 \text{ otherwise}$$

5. Blackman window

The Hanning and Hamming have a constant and a cosine term; the Blackman window adds a cosine at twice the frequency

$$w(n) = 0.42 - 0.5 \cos \left(\frac{2\pi n}{M} \right) + 0.08 \cos \left(\frac{4\pi n}{M} \right) \quad 0 \leq n \leq M, \\ = 0 \text{ otherwise}$$

After estimating the window function coefficients, it is necessary to find the ideal filter frequency samples. The expressions used for computing these samples are discussed in section 2.2.3 under Ideal filter approximation. The final objective of this step is to obtain the coefficients $h_d[n]$. Two sequences $w[n]$ and $h_d[n]$ have the same number of elements.

The next step is to compute the frequency response of designed filter $h[n]$ using the following expression:

$$h[n] = w[n] \cdot h_d[n]$$

Lastly, the transfer function of designed filter will be found by transforming impulse response via Fourier transform:

$$H(e^{j\omega}) = \sum_{n=0}^N h[n] \cdot e^{-jn\omega}$$

or via Z-transform:

$$H(z) = \sum_{n=0}^N h[n]z^{-n}$$

If the transition region of designed filter is wider than needed, it is necessary to increase the filter order, reestimate the window function coefficients and ideal filter frequency samples, multiply them in order to obtain the frequency response of designed filter and reestimate the transfer function as well. If the transition region is narrower than needed, the filter order can be decreased for the purpose of optimizing hardware and/or software resources. It is also necessary to reestimate the filter frequency coefficients after that.

PROBLEMS

Use the window design method to design a linear phase FIR filter of order $N = 24$ to approximate the following ideal frequency response magnitude

$$|H_d(e^{j\omega})| = \begin{cases} 1 & |\omega| \leq 0.2\pi \\ 0 & 0.2\pi < |\omega| \leq \pi \end{cases}$$

The ideal filter that we would like to approximate is a low-pass filter with a cutoff frequency = 0.2. With $N = 24$, the frequency response of the filter that is to be designed has the form

$$H(e^{j\omega}) = \sum_{n=0}^{24} h(n)e^{-jn\omega}$$

Therefore, the delay of $h(n)$ is $N/2 = 12$, and the ideal unit sample response that is to be windowed is

$$h_d(n) = \frac{\sin[0.2\pi(n - 12)]}{(n - 12)\pi}$$

All that is left to do in the design is to select a window. With the length of the window fixed, there is a trade-off between the width of the transition band and the amplitude of the passband and stopband ripple. With a rectangular window, which provides the smallest transition band,

$$\Delta\omega = 2\pi \cdot \frac{0.9}{24} = 0.075\pi$$

and the filter is

$$h(n) = \begin{cases} \frac{\sin[0.2\pi(n-12)]}{(n-12)\pi} & 0 \leq n \leq 24 \\ 0 & \text{otherwise} \end{cases}$$

However, the stopband attenuation is only 21 dB, which is equivalent to a ripple of 0.089. With a Hamming window, on the other hand,

$$h(n) = \left[0.54 - 0.46 \cos\left(\frac{2\pi n}{24}\right) \right] \cdot \frac{\sin[0.2\pi(n-12)]}{(n-12)\pi} \quad 0 \leq n \leq 24$$

and the stopband attenuation is 53 dB, or $\delta_s = 0.0022$. However, the width of the transition band increases to

$$\Delta\omega = 2\pi \cdot \frac{3.3}{24} = 0.275\pi$$

which, for most designs, would be too wide.

.Frequency sampling method:

The frequency sampling method allows us to design recursive and nonrecursive FIR filters for both standard frequency selective and filters with arbitrary frequency response. A. No recursive frequency sampling filters : The problem of FIR filter design is to find a finite-length impulse response $h(n)$ that corresponds to desired frequency response. In this method $h(n)$ can be determined by uniformly sampling, the desired frequency response $H_D(\omega)$ at the N points and finding its inverse DFT of the frequency samples.

Problem

Design of Optimum Equiripple Linear-Phase FIR

The window method and the frequency-sampling method are relatively simple

techniques for designing linear-phase FIR filters. However, they also possess some minor disadvantages, which may render them undesirable for some applications. A major problem is the lack of precise control of the critical frequencies such as ω_s . The filter design method described in this section is formulated as a Chebyshev approximation problem. It is viewed as an optimum design criterion in the sense that the weighted approximation error between the desired frequency response and the actual frequency response is spread evenly across the passband and evenly across the stopband of the filter minimizing the maximum error. The resulting

filter designs have ripples in both the passband and the stopband. To describe the design procedure, let us consider the design of a lowpass filter with passband edge frequency ω_p and stopband edge frequency ω_s .

Structure realization of FIR Filters

In signal processing, a digital filter is a system that performs mathematical operations on a sampled, discrete-time signal to reduce or enhance certain aspects of that signal. This is in contrast to the other major type of electronic filter, the analog filter, which is an electronic circuit operating on continuous-time analog signals.

A digital filter system usually consists of an analog-to-digital converter to sample the input signal, followed by a microprocessor and some peripheral components such as memory to store data and filter coefficients etc. Finally a digital-to-analog converter to complete the output stage. Program Instructions (software) running on the microprocessor implement the digital filter by performing the necessary mathematical operations on the numbers received from the ADC. In some high performance applications, an FPGA or ASIC is used instead of a general purpose microprocessor, or a specialized DSP with specific paralleled architecture for expediting operations such as filtering.

Digital filters may be more expensive than an equivalent analog filter due to their increased complexity, but they make practical many designs that are impractical or impossible as analog filters. When used in the context of real-time analog systems, digital filters sometimes have problematic latency (the difference in time between the input and the response) due to the associated analog-to-digital and digital-to-analog conversions and anti-aliasing filters, or due to other delays in their implementation.

Digital filters are commonplace and an essential element of everyday electronics such as radios, cellphones, and AV receivers.

Characterization

A digital filter is characterized by its transfer function, or equivalently, its difference equation. Mathematical analysis of the transfer function can describe how it will respond to any input. As such, designing a filter consists of developing specifications appropriate to the problem (for example, a second-order low pass filter with a specific cut-off frequency), and then producing a transfer function which meets the specifications.

The transfer function for a linear, time-invariant, digital filter can be expressed as a transfer function in the Z-domain; if it is causal, then it has the form:

$$H(z) = \frac{B(z)}{A(z)} = \frac{b_0 + b_1 z^{-1} + b_2 z^{-2} + \dots + b_N z^{-N}}{1 + a_1 z^{-1} + a_2 z^{-2} + \dots + a_M z^{-M}}$$

where the order of the filter is the greater of N or M . See Z-transform's LCCD equation for further discussion of this transfer function.

This is the form for a recursive filter with both the inputs (Numerator) and outputs (Denominator), which typically leads to an IIR infinite impulse response behaviour, but if

the denominator is made equal to unity i.e. no feedback, then this becomes an FIR or finite impulse response filter.

The impulse response, often denoted $h(k)$ or h_k , is a measurement of how a filter will respond to the Kronecker delta function. Digital filters are typically considered in two categories: infinite impulse response (IIR) and finite impulse response (FIR). In the case of linear time-invariant FIR filters, the impulse response is exactly equal to the sequence of filter coefficients:

$$y_n = \sum_{k=0}^{n-1} h_k x_{n-k}$$

IIR filters on the other hand are recursive, with the output depending on both current and previous inputs as well as previous outputs. The general form of an IIR filter is thus:

$$\sum_{m=0}^{M-1} a_m y_{n-m} = \sum_{k=0}^{N-1} b_k x_{n-k}$$

Plotting the impulse response will reveal how a filter will respond to a sudden, momentary disturbance.

1. Difference equation

In discrete-time systems, the digital filter is often implemented by converting the transfer function to a linear constant-coefficient difference equation (LCCD) via the Z-transform. The discrete frequency-domain transfer function is written as the ratio of two polynomials. For example:

$$H(z) = \frac{(z+1)^2}{(z-\frac{1}{2})(z+\frac{3}{4})}$$

This is expanded:

$$H(z) = \frac{z^2 + 2z + 1}{z^2 + \frac{1}{4}z - \frac{3}{8}}$$

and to make the corresponding filter causal, the numerator and denominator are divided by the highest order of z :

$$H(z) = \frac{1 + 2z^{-1} + z^{-2}}{1 + \frac{1}{4}z^{-1} - \frac{3}{8}z^{-2}} = \frac{Y(z)}{X(z)}$$

The coefficients of the denominator, a_k are the 'feed-backward' coefficients and the coefficients of the numerator are the 'feed-forward' coefficients, b_k . The resultant linear difference equation is:

$$y[n] = - \sum_{k=1}^M a_k y[n-k] + \sum_{k=0}^N b_k x[n-k]$$

or, for the example above:

$$\frac{Y(z)}{X(z)} = \frac{1 + 2z^{-1} + z^{-2}}{1 + \frac{1}{4}z^{-1} - \frac{3}{8}z^{-2}}$$

rearranging terms:

$$\Rightarrow (1 + \frac{1}{4}z^{-1} - \frac{3}{8}z^{-2})Y(z) = (1 + 2z^{-1} + z^{-2})X(z)$$

then by taking the inverse z-transform:

$$\Rightarrow y[n] + \frac{1}{4}y[n-1] - \frac{3}{8}y[n-2] = x[n] + 2x[n-1] + x[n-2]$$

and finally, by solving for $y[n]$:

$$y[n] = -\frac{1}{4}y[n-1] + \frac{3}{8}y[n-2] + x[n] + 2x[n-1] + x[n-2]$$

This equation shows how to compute the next output sample, $y[n]$, in terms of the past outputs, $y[n-p]$, the present input, $x[n]$, and the past inputs. Applying the filter to an input in this form is equivalent to a Direct Form I or II realization, depending on the exact order of evaluation. After a filter is designed, it must be *realized* by developing a signal flow diagram that describes the filter in terms of operations on sample sequences.

A given transfer function may be realized in many ways. Consider how a simple expression such as $ax + bx + c$ could be evaluated – one could also compute the

equivalent $x(a + b) + c$. In the same way, all realizations may be seen as "factorizations" of the same transfer function, but different realizations will have different numerical properties. Specifically, some realizations are more efficient in terms of the number of operations or storage elements required for their implementation, and others provide advantages such as improved numerical stability and reduced round-off error. Some structures are better for fixed-point arithmetic and others may be better for floating-point arithmetic.

1. Direct Form I

A straightforward approach for IIR filter realization is Direct Form I, where the difference equation is evaluated directly. This form is practical for small filters, but may be inefficient and impractical (numerically unstable) for complex designs.^[3] In general, this form requires $2N$ delay elements (for both input and output signals) for a filter of order N .

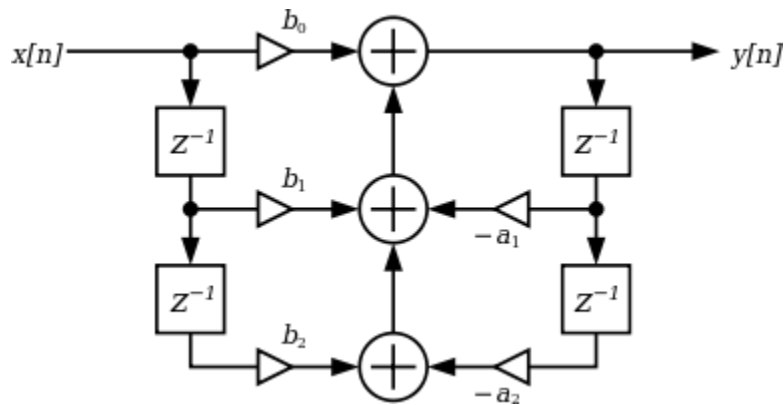


Fig.2.3. Direct form I

2. Direct Form II

The alternate Direct Form II only needs N delay units, where N is the order of the filter – potentially half as much as Direct Form I. This structure is obtained by reversing the order of the numerator and denominator sections of Direct Form I, since they are in fact two linear systems, and the commutativity property applies. Then, one will notice that there are two columns of delays (z^{-1}) that tap off the center net, and these can be combined since they are redundant, yielding the implementation as shown below.

The disadvantage is that Direct Form II increases the possibility of arithmetic overflow for filters of high Q or resonance.^[4] It has been shown that as Q increases, the round-off noise of both direct form topologies increases without bounds.^[5] This is because, conceptually, the signal is first passed through an all-pole filter (which normally boosts gain at the resonant frequencies) before the result of that is saturated, then passed through an all-zero filter (which often attenuates much of what the all-pole half amplifies).

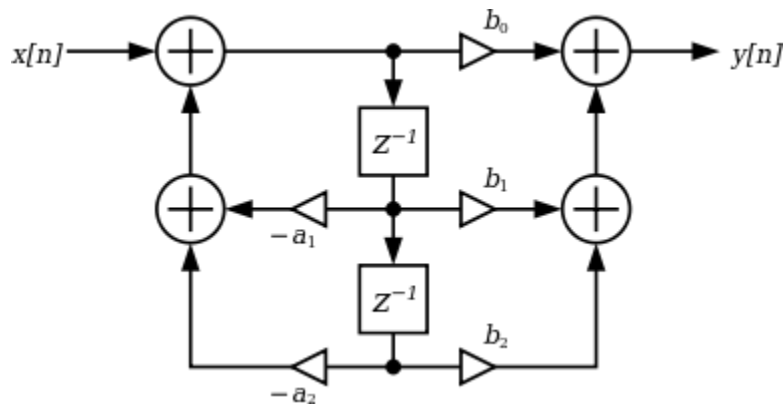


Fig.2.4. Direct form II

3. Cascaded second-order sections

A common strategy is to realize a higher-order (greater than 2) digital filter as a cascaded series of second-order "biquadratic" (or "biquad") sections^[6] (see digital biquad filter). The advantage of this strategy is that the coefficient range is limited.

Cascading direct form II sections results in N delay elements for filters of order N. Cascading direct form I sections results in N+2 delay elements since the delay elements of the input of any section (except the first section) are redundant with the delay elements of the output of the preceding section.

4. Linear-Phase FIR Structures

The symmetry (or antisymmetry) property of a linear-phase FIR filter can be exploited to reduce the number of multipliers into almost half of that in the direct form implementations

Consider a length-7 Type 1 FIR transfer function with a symmetric impulse response:

$$H(Z) = h(0) + h(1)Z^{-1} + h(2)Z^{-2} + h(3)Z^{-3} + h(2)Z^{-4} + h(1)Z^{-5} + h(0)Z^{-6}$$

Rearranging, we get

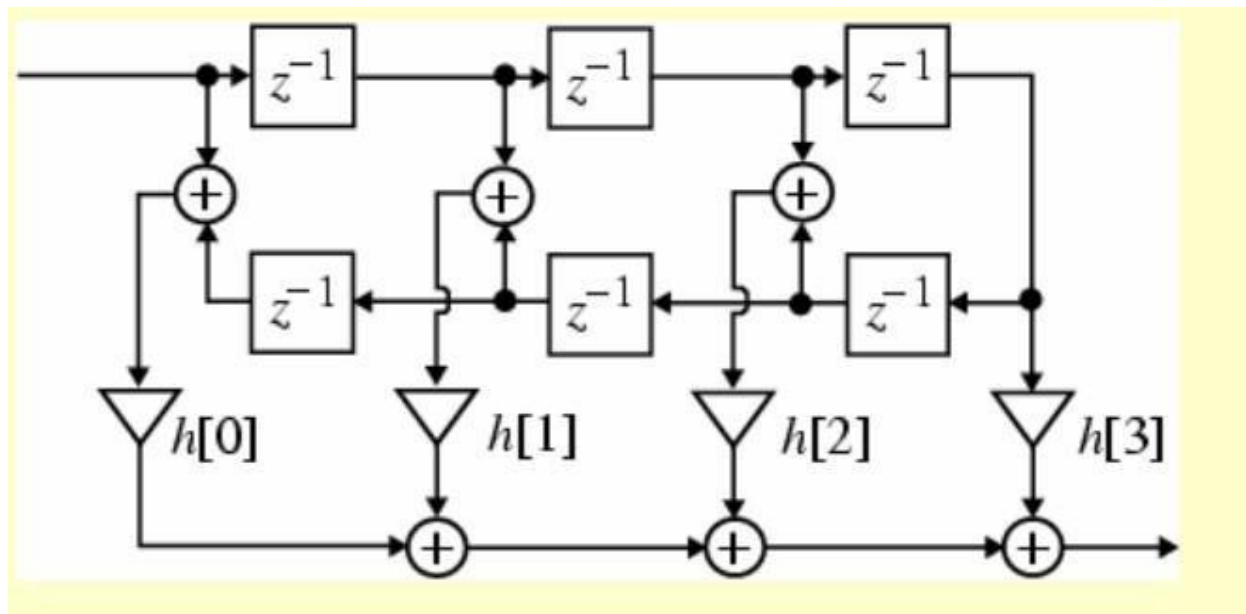


Fig.2.5. Linear phase FIR

5. Polyphase FIR Structures

The polyphase decomposition of $H(z)$ leads to a parallel form structure.

To illustrate this approach, consider a causal FIR transfer function $H(z)$ with $N = 8$:

$$H(Z) = h(0) + h(1)Z^{-1} + h(2)Z^{-2} + h(3)Z^{-3} + h(4)Z^{-4} + h(5)Z^{-5} + h(6)Z^{-6} + h(7)Z^{-7} + h(8)Z^{-8}$$

$H(z)$ can be expressed as a sum of two terms, with one term containing the even indexed coefficients and the other containing the odd-indexed coefficients:

$$\begin{aligned} H(Z) &= h(0) + h(2)Z^{-2} + h(4)Z^{-4} + h(6)Z^{-6} + h(8)Z^{-8} \\ &\quad + Z^{-1}[h(1) + h(3)Z^{-2} + h(5)Z^{-4} + h(7)Z^{-6}] \\ &= E_0(Z^2) + Z^{-1}E_1(Z^2) \end{aligned}$$

Putting $H(Z)$

The subfilters in the polyphase realization of an FIR transfer function are also FIR filters and can be realized using any methods. However, to obtain a canonic realization of the overall structure, the delays in all subfilters must be shared.

The filters designed by considering all the infinite samples of impulse response are called IIR (Infinite Impulse Response) filters. In digital domain, the processing of infinite samples of impulse response is practically not possible. Hence direct design of IIR filter is not possible. Therefore, the IIR filters are designed via analog filters. In design of IIR filter, the specification of an IIR filter is transformed to specification of an analog filter and an analog filter with transfer function, $H(s)$ is designed to satisfy the specification. Then the analog filter is transformed to digital filter with transfer function, $H(z)$. We know that the analog filter with transfer function $H(s)$ is stable if all its poles lie in the left half of the s -plane. Consequently, if the conversion technique is to be effective, it should possess the following desirable properties. 1. The imaginary axis in the s -plane should map into the unit circle in the z -plane. Thus there will be a direct relationship between the two frequency variables in the two domains. 2. The left-half of the s -plane should map into the interior of the unit circle in the z -plane. Thus a stable analog filter will be converted to a stable digital filter. The analog filter is designed by approximating the ideal frequency response using an error function. A number of solutions to the approximation problem of analog filter design are well developed. The popular among them are Butterworth and Chebyshev approximation. The popular transformation techniques used for transforming analog filter transfer function $H(s)$ to digital filter transfer function $H(z)$ are bilinear and impulse invariant transformation. The digital transfer function $H(z)$ can be realized in a software that runs on a digital hardware (or it can be implemented in firmware). The frequency response $H(e^{j\omega})$ by letting $z = e^{j\omega}$ in the transfer function $H(z)$ of the filter.

6. Design of IIR filters

The ideal magnitude response, $|H_d(j\omega)|$ of the four basic types of analog filters are shown in fig (a), (b), (c) and (d). The ideal magnitude response has sudden transition from passband to stopband which is practically not realizable. Hence the ideal response is approximated using a filter approximation function. The approximation problem is solved to meet a specified tolerance in the passband and stopband. The shaded areas in the fig 7.1 shows the tolerance regions of the ideal frequency response. In the passband the magnitude is approximated to unity within an error of δ_p . In the stopband the magnitude is approximated to zero within an error of δ_s . Here the δ_p and δ_s are the limits of the tolerance in the passband and stopband. The δ_p and δ_s are also called ripples. The magnitude response of practical or approximated analog filters, $|H(j\omega)|$ are shown in fig 7.1 (e), (f), (g) and (h). The frequency response of practical analog filter shows edges for passband and stopband so that the tolerances are within specified limits. Now, the specification of practical analog filter will be the following. ω_p = Passband edge frequency in rad/second. ω_s = Stopband edge frequency in rad/second. A_p = Gain at passband edge frequency A_s = Gain at stopband edge frequency.

Frequency selective filters: Ideal filter characteristics

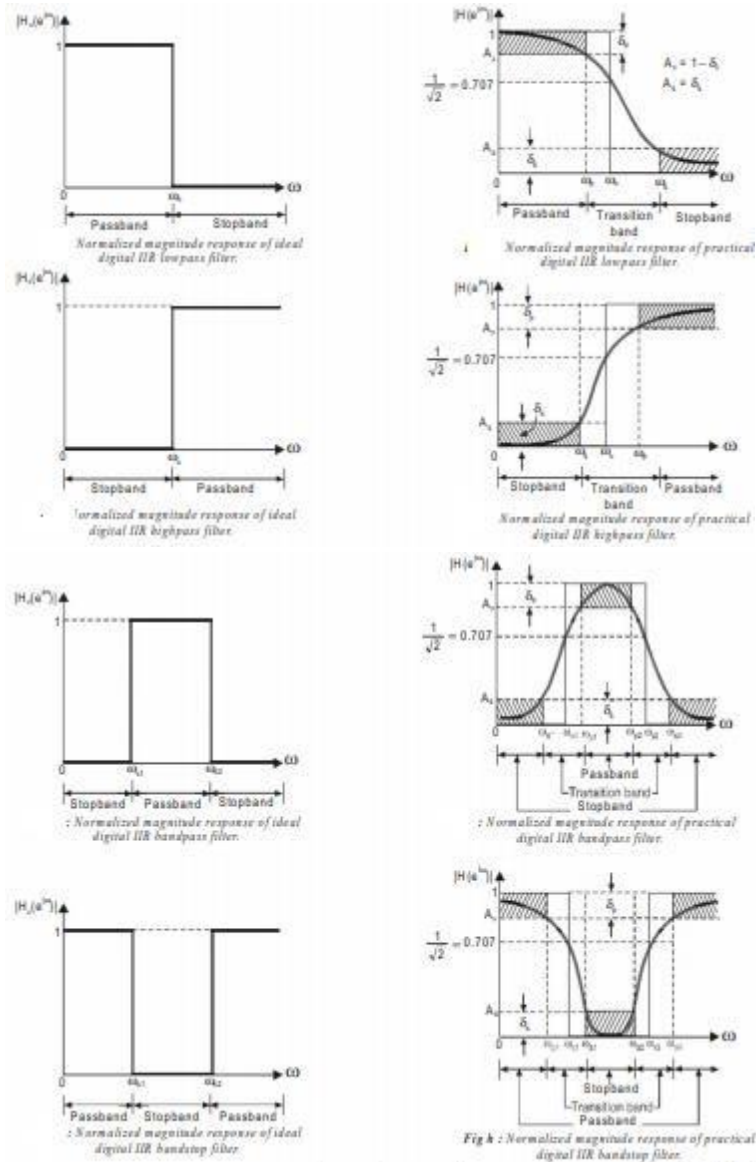


Fig.2.6. Ideal filter characteristics

Impulse Invariant Transformation

The objective of impulse invariant transformation is to develop an IIR filter transfer function whose impulse response is the sampled version of the impulse response of the analog filter. The main idea behind this technique is to preserve the frequency response characteristics of the analog filter. It can be stated that the frequency response of digital filter will be identical with the frequency response of the corresponding analog filter if the sampling time period T is selected sufficiently small (or the sampling frequency should be high) to minimize (or avoid completely) the effects of aliasing.

$$\frac{1}{s + p_i} \xrightarrow{\text{(is transformed to)}} \frac{1}{1 - e^{-p_i T} z^{-1}}$$

Relation Between Analog and Digital Frequency in Impulse Invariant Transformation

Let, W = Analog frequency in rad/second.

w = Digital frequency in rad/sample

$$\boxed{\text{Digital frequency, } w = WT} \quad \text{or} \quad \boxed{\text{Analog frequency, } \Omega = \frac{w}{T}}$$

Thus the mapping from the analog frequency W to the digital frequency w is many-to-one. This reflects the effects of aliasing due to sampling.

Useful Impulse Invariant Transformation

$$\begin{aligned} \frac{1}{(s + p_i)^m} &\longrightarrow \frac{(-1)^{m-1}}{(m-1)!} \frac{d^{m-1}}{dp_i^{m-1}} \frac{1}{1 - e^{-p_i T} z^{-1}} \\ \frac{(s + a)}{(s + a)^2 + b^2} &\longrightarrow \frac{1 - e^{-aT} (\cos bT) z^{-1}}{1 - 2e^{-aT} (\cos bT) z^{-1} + e^{-2aT} z^{-2}} \\ \frac{b}{(s + a)^2 + b^2} &\longrightarrow \frac{e^{-aT} (\sin bT) z^{-1}}{1 - 2e^{-aT} (\cos bT) z^{-1} + e^{-2aT} z^{-2}} \end{aligned}$$

Bilinear Transformation

The bilinear transformation is a conformal mapping that transforms the imaginary axis of s-plane into the unit circle in the z-plane only once, thus avoiding aliasing of frequency components. In this mapping all points in the left half of s-plane are mapped inside the unit circle in the z-plane and all points in the right half of s-plane are mapped outside the unit circle in the z-plane. The bilinear transformation can be linked to the trapezoidal formula for numerical integration. Any analog system is governed by a differential equation in time domain.

In the s-domain transfer function, if "s" is substituted by the term $\frac{2}{T} \frac{1-z^{-1}}{1+z^{-1}}$ the resulting transfer function will be z-domain transfer function.

Relation Between Analog and Digital Filter Poles in Bilinear Transformation

The mapping of s-domain function to z-domain function by bilinear transformation is a one to one mapping, that is, for every point in z-plane, there is exactly one corresponding point in s-plane and vice versa. The transformation is accomplished when,

$$s = \frac{2}{T} \frac{1-z^{-1}}{1+z^{-1}}$$

$$\therefore \text{Analog frequency, } \Omega = \frac{2}{T} \tan \frac{\omega}{2}$$

$$\therefore \text{Digital frequency, } \omega = 2 \tan^{-1} \frac{\Omega T}{2}$$

Specifications of Digital IIR Lowpass Filter

Let, $H(e^{j\omega})$ = Frequency response of IIR filter.

$|H(e^{j\omega})|$ = Magnitude response of IIR filter.

The magnitude response, $|H(e^{j\omega})|$ of IIR filter will have a passband, transition band and stop band.

The specification of the IIR filter can be expressed in any one of the following three different ways.

Case i : Gain at passband and stopband edge frequency

Case ii : Attenuation at passband and stopband edge frequency

Case iii : Ripple at passband and stopband edge frequency

The gain can be expressed either in normal values or in decibels (dB). The maximum value of normalized gain is unity and so the gain at band edge frequencies will be less than 1. Therefore, the dB-gain will be negative.

Let, ω_p = Passband edge digital frequency in rad/sample.

ω_s = Stopband edge digital frequency in rad/sample.

$A_p = |H(e^{j\omega})|_{\omega = \omega_p}$ = Gain (or magnitude) at passband edge frequency.

$A_s = |H(e^{j\omega})|_{\omega = \omega_s}$ = Gain (or magnitude) at stopband edge frequency.

$A_{p,dB} = 20 \log [|H(e^{j\omega})|_{\omega = \omega_p}]$ = dB-Gain (or dB-magnitude) at passband edge frequency.

$A_{s,dB} = 20 \log [|H(e^{j\omega})|_{\omega = \omega_s}]$ = dB-Gain (or dB-magnitude) at stopband edge frequency. The gain in normal values can be converted to dB-gain or vice versa as shown below.

$$A_{p,dB} = 20 \log A_p$$

$$A_p = 10^{(A_{p,dB}/20)}$$

$$A_{s,dB} = 20 \log A_s$$

$$A_s = 10^{(A_{s,dB}/20)}$$

The attenuation is usually expressed in decibels (dB). Since the gain at edge frequencies are less than 1, the attenuation in normal values will be greater than 1, and the dB-attenuation is positive.

$$\text{Let, } \alpha_p = \frac{1}{A_p} = \frac{1}{|H(e^{j\omega})|_{\omega = \omega_p}} = \text{Attenuation at passband edge frequency}$$

$$\alpha_s = \frac{1}{A_s} = \frac{1}{|H(e^{j\omega})|_{\omega = \omega_s}} = \text{Attenuation at stopband edge frequency}$$

$$\alpha_{p,dB} = 20 \log \left[\frac{1}{A_p} \right] = 20 \log \left[\frac{1}{|H(e^{j\omega})|_{\omega = \omega_p}} \right] = \text{dB-Attenuation at passband edge frequency}$$

$$\alpha_{s,dB} = 20 \log \left[\frac{1}{A_s} \right] = 20 \log \left[\frac{1}{|H(e^{j\omega})|_{\omega = \omega_s}} \right] = \text{dB-Attenuation at stopband edge frequency}$$

Ripple at passband and stopband edge frequency:

$$\begin{array}{l|l} A_p = 1 - \delta_p & \alpha_p = \frac{1}{A_p} = \frac{1}{1 - \delta_p} \\ A_s = \delta_s & \alpha_s = \frac{1}{A_s} = \frac{1}{\delta_s} \end{array}$$

7. Transfer function of Analog Butterworth Lowpass Filter:

The analog filter transfer function of normalized and unnormalized butterworth lowpass filters are given below. Let, N be the order of the filter. Let, $H(s_n)$ be the normalized Butterworth lowpass filter transfer function. When N is even

$$H(s_n) = \prod_{k=1}^{\frac{N}{2}} \frac{1}{s_n^2 + b_k s_n + 1}$$

When N is odd,

$$H(s_n) = \frac{1}{s_n + 1} \prod_{k=1}^{\frac{N-1}{2}} \frac{1}{s_n^2 + b_k s_n + 1}$$

where, $b_k = 2 \sin \left[\frac{(2k-1)\pi}{2N} \right]$

Table. Summary of Butterworth Lowpass Filter Normalized Transfer

Order, N	Normalized transfer function, $H(s_n)$
1	$\frac{1}{s_n + 1}$
2	$\frac{1}{s_n^2 + 1.414s_n + 1}$
3	$\frac{1}{(s_n + 1)(s_n^2 + s_n + 1)}$
4	$\frac{1}{(s_n^2 + 0.765s_n + 1)(s_n^2 + 1.848s_n + 1)}$
5	$\frac{1}{(s_n + 1)(s_n^2 + 0.618s_n + 1)(s_n^2 + 1.618s_n + 1)}$
6	$\frac{1}{(s_n^2 + 1.932s_n + 1)(s_n^2 + 1.414s_n + 1)(s_n^2 + 0.518s_n + 1)}$

Function

8. Order of the Lowpass Butterworth Filter

In Butterworth filters the frequency response of the filter depends on the order, N . Hence the order N has to be estimated to satisfy the given specifications. Usually the specifications of the filter are given in terms of gain at a passband and stopband frequency. Let, A_p = Gain or Magnitude at a passband frequency ω_p .

A_s = Gain or Magnitude at a stopband frequency ω_s .

$$N_1 = \frac{1}{2} \frac{\log \left[\frac{(1/A_s^2) - 1}{(1/A_p^2) - 1} \right]}{\log \left(\frac{\omega_s}{\omega_p} \right)} \quad N_1 = \frac{\log \left[\left(\frac{10^{0.1\alpha_{s,dB}} - 1}{10^{0.1\alpha_{p,dB}} - 1} \right)^{\frac{1}{2}} \right]}{\log \frac{\omega_s}{\omega_p}}$$

$$\text{Cutoff frequency, } \Omega_c = \frac{\Omega_s}{\left[(1/A_s^2) - 1 \right]^{\frac{1}{2N}}}$$

Alternatively,

$$\text{Cutoff frequency, } \Omega_c = \frac{\Omega_p}{\left[(1/A_p^2) - 1 \right]^{\frac{1}{2N}}}$$

For bilinear transformation,

$$\Omega_p = \frac{2}{T} \tan \frac{\omega_p}{2} ; \quad \Omega_s = \frac{2}{T} \tan \frac{\omega_s}{2}$$

For impulse invariant transformation,

$$\Omega_p = \frac{\omega_p}{T} ; \quad \Omega_s = \frac{\omega_s}{T}$$

where T is the sampling time.

$$\text{Cutoff frequency, } \Omega_c = \frac{\Omega_s}{\left(10^{0.1\alpha_{s,dB}} - 1 \right)^{\frac{1}{2N}}}$$

Alternatively,

$$\text{Cutoff frequency, } \Omega_c = \frac{\Omega_p}{\left(10^{0.1\alpha_{p,dB}} - 1 \right)^{\frac{1}{2N}}}$$

9. Design Procedure for Lowpass Digital Butterworth IIR Filter

- The process of filter design begins with filter specifications which include the filter characteristics (Lowpass, high-pass, band-pass, band-stop filter), filter type, passband frequency, stopband frequency, transition width frequency, sampling frequency and filter length.)
- The second step is obtain filter response, $H(\omega)$
- Third step is to find the filter coefficient and acceptable filter.
- The last step is to implement filter coefficient and choose ω appropriate filter

structure for filter implementation.

- There are 2 common IIR filter designs
- 1. Butterworth (As the Filter Order, N increases, the transition band becomes narrower).
- 2. Chebyshev Type
- The analog filter will be mapped to digital filter using transformation of s-domain to z-domain. 2 methods to convert the analog filter to digital filter and vice versa;
- 1. Impulse Invariance method
-

2. Bilinear Transformation method

1. Choose either bilinear or impulse invariant transformation, and determine the specifications of equivalent analog filter. The gain or attenuation of analog filter is same as digital filter. The band edge frequencies are calculated using the following equations.

Let, ω_p = Passband edge analog frequency corresponding to ω_p .

ω_s = Stopband edge analog frequency corresponding to ω_s .

For bilinear transformation,

$$\Omega_p = \frac{2}{T} \tan \frac{\omega_p}{2}$$

$$\Omega_s = \frac{2}{T} \tan \frac{\omega_s}{2}$$

.5

*Note : If either T or F_s is not specified then take $T = 1$ second.
If F_s is specified, then $T = \frac{1}{F_s}$*

For impulse invariant transformation,

$$\Omega_p = \frac{\omega_p}{T}$$

$$\Omega_s = \frac{\omega_s}{T}$$

2. Decide the order N of the filter. In order to estimate the order N , calculate a parameter N_1 using the following equation.

$$N_1 = \frac{1}{2} \frac{\log \left[\frac{(1/A_s^2) - 1}{(1/A_p^2) - 1} \right]}{\log \left(\frac{\Omega_s}{\Omega_p} \right)}$$

Choose N such that, $N \geq N_1$. Usually N is chosen as nearest integer just greater than N_1 .

3. Determine the normalized transfer function, $H(s_n)$ of the analog lowpass filter.

When N is even,

$$H(s_n) = \prod_{k=1}^{\frac{N}{2}} \frac{1}{s_n^2 + b_k s_n + 1}$$

When N is odd,

$$H(s_n) = \frac{1}{s_n + 1} \prod_{k=1}^{\frac{N-1}{2}} \frac{1}{s_n^2 + b_k s_n + 1}$$

where, $b_k = 2 \sin \left[\frac{(2k-1)\pi}{2N} \right]$

4. Calculate the analog cutoff frequency, ω_c .

$$\text{Cutoff frequency, } \Omega_c = \frac{\Omega_s}{\left[(1/A_s^2) - 1 \right]^{\frac{1}{2N}}}$$

5. Determine the unnormalized analog transfer function $H(s)$ of the lowpass filter.

$$H(s) = H(s_n) \Big|_{s_n = \frac{s}{\Omega_c}}$$

When the order N is even, $H(s)$ is obtained by letting $s_n \otimes s/\omega_c$ in equation (7.58).

$$\therefore H(s) = \prod_{k=1}^{\frac{N}{2}} \frac{1}{s_n^2 + b_k s_n + 1} \Big|_{s_n = \frac{s}{\Omega_c}} = \prod_{k=1}^{\frac{N}{2}} \frac{\Omega_c^2}{s^2 + b_k \Omega_c s + \Omega_c^2}$$

When the order N is odd, $H(s)$ is obtained by letting $s_n \otimes s/\omega_c$ in equation (7.59).

$$\therefore H(s) = \frac{1}{s_n + 1} \prod_{k=1}^{\frac{N-1}{2}} \frac{1}{s_n^2 + b_k s_n + 1} \Big|_{s_n = \frac{s}{\Omega_c}} = \frac{\Omega_c}{s + \Omega_c} \prod_{k=1}^{\frac{N-1}{2}} \frac{\Omega_c^2}{s^2 + b_k \Omega_c s + \Omega_c^2}$$

6. Determine the transfer function of digital filter, $H(z)$. Using the chosen transformation in step-1, transform $H(s)$ to $H(z)$. When impulse invariant transformation is employed, if $T < 1$, then multiply $H(z)$ by T to normalize the magnitude.

7. Realize the digital filter transfer function $H(z)$ by a suitable structure. 8. Verify the design by sketching the frequency response $H(e^{j\omega})$.

$$H(e^{j\omega}) = H(z) \Big|_{z=e^{j\omega}}$$

10. Design of Lowpass Digital Chebyshev Filter

The analog Chebyshev filter is designed by approximating the ideal frequency response using an error function. The approximation function is selected such that the error is minimized over a prescribed band of frequencies.

1. Choose either bilinear or impulse invariant transformation, and determine the specifications of equivalent analog filter. The gain or attenuation of analog filter is same as digital filter. The band edge frequencies are calculated using the following equations.

Let, ω_p = Passband edge analog frequency corresponding to ω_p .

ω_s = Stopband edge analog frequency corresponding to ω_s .

For bilinear transformation,

$$\Omega_p = \frac{2}{T} \tan \frac{\omega_p}{2}$$

$$\Omega_s = \frac{2}{T} \tan \frac{\omega_s}{2}$$

*Note : If either T or F_s is not specified then take $T = 1$ sec.
If F_s is specified, then $T = \frac{1}{F_s}$*

For impulse invariant transformation,

$$\Omega_p = \frac{\omega_p}{T}$$

$$\Omega_s = \frac{\omega_s}{T}$$

2. Decide the order N of the filter. In order to estimate the order N , calculate a parameter N_1 using the following equation. Choose N such that $N \geq N_1$. Usually N is chosen as nearest integer just greater than N_1 .

$$N_1 = \frac{\cosh^{-1} \left[\left(\frac{(1/A_s^2) - 1}{(1/A_p^2) - 1} \right)^{\frac{1}{2}} \right]}{\cosh^{-1} \left(\frac{\Omega_s}{\Omega_p} \right)}$$

3. Determine the normalized transfer function $H(s_n)$, of the filter.

When the order N is even,

$$H(s_n) = \prod_{k=1}^{\frac{N}{2}} \frac{B_k}{s_n^2 + b_k s_n + c_k}$$

When the order N is odd,

$$H(s) = \frac{B_0}{s + c_0} \prod_{k=1}^{\frac{N-1}{2}} \frac{B_k}{s^2 + b_k s + c_k}$$

$$\text{where, } b_k = 2 y_N \sin\left(\frac{(2k-1)\pi}{2N}\right)$$

$$c_k = y_N^2 + \cos^2\left(\frac{(2k-1)\pi}{2N}\right)$$

$$c_0 = y_N$$

$$y_N = \frac{1}{2} \left\{ \left[\left(\frac{1}{\epsilon^2} + 1 \right)^{\frac{1}{2}} + \frac{1}{\epsilon} \right]^{\frac{1}{N}} - \left[\left(\frac{1}{\epsilon^2} + 1 \right)^{\frac{1}{2}} + \frac{1}{\epsilon} \right]^{-\frac{1}{N}} \right\}$$

$$\epsilon = \left[\left(1/\Lambda_p^2 \right) - 1 \right]^{\frac{1}{2}}$$

For even values of N , find B_k such that,

$$H(0) = \frac{1}{(1 + \epsilon^2)^{\frac{1}{2}}}$$

For odd values of N , find B_k such that,

$$H(0) = 1$$

(It is normal practice to take $B_0 = B_1 = B_2 \dots = B_k$).

4. Determine the unnormalized analog transfer function $H(s)$ of the lowpass filter.

$$H(s) = H(s_n) \Big|_{s_n = \frac{s}{\Omega_c}}$$

Here, $\omega_c = \omega_p$ = Passband edge frequency.

When the order N is even, $H(s)$ is obtained by letting $s_n = s/\omega_c$ in equation (7.88).

$$\therefore H(s) = \prod_{k=1}^{\frac{N}{2}} \frac{B_k}{s_n^2 + b_k s_n + c_k} \Big|_{s_n = \frac{s}{\Omega_c}} = \prod_{k=1}^{\frac{N}{2}} \frac{B_k \Omega_c^2}{s^2 + b_k \Omega_c s + c_k \Omega_c^2}$$

When the order N is odd, $H(s)$ is obtained by letting $s_n \rightarrow s/\omega_c$ in equation (7.89).

$$\therefore H(s) = \frac{B_0}{s + c_0} \prod_{k=1}^{\frac{N-1}{2}} \frac{B_k}{s_n^2 + b_k s_n + c_k} \bigg|_{s_n = \frac{s}{\omega_c}} = \frac{B_0 \omega_c}{s + c_0 \omega_c} \prod_{k=1}^{\frac{N-1}{2}} \frac{B_k \omega_c^2}{s^2 + b_k \omega_c s + c_k \omega_c^2}$$

5. Determine the transfer function of digital filter, $H(z)$. Using the chosen transformation, in step-1 transform $H(s)$ to $H(z)$. When impulse invariant transformation is employed, if $T < 1$, then multiply $H(z)$ by T to normalize the magnitude.
6. Realize the digital filter transfer function $H(z)$ by a suitable structure.
7. Verify the design by sketching the frequency response $H(e^{j\omega})$.

$$H(e^{j\omega}) = H(z) \big|_{z=e^{j\omega}}$$

TEXT / REFERENCE BOOKS:

1. John G. Proakis & Dimitris G. Manolakis, "Digital Signal Processing - Principles, Algorithms & Applications", Fourth Edition, Pearson education / Prentice Hall, 2009
2. Sanjit K. Mitra, "Digital Signal Processing: A Computer - Based Approach", McGrawHill Education, 4th Edition, 2013
3. B.P. Lathi, "Signal Processing & Linear systems", Oxford University Press, 2nd edition, 2009
4. Lyons, "Understanding Digital Signal Processing", Prentice Hall, 3rd edition, 2010
5. Johny R. Johnson, "Introduction to Digital Signal Processing", PHI, 2006
6. Alan V. Oppenheim, Ronald W. Schaffer, "Discrete-Time Signal Processing", Pearson, 3rd Edition, 2010
7. Salivahanan, "Digital Signal Processing, 2nd Edition, TMH, 2010.
8. A. Nagoor Kani, "Digital Signal Processing", Tata McGrawHill Education, 4th Edition, 2012



SATHYABAMA

INSTITUTE OF SCIENCE AND TECHNOLOGY

(DEEMED TO BE UNIVERSITY)

Accredited "A" Grade by NAAC | 12B Status by UGC | Approved by AICTE

www.sathyabama.ac.in

SCHOOL OF ELECTRICAL AND ELECTRONICS

DEPARTMENT OF ELECTRONICS AND COMMUNICATION
ENGINEERING

UNIT - III

SECA1506- Digital Signal Processing

UNIT III Finite Word Length Effects

In digital representation the signals are represented as an array of binary numbers, and the digital system employ a fixed size of binary called “word size or word length” for number representation. This finite word size for number representation leads to errors in input signals, intermediate signals in computations and in the final output signals. In general, the various effects due to finite precision representation of numbers in digital systems are called finite word length effects.

Some of the finite word length effects in digital systems are given below.

- Errors due to quantization of input data.
- Errors due to quantization of filter coefficients.
- Errors due to rounding the product in multiplication.
- Errors due to overflow in addition.
- Limit cycles in recursive computations.

The two major methods of representing binary numbers are fixed point representation and floating point representation.

Fixed point representation the digits allotted for integer part and fraction part are fixed, and so the position of binary point is fixed. Since the number of digits is fixed it is impossible to represent too large and too small numbers by fixed point representation. Therefore the range of numbers that can be represented in fixed point representation for a given binary word size is less when compared to floating point representation.

In fixed point representation there are three different formats for representing negative binary fraction numbers. They are,

1. Sign-magnitude format
2. One's complement format
3. Two's complement format

In sign magnitude format the negative value of a given number differ only in sign bit (i.e., digit d₀). The sign digit d₀ is zero for positive number and one for negative number.

In one's complement format the negative of the given number is obtained by bit by bit complement of its positive representation.

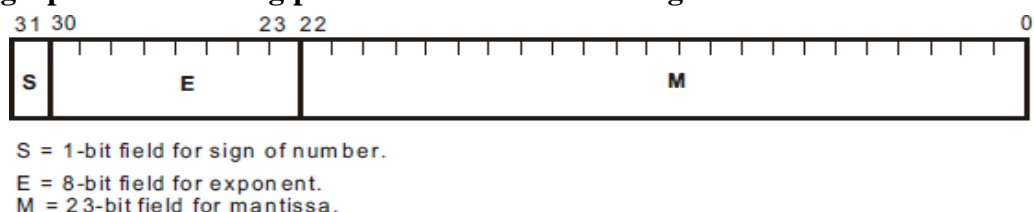
In two's complement format the negative of the given number is obtained by

taking one's complement of its positive representation and then adding one to the least significant bit.

Floating point representation the binary point can be shifted to desired position so that number of digits in the integer part and fraction part of a number can be varied. This leads to larger range of number that can be represented in floating point representation.

$$\text{Floating point number, } N_f = M \times 2^E$$

In various digital systems or computers, a variety of formats are employed for floating point representation. The IEEE (Institute of Electrical and Electronic Engineers) has proposed a standard format for floating point representation, which is widely followed in digital computers. The IEEE-754 standard format for 32-bit single precision floating point number is shown in fig



IEEE-754 format for 32-bit floating point number.

Fig 3.1 IEEE-754 format for 32 bit-floating point number

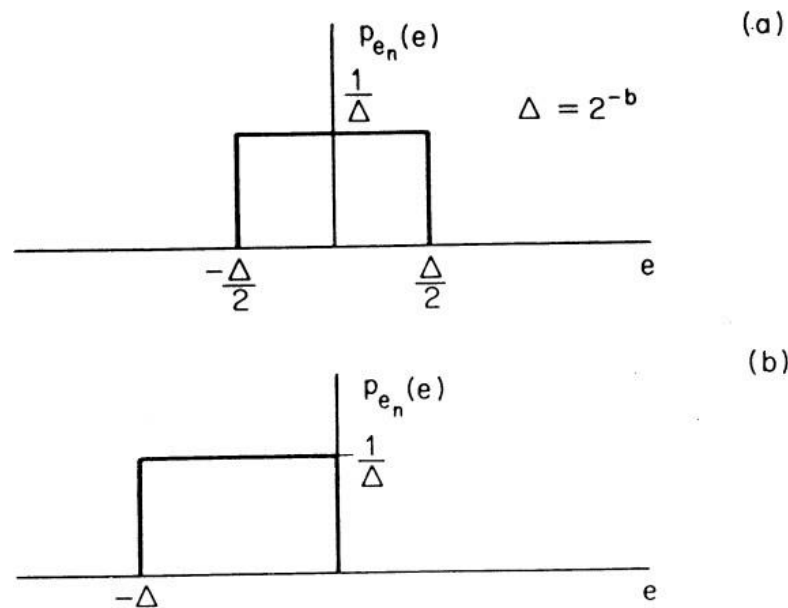
Comparison of Fixed Point and Floating Point Representation

Fixed point representation	Floating point representation
1. In a b-bit binary the range of numbers represented is less when compared floating point representation.	1. In a b-bit binary the range of numbers represented is large when compared to fixed point representation.
2. The position of binary point is fixed	2. The position of binary point is variable.
3. The resolution is uniform throughout	3. The resolution is variable

Truncation and Rounding error

In fixed point or floating point arithmetic the size of the result of an operation (sum or product) may be exceeding the size of binary used in the number system. In such cases the low order bits has to be eliminated in order to store the result. The two methods of eliminating these low order bits are truncation and rounding. This process is also referred to as quantization via truncation and rounding. The effect of rounding and truncation is to

introduce an error whose value depends on the number of bits eliminated. The characteristics of the errors introduced through either truncation or rounding depend on the type of number representation. The truncation is the process of reducing the size of binary number (or reducing the number of bits in a binary number) by discarding all bits less significant than the least significant bit that is retained. In the truncation of a binary number to b bits, all the less significant bits beyond b^{th} bit are discarded. Rounding is the process of reducing the size of a binary number to finite word size of b -bits such that the rounded b -bit number is closest to the original unquantized number. The rounding process consists of truncation and addition. In rounding of a number to b -bits, first the unquantized number is truncated to b -bits by retaining the most significant b -bits. Then a zero or one is added to the least significant bit of the truncated number depending on the bit that is next to the least significant bit that is retained.



Probability density functions for (a) rounding; (b) truncation.

Fig 3.2 Probability density function for (a) rounding (b) Truncation

Quantization Steps

The decimal numbers that are encountered as filter coefficients, sum, product, etc., in DSP applications will usually lie in the range of -1 to $+1$. When “B” bit binary is selected to represent the decimal numbers, then 2^B binary codes are possible. Hence the range of decimal numbers has to be divided into 2^B steps and each step is represented by a binary code. Each step of decimal number is also called quantization step.

$$\therefore \text{Quantization step size, } q = \frac{R}{2^B} = \frac{1 - (-1)}{2^B} = \frac{2}{2^B} = \frac{1}{2^{B-1}}$$

$$= \frac{1}{2^{B-1}} = \frac{1}{2^b} = 2^{-b}$$

Where, R = Range of decimal number

B = Size of binary including sign bit

b = B - 1 = Size of binary excluding sign bit

Steady State Output Noise Variance (Power) Due to the Quantization Error Signal

The quantized input signal of a digital system can be represented as a sum of unquantized signal $x(n)$ and error signal $e(n)$ as shown in fig

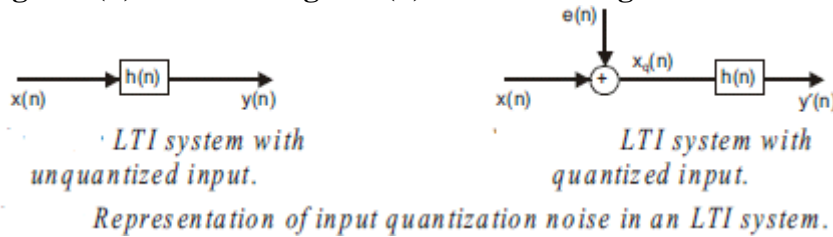


Fig 3.3 Representation of input quantization noise in an LTI system.

In fig $h(n)$ is the impulse response of the system and $y(n)$ is the response or output of the system due to input and error signal. The response of the system is given by convolution of input and impulse response. For linear systems using distributive property of convolution the response $y(n)$ can be written as shown in equation

$$\begin{aligned} y(n) &= x_q(n) * h(n) \\ &= [x(n) + e(n)] * h(n) \\ &= [x(n) * h(n)] + [e(n) * h(n)] \end{aligned}$$

Let, $y(n) = y(n) + e(n)$

where, $y(n) = x(n) * h(n)$ = Output due to input signal $x(n)$.

$e(n) = e(n) * h(n)$ = Output due to error signal $e(n)$.

The variance of the signal $e(n)$ is called output noise power or steady state output noise power (or variance) due to the quantization error signal. Using autocorrelation function and the definition for variance of a discrete time signal, the expression for output noise power is

$$= \sigma_e^2 \sum_{i=1}^N \left[(z - p_i) H(z) H(z^{-1}) z^{-1} \right]_{z=p_i}$$

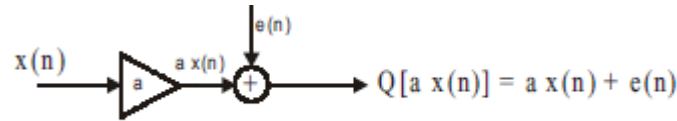
where, p_i are poles of $H(z) H(z^{-1}) z^{-1}$ only the poles that lie inside the unit circle in z -plane are considered.

Product Quantization Error

In realization structures of IIR system, multipliers are used to multiply the signal by constants. The output of the multipliers i.e, the products are quantized to finite word length in order to store them in registers and to be used in subsequent

calculations. The error due to the quantization of the output of multiplier is referred to as product quantization error.

The Noise Transfer Function (NTF) is defined as transfer function from the noise source to the filter output (i.e., NTF is the transfer function obtained by treating the noise source as actual input).

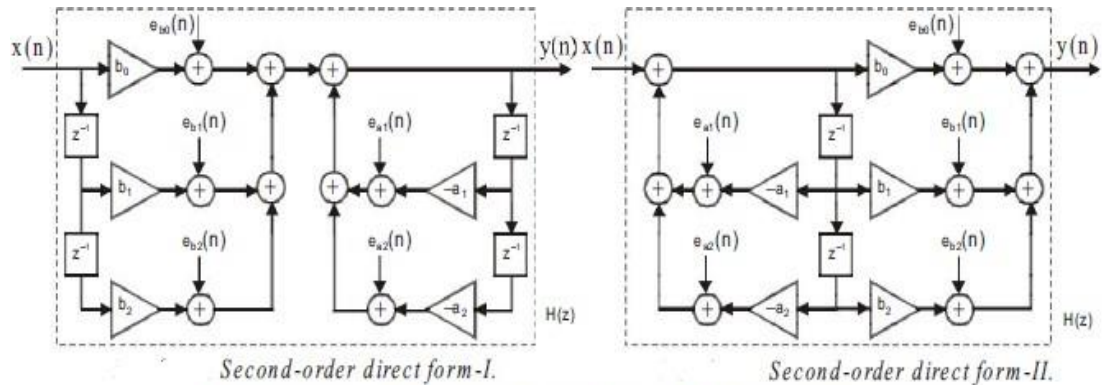


Statistical model of fixed point product quantization.

Quantized product = $Q[a x(n)] = a x(n) + e(n)$

where, $a x(n)$ = Unquantized product

$e(n)$ = Product quantization error signal



Product quantization noise models of IIR systems for direct form realization.

Fig 3.4 Product quantization noise models of IIR systems for direct form realization

The total steady state noise variance at the output of the system due to product quantization errors is given by the sum of the output noise variances due to all the noise sources.

Output Noise Power (Roundoff Noise Power) Due to Product Quantization

Let, σ_{eTop}^2 = Total output noise power due to product quantization error
(or Total roundoff noise power)

$$\therefore \sigma_{eTop}^2 = \sigma_{e1op}^2 + \sigma_{e2op}^2 + \dots + \sigma_{eMop}^2$$

Limit Cycles

During periodic oscillations, the output $y(n)$ of a system will oscillate between a finite positive and negative value for increasing n or the output will become constant for increasing n . Such oscillations are called limit cycles. These oscillations are due to round-off errors in multiplication and overflow in addition.

Limit cycle oscillations are clearly unwanted (e.g. may be audible in speech/audio applications)

Limit cycle oscillations can only appear if the filter has feedback. Hence FIR filters cannot have limit cycle oscillations.

Types

1. zero input limit cycles
2. Overflow limit cycles

In recursive systems, if the system output enters a limit cycle, it will continue to remain in limit cycle even when the input is made zero. Hence these limit cycles are also called zero input limit cycles. In fixed point addition of two binary numbers the overflow occurs when the sum exceeds the finite word length of the register used to store the sum. The overflow in addition may lead to oscillations in the output which is referred to as overflow limit cycles

In a limit cycle the amplitudes of the output are confined to a range of values, which is called the dead band of the filter. For a first-order system described by the equation, $y(n) = a y(n-1) + x(n)$, the dead band is given by,

$$\text{Dead band} = \pm \frac{2^{-B}}{1 - |a|} = - \frac{2^{-B}}{1 - |a|} \text{ to } + \frac{2^{-B}}{1 - |a|}$$

where, B = Number of binary bits (including sign bit) used to represent the product. For a second-order system described by the equation, $y(n) = a_1 y(n-1) + a_2 y(n-2) + x(n)$, the dead band of the filter is given by,

$$\text{Dead band} = \pm \frac{2^{-B}}{1 - |a_2|} = - \frac{2^{-B}}{1 - |a_2|} \text{ to } + \frac{2^{-B}}{1 - |a_2|}$$

Scaling to Prevent Overflow

The two methods of preventing overflow are saturation arithmetic and scaling the input signal to the adder. In saturation arithmetic, undesirable signal distortion is introduced. In order to limit the signal distortion due to frequent overflows, the input signal to the adder can be scaled such that the overflow becomes a rare event.

TEXT / REFERENCE BOOKS:

1. John G. Proakis & Dimitris G. Manolakis, “Digital Signal Processing - Principles, Algorithms & Applications”, Fourth Edition, Pearson education / Prentice Hall, 2009
2. Sanjit K. Mitra, Digital Signal Processing: A Computer - Based Approach, McGrawHill Education, 4th Edition, 2013
3. B.P. Lathi, “Signal Processing & Linear systems”, Oxford University Press, 2nd edition, 2009
4. Lyons, “Understanding Digital Signal Processing”, Prentice Hall, 3rd edition, 2010

5. Johny R. Johnson, "Introduction to Digital Signal Processing", PHI, 2006
6. Alan V. Oppenheim, Ronald W. Schaffer, Discrete-Time Signal Processing, Pearson, 3rd Edition, 2010
7. Salivahanan, "Digital Signal Processing, 2nd Edition, TMH, 2010.
8. A.Nagoor Kani, "Digital Signal Processing", Tata McGrawHill Education, 4th Edition, 2012



SATHYABAMA

INSTITUTE OF SCIENCE AND TECHNOLOGY

(DEEMED TO BE UNIVERSITY)

Accredited "A" Grade by NAAC | 12B Status by UGC | Approved by AICTE

www.sathyabama.ac.in

SCHOOL OF ELECTRICAL AND ELECTRONICS

DEPARTMENT OF ELECTRONICS AND COMMUNICATION
ENGINEERING

UNIT - IV

SECA1506 - Digital Signal Processing

IV MULTIRATE SIGNAL PROCESSING

Introduction to Multirate signal processing

Single-rate systems: Sampling rates at the input and at the output and all internal nodes are the same.

Multirate systems: DSP systems with unequal sampling rates at various parts of the system.

The process of converting a signal from one sampling rate to another sampling rate is called **sampling rate conversion**.

There are two ways for sampling rate conversion in the digital domain. They are,

1. Up-sampler / Up- Converter/ Interpolator
2. Down-sampler/ Decimator / Sub-sample

Downsampling (or Decimation)

Down sampling or decimation is the process of reducing the sampling rate by an integer factor D .

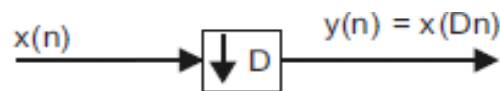


Fig.4.1 Decimator.

$x(n)$ = Discrete time signal

D = Sampling rate reduction factor (and D is an integer)

Now, $x(Dn)$ = Downsampled version of $x(n)$

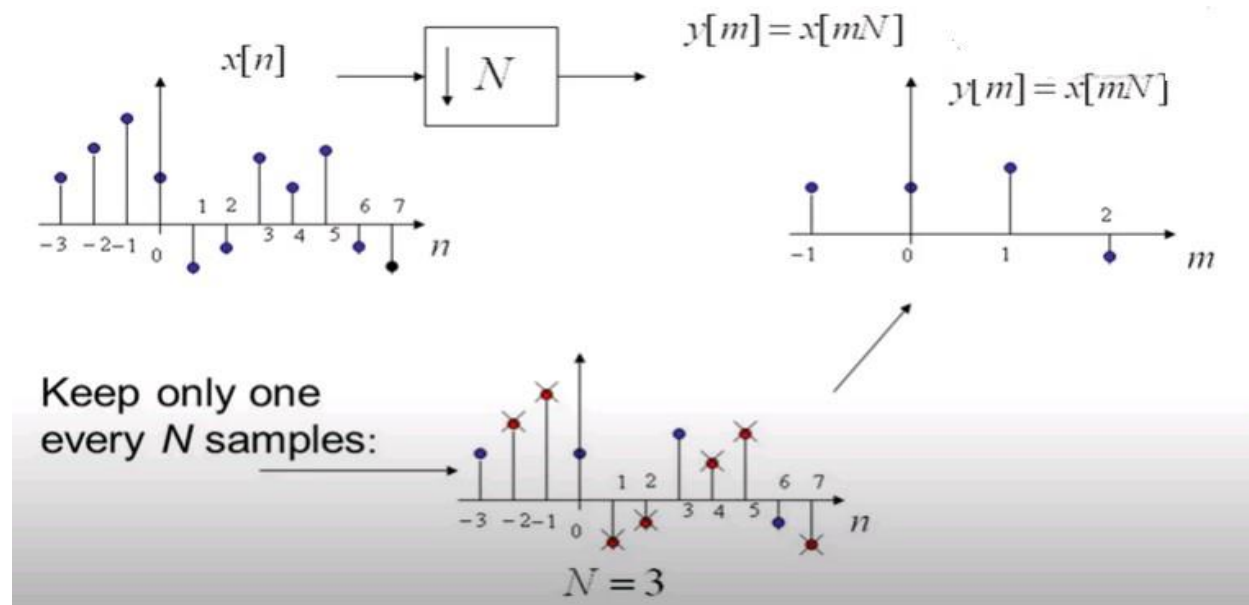


Fig 4.2 Time domain representation of decimation

Spectrum of Down sampler

The spectrum of Down sampler is given by

$$Y(e^{j\omega}) = \frac{1}{D} \sum_{k=0}^{D-1} X(e^{j(\omega - 2\pi k)/D})$$

Anti-aliasing Filter

When the input signal to the decimator is not bandlimited then the spectrum of decimated signal has aliasing. In order to avoid aliasing the input signal should be bandlimited to π/D for decimation by a factor D . Hence the input signal is passed through a lowpass filter with a bandwidth of π/D before decimation. Since this lowpass filter is designed to avoid aliasing in the output spectrum of decimator, it is called anti-aliasing filter.

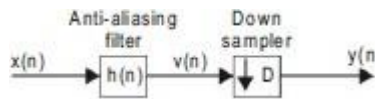


Fig 4.3 decimator with anti-aliasing filter.

Problem

sketch the spectrum of a down sampled signal for sampling rate reduction factor $D = 2, 3$ and 4.

$$\begin{aligned} Y(e^{j\omega}) &= \frac{1}{2} \sum_{k=0}^1 X(e^{j(\omega - 2\pi k)/2}) \\ &= \frac{1}{2} X(e^{j\omega/2}) + \frac{1}{2} X(e^{j(\omega - 2\pi)/2}) \end{aligned}$$

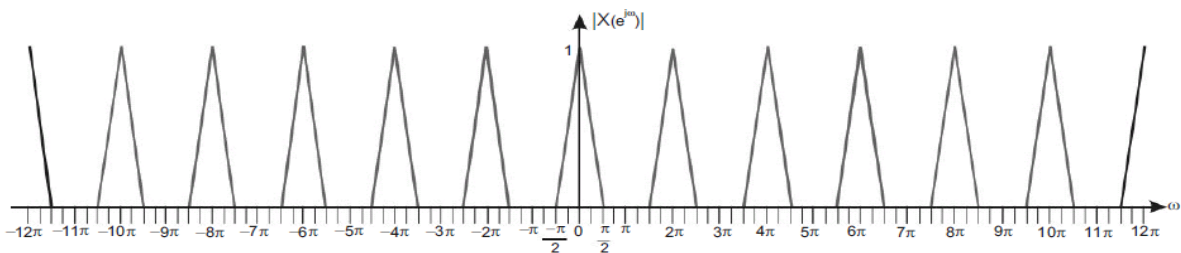


Fig 1.

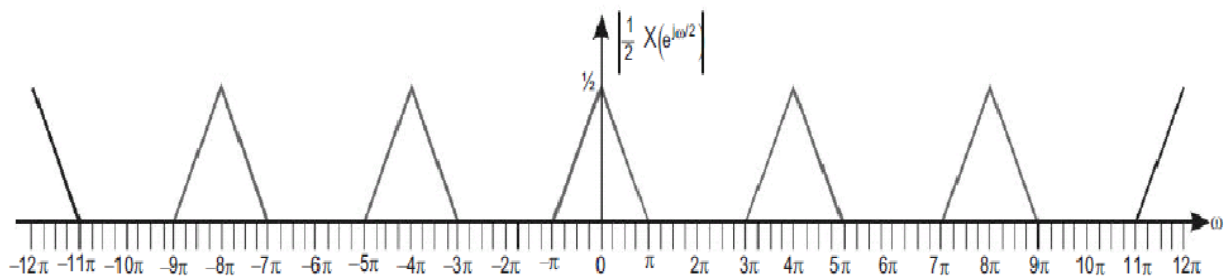


Fig 2.

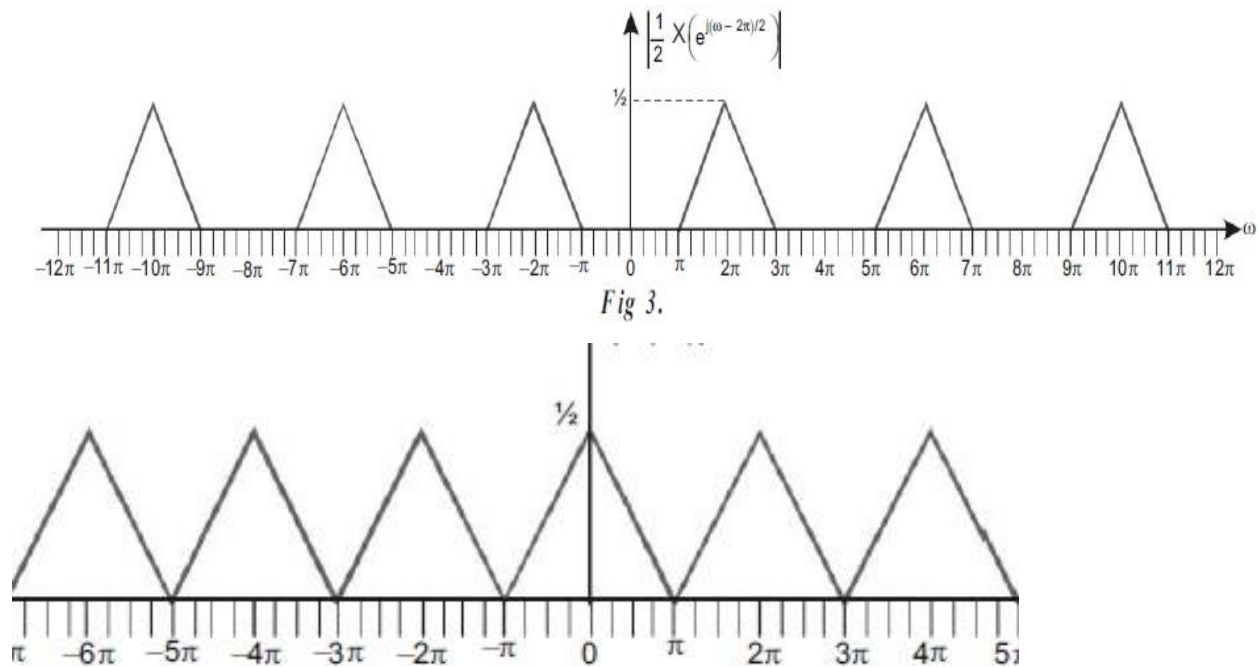


Fig. 4.4 spectrum of a down sampled signal for sampling rate reduction factor $D = 2$

Problem

Consider the discrete time signal shown in fig 1. Sketch the down sampled version of the signals for the sampling rate reduction factors, a) $D = 2$ b) $D = 3$.

$$x(n) = \{1, -1, 1, -1, 2, -2, 2, -2, 3, -3, 3, -3\}$$

Sampling rate reduction factor, $D = 2$.

$$x(2n) = x_{D2}(n) = \{1, 1, 2, 2, 3, 3\}$$

Sampling rate reduction factor, $D = 3$.

$$x(3n) = x_{D3}(n) = \{1, -1, 2, -3\}$$

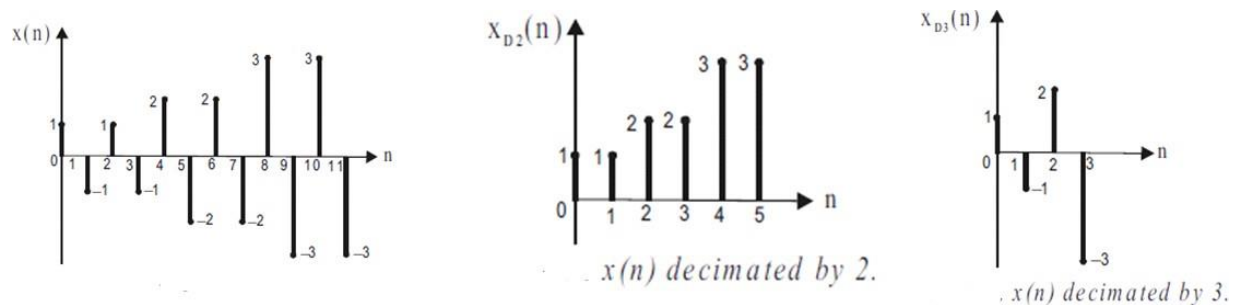


Fig 4.5 Down sampled version of the signals for the sampling rate reduction factors $D = 2$

Upsampling (or Interpolation)

The upsampling (or interpolation) is the process of increasing the samples of the discrete

time signal.

Let, $x(n)$ = Discrete time signal

I = Sampling rate multiplication factor (and I is an integer).

The device which perform the process of upsampling is called upsampler (or interpolator).

Symbolically, the upsampler can be represented as shown in fig



Fig 4.6 interpolator

Up sampling or interpolation is the process of increasing the sampling rate by an integer factor I .

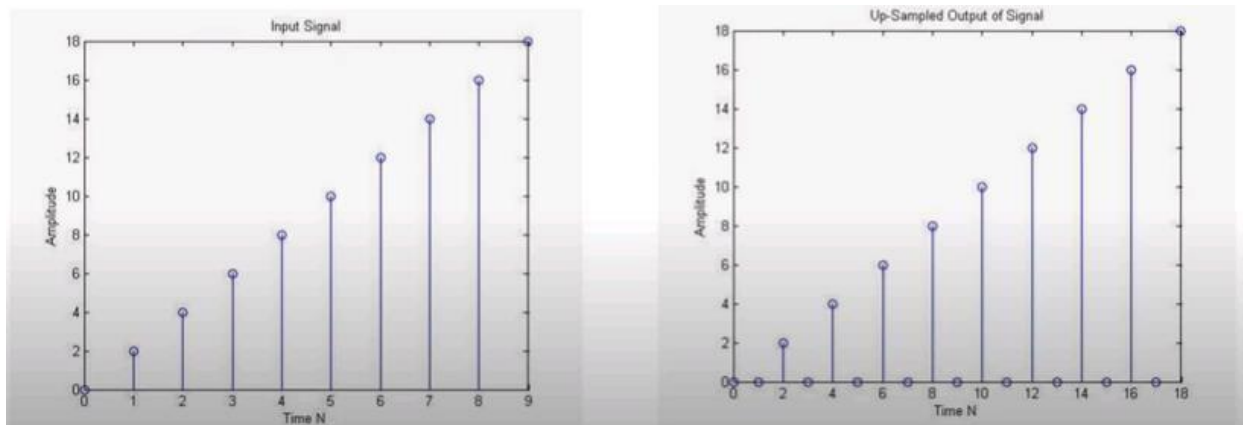


Fig 4.7 Time domain representation of interpolator

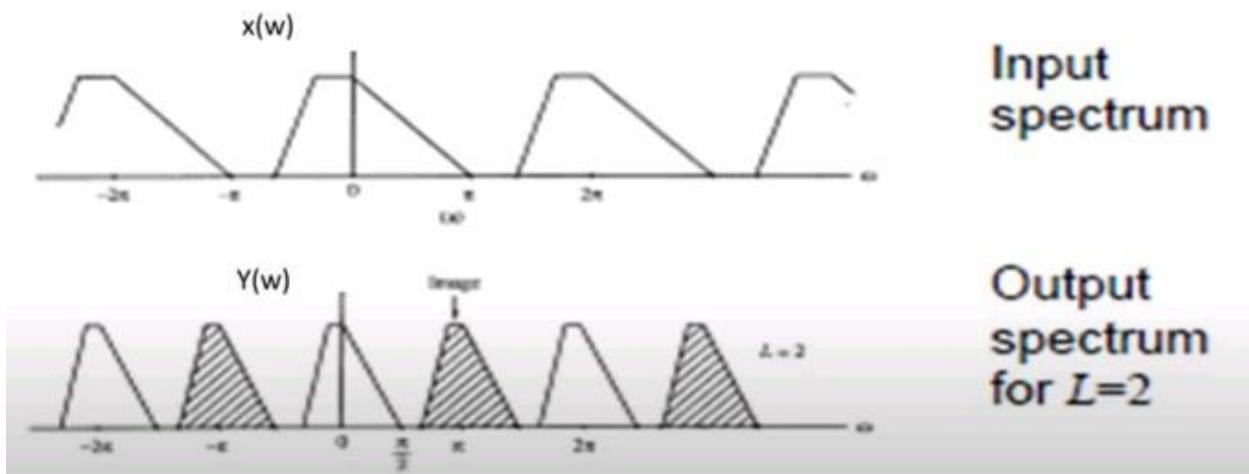


Fig 4.8 Spectrum of a upsampled signal for sampling rate reduction factor $L = 2$

Anti-imaging Filter

The output spectrum of interpolator is compressed version of the input spectrum, therefore, the spectrum of upsampled signal has multiple images in a period of 2π . When

upsampled by a factor of I , the output spectrum will have I images in a period of 2π , with each image band limited to π/I . Since the frequency spectrum in the range 0 to π/I are unique, we have to filter the other images. Hence the output of upsampler is passed through a lowpass filter with a bandwidth of π/I . Since this lowpass filter is designed to avoid multiple images in the output spectrum, it is called anti-imaging filter.

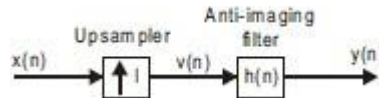


Fig 4.9. Interpolator with anti-imaging filter.

Poly phase implementation of FIR filters for interpolator and decimator

Potential computational savings can be made within the process of decimation, interpolation, and sampling-rate conversion. Polyphase filters is the name given to certain realisations of multirate filtering operations, which facilitate computational savings in both hardware and software.

Polyphase Structure of Decimator

In decimator, a lowpass filter called anti-aliasing filter is employed at the input in order to bandlimit the input signal, so that aliasing is avoided in the output spectrum of decimator. In order to reduce the computations in FIR filter, polyphase decomposition can be applied to FIR filter to decompose into L sub-filters.

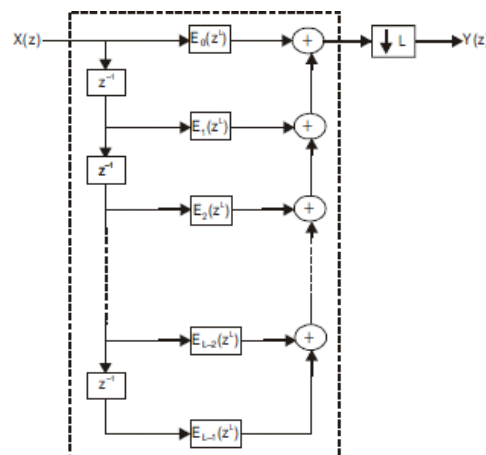


Fig 4.10 decimator with antialiasing filter

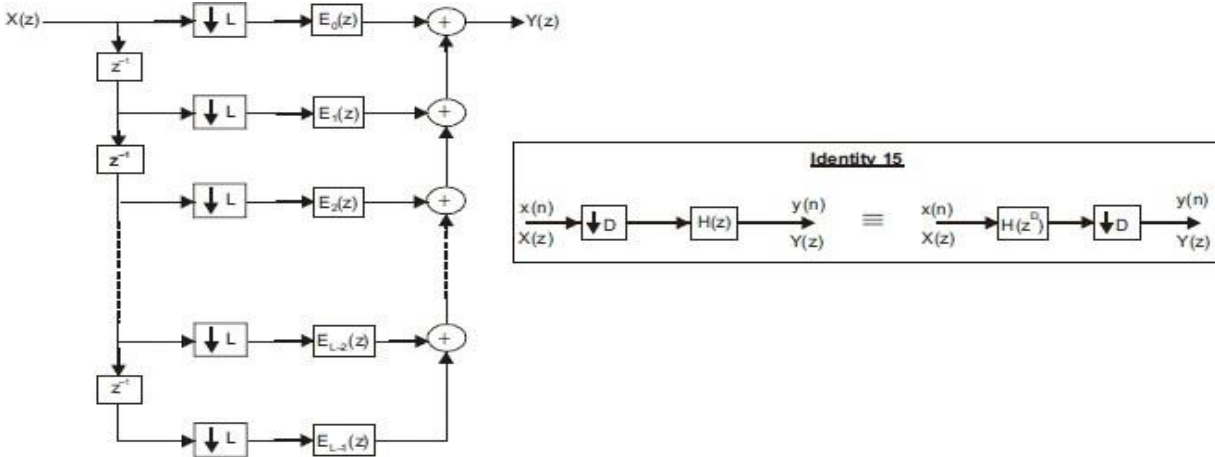


Fig 4.11 Decimator with antialiasing filter further deduction of fig 4.10 using identity.

Polyphase Structure of Interpolator

In interpolator, a lowpass filter called anti-imaging filter is employed at the output in order to eliminate the multiple images in the output spectrum of interpolator.

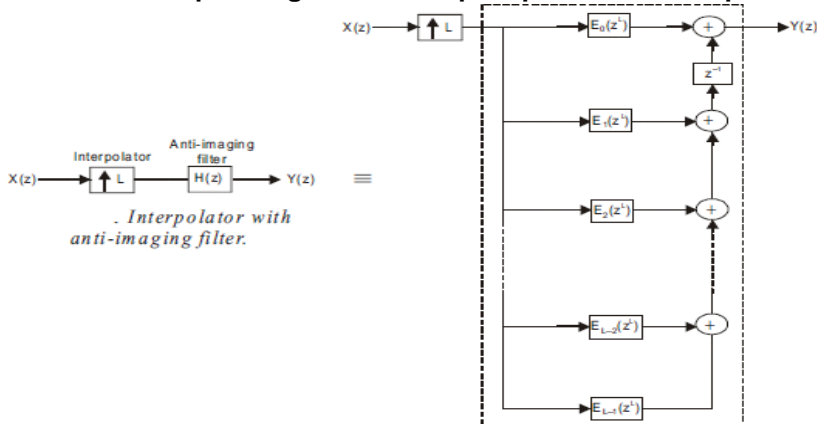


Fig 4.12 Interpolator with antialiasing filter

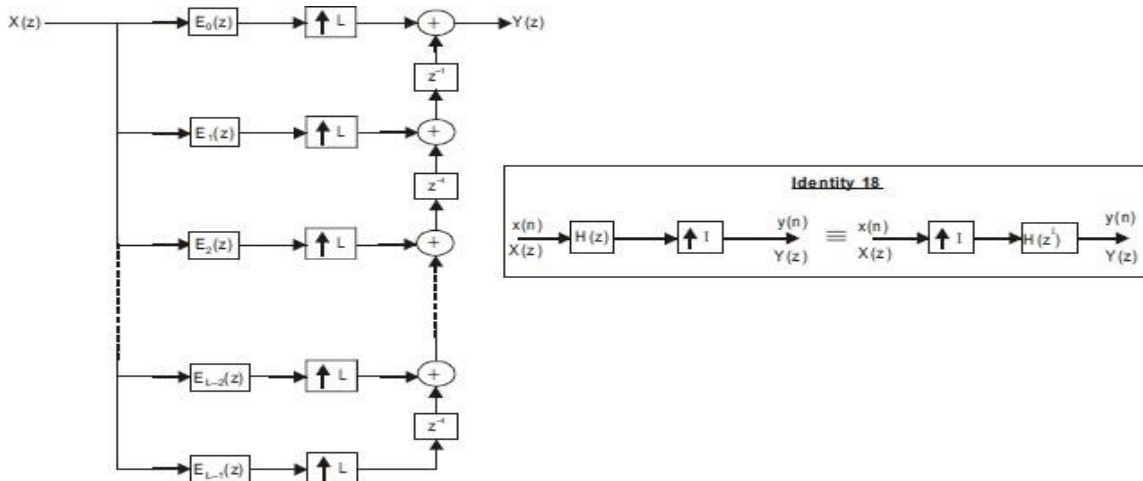


Fig 4.11 Interpolator with antialiasing filter further deduction of fig 4.12 using identity

Sampling rate conversion

A common use of multirate signal processing is for sampling-rate conversion. Suppose a digital signal $x[n]$ is sampled at an interval T_1 , and we wish to obtain a signal $y[n]$ sampled at an interval T_2 . Then the techniques of decimation and interpolation enable this operation, providing the ratio T_1/T_2 is a rational number i.e. L/M .

Sampling-rate conversion can be accomplished by L -fold expansion, followed by low-pass filtering and then M -fold

Decimation, It is important to emphasize that the interpolation should be performed first and decimation second, to preserve the desired spectral characteristics of $x[n]$. Furthermore by cascading the two in this manner, both of the filters can be combined into one single low-pass filter.

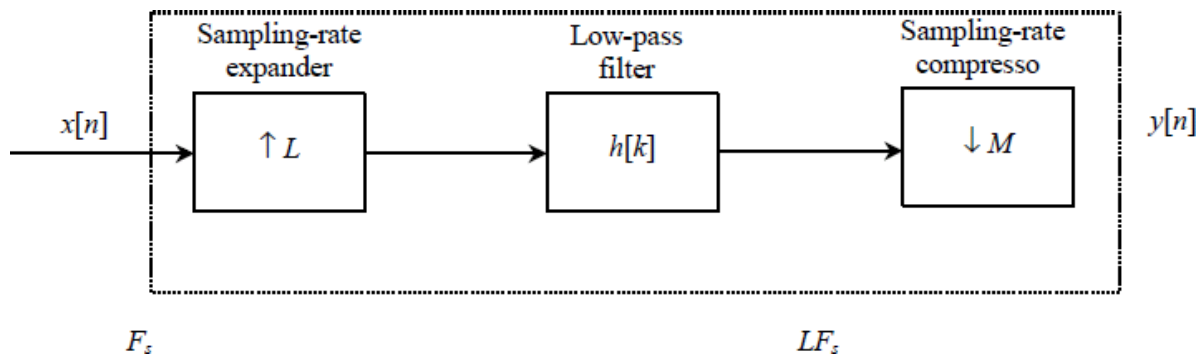


Fig 4.12. Sampling-rate conversion by expansion, filtering, and decimation

An example of sampling-rate conversion would take place when data from a CD is transferred onto a DAT. Here the sampling-rate is increased from 44.1 kHz to 48 kHz. To enable this process the non-integer factor has to be approximated by a rational number:

$$\frac{L}{M} = \frac{48}{44.1} = \frac{160}{147} = 1.08844$$

Design of narrow band filters

A common need in electronics and DSP is to isolate a narrow band of frequencies from a wider bandwidth signal. For example, you may want to eliminate 60 hertz interference in an instrumentation system, or isolate the signaling tones in a telephone network. Two types of frequency responses are available: the band-pass and the band-reject (also called a notch filter). Figure 4.13 shows the frequency response of these filters,

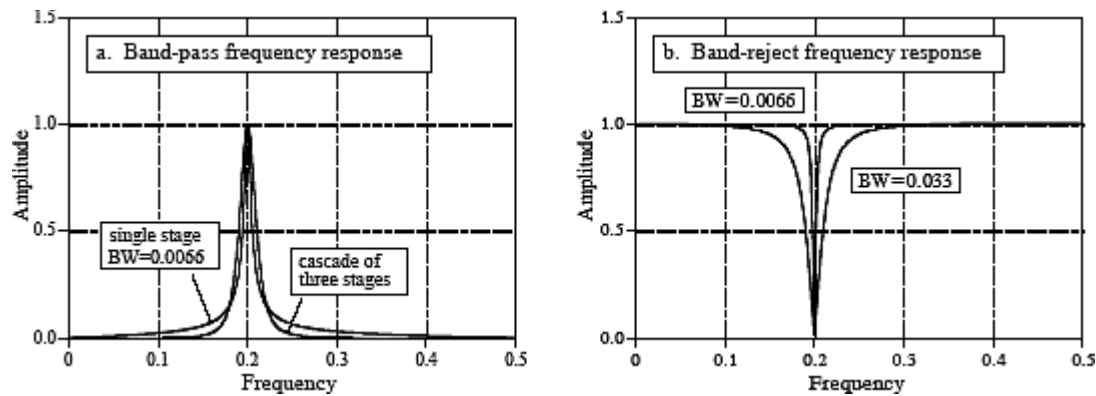


Figure 4.13 shows the frequency response of narrow band filters

Applications of Multirate signal processing

Applications of Multirate DSP Systems

Multirate signal processing is employed in the following systems.

1. Sub-band coding of speech signals and image compression
2. QMF (Quadrature Mirror Filters) for realizing alias-free LTI multirate systems
3. Narrowband FIR and IIR filters for various applications
4. Digital transmultiplexers for converting TDM (Time Division Multiplexed) signals to FDM (Frequency Division Multiplexed) signals and vice versa
5. Oversampling A/D (Analog-to-Digital) and D/A (Digital-to-Analog) converters for high quality digital audio systems and data loggers (or digital storage systems)
6. In digital audio systems the sampling rates of broadcasted signal, CD (Compact Disc), MPEG (Motion Picture Expert Group) standard CD, etc., are different. Hence to access signals from all these devices, sampling rate converters are needed in digital audio systems.
7. In video broadcasting the American standard NTSC (National Television System Committee) and European standard PAL (Phase Alternating Line) employ different sampling rates. Hence to receive both the signals sampling rate converters are needed in video receivers.

Advantages of Multirate Processing

The advantages of multirate processing of discrete time signals are given below.

1. The reduction in number of computations

2. The reduction in memory requirement (or storage) for filter coefficients and intermediate results.
3. The reduction in the order of the system
4. The finite word length effects are reduced

Digital Filter Banks

A digital filter bank is a set of bandpass filters. The digital filter banks can be classified into two types. They are,

- i) Analysis filter banks
- ii) Synthesis filter banks

Analysis Filter Banks

An analysis filter bank is a set of bandpass filters with common input. The analysis filter bank is used for spectrum analysis in which a signal is divided into a set of sub-band signals. The analysis filter bank consists of M numbers of sub-band filters so that the input signal $x(n)$ is divided into M-numbers of sub-band signals.

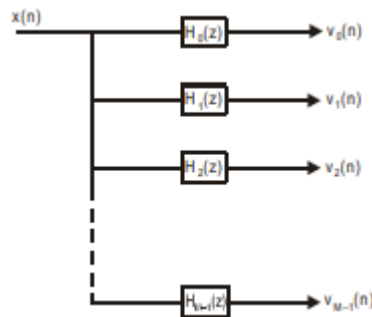


Figure 4.14 Analysis filter banks

Synthesis Filter Bank

A synthesis filter bank is a set of bandpass filters used to combine or synthesis a number of sub-band Signals into a single composite signal as shown in fig 9.31. The synthesis filter accepts M-numbers of sub-band signals $w_0(n)$, $w_1(n)$, $w_2(n)$, $w_{M-1}(n)$, combined to give a signal, $y(n)$. In fact the synthesis filter bank perform the reverse process of analysis filter bank.

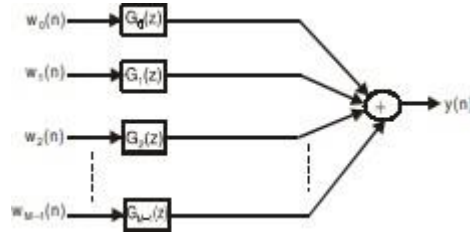


Figure 4.15 Synthesis Filter Bank

Applications of Multirate signal processing

In the digital audio industry, it is a common requirement to change the sampling rates of band-limited sequences. This arises for example when an analog music waveform $x_a(t)$ is to be digitized. Assuming that the significant information is in the band 22 kHz a minimum sampling rate of 44 kHz is suggested. It is, however, necessary to perform analog filtering before sampling to eliminate aliasing of out-of-band noise. Now the requirements on the analog filter it should have a fairly flat passband and a narrow transition band (so that only a small amount of unwanted energy is let in). Optimal filters for this purpose (such as elliptic filters, which are optimal in the minimax sense) have a very nonlinear phase response around the bandedge (i.e., around 22 kHz). In highquality music this is considered to be objectionable. A common strategy to solve this problem is to oversample $x_a(t)$ by a factor of two (and often four). Further applications of multirate filter banks in digital audio are Subband Coding of Speech and Image Signals.

Sub-band Coding of Speech Signals

In sub-band coding of speech signals, the speech signal is divided into sub-bands, decimated, encoded and transmitted to the receiver system. On the receiver side the subband signals are decoded, interpolated and synthesized into the original speech signal. The figure below shows the subband coding of speech signal.

In the transmission side, the input signal is split into M -numbers of non-overlapping frequency bands using an analysis filter bank consisting of M -numbers of bandpass filters. The output of each bandpass filter is decimated by a factor of D . The output of decimators

are encoded and transmitted. On the reception side, the received sub-band signals are decoded and then interpolated to recover the missing samples. The output of interpolators are applied to a synthesis filter bank consisting of M-numbers of bandpass filters to recover the original signal.

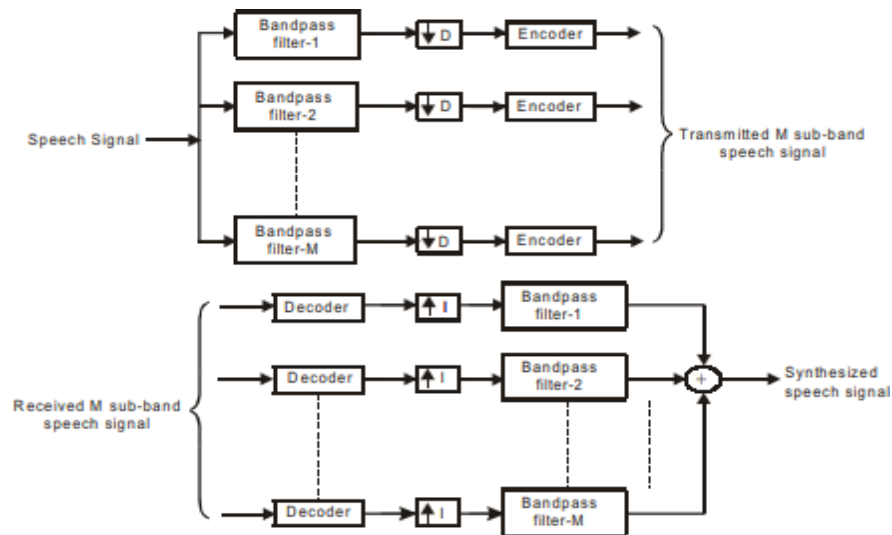


Figure 4.16 Sub-bands Coding of Speech Signals.

Speech compression

The processing of speech involves the analysis, coding, decoding, and synthesis of speech sounds. The speech analyzer consists of normalizers, syllable, segmenters, sound recognizers, sequencers, adapters, and memories which convert the speech elements into a code. The speech synthesizer converts the code to speech by reproducing prerecorded speech elements. There are many applications for the speech analyzer and synthesizer ranging from limited vocabulary to complete communication systems. The most important systems for the communication of speech information are the telephone, phonograph, radio, sound motion picture, and television.

The main objective in the analysis of speech as applied to communication systems is to provide a savings in the channel capacity required for transmission. There are several considerations involved in the use of the different speech elements in communication systems as follows: the bit rate for the transmission of speech, the segmentation of speech,

the analysis of speech, the synthesis of speech. In order to analyze the different types of speech, there must be some means for the segmentation of the flow of speech. The segmentation involves sentences, word , syllables and phonemes.

Segmentation of speech into syllables reduces the number of speech segments and Reduction of bandwidth. In conventional speech processing applications, speech signal is encoded using fixed number of bits over the entire speech signal band. During the process, the bandwidth requirement for speech transmission is relatively high which is of concern. The QMF (Quadrature Mirror Filter) banks are the fundamental building blocks for spectral splitting. The aim is to design a QMF filter and then pass a speech signal through it. In speech signals most of the energy is present in the lower frequency bands. Signal coding is the act of transforming the signal at hand to a more compact form, which can then be transmitted with considerably smaller memory. The motivation behind this is the fact that access to the unlimited amount of bandwidth, which is not possible.

Therefore there is a need to code and compress speech signals. By taking advantage of the fact that most of the energy is present in a particular frequency band we can split the signal into various bands depending on the information content and then code the subband signals separately. The basic theory of multirate digital signal processing is introduced in this section along with the two Sampling rate alteration devices namely up-sampler and down-sampler.

Elimination of interference:

Multirate digital signal processing has a very important role in sub band coding of speech, audio ,video and multiple carrier data transmission because of the high computational efficiency of the multirate algorithms. The performance of a filter bank based interference detection and suppression method to extract the original speech from the interference contaminated speech using the perfect reconstruction (PR) property of the Cosine Modulated filter bank.

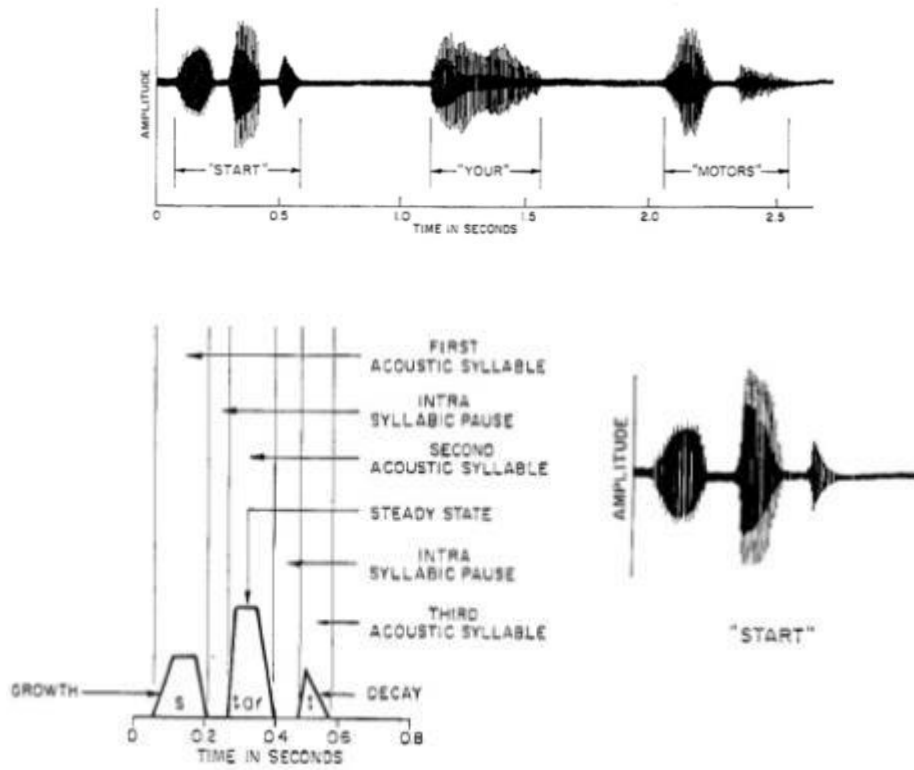


Figure 4.17 Segmentation of speech into syllables

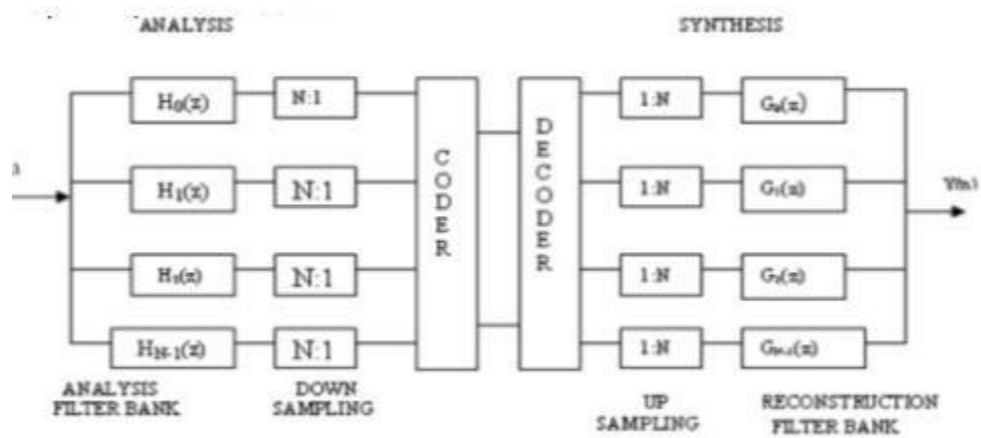


Figure 4.18 QMF filter

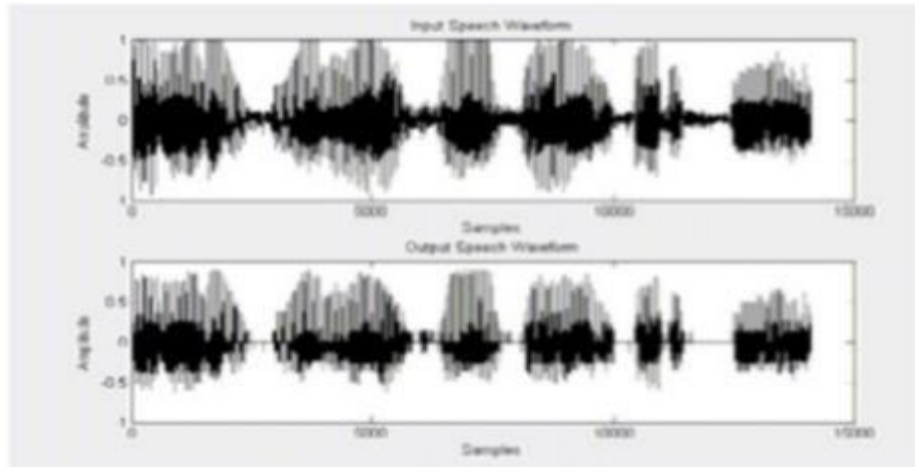


Figure 4.19 Sampled output of speech signal.

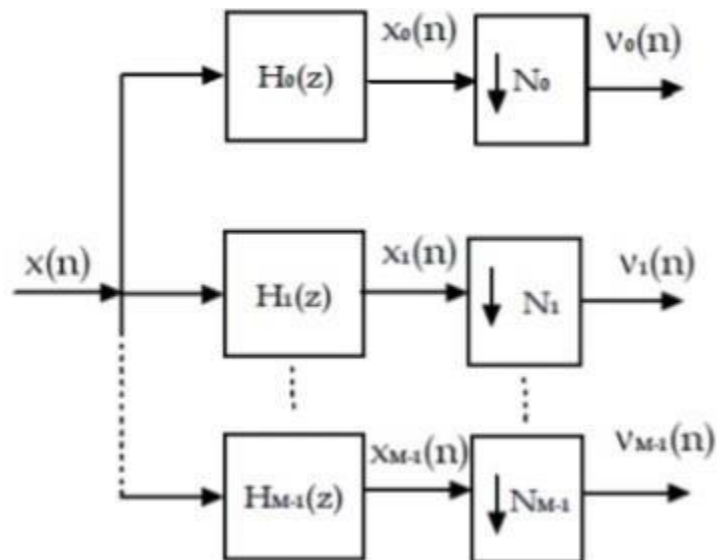


Figure 4.20 Cosine modulated filter bank.

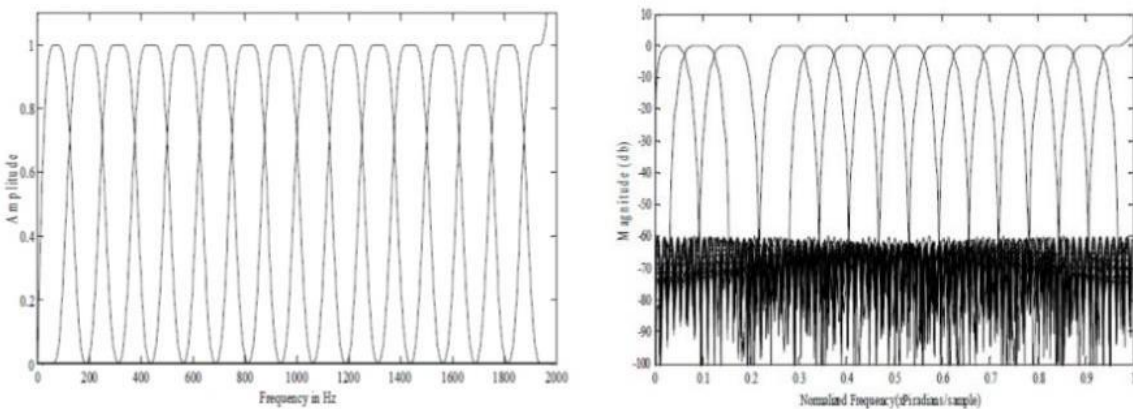


Figure 4.21 Signal with added interference

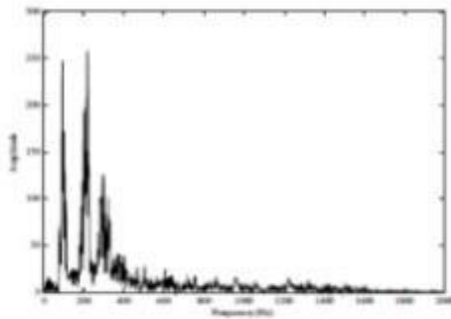


Figure Spectrum of the original signal

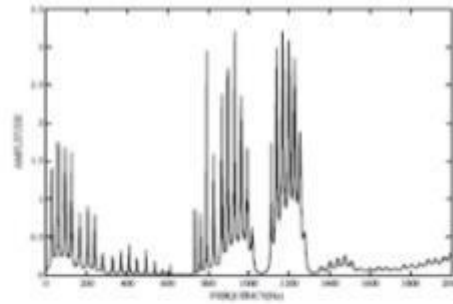


Figure Spectrum of the interference added signal

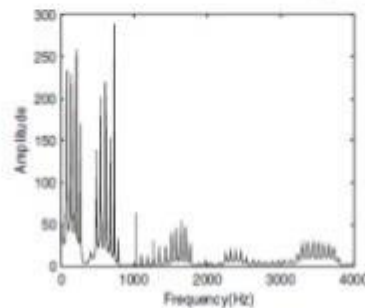


Figure The spectrum of the signal with interference suppression resulting the original spectrum

Figure 4.22 Simulated response of speech process.

The interference suppressor is a critically sampled filter bank system. Modulated filter banks are used to form analysis-synthesis filter banks that divide the received signal into several channels (analysis part), and reconstruct the original signal from the sub-channels (synthesis part). When a signal with added interference is applied to the analysis filter banks, the signal interference appears at the output of one of the filter banks. The spectrum of each sub band signal is estimated to identify the interference bands. For interference suppression, the sub channels affected by the interference are not included in the synthesis filter bank, resulting in notch filtering

Adaptive filter

The goal of adaptive filters are to maintain or derive desired output signal characteristics from a FIR or IIR filter. This goal is obtained via a feedback loop structure that feeds measure of undesired signal characteristics (error) to the filter under consideration and subsequently the filter updates its filter kernel with the fed coefficients to generate or maintain the desired output signal characteristics. The calculation of new coefficients based on the error signal feedback which is to be minimized is powered by some adapting

algorithms. The error is defined as the deviation of output signal from the desired signal characteristics, such that, where $d(n)$ is the desired signal, $y(n)$ is the output signal and $e(n)$ is the error signal, then the following formulas holds.

$$y(n) = \sum_{i=0}^{N-1} W_i(n) x(n - i)$$

To derive the desired signal from the system, we first have to measure the error signal through finding out mathematical correlation between samples of output signal and desired signal. In short, from a higher point of view, this error signal is measured by subtracting the first signal from the latter signal. Then, this error signal is optimally minimized via updating operating filter's coefficients through a live feedback loop.

The use of adaptive filters can be divided majorly into two groups. Firstly, to continuously maintain the output signal unchanged from a running filter. Secondly, to approximate a desired signal from the output signal of a filter. These both approach use the same fundamental structure of the adaptive filter but they varies in terms of orientation and applications.

Adaptive filters can be mainly structurally realized into two ways, namely, spatially and functionally. Spatial structure discusses about the organization of filter components without restricting corresponding filters desired functional output. On the other hand, functional structure discusses about the functional role of the sub-systems of each adaptive filter.

Spatial Structure or Block Diagram

The most common used structure are direct form, cascade form, parallel form and lattice. Transversal layout of adaptive filters are most commonly used, however, lattice layout is also used when its advantages overrides the advantages of transversal layout.

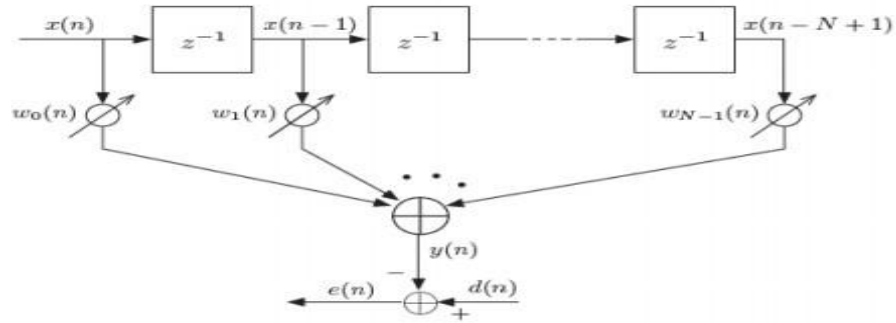


Figure 4.23. Spatial Structure or Block Diagram

Error signal is the difference between output signal and desired signal. That is to say that, error signal is the amount of signal component that adaptive filter optimally removes when it converges and thus arriving at the desired condition.

Adaptive control algorithm is the algorithm that adaptive filter uses to iteratively calculate the new coefficients that optimally reduces the power of error signal. The choice of adaptive control algorithm depends on the data class, memory resources, computational time, energy requirements and overall cost. The L-MSE and LSE are two commonly used algorithm to calculate the updated coefficients.

Musical sound processing:

The musical sound generated by a musical instrument is due to mechanical vibrations produced by a primary oscillator and then making other parts of the instrument to vibrate. For example, in a violin the primary oscillator is a stretched piece of string and it is vibrated by drawing a bow across it, which in turn vibrates the wooden body of the violin, and these vibrations make the surrounding air to vibrate, which produces the musical sound.

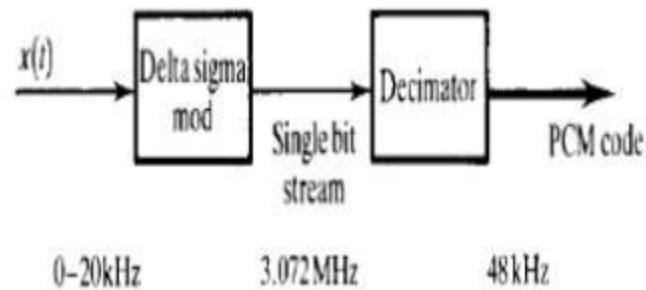


Figure 4.24. High quality Analog to Digital conversion for digital audio

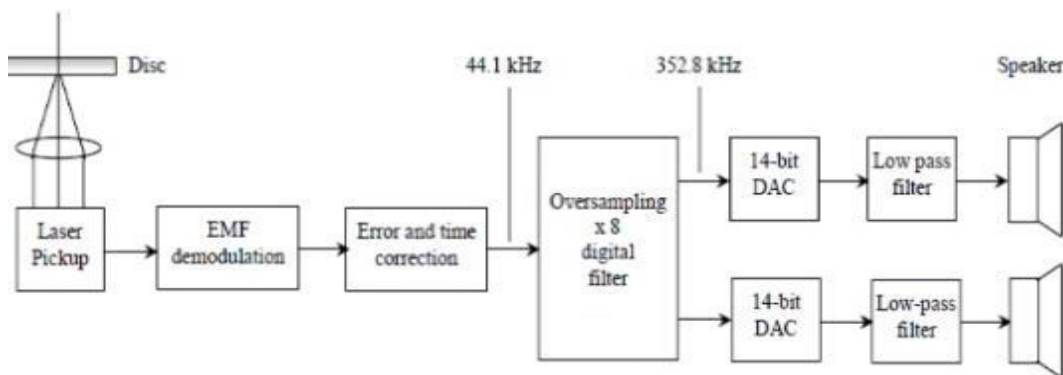


Figure 4.25. Multirate systems are used in a CD player when the music signal is converted from digital into analog

Digital Music Synthesis: Music synthesis plays an important role in multimedia applications, modern entertainment, and professional music systems. The various music synthesis techniques used in the commercial systems are wavetable synthesis, spectral modeling synthesis, nonlinear synthesis (or FM synthesis) and physical modeling synthesis. In wavetable synthesis method, the digital data of one period of the desired musical tone is stored in a table called wavetable. Then, using an IIR filter with no input and the stored data as initial condition, the musical signal is constructed whenever needed. In spectral modeling synthesis, the mathematical equation representing the sound signal is used to generate the required music. The musical sound can be represented by an equation consisting of summation of sinusoidal signals. A musical tone consists of a fundamental tone frequency and its harmonics. Using suitable signal generation algorithm, the desired

musical tone can be generated. In nonlinear synthesis, the musical sound signal is represented as a nonlinear frequency modulated sinusoidal signal containing a fundamental frequency and harmonics of modulating signal. Using signal generation algorithm, various musical tones can be generated for various fundamental frequency. This method cannot be used to generate musics of natural instruments. In physical modeling synthesis, a model of musical instrument like transfer function is constructed and the system model is implemented in a digital hardware, that can be used to generate the musics of an instrument.

The recording of musical programs are generally made in an acoustically inert studio. The sound of each instrument is separately recorded using microphones placed closed to it and then they are mixed using mixing system by a sound engineer. During mixing phase, various audio effects are artificially generated using signal processing circuits and devices. The modern trend is to use digital signal processing for these applications. Some of the special effects that can be implemented during mixing process are echo generation, reverberation, and chorus generation. Also, the musical sound signals can be passed through equalizers to provide amplification or attenuation of some of the tone frequencies

Image enhancement.

The Mach band phenomenon is a good example of this property of the HVS. In an Image 21 consisting of adjacent rectangular bands of different gray levels (called Mach band), the perceived gray level near the edges is different than in the middle of the rectangles. The edge near the darker band appears lighter and the one near the lighter band appears darker than the middle of the rectangle.

For 2-D signals (images) only, however the concepts can be extended to M-D signals. A 2-D analog signal $x_a(t)$ is a function of the variable t which can be defined as a column vector

$$\bar{t} = \begin{bmatrix} t_1 \\ t_2 \end{bmatrix} \quad T_1 = \begin{bmatrix} T_{11} \\ T_{21} \end{bmatrix} \quad T_2 = \begin{bmatrix} T_{12} \\ T_{22} \end{bmatrix} \quad \begin{bmatrix} t_1 \\ t_2 \end{bmatrix} = \begin{bmatrix} T_{11} & T_{12} \\ T_{21} & T_{22} \end{bmatrix} \begin{bmatrix} n_1 \\ n_2 \end{bmatrix}$$

D is the sampling matrix made up of sampling vectors **T1** and **T2**

The matrix **D** that generates **LAT(D)** is not unique and the lattice may or may not be separable. A separable lattice is a lattice that can be represented by a diagonal matrix. For example, the rectangular lattice has a sampling matrix form of **D_r** and matrix **D_h** can generate a hexagonal sampling lattice.

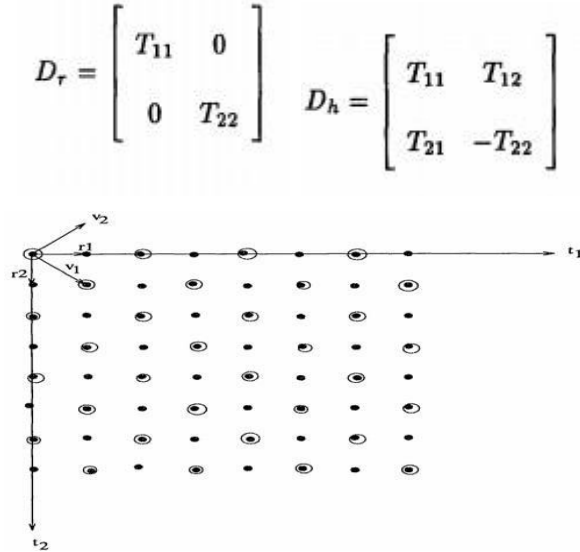


Figure 4.26 Hexagonal resampling and decimation by 2 of a rectangular grid

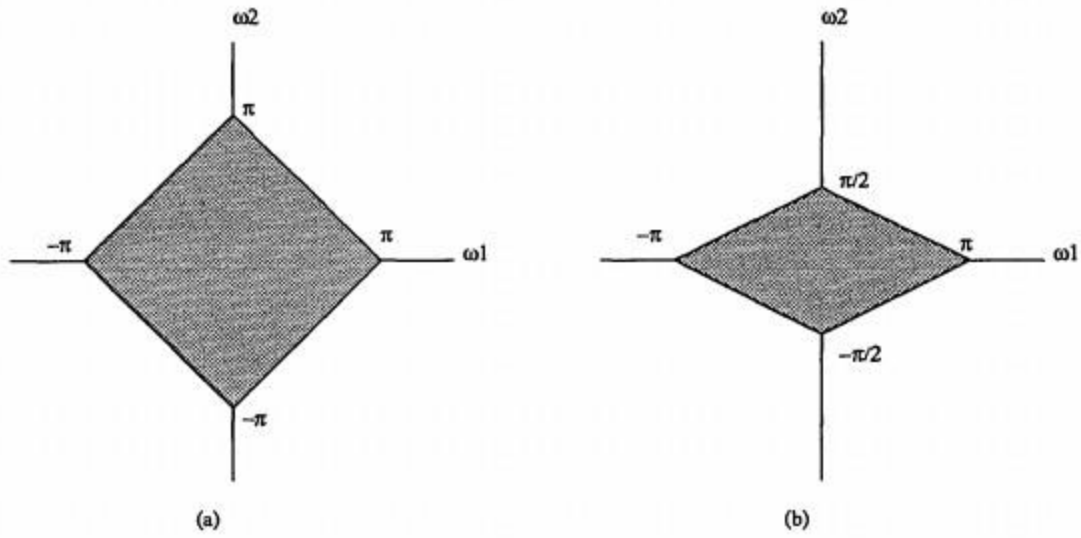


Figure 4.27 Frequency domain support for hexagonal decimation filters: (a) Hex decimation by 2 (b) Hex decimation by 4

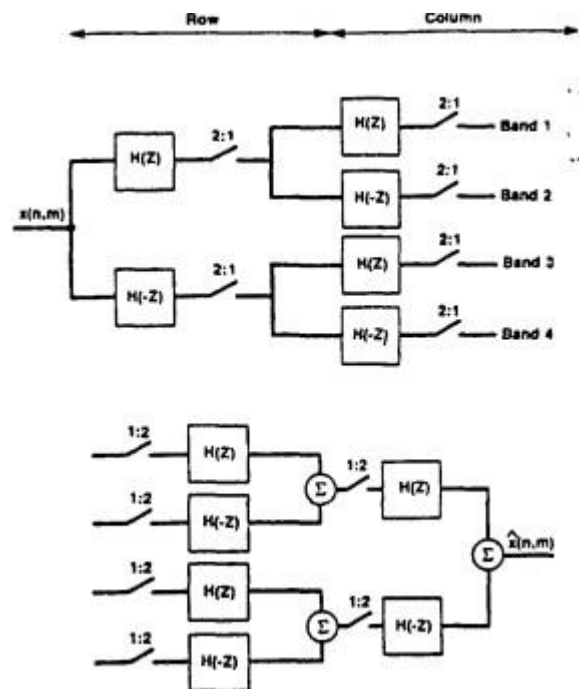


Figure 4.28 Two-dimensional separable QMF bank system.

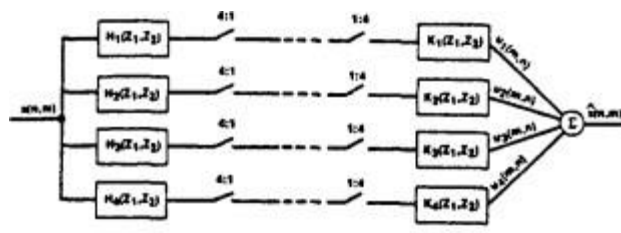
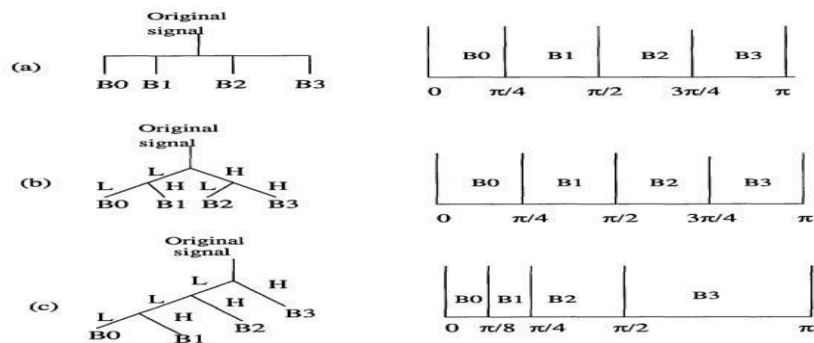


Figure 4.29. Two-dimensional non-separable filter bank system



L stands for lowpass branch and H stands for highpass branch

Figure 4.30. 1-D Subband decomposition structures.

TEXT / REFERENCE BOOKS:

1. John G. Proakis & Dimitris G. Manolakis, "Digital Signal Processing - Principles, Algorithms & Applications", Fourth Edition, Pearson education / Prentice Hall, 2009
2. Sanjit K. Mitra, Digital Signal Processing: A Computer - Based Approach, McGrawHill Education, 4th Edition, 2013
3. B.P. Lathi, "Signal Processing & Linear systems", Oxford University Press, 2nd edition, 2009
4. Lyons, "Understanding Digital Signal Processing", Prentice Hall, 3rd edition, 2010
5. Johny R. Johnson, "Introduction to Digital Signal Processing", PHI, 2006
6. Alan V. Oppenheim, Ronald W. Schaffer, Discrete-Time Signal Processing, Pearson, 3rd Edition, 2010
7. Salivahanan, "Digital Signal Processing, 2nd Edition, TMH, 2010.
8. A. Nagoor Kani, "Digital Signal Processing", Tata McGrawHill Education, 4th Edition, 2012



SATHYABAMA

INSTITUTE OF SCIENCE AND TECHNOLOGY
(DEEMED TO BE UNIVERSITY)

Accredited "A" Grade by NAAC | 12B Status by UGC | Approved by AICTE

www.sathyabama.ac.in

SCHOOL OF ELECTRICAL AND ELECTRONICS

**DEPARTMENT OF ELECTRONICS AND COMMUNICATION
ENGINEERING**

UNIT - V
DIGITAL SIGNAL PROCESSING –SECA1506

V.REALTIME DIGITAL SIGNAL PROCESSING

1 Introduction

Digital Signal Processors (DSPs) are microprocessors with the following characteristics:

- a) Real-time digital signal processing capabilities. DSPs typically have to process data in real time, i.e., the correctness of the operation depends heavily on the time when the data processing is completed.**
- b) High throughput. DSPs can sustain processing of high-speed streaming data, such as audio and multimedia data processing.**
- c) Deterministic operation. The execution time of DSP programs can be foreseen accurately, thus guaranteeing a repeatable, desired performance.**
- d) Re-programmability by software. Different system behavior might be obtained by re-coding the algorithm executed by the DSP instead of by hardware modifications.**

DSPs appeared on the market in the early 1980s. Over the last 15 years they have been the key enabling technology for many electronics products in fields such as communication systems, multimedia, automotive, instrumentation and military. Table 1 gives an overview of some of these fields and of the corresponding typical DSP applications.

Figure 5.1 shows a real-life DSP application, namely the use of a Texas Instruments (TI) DSP in a MP3 voice recorder–player. The DSP implements the audio and encode functions. Additional tasks carried out are file management, user interface control, and post-processing algorithms such as equalization and bass management.

Table 1: A short selection of DSP fields of use and specific applications

Field		Application
Communication	Broadband	Video conferencing / phone
		Voice / multimedia over IP
		Digital media gateways (VOD)
	Wireless	Satellite phone
		Base station
Consumer	Security	Biometrics
		Video surveillance
	Entertainment	Digital still /video camera
		Digital radio
		Portable media player / entertainment console
	Toys	Interactive toys
		Video game console
Industrial and entertainment	Medical	MRI
		Ultrasound
		X-ray
	Point of sale	Scanner
		Vending machine
	Industrial	Factory automation
		Industrial / machine / motor control
		Vision system
Military and aerospace		Guidance (radar, sonar)
		Avionics
		Digital radio
		Smart munitions, target detection

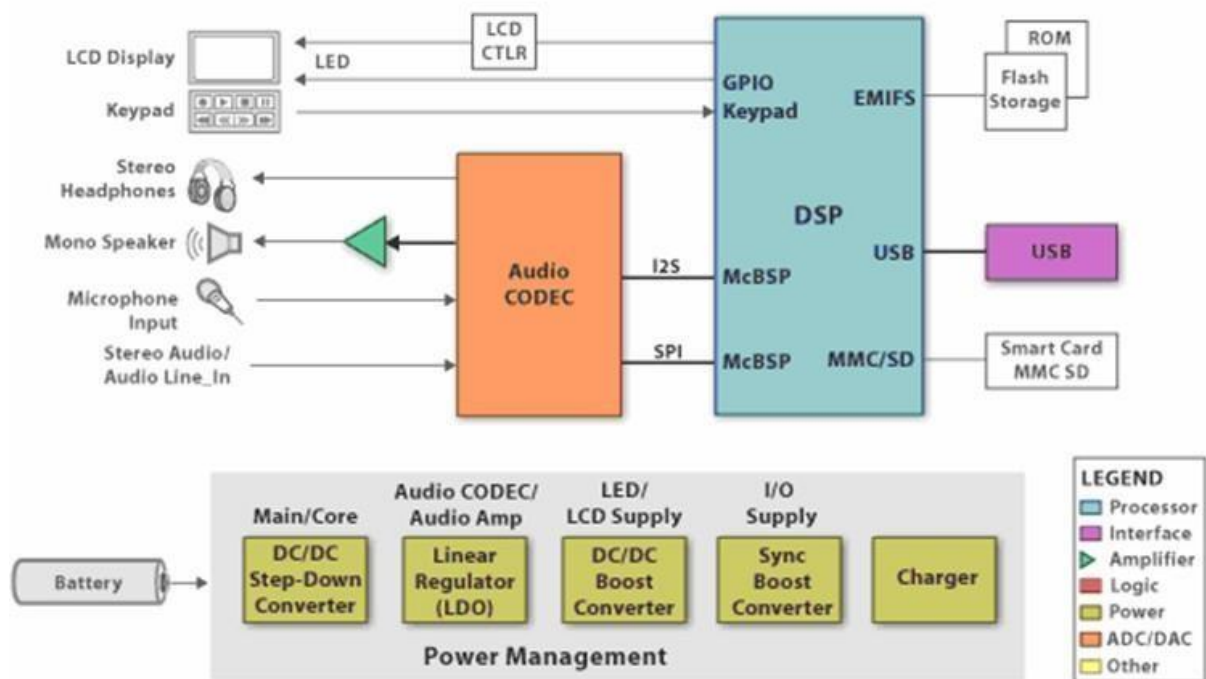


Fig. 5.1: Use of Texas Instruments DSP in a MP3 player/recorder system. Picture courtesy of Texas Instruments from www.ti.com.

Use in accelerators

DSPs have been used in accelerators since the mid-1980s. Typical uses include diagnostics, machine protection and feedforward/feedback control. In diagnostics, DSPs implement beam tune, intensity, emittance and position measurement systems. For machine protection, DSPs are used in beam current and beam loss monitors. For control, DSPs often implement beam controls, a complex task where beam dynamics plays an important factor for the control requirements and implementations. Other types of control include motor control, such as collimation or power converter control and regulation.

DSPs are located in the system front-end. Figure 5.2 shows CERN's hierarchical controls infrastructure, a three-tier distributed model providing a clear separation between Graphical User Interface (GUI), server, and device (front-end) tiers.

DSPs are typically hosted on VME boards which can include one or more programmable devices such as Complex Programmable Logic Devices (CPLDs) or Field Programmable Gate Arrays (FPGAs). Daughtercards, indicated in Fig.5.2 as dashed boxes, are often used; their aim is to construct a system from building blocks and to customize it by different FPGA/DSP codes and by the daughtercards type. DSPs and FPGAs are often connected to other parts of the system via low-latency data links. Digital input/output, timing, and reference signals are also typically available. Data are exchanged between the front-end computer and the DSP over the VME bus via a driver.

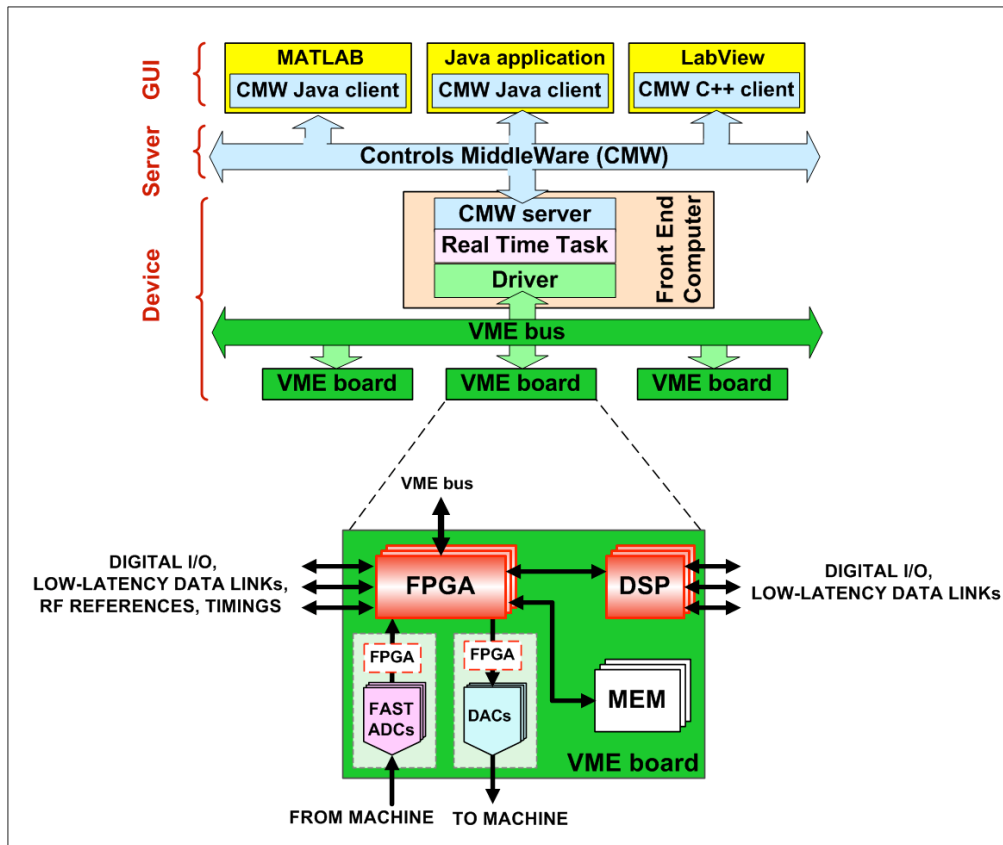


Fig.5.2: Typical controls infrastructure used at CERN and DSP characteristics location

2 DSP evolution and current scenery

DSPs appeared on the market in the early 1980s. Since then, they have undergone an intense evolution in terms of hardware features, integration, and software development tools. DSPs are now a mature technology. This section gives an overview of the evolution of the DSP over their 25-year life span; specialized terms such as ‘Harvard architecture’, ‘pipelining’, ‘instruction set’ or ‘JTAG’ are used.

DSP evolution: hardware features

In the late 1970s there were many chips aimed at digital signal processing; however, they are not considered to be digital signal processing owing to either their limited programmability or their lack of hardware features such as hardware multipliers. The first marketed chip to qualify as a programmable DSP was NEC’s MPD7720, in 1981: it had a hardware multiplier and adopted the Harvard architecture. Another early DSP was the TMS320C10, marketed by TI in 1982. Figure 5.3 shows a selective chronological list of DSPs that have been marketed from the early 1980s until now.

From a market evolution viewpoint, we can divide the two and a half decades of DSP life span into two phases: a development phase, which lasted until the early 1990s, and a consolidation phase, lasting until now. Figure 5.3 gives an overview of the evolution of DSP features together with the first year of marketing for some DSP families.

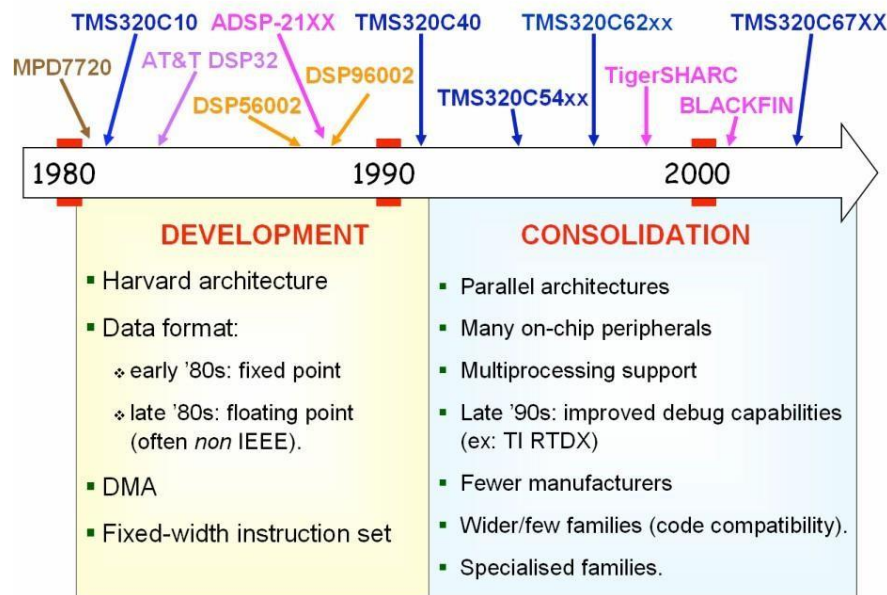


Fig.5.3: Evolution of DSP features from their early days until now. The first year of marketing is indicated at the top for some DSP families.

During the market development phase, DSPs were typically based upon the Harvard architecture. The first generation of DSPs included multiply, add, and accumulator units. Examples are TI's TMS320C10 and Analog Devices' (ADI) ADSP-2101. The second generation of DSPs retained the architectural structure of the first generation but added features such as pipelining, multiple arithmetic units, special address generator units, and Direct Memory Access (DMA). Examples include TI's TMS320C20 and Motorola's DSP56002. While the first DSPs were capable of fixed- point operations only, towards the end of the 1980s DSPs with floating point capabilities started to appear. Examples are Motorola's DSP96001 and TI's TMS320C30. It should be noted that the floating-point format was not always IEEE-compatible. For instance, the TMS320C30 internal calculations were carried out in a proprietary format; a hardware chip converter. was available to convert to the standard IEEE format. DSPs belonging to the development phase were characterized by fixed-width instruction sets, where one of each instruction was executed per clock cycle. These instructions could be complex, and encompassing several operations. The width of the instruction was typically quite short and did not overcome the DSP native word width. As for DSP producers, the market was nearly equally shared between many manufacturers such as AT&T, Fujitsu, Hitachi, IBM, NEC, Toshiba, Texas Instruments and, towards the end of the 1980s, Motorola, Analog Devices and Zoran.

During the market consolidation phase, enhanced DSP architectures such as Very Long Instruction Word (VLIW) and Single Instruction Multiple Data (SIMD) emerged. These architectures increase the DSP performance through parallelism. Examples of DSPs with enhanced architectures are TI's TMS320C6xxx DSPs, which was the first DSP to implement the VLIW architecture, and ADI's TigerSHARC, that includes both VLIW and SIMD features. The number of on-chip peripherals increased greatly during this phase, as well as the hardware features that allow many processors to work together. Technologies that allow real-time data exchange between host processor and DSP started to appear towards the end of the 1990s. This constituted a real sea change in DSP system debugging and helped the developers enormously. Another phenomenon observed during this phase was the reduction of the number of DSP manufacturers. The number of DSP families was also greatly reduced, in favour of wider families that granted increased code compatibility between DSPs of different generations belonging to the same family. Additionally, many DSP families are not 'general-purpose' but are focused on specific digital signal processing applications, such as audio equipment or control loops.

DSP evolution: device integration

Table 2 shows the evolution over the last 25 years of some key device characteristics and their expected values after the year 2010.

Table 2: Overview of DSP device characteristics as a function of time.
The last column refers to expected values.

Characteristic	Year	1980	1990	2000	> 2010
Wafer size [inches]		3	6	12	18
Die size [mm]		50	50	50	5
Feature [µm]		3	0.8	0.1	0.02
RAM [Bytes]		256	2000	32000	1 million
Clock frequency [MHz]		20	80	1000	10000
Power [mW/MIPS]		250	12.5	0.1	0.001
Price [USD]		150	15	5	0.15

Wafer, die, and feature sizes are the basic key factors that define a chip technology. The wafer size is the diameter of the wafer used in the semiconductor manufacturing process. The die size is the size of the actual chips carved up in a wafer. The feature size is the size of the smallest circuit component (typically a transistor) that can be etched on a wafer; this is used as an overall indicator of the density of an Integrated Circuit (IC) fabrication process. The trend in industry is to

go towards larger wafers and chip dies, so as to increase the number of working chips that can be obtained from the same wafer; also called yield. For instance, the current typical wafer size is 12 inches (300 mm), and some leading chip maker companies plan to move to 18 inches (450 mm) within the first half of the next decade. (It should be added that the issue is somewhat controversial, as many equipment manufacturers fear that the 18 inches wafer size will lead to scale problems even worse than for the 12 inches.) Feature size is decreasing, allowing one to either have more functionality on a die or to reduce the die size while keeping the same functionality. Transistors with smaller sizes require less voltage to drive them; this results in a decrease of the core voltage from 5 V to 1.5 V. The I/O voltage has been lowered as well, with the caveat that it remains compatible with the external devices used and their standard. A lower core voltage has been one of the key factors enabling higher clock frequencies: in fact, the gap between high and low state thresholds is tightened thus allowing a faster logic level transition. Additionally, the reduced die size and lowered core voltage allow lower power consumption, an important factor for portable or mobile system. Finally, the global cost of a chip has decreased by at least a factor 30 over the last 25 years.

The trend towards a faster switching hardware (including chip over-clocking) and smaller feature size carries the benefit of increased processing power and throughput. There is a downside to it, however, represented by the electromigration phenomenon. Electromigration occurs when some of the momentum of a moving electron is transferred to a nearby activated ion, hence causing the ion to move from its original position. Gaps or, on the contrary, unintended electrical connections can develop with time in the conducting material if a significant number of atoms are moved far from their original position. The consequence is the electrical failure of the electronic interconnects and the consequent shortened chip lifetime.

DSP evolution: software tools

The improvement of DSP software tools from the early days until now has been spectacular.

Code compilers have evolved greatly to be able to deal with the underlying hardware complexity and the enhanced DSP architectures. At the same time, they allow the developer to program more and more efficiently in high-level languages as opposed to assembly coding. This speeds up considerably the code development time and makes the code itself more portable across different platforms.

Advanced tools now allow the programming of DSPs graphically, i.e., by interconnecting pre-defined blocks that are then converted to DSP code. Examples of these tools are MATLAB Code Generation and embedded target products and National Instruments' LabVIEW DSP Module.

High-performance simulators, emulator and debugging facilities allow the developer to have a high visibility into the DSP with little or no interference on the program execution. Additionally, multiple DSPs can be accessed in the same JTAG chain for both code development and debugging.

DSP current scenery

The number of DSP vendors is currently somewhat limited: Analog Devices (ADI), Freescale (formerly Motorola), Texas Instruments (TI), Renesas, Microchip and VeriSilicon are the basic players. Amongst them, the biggest share of the market is taken by only three vendors, namely ADI, TI and Freescale. In the accelerator sector one can find mostly ADI and TI DSPs, hence most of the examples in this document will be focused on them. Table 3 lists the main DSP families for ADI and TI DSPs, together with their typical use and performance.

Table 3: Main ADI and TI DSP families, together with their typical use and performance

Manufacturer	Family	Typical use and performance
TI	TMS320C2x	Digital signal controllers
	TMS320C5x	Power efficient
	TMS320C6x	High performance
ADI	SHARC	Medium performance. First ADI family (now three generations)
	TigerSHARC	High performance for multi-processor systems
	Blackfin	High performance and low power

3 DSP core architecture

DSP architecture has been shaped by the requirements of predictable and accurate real-time digital signal processing. An example is the Finite Impulse Response (FIR) filter, with the corresponding mathematical equation, where y is the filter output, x is the input data and a is a vector of filter coefficients. Depending on the application, there might be just a few filter coefficients or many hundreds or more.

$$y(n) = \sum_{k=0}^M a_k \cdot x(n-k) \quad \text{----->(1)}$$

As shown in above Equation, the main component of a filter algorithm is the ‘multiply and accumulate’ operation, typically referred to as MAC. Coefficients data have to be retrieved from the memory and the whole operation must be executed in a predictable and fast way, so as to sustain a high throughput rate. Finally, high accuracy should typically be guaranteed. These requirements are common to many other algorithms performed in digital signal processing, such as Infinite Impulse Response (IIR) filters and Fourier Transforms. Table 4 shows a selection of processing requirements together with the main DSP hardware features satisfying them.

Table 4: Main requirements and corresponding DSP hardware implementations for predictable and accurate real-time digital signal processing. The numbers in the first column refer to the section treating the topic.

Processing requirements	Hardware implementations satisfying the requirement
3.2 Fast data access	<ul style="list-style-type: none"> • High-bandwidth memory architectures • Specialized addressing modes • Direct Memory Access (DMA)
3.3 Fast computation	<ul style="list-style-type: none"> • MAC-centred • Pipelining • Parallel architectures (VLIW, SIMD)
3.4 Numerical fidelity	<ul style="list-style-type: none"> • Wide accumulator registers, guard bits, etc.
3.5 Fast execution control	<ul style="list-style-type: none"> • Hardware-assisted, zero-overhead loops, shadow registers, etc.

Fast data access

Fast data access refers to the need of transferring data to / from memory or DSP peripherals, as well as retrieving instructions from memory. The hardware implementations considered for this are three, namely a) high-bandwidth memory architectures,

High-bandwidth memory architectures

Traditional general-purpose microprocessors are based upon the Von Neumann architecture, shown in Fig.5.4(a). This consists of a single block of memory, containing both data and program instructions, and of a single bus (called data bus) to transfer data and instructions from/to the CPU. The disadvantage of this architecture is that only one memory access per instruction cycle is possible, thus constituting a bottleneck in the algorithm execution.

DSPs are typically based upon the Harvard architecture, shown in Fig.5.4(b), or upon modified versions of it, such as the Super-Harvard architecture shown in Fig.5.4(c). In the Harvard architecture there are separate memories for data and program instructions, and two separate buses connect them to the DSP core. This allows fetching program instructions and data at the same time, thus providing better performance at the price of an increased hardware complexity and cost. The Harvard architecture can be improved by adding to the DSP core a small bank of fast memory, called ‘instruction cache’, and allowing data to be stored in the

program memory. The last-executed program instructions are relocated at run time in the instruction cache. This is advantageous for instance if the DSP is executing a loop small enough so that all its instructions can fit inside the instruction cache: in this case, the instructions are copied to the instruction cache the first time the DSP executes the loop. Further loop iterations are executed directly from the instruction cache, thus allowing data retrieval from program and data memories at the same time.

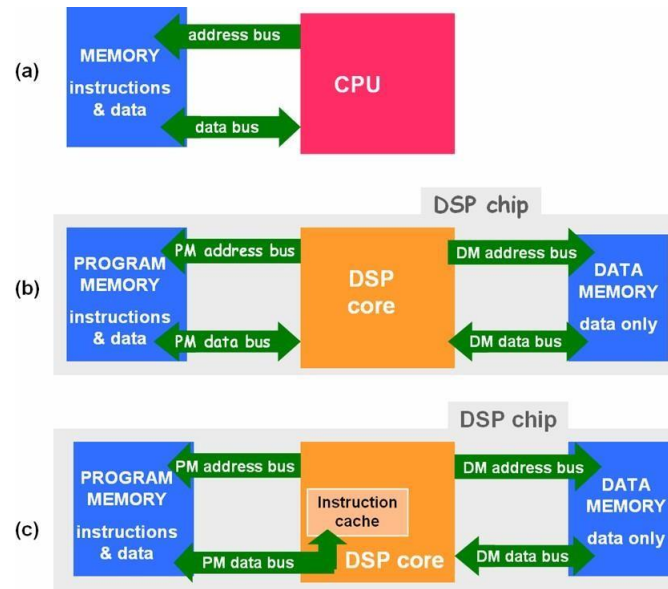


Fig.5.4: (a) Von Neumann architecture, typical of traditional general-purpose microprocessors.

(b) Harvard and (c) Super-Harvard architectures, typical of DSPs.

Another more recent improvement of the Harvard architecture is the presence of a ‘data cache’, namely a fast memory located close to the DSP core which is dynamically loaded with data. Of course, the fact of having the cache memory very close to the DSP allows clocking it at high speed, as routing wire delays are short. Figure 5.5. shows the cache architecture for TI TMS320C67xx DSP, including both program and data cache. There are two levels of cache, called Level 1 (L1) and Level 2 (L2). The L1 cache comprises 8 kbyte of memory divided into 4 kbyte of program cache and 4 kbyte of data cache. The L2 cache comprises 256 kbyte of memory divided into 192 kbyte mapped-SRAM memory and 64 kbyte dual cache memory. The latter can be configured as mapped memory, cache or a combination of the two.

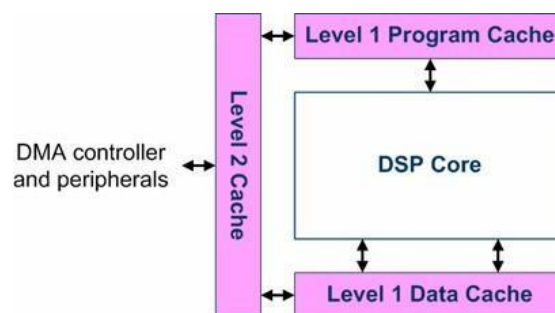


Fig. 5.5: TI DSP TMS320C67xx family two-level cache architecture

Figure 6 shows the hierarchical memory architecture to be found in a modern DSP. Typical levels of memory and corresponding access time, hardware implementation, and size are also shown. As remarked above, a hierarchical memory allows one to take advantage of both the speed and the capacity of different memory types. Registers are banks of very fast internal memory, typically with single-cycle access time. They are a precious DSP resource used for temporary storage of coefficients and intermediate processing values. The L1 cache is typically high-speed static RAM made of five or six transistors. The amount of L1 cache available thus depends directly on the available chip space. A L2 cache needs typically a smaller number of transistors hence can be present in higher quantities inside the DSPs. Recent years have also seen the integration of DRAM memory blocks into the DSP chip, thus guaranteeing larger internal memories with relatively short access times. The Level 3 (L3) memory shown in Fig.5.6 is rarely present in DSPs while the external memory is typically available. This is often a large memory with long access times.

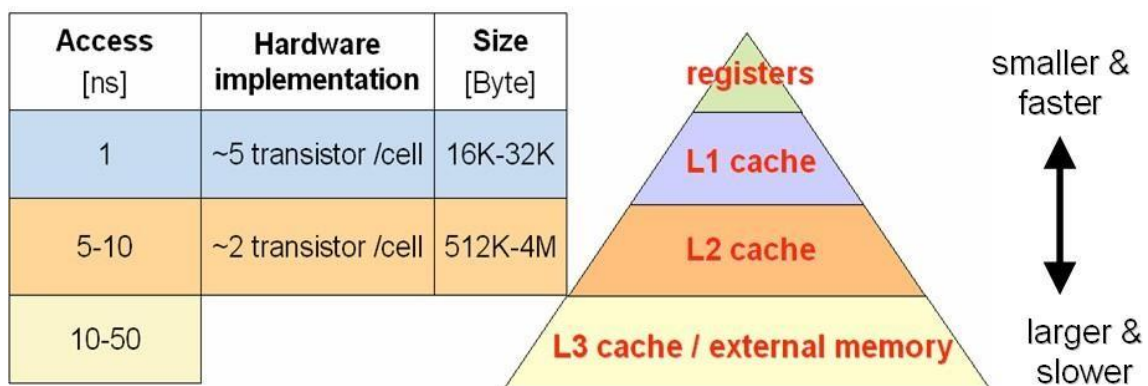


Fig.5.6: DSP hierarchical memory architecture and typical number of access clock cycles, hardware implementation, and size for different memory types

As shown above, cache memories improve the average system performance. However, there are drawbacks to the presence of a cache in DSP-based systems, owing to the lack of full predictability for cache hits. A missing cache hit happens when the data or the instructions needed by the DSP are not stored in cache memory, hence they have to be fetched from a slower memory with an execution speed penalty. A situation causing a missing cache hit is, for instance, the flow change due to branch instructions. The consequence is a difficult worst-case-scenario prediction, which is particularly negative for DSP-based systems where it is important to be able to calculate and predict the system time response. There may, however, be methods used to limit these effects, such as the possibility for the user to lock the cache so as to execute time-critical sections in a deterministic way. Advanced cache organizations characterized by a uniform memory addressing are also under study .

Specialized addressing modes

DSPs include specialized addressing modes and corresponding hardware support to allow a rapid access to instruction operands through rapid generation of their

location in memory. DSPs typically support a wide range of specialized addressing modes, tailored for an efficient implementation of digital signal processing algorithms.

Figure 5.7 adds the address generator units to the basic DSP architecture shown in Fig. 5.4(c). As in general-purpose processors, DSPs include a Program Sequencer block, which manages program structure and program flow by supplying addresses to memory for instruction fetches. Unlike general-purpose processors, DSPs include address generator blocks, which control the address generation for specialized addressing modes such as indexing addressing, circular buffers, and bit-reversal addressing. The two last addressing modes are discussed below.

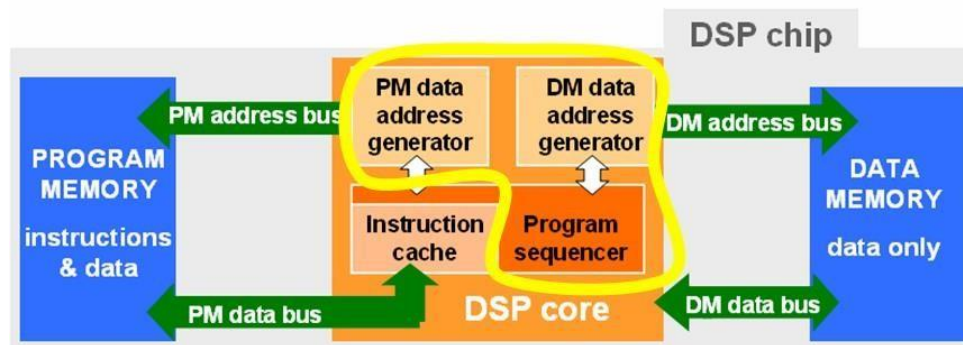


Fig. 5.7: Program sequencer and address generator units location within a generic DSP core architecture

Circular buffers are limited memory regions where data are stored in a First-In First-Out (FIFO) way; these memory regions are managed in a ‘wrap-around’ way, i.e., the last memory location is followed by the first memory location. Two sets of pointers are used, one for reading and one for writing; the length of the step at which successive memory locations are accessed is called ‘stride’. Address generator units allow striding through the circular buffers without requiring dedicated instructions to determine where to access the following memory location, error detection and so on. Circular buffers allow storing bursts or continuous streams of data and processing them in the order in which they have arrived. Circular buffers are used for instance in the implementation of digital filters; strides higher than one are useful in case of multi-rate signal processing. Figure 5.8 shows the order in which data are accessed for a read operation in case of an eleven-element circular buffer and with a stride equal to four.

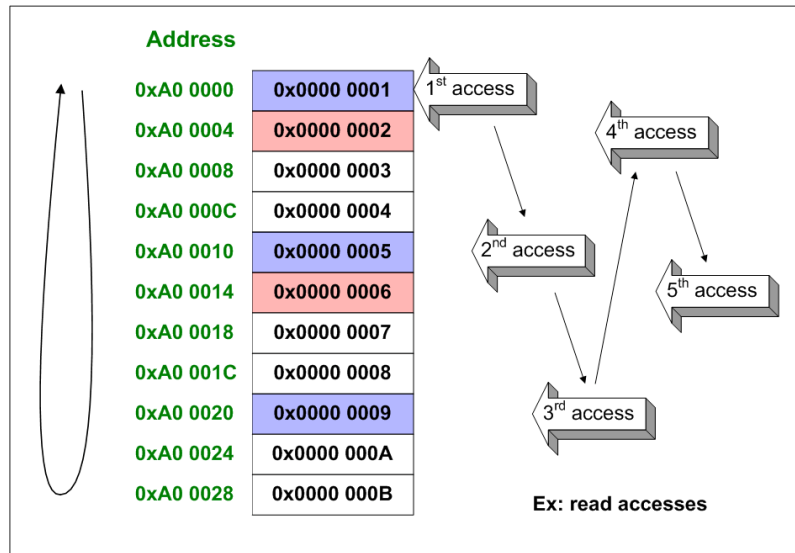


Fig. 5.8: Example of read data access order in a circular buffer composed of 11 elements and with stride equal to 4 elements

Bit-reversal addressing, shown in Fig.5.9, is an essential step in the discrete Fourier transforms calculation. In fact, many implementations of the Fourier transforms require a re-ordering of either the input or the output data that corresponds to reversing the order of the bits in the array index. Figure 5.9 gives an example of the bit-reversal mechanism. Carrying it out by software is very demanding and would result in using many CPU cycles, which are saved thanks to the hardware bit-reversal functionality.

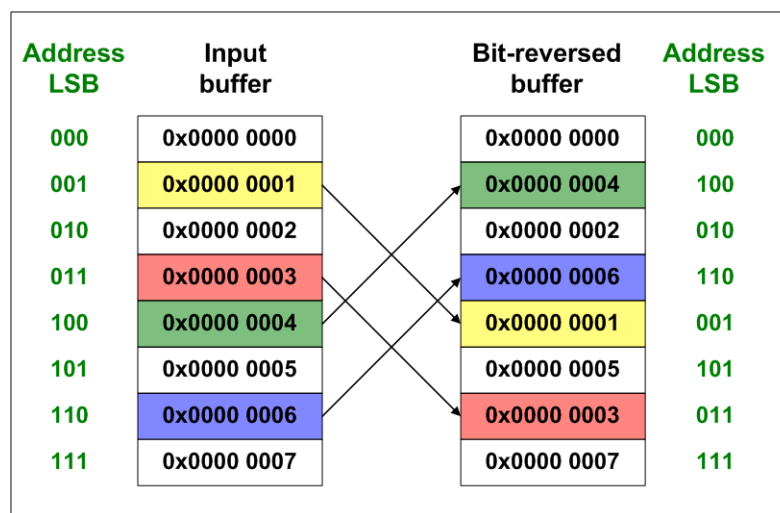


Fig.5.9: Bit-reversal mechanism

Direct Memory Access (DMA) controller

The DMA controller is a second processor working in parallel with the DSP core and dedicated to transferring information between two memory areas or between peripherals and memory. In doing so the DMA controller frees the DSP core for

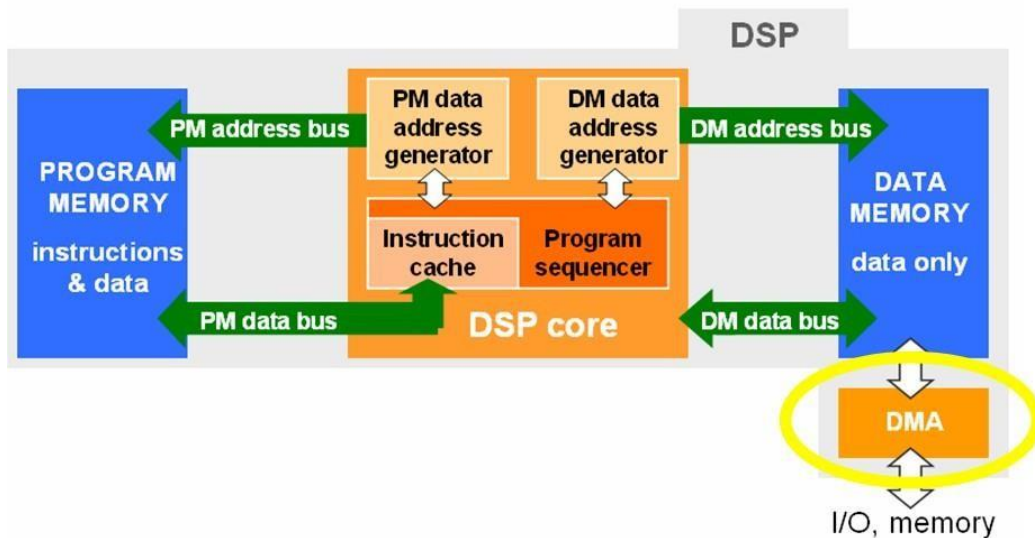


Fig.5.10: An example of DMA controller location within a generic DSP core

other processing tasks. Figure 5.10 shows an example of the DMA location within a general DSP core architecture.

A DMA coprocessor can transfer data as well as program instructions, the latter transfer corresponding typically to the case of code overlay, i.e., of code stored in an external memory and moved to an internal memory (for instance L1) when needed. Multiple and independent DMA channels are also available for greater flexibility. Bus arbitration between the DMA and the DSP core is needed to avoid colliding memory accesses when the DMA and the DSP core share the same bus to access peripherals and/or memories. To prevent bottlenecks, recent DSPs typically fit DMA controllers with dedicated buses.

Figure 5.11 shows the advantages of DMA for the DSP core efficient use: the DSP core must set up the DMA but still there is a net gain in the DSP core availability for other processing activities. Nowadays there are two classes of DMA transfer configurations: register-based and RAM-based, the latter one also called descriptor-based. In register-based DMA controllers the transfer set-up is done by the DSP core via the registers set-up. This method is very efficient but allows mainly simple DMA operations. In RAM-based DMA controllers the set-up parameters are stored in memory. This method is preferred by powerful and recent DSPs as it

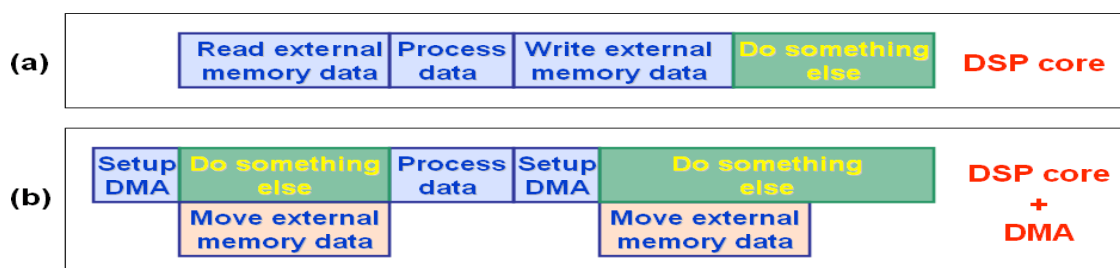


Fig. 5.11: (a) Read-process-write data when the DSP core only is

present; (b) same activity when the DMA takes care of data transfers allows great DMA transfer flexibility.

Figure 5.12 provides two examples of transfer configurations. Plot (a) shows a chained DMA transfer, where the completion of a data transfer triggers a new transfer. This type of data transfer is particularly suited to applications that require a continuous data stream in input. Plot (b) shows a multi-dimensional data transfer, obtained by changing the stride of the DMA transfer. This type of data transfer is particularly useful for video applications.

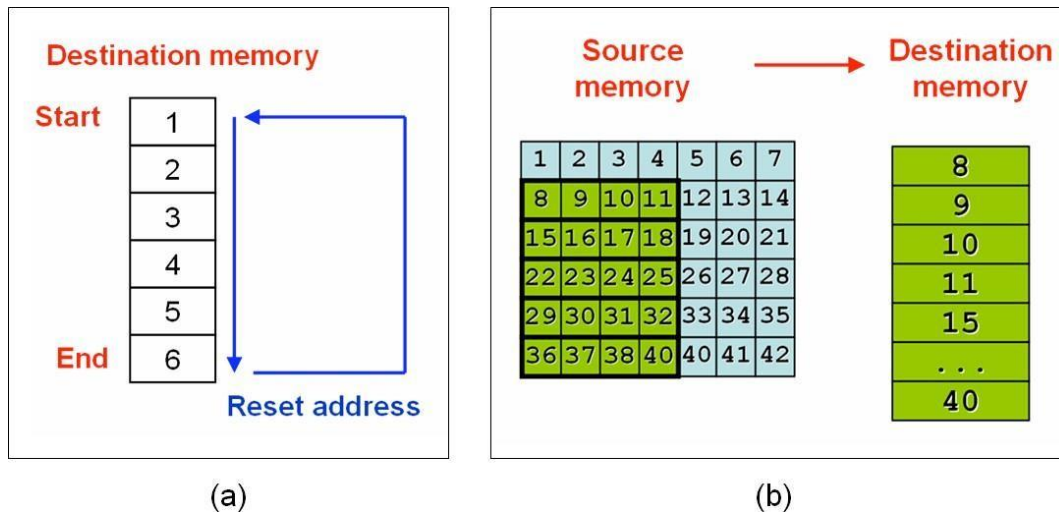


Fig. 5.12: Examples of DMA transfer configurations. (a): chained DMA transfer; (b): Multi- dimensional data transfer.

DSP external events and interrupts can be used to trigger a DMA data transfer. DMA controllers can also generate interrupts to communicate with the DSP core, for instance to inform it that a data transfer has been completed. An example of a powerful and highly flexible DMA controller is that implemented for TI's TMS320C6000 family.

MAC-centred

The basic DSP arithmetic processing blocks are a) many registers; b) one or more multipliers; c) one or more Arithmetic Logic Units (ALUs); d) one or more shifters. These blocks work in parallel during the same clock cycle thus optimizing MAC as well as other arithmetic operations. The blocks are shown in Fig.5.13 and are briefly described below.

- a) **Registers:** these are banks of very fast memory used to store intermediate data processing. Very often they are wider than the DSP normal word width, so as to provide a higher resolution during the processing.
- b) **Multiplier:** it can carry out single-cycle multiplications and very often it includes very wide accumulator registers to reduce round-off or truncation errors. As a consequence, truncation and round-off errors will happen only at the end of the data processing, when the data is stored onto memory. Sometimes an adder is integrated in the multiplier unit.

- c) **ALU:** it carries out arithmetic and logical operations.
- d) **Shifters:** it shifts the input value by one or more bits, left or right. In the latter case, the shifter is called a barrel shifter and is especially useful in the implementation of floating point add and subtract operations.

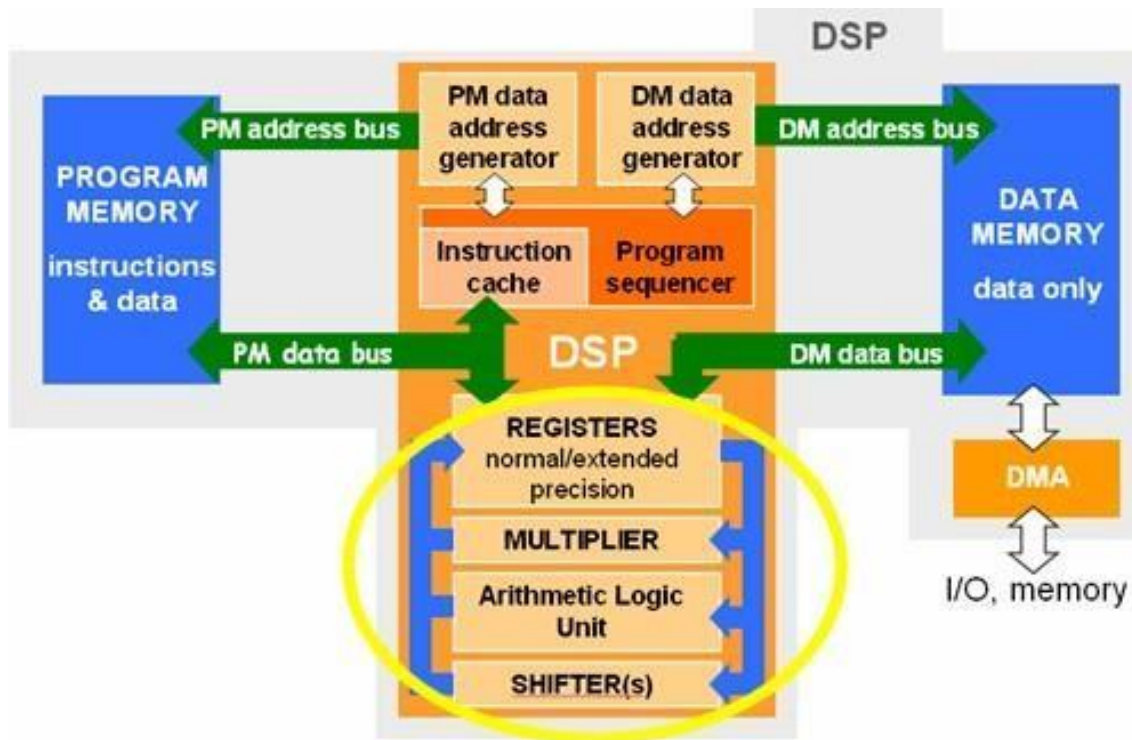


Fig. 5.13: Basic DSP arithmetic processing blocks. The structure shown is that of ADI SHARC.

Instruction pipelining

Instruction pipelining has become an important element to achieve high DSP performance. It consists of dividing the execution of instructions into different stages and executing the different instructions in parallel stages. The net result is an increased throughput of the instruction execution. The whole process can be compared to a factory assembly line, which produces cars for instance: more than one car is in the assembly line at the same moment, at different stages of assembly. This provides a production higher than the case where only one car at a time is produced, where many specialized crews are idle waiting for the next car to require their work.

Table 5 shows the basic pipelining stage into which each instruction is divided:

1. **Fetch.** The DSP calculates the address of the next instruction to execute and retrieve the op- code, i.e., the binary word containing the operands and the operation to be carried out on them.
2. **Decode.** The op-code is interpreted and sent to the corresponding functional unit. The instruction is interpreted and the operands are retrieved.
3. **Execute.** The instruction is executed and the results are written onto the registers.

Table 5: The three basic pipelining stages and corresponding actions

Basic pipelining stages	Action
Fetch	<ul style="list-style-type: none"> • Generate program fetch address • Read op-code
Decode	<ul style="list-style-type: none"> • Route op-code to functional unit • Decode instruction • Read operands
Execute	<ul style="list-style-type: none"> • Execute instruction • Write results back to registers

Figure 5.14 shows the advantage of a pipelined CPU with respect to a non-pipelined CPU, in terms of processing time gain. In a non-pipelined CPU the different instructions are executed serially, while in a pipelined CPU only the same type of stages (e.g. Fetch, Decode and Execute) are serialized and different instructions are executed in parallel. A pipeline is called fully-loaded if all stages are executed at the same time; this corresponds to the maximum possible instruction throughput. The depth of the pipeline, i.e., the number of stages into which an instruction is divided, can vary from one processor to another. Generally speaking a deeper pipeline allows the processor to execute faster, hence many processors subdivide pipeline stages into smaller steps, each one executed at each clock cycle. The smaller the step, the faster the processor clock speed can be. An example of deep pipeline is the TI TMS320C6713 DSP, which includes four fetch stages, two decode stages, and up to ten execution stages.

There are drawbacks and limitations to the pipelining technique. One drawback is the hardware and programming complexity required by it, for instance in terms of capabilities needed in the compiler and the scheduler. This is especially true in the case of deep pipelines. A limitation in the effective instruction execution throughput is given by situations that prevent the pipeline from being fully-loaded. These situations include pipeline flushes due to changes in the program flow, such as code branches or interrupts. In this case, the DSP does not know which instructions it should execute next until the branch instruction is executed. Other situations are data hazards, namely when one instruction needs the result of a previous instruction to be executed. Apart from a reduced throughput,

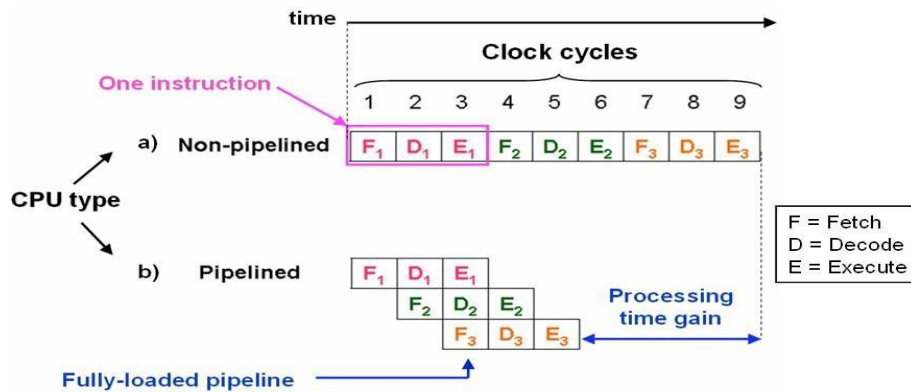


Fig. 5.14: Instruction execution and processing time gain of a pipelined CPU (plot b) with respect to a non-pipelined one (plot a)

these pipeline limitations cause a more difficult prediction of the worst-case scenario. Techniques not described here are available to provide the DSP programmer with a pipeline control; they include time-stationary pipeline control, data-stationary control, and interlocked pipeline.

Parallel architectures

The DSP performance can be increased by an increased parallelism in the instructions execution. Parallel-enhanced DSP architectures started to appear on the market in the mid 1990s and were based on instruction-level parallelism, data-level parallelism, or a combination of both. These two approaches are called Very Long Instruction Word (VLIW) and Single-Input Multiple-Data (SIMD), respectively and are discussed below.

VLIW architectures are based upon instruction level parallelism, i.e., many instructions are issued at the same time and are executed in parallel by multiple execution units. As a consequence, DSPs based on this architecture are also called ‘multi-issue’ DSP. This is an innovative architecture that was first used in the TI TMS320C62xx DSP family. Figure 5.15 shows an example of the VLIW architecture: eight, 32-bit instructions are packed together in a 256-bit wide instruction which is fed to eight separate execution units. Characteristics of VLIW architectures include simple and regular instruction sets. Instruction scheduling is done at compile-time and not at run-time so as to guarantee a deterministic behaviour. This means that the decision on which instructions have to be executed in parallel is done when the program is compiled, hence the order does not change during the program execution. A run-time scheduling would instead make the scheduling dependent on data and resources availability, which could change for different program executions. An important advantage of the VLIW architecture is that it can increase the DSP performance for a wide range of algorithms. Additionally, the architecture is potentially scalable, i.e., more execution units could be added to allow a higher number of instructions to be executed in parallel. There are disadvantages as well, such as the high memory use and power consumption

required by this architecture. From a programmer's viewpoint, writing assembly code for VLIW architecture is very complex and the optimization is often better left to the compiler.

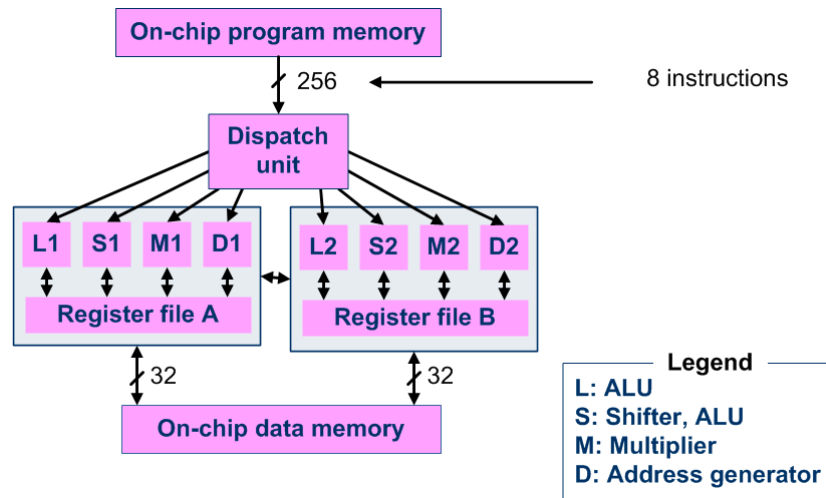


Fig.5.15: TI TMS320C6xxx family VLIW architecture

SIMD architectures are based on data-level parallelism, i.e., only one instruction is issued at a time but the same operation specified by the instruction is performed on multiple data sets. Figure 5.16 shows the example of a DSP based upon the SIMD architecture: two 32-bit input registers provide four, 16-bit each, data inputs. They are processed in parallel by two separate execution units that carry out the same operation. The two, 16-bit data outputs are packed into a 32-bit register. Typical SIMD architecture can support multiple data width and is most effective on algorithms that require the processing of large data chunks. The SIMD operation mode can be switched ON or OFF, for instance in the ADI SHARC DSP. An advantage of the SIMD architecture is that it is applicable to other architectures; an example is the ADI TigerSHARC DSP that comprises both VLIW and SIMD characteristics. SIMD drawbacks include the fact that SIMD architectures are not useful for algorithms that process data serially or that contain tight feedback loops. It is sometimes possible to convert serial algorithms to parallel ones; however, the cost is in reorganization penalties and in a higher program- memory usage, owing to the need to re-arrange the instructions.

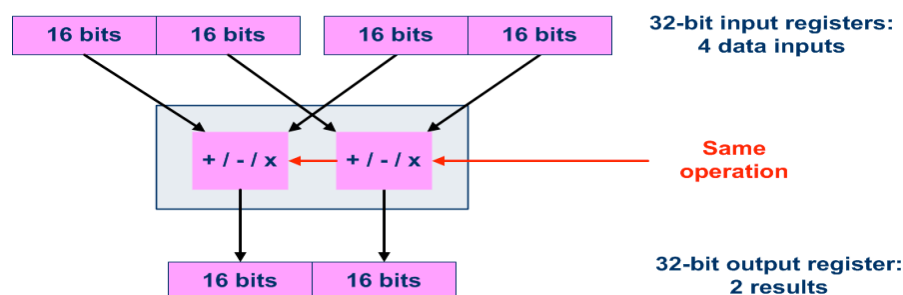


Fig. 5.16: Simplified schematics for ADI SHARC DSP as an example of SIMD architecture

Numerical fidelity

Arithmetic operations such as additions and multiplications are the heart of DSP systems. It is thus essential that the numerical fidelity be maximized, i.e., that errors due to the finite number of bits used in the number representation and in the arithmetic operations be minimized. DSPs have many ways to obtain this, ranging from the numeric representation to dedicated hardware features.

As far as the number representation is concerned, DSPs can be divided into two categories: fixed point and floating point.

Fixed-point DSPs perform integer as well as fractional arithmetic, and can support data widths of 16, 24 or 32 bits. A fixed-point format can represent both signed and unsigned integers and fractions. Fractional numbers can take values in the $[-1.0, 1.0]$ range and are often indicated as $Q_{x,y}$, where 'x' indicates the number of bits located before the binary point and 'y' the number of bits after it. Figure 5.17(a) shows how 16-bit signed fractional point numbers are coded. Signed fractional numbers with 24-bit and 32-bit data width are coded in an equivalent way as $Q_{1.23}$ and $Q_{1.31}$, respectively. They can take values in the same $[-1.0, 1.0]$ range, however, their resolution is higher than the 16-bit implementation.

Floating-point DSPs represent numbers with a mantissa and an exponent, nowadays following the IEEE 754 standard shown in Fig.5.17(b) for a 32-bit number. The mantissa dictates the number precision and the exponent controls its dynamic range. Numbers are scaled so as to use the full word-length available, hence maximizing the attainable precision.

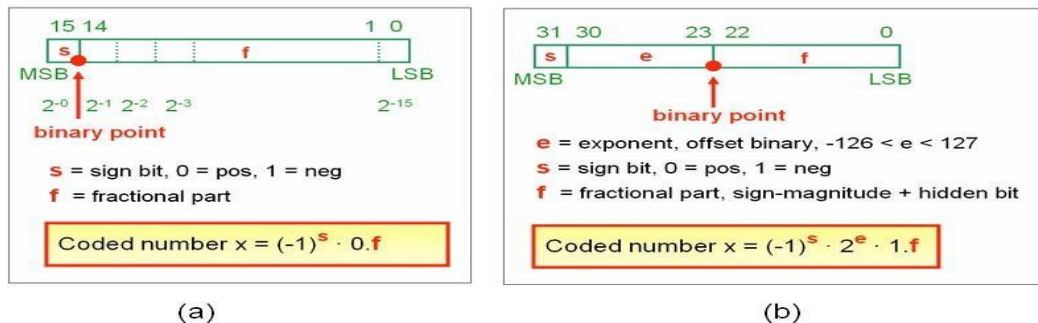


Fig.5. 17: (a): 16-bit signed fractional point, often indicated as $Q_{1.15}$.
(b): IEEE 754 normalized representation of a single precision floating point number.

Floating-point numbers provide a higher dynamic range, which can be essential when dealing with large data sets and with data sets whose range cannot be easily predicted. The dynamic range for a 32-bit number represented as fixed-point and as floating-point is shown in Fig.5.18.

$$\text{Dynamic range}_{\text{dB}} = 20 \log_{10} \left[\frac{\text{largest value}}{\text{smallest value}} \right] \begin{cases} \text{Fixed point} \sim 180 \text{ dB} \\ \text{Floating point} \sim 1500 \text{ dB} \end{cases}$$

Fig.5.18: Dynamic range for 32-bit data, represented as 32-bit signed fractional point and IEEE 754 normalized number

In addition to the different number formats available, DSPs provide hardware ways to improve numerical fidelity. One example is represented by the large accumulator registers, used to hold intermediate and final results of arithmetic operations. These registers are several bits (at least four) wider than the normal registers in order to prevent overflow as much as possible during accumulation operations. The extra bits are called guard bits and allow one to retain a higher precision in intermediate computation steps. Flags to indicate that an overflow/underflow has happened are also available. These flags are often connected to interrupts, thus allowing exception-handling routines to be called. Another means DSPs have to improve numerical fidelity is saturated arithmetic. This means that a number is saturated to the maximum value that can be represented, so as to avoid wrap-around phenomena.

Fast-execution control

Here we show two important examples of how DSP can fast-execute control instructions. The first example is the zero-overhead hardware loop and refers to the program flow control in loops. The second example refers to how DSPs react to interrupts.

Looping is a critical feature in many digital signal processing algorithms. An important DSP feature is the implementation by hardware of looping constructs, referred to as ‘zero-overhead hardware loop’. This allows DSP programmers to initialize loops by setting a counter and defining the loop bounds, without spending any software overhead to update and test loop counters or branching back to the beginning of the loop.

The capability to service interrupts very quickly and in a deterministic way is an important DSP characteristic. Interrupts are internal (for instance generated by internal timers) or external (brought to the DSP code via pins) events that change the DSP execution flow when they are serviced. The latency is the time elapsed from when the interrupt event is triggered and when the DSP starts to execute the first instruction of the corresponding Interrupt Service Routine (ISR). When an interrupt is received and if the interrupt has a sufficiently-high priority, the DSP must carry out the following actions:

- a) stop its current activity;
- b) save the information related to the interrupted activity (called context) into the DSPstack;
- c) start servicing the interrupt.

The context corresponding to the interrupted activity can be restored when the ISR has been executed and the previous activity is continued.

Table 6: Interrupt dispatchers available on the ADI ADSP21160M DSP. The instruction cycle is 12.5 μ s, hence the number of cycles can easily be converted to time.

Interrupt dispatcher	Cycles before ISR	Cycles after ISR
Normal	183	109
Fast	40	26
Super-fast (with alternate registers set)	34	10
Final	24	15

More than one interrupt dispatcher is typically available in a DSP; this means that the user can select the amount of context to be saved, knowing that a higher number of saved registers implies a longer context switching time. An interesting feature available in some DSPs, such as the ADI SHARC AD21160, is the presence of two register sets, called ‘primary’ and ‘alternate’ for all the CPU’s key registers. When an interrupt occurs, the alternate register set can be used, thus allowing a very fast context switch. Table 6 shows the four interrupt dispatchers available on the ADSP21160M DSP and their corresponding latency (‘Cycles before ISR’) and context restore time (‘Cycles after ISR’). The ‘Final’ dispatcher is intended for use with user-written assembly functions or C functions that have been compiled using ‘*#pragma interrupt*’. In particular, this dispatcher relies on the compiler (or assembly routine) to save and restore all appropriate registers.

DSP core example: TI TMS320C67x

Figure 5.19 shows TI’s TMS320C6713 DSP core architecture, as an example of modern VLIW architecture implementing many of the characteristics described in Section 3. This DSP is that used in the laboratory companion of the lectures upon which this paper is based.

Boxes inside the yellow square belong to the DSP core architecture, which here is considered to include the cache memory as well as the DMA controller. The white boxes are components common to all C6000 devices; grey boxes are additional features on the TMS320C6713 DSP.

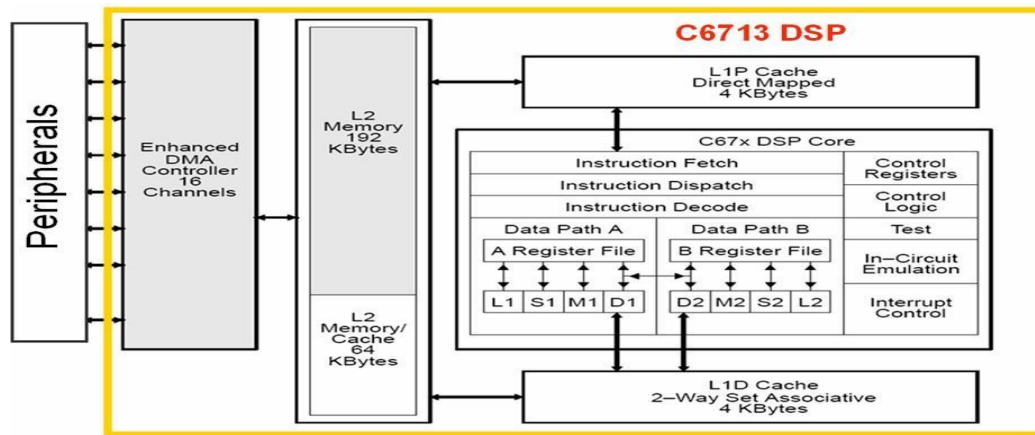


Fig. 5.19: TI TMS320C6713 DSP core architecture. Picture courtesy of TI .

The TMS320C6713 DSP is a floating point DSP with VLIW architecture. The internal program memory is structured so that a total of eight instructions can be fetched at every cycle. To give a numerical example, with a clock rate of 225 MHz the C6713 DSP can fetch eight, 32-bit instructions every 4.4 ns. Features of the C6713 include 264 kBytes of internal memory: 8 kB as L1 cache and 256 kB as L2 memory shared between program and data space. The processing of instructions occurs in each of the two data paths (A and B), each of which contains four functional units (.L, .S, .M, .D). An Enhanced DMA (EDMA) controller supports up to 16 EDMA channels. Four of the sixteen channels (channels 8–11) are reserved for EDMA chaining, leaving twelve EDMA channels available to service peripheral devices.

4 DSP peripherals

The available peripherals are an important factor for the DSP choice. Peripherals are here considered as belonging to two categories:

- a) interconnect, discussed in Section 4.2;
- b) services, such as timers, PLL and power management, discussed in Section 4.3.

DSP developers must in fact carefully evaluate the needs of their system in terms of interconnect and services required, to avoid bottlenecks and reduced system performance.

Modern DSPs often have several peripherals integrated on-chip, such as UARTs, serial, USB and video ports. There are benefits in using embedded peripherals, such as fast performance and reduced power consumption. There are, however, drawbacks, in that embedded peripherals can be less flexible across applications and their unit cost might be higher.

The evolution of DSP-supported peripherals has been terrific over the last 20

years. From the original few parallel and serial ports, DSP can now support a wide peripherals range, including those needed by audio/video streaming applications. Often the DSP chip does not have pins to allow using all supported peripherals at the same time. To overcome this limitation, the pins are multiplexed, i.e., the DSP developer must select at boot time which peripherals he/she needs to have available. An example of pin multiplexing referred to TI's TMS320C6713 DSP is given in Section 4.4.

An overview of interconnect and DSP services is given in Sections 4.2 and 4.3, respectively. Hints on different interfacing possibilities to external memories and data converter memories are provided in Sections 4.5 and 4.6, respectively. Finally, a brief outline of the DSP booting process is given in Section 4.7.

Interconnect

The amount of supported interconnect and data I/O is huge, so only a few examples are given below, divided per interconnect type.

Serial interfaces

- a) **Serial Peripheral Interface (SPI):** this is an industry-standard synchronous serial link that supports communication with multiple SPI compatible devices. The SPI peripheral is a synchronous, four-wire interface consisting of two data pins, one device select pin, and a gated clock pin. With the two data pins, it allows for full-duplex operation to other SPI compatible devices. An example of DSP fitted with a SPI port is ADI's Blackfin ADSP-BF533 .
- b) **Multichannel Buffered Serial Ports (McBSP)** on TI's DSPs: this serial interface is based upon the standard serial port found in TMS320C2x and TMS320C5x DSPs.
- c) **Multichannel Audio Serial Port (McASP)** on TI's DSPs: this is a serial port optimized for the needs of multichannel audio applications. Each McASP includes transmit and receive sections that can operate synchronized as well as completely independent, i.e., with separate master clocks, bit clocks, and data stream formats.

Parallel interfaces

- a) **ADI's linkports** are parallel interfaces that allow DSP–DSP as well as DSP–peripheral connection. An example of their use for inter-DSP communication to build multi-DSP systems is given in Sub-section 9.3.1.
- b) **Parallel Peripheral Interface (PPI)** on ADI's Blackfin DSP: this is a multifunction parallel interface, configurable between 8 and 16 bits in width. It supports bidirectional data flow and it includes three synchronization lines and a clock pin for connection to an externally-supplied clock. The PPI can receive data at clock speeds of up to 65 MHz, while transmit rates can approach 60 MHz.

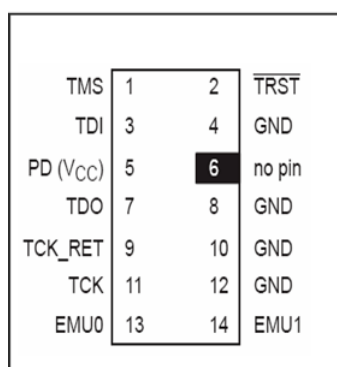
Other interfaces commonly found, for instance in TI DSPs, are Peripheral Component Interconnect (PCI) [29], Inter-Integrated Circuit (I2C) , Host-Port

Interface (HPI) and General-Purpose Input/Output (GPIO) .

Services

System services provide functionality that is common to embedded systems; the on-chip hardware is generally accompanied by an API that allows one to easily interface to them. A few examples of services are given below.

- Timers:** DSPs are typically fitted with one or more general-purpose timers that are used to time or count events, generate interrupts to the CPU, or send synchronization events to a DMA/EDMA controller.
- PLL controller:** it generates clock pulses for the DSP code and the peripherals from internal or external clock signals.
- Power Management:** the power-down logic allows the reduction of clocking so as to reduce power consumption. In fact, most of the operating power of CMOS logic dissipates during circuit switching from one logic state to the other. Significant power can be saved by preventing some of these level switches.
- Boot configuration:** a variety of boot configurations are often available in DSPs. They are user-programmable and determine what actions the DSP performs after it has been reset to prepare for the initialization. These actions include loading the DSP code load from external memory or from an external host. Some boot modes are outlined in Section 4.7
- JTAG:** this interface implements the IEEE standard 1149.1 and allows emulation and debugging. A detailed description of its use can be found in Section 7.2. Figure 5.20 shows a typical JTAG connector and corresponding signals .



Signal	Description	Emulator state	Target state
TMS	Test mode select	OUT	IN
TDI	Test data input	OUT	IN
TDO	Test data output	IN	OUT
TCK	Test clock: 10.368 MHz clock source from emulation cable pod, that can be used to drive the system test clock.	OUT	IN
TRST	Test reset	OUT	IN
EMU0	Emulation pin 0	IN	IN/OUT
EMU1	Emulation pin 1	IN	IN/OUT
PD(V _{CC})	Presence detect: it indicates that the emulation cable is connected and that the power is powered up	IN	OUT
TCK_RET	Test clock return, input to the emulator	IN	OUT
GND	Ground		

Fig.5.20: Fourteen-pin JTAG header and corresponding signals.

Picture courtesy of TI.

TI C6713 DSP example

The peripherals available on TI's TMS320C6713 DSP are shown in Fig. 21 as boxes encircled by a yellow shape. The white boxes are components common to all C6000 devices, while grey boxes are additional features on the TMS320C6713 DSP.

Many peripherals are available on this DSP; however, there are pins that are shared by more than one peripheral and are internally multiplexed. Most of these pins are configured by software via a configuration register, hence they can be programmed to switch functionality at any time. Others (such as the HPI pins) are configured by external pullup/pulldown resistors at DSP chip reset; as a consequence, only one peripheral has primary control of the function of these pins after reset.

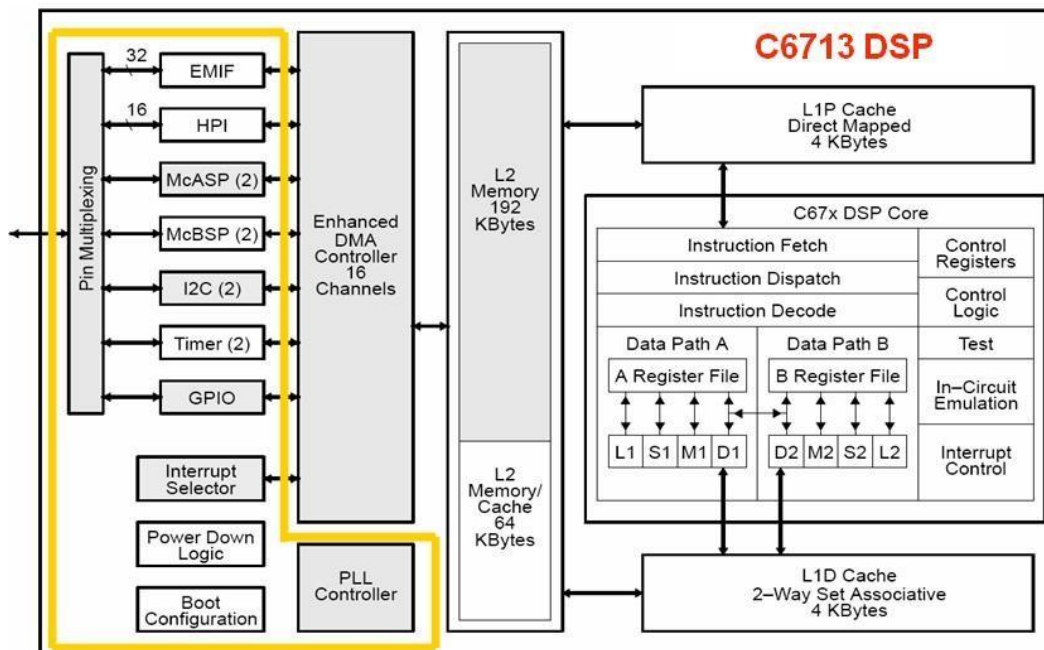


Fig.5.21: TI TMS320C6713 DSP available peripherals. Picture courtesy of TI.

Memory interfacing

DSPs often have to interface with external memory, typically shared with host processors or with other DSPs. The two main mechanisms available to implement the memory interfacing are to use hardware interfaces already existing on the DSP chip or to provide external hardware that carries out the memory interfacing. These two methods are briefly mentioned below.

Hardware interfaces are often available on TI as well as on ADI DSPs. An example is TI External Memory Interface (EMIF), which is a glueless interface to memories such as SRAM, EPROM, Flash, Synchronous Burst SRAM (SBSRAM) and Synchronous DRAM (SDRAM). On the TMS320C6713 DSP, for instance, the EMIF provides 512 Mbytes of addressable external memory space. Additionally, the EMIF supports memory width of 8 bits, 12 bits and 32 bits, including read/write of both big- and little-endian devices.

When no dedicated on-chip hardware is available, the most common solution for interfacing a DSP to an external memory is to add external hardware between memory and DSP, as shown in Fig.5.22. Typically this is done by using a CPLD or an FPGA which implements address decoding and access arbitration. Care must be taken when programming the access priority and/or interleaved memory access in the CPLD/FPGA. This is essential to preserve the data integrity. Synchronous mechanisms should be preferred over asynchronous ones to carry out the data interfacing.

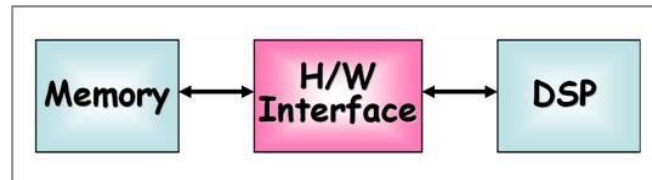


Fig. 5.22: Generic DSP–external memory interfacing scheme. Very often the h/w interface consists of a CPLD or an FPGA.

Data converter interfacing

DSPs provide a variety of methods to interface with data converters such as ADCs. On-chip peripherals are a very convenient data transfer mechanism, since data converters are typically much slower than the DSPs they are interfaced with, hence asking the DSP core to directly retrieve data from the converters is a waste of valuable processing time.

Serial interfaces are often available in TI's DSPs: peripherals such as McBSP and McASP plus the powerful DMA allow an easy interface to many data converter types. Another possible solution for TI DSPs is to use the EMIF in asynchronous mode together with the DMA.

In addition to serial interfaces, ADI Blackfin DSP provides a parallel interface, namely the PPI interface mentioned in Section 4.2, as a convenient way to interact with many converters. This interface typically allows higher sampling rates than the serial interfaces.

A general solution for implementing the DSP–data converter interface is to use an FPGA between DSP and converter, so as to re-buffer the data. Additional pre-processing, such as filtering or down-conversion, can also be carried out in the FPGA. This is the case for instance in CERN's LEIR LLRF system, where converters such as ADCs and DACs are hosted on daughtercards. Powerful FPGAs located on the same daughtercards carry out pre-processing and diagnostics actions under full DSP control.

Finally, mixed-signal DSPs, i.e., DSPs with embedded ADCs and/or DACs, are also available. An example of mixed-signal DSP is ADI's ADSP-21990, containing a pipeline flash converter with eight independent analog inputs and sampling frequency of up to 20 MHz.

DSP booting

The actions executed by the DSP immediately after a power-down or a reset are

called DSP boot and are defined by a certain number of configurable input pins. This paragraph will focus on how the executable file(s) is uploaded to the DSP after a power-down or reset. Two methods are available, which typically correspond to differently built executables. More information on the code building process and on the many file extensions can be found in Section 6.4.

The first method is to use the JTAG connector to directly upload to the executable in the DSP. Upon a DSP power-down the code will typically not be retained in the DSP and another code upload will be necessary. This method is used during the system debugging phase, when additional useful information can be gathered via the JTAG.

On operational systems the DSP loads the executable code without a JTAG cable. Many methods are available for doing this, depending on the DSP family and manufacturer; some general ways are described below.

- a) **No-boot.** The DSP fetches instructions directly from a pre-determined memory address, corresponding to EPROM or Flash memory and executes them. On SHARC DSPs, for instance, the pre-defined start address is typically 0x80 0004.
- b) **Host-boot.** The DSP is stalled until the host configures the DSP memory. For TI TMS320C6xxx DSPs, for instance, this is done via the HPI interface. When all necessary memory is initialized, the host processor takes the DSP out of the reset state by writing in a HPI register.
- c) **ROM-boot.** A boot kernel is uploaded from ROM to DSP at boot time and starts executing itself. The kernel copies data from an external ROM to the DSP by using the DMA controller and overwrites itself with the last DMA transfer. After the transfer is completed the DSP begins its program execution. Figure 5.23 visualizes the TI DSP process of booting from ROM memory: the program (shown in green) has been moved from ROM to L2 and L1 Program (L1P) cache via EMIF and DMA.

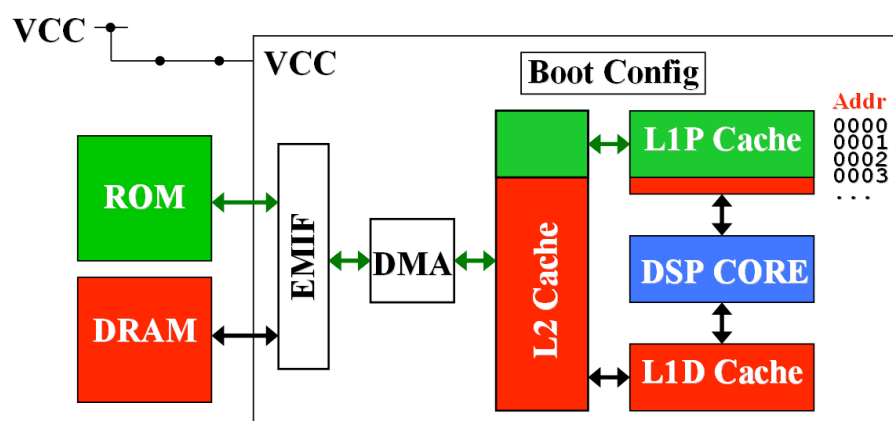


Fig. 5.23: Example of TI TMS320C6x DSP booting from ROM memory. The picture is courtesy of TI.

5 Real-time design flow: introduction

Figure 5.24 shows a time-ordered view of the various activities or phases that a real-time system developer may be required to carry out during a new system development. These activities will be treated in this document in a didactic rather than in a time-related order, to allow even the un-experienced reader to build up the knowledge needed at each step. It should be underlined that the real-time design flow may be not totally forward-directed, and at each step the developer may have to go back to a previous phase to make modifications or carry out additional tests.

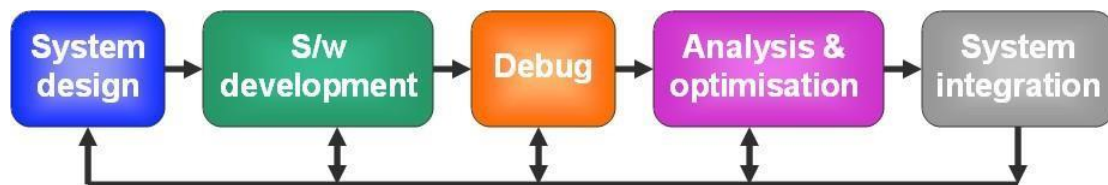


Fig.5.24: Activities typically required to develop a new, DSP-based system

The ‘system design’ phase may include both hardware and software design. For hardware design, the developer must make choices such as the DSP type to use, the hardware architecture/interfaces, and so on. For software design, choices such as the code structure, the data flow and data exchange interfaces must be made. This phase is treated in Section 9.

The ‘software development’ phase includes creating the DSP project and writing the actual DSP code. Basic and essential information for this phase is given in Section 6.

The ‘debug’ phase is a very critical one, where the developer must verify that the code executes what it was meant to. Some debugging techniques as well as different methodologies available (such as simulation and emulation) are described in Section 7.

The ‘analysis and optimization’ phase allows the developer to optimize the system for different goals, such as speed, memory, input/output bandwidth, or power consumption. Analysis and optimization tools are described in Section 8, together with some optimization guidelines.

Finally, the ‘system integration’ is the essential phase where the system is integrated within the existing infrastructure and is therefore made fully operational. It is not possible to give precise details on this phase owing to the many existing control infrastructures. However, general guidelines and good practices are discussed in Section 10.

6 Real-time design flow: software development

DSPs are programmed by software via a cross-compilation. This means that the executable is created in a platform (such as a Windows- or a SUN-based machine) different from the one that it runs on, i.e., the DSP itself. One reason for this is that DSPs have limited and dedicated resources, hence it would not be convenient or even possible to run a file system with a user-friendly development environment.

The choice of programming languages is vast, including native assembly language as well as high-level languages such as C, C++, C extensions and dialects, Ada and so on. High-level software tools such as MATLAB and National Instruments allow one to automatically generate code files from graphical interfaces, thus providing rapid prototyping methods.

The code-building tools are very often provided by the DSP manufacturers themselves. Compilers and Integrated Development Environments (IDEs) are also available from other sources, such as Green Hills Software. The trend is now towards more powerful and user-friendly tools, capable of taming and using in the best possible way the underlying hardware and software complexity.

Development set-up and environment

DSP executables are developed by using Integrated Development Environments (IDEs) provided by DSP manufacturers; they integrate many functions, such as editing, debugging, project files management, and profiling. Very often the licences are bought on a 'per-project' basis, even if ADI provides also floating (i.e., networked) licences. The development environment for TI and ADI DSPs are called 'Code Composer Studio' and 'VisualDSP++', respectively; they provide very similar functionalities. It should be underlined that TI has recently made available free of charge the compiler, assembler, optimizer and linker to non-commercial users. However, neither the IDE nor a debugger were included, thus the developer must still use the proprietary tools.

Figure 25 gives an example of a typical Code Composer screen. On the left-hand side there is the list of all files included in the software project. At the centre of the screen two windows show the code, as a C file (*process.c*) and as assembly code (Dis-assembly window). A breakpoint has been set and the execution is stopped there. Below the code windows, two memory windows are also visible, detailing the data present at addresses 0x80000000 and following, and at addresses 0x40000030 and following. Data at address 0x80000002 is of a different colour because its value changed recently. At the bottom of the IDE screen the following items are displayed: a) the Compile/Link window, which details the results from the last code compilation; b) the Watch window, which displays the value assumed by two C-language variables and c) the Register window, which details the contents of all DSP registers. On the right-hand side there are three graphs: the yellow ones show memory regions, while the green one shows the Fast Fourier Transform of data stored in memory as calculated by the

IDE.

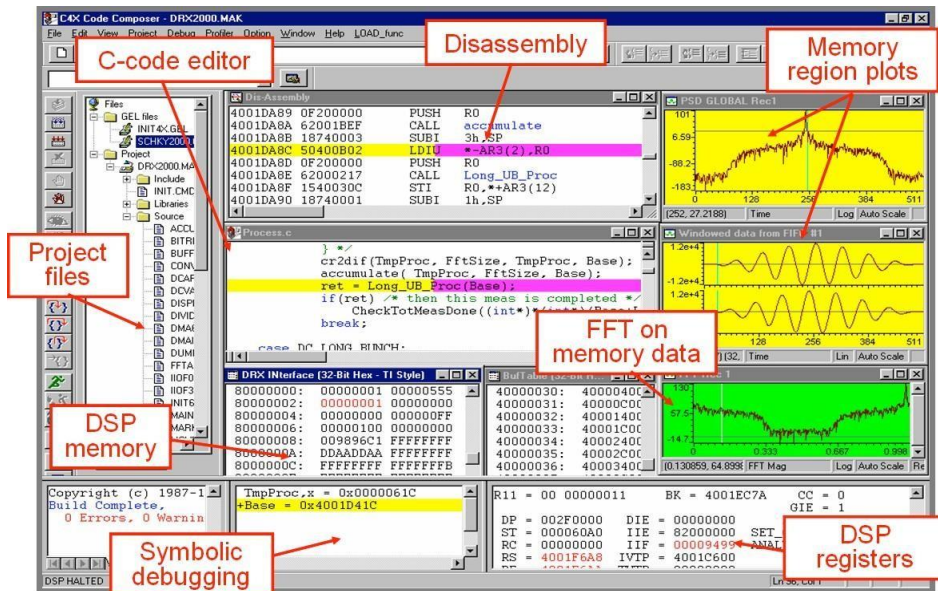


Fig. 5.25: Screenshot from Code Composer, i.e., the TI DSP IDE. The picture was taken in 1998 from the development of CERN's AD Schottky system.

Figure 5.26 shows a typical DSP-based system set-up. On the left-hand side the DSP IDE runs on a PC, which is connected to the DSP via a JTAG emulator and pod. This allows one to edit the code, compile it, download it to the hardware and retrieve debug information. On the right-hand side the system exploitation is shown whereby the DSP runs its program and a PowerPC board, running LynxOS and acting as master VME, controls the DSP actions, downloads the control parameters, and retrieves the resulting data.

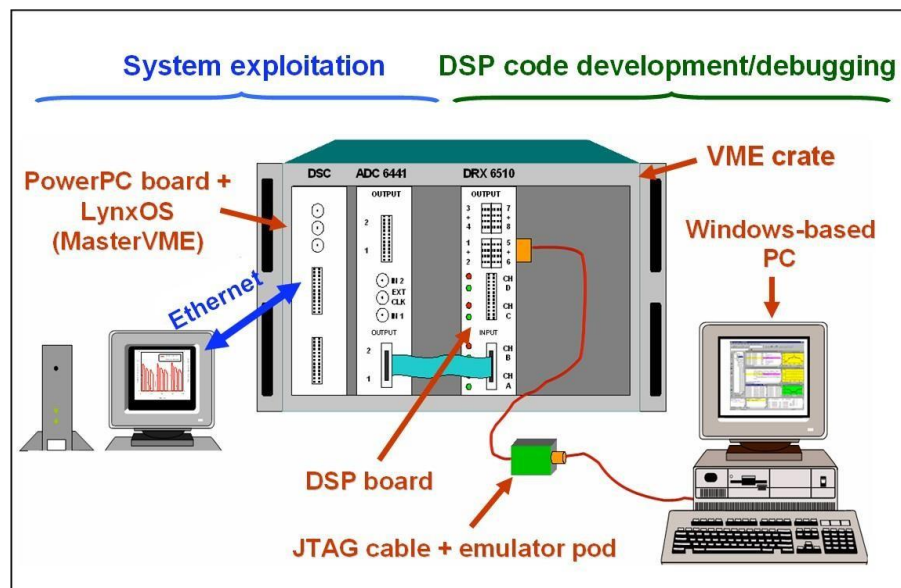


Fig. 5.26: Typical system exploitation (on the left-hand side) and code development (on the right-hand side) set-ups.

Languages: assembly, C, C++, graphical

The choice of the language(s) to be used for the DSP development is very important and depends mainly on the selected DSP, as different DSPs may support different languages. Often a DSP system will include both assembly and high-level languages; the language choice or the chosen balance between the languages depends also on the required processor workload, i.e., on how much the code should be optimized to satisfy the requirements. The language choice is nowadays much larger than in the past, mainly thanks to the improvements of compilers. Additionally, the increased complexity of DSP hardware (see Section 3), such as deep pipelining, makes the hand-optimization much more difficult. The main language choices include: a) assembly language; b) high-level languages such as C, C dialects/extensions and C++; c) graphical languages such as Matlab. These three choices are discussed below.

Assembly language

The assembly language is very close to the hardware, as it explicitly works with registers and it requires a detailed knowledge of the inner DSP architecture. To write assembly code typically takes longer than to write high-level languages; additionally, it is often more difficult to understand other people's assembly programs than to understand programs written in high-level languages. The assembly grammar/style and the available instruction set/peripherals depend not only on the DSP manufacture, but also on the DSP family and on the targeted DSP. As a consequence, it might be difficult or even impossible to port assembly programs from one DSP to another. For instance, for DSPs belonging to the TI C6xxx family there is about an 85% assembly code compatibility, i.e., when going from a C62x to a C64x DSP there are no issues but if moving from a C64x to a C62x one might have to introduce some changes in the code owing to the different instruction set.

DSP applications have typically very demanding processing requirements. The need to obtain the maximum processing performance has often led DSP programmers to use assembly programming extensively. Nowadays the improvements in code compilers and the increasing difficulty in hand-optimizing assembly code have prompted DSP developers to use high-level languages more often. However, in some DSPs there are still features available only in assembly, such as the super-fast interrupt dispatcher for ADI's ADSP21160M DSP shown in Table 6. Very often, the bulk of the DSP code is written in high-level languages and the parts needing a better performance may be written in assembly.

Figure 5.27 gives an example of how one line of C code is converted to the corresponding assembly code for the TI C6317 DSP. The upper window shows part of the 'SIN_to_output_RTDX.c' file, which was included in the DSP laboratory companion of the lectures described in this document; the lower 'Disassembly' window shows the resulting assembly code.

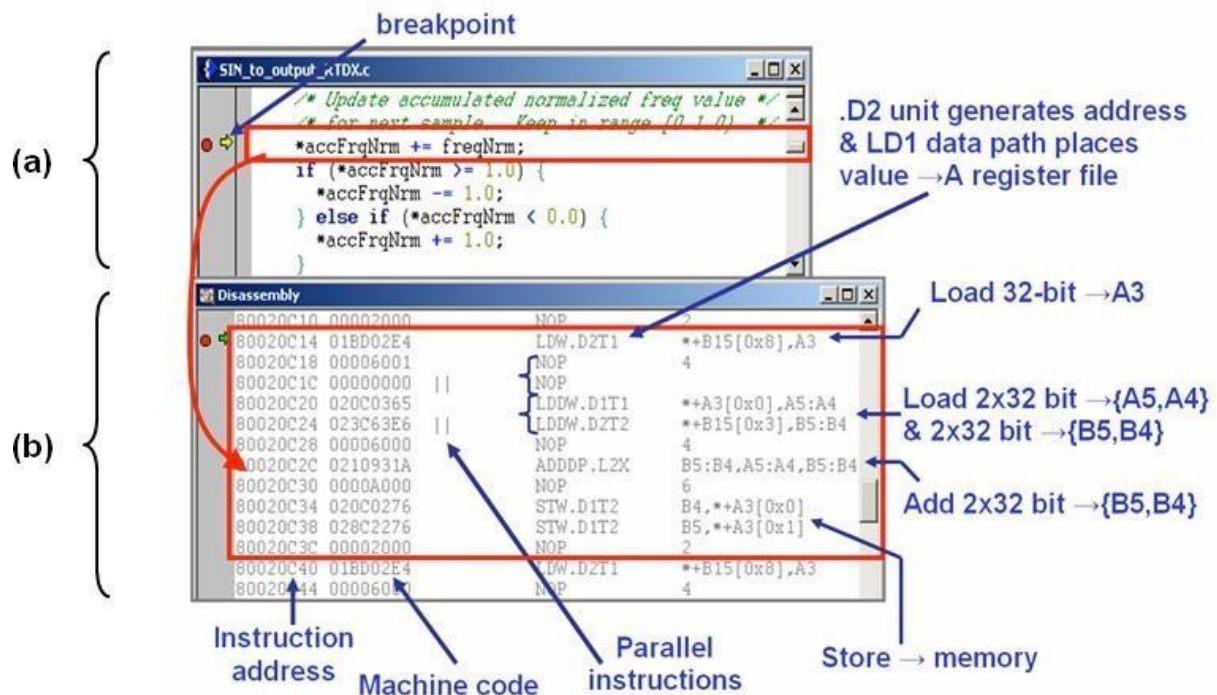


Fig.5. 27: C and assembly language examples for the TI C6713 DSP. Window (a): C source code. Window (b): assembly code resulting from the first C-code line in window (a).

High-level languages: C

The C language was developed in the early 1970s; three main standards exist, referred to as ANSI, ISO, and C99 respectively. There are many reasons why it is convenient to use the C language in DSP-based systems. The C language is very popular and known by engineers and software developers alike; it is typically easier to understand and faster to develop than assembly code. It supports features useful for embedded systems such as control structures and low-level bit manipulations. All DSPs are provided with a C compiler, hence it may be possible to port the C code from one DSP to another.

There are, however, drawbacks to the use of standard C languages in DSP-based systems. First, the executable resulting from a C-language source code is typically slower than that derived from optimized assembly code and has a larger size. The ANSI/ISO C language does not have support for typical DSP hardware features such as circular buffers or non-flat memory spaces. Additionally, the address at which data must be aligned can vary between different DSP architectures: on some DSPs a 4-byte integer can start at any address, but on other DSPs it could start for instance at even addresses only. As a consequence, the data alignment obtained with ANSI/ISO C compilers may be incompatible with the data

alignment required by the DSP, thus leading to deadly bus errors. In the standard C language there is no native support for fixed-point fractional variables, a serious drawback for many DSPs and signal processing algorithms. Finally, the standard C compiler data-type sizes are not standardized and may not fit the DSP native data size, leading for instance to the replacement of fast hardware implementations with slower software emulations. For instance, 64-bit double operations are available in ADI's TigerSHARC as software emulations only; hence the declaration of variables as double and not as float will result in slower execution. Table 8 shows how data-type sizes can vary for different DSPs.

Table 8: Examples of data-type size for different DSPs

char size	Processor	int size	Processor
8	ADI Blackfin	16	ADI '21xx, TI 'C54, C55
16	ADI '21xx, TI 'C54, 'C55	24	Freescale 56x
24	Freescale 56x	32	ADI Blackfin, TI 'C6x
32	ADI Blackfin, TI 'C6x	32	ADI SHARC, TigerSHARC
32	ADI SHARC, TigerSHARC		

(a) (b)

Table 9 shows the data-type sizes and number format for the TI C6713 DSP. The 2's complement and binary formats are used for signed and unsigned numbers, respectively.

Table 9: Data-type sizes and number format for the TI C6713 DSP

TI 'C6713 DSP		
Data type	# bits	Representation
char	8	ASCII
short	16	2's complement / binary
int	32	2's complement / binary
long	40	2's complement / binary
float	32	IEEE 32-bit
double	64	IEEE 64-bit

There are two main approaches to adapting the C language to specific DSPs hardware and to the needs of signal processing applications. The first approach is the definition of 'intrinsic' functions, i.e., of functions that map directly to optimized DSP instructions. Table 10 shows some examples of intrinsic functions available in TI C6713 DSPs. The second approach is to 'extend' the C-language so as to include specialized data types and constructs. Of course, the drawback of the latter approach is a reduced portability of the resulting C language.

Table 10: TI C6713 intrinsic functions – some examples.

Intrinsic	Description
double _rsqrdp(double src);	Returns approximate 64-bit double square root reciprocal
double _fabs(double src);	Returns absolute value of src
unit _enable_interrupts(void);	Returns previous interrupt state & enables interrupts

High-level languages: C++

The C++ programming language supports object-oriented programming and is the language of choice for many business computer applications. C++ compilers are often available for DSPs; some advantages of using it are the ability to provide a higher abstraction layer and the upwards compatibility with the C language. There are, however, several disadvantages, for instance the increased memory requirements due to the more general constructs. Additionally, many application programs and libraries rely on functions such as *malloc()* and *free()*, which need a heap.

While the way to adapt the C-language to DSPs is to add features, the C++ language is adapted by trimming its features. C++ characteristics typically removed are multiple inheritance and exception handling; the resulting code is more efficient and the executable is smaller.

Graphical languages

A trend which has developed over the last five to ten years is to use graphical programming to generate DSP code. Examples of programs and tools aimed at this are the MATLAB, Hypersignal RIDE (now acquired by National Instruments) and the LabVIEW DSP Module. These methodologies generate DSP executables that often are not highly optimized, therefore not suitable for the implementation of demanding DSP-based systems. However, they allow one to quickly move from the design to the implementation phase, thus providing a rapid prototyping methodology.

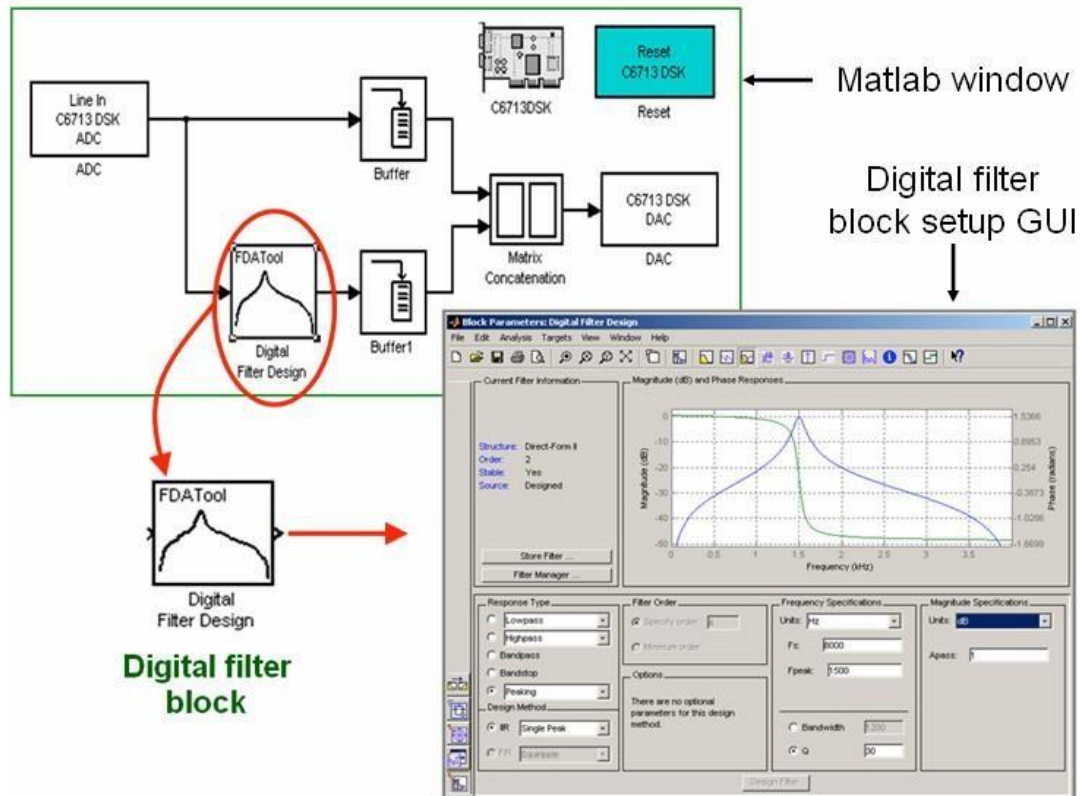


Fig. 5.28: MATLAB graphical programming used in the DSP laboratory companion in these notes. The digital filter block can easily be set up by using a user-friendly set-up GUI.

As an example, MATLAB provides tools such as Simulink, Real-Time Workshop, Real-Time Workshop Embedded Coder, Embedded Target for TI C6xxx DSPs and Link for Code Composer that allow generating embedded code for TI DSPs and downloading it directly into a DSP evaluation board. These tools provide interfaces for the DSP peripherals, too. The DSP laboratory companion on these notes was based upon TI C6713 DSK and MATLAB tools. Figure 5.28 shows the MATLAB graphical program that constituted one of the laboratory exercises. MATLAB allows not only to interface immediately with the on-board CODEC by using the ADC and DAC blocks, but also to set up through a user-friendly GUI the digital filter to be implemented.

Real-time operating system

A Real-Time Operating System (RTOS) is a program that has real-time capabilities, is downloaded to the DSP at boot time, and manages all DSP programs, typically referred to as tasks. The RTOS interfaces tasks with peripherals such as DMA, I/O and memory, via an Application Program Interface (API), as shown in Fig.5.29.

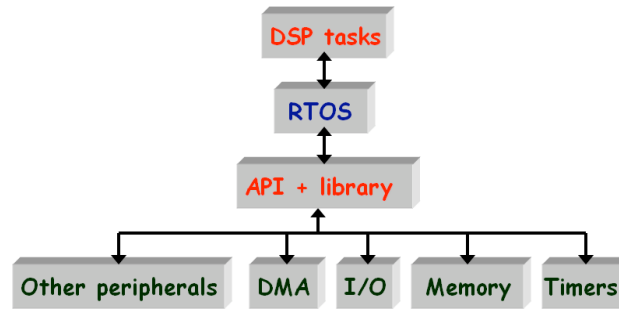


Fig.5.29: Embedded DSP software components

A RTOS is typically task-based and supports multiple tasks (often referred to as threads) by time-sharing, i.e., by multiplexing the processor time over the active tasks set. Each task has a priority associated to it and the RTOS schedules which task should run depending on the priority. Very often this is done in a pre-emptive way, meaning that when a high-priority task becomes ready for execution, it pre-empts the execution of a lower-priority task, without having to wait for its turn in the regular re-scheduling. Finally, RTOS have a small memory footprint, so as not to have too negative an impact on the DSP executable size.

There are many advantages when using a RTOS to develop a DSP-based system. For instance, the API and library shown in Fig.5.29 provide a device abstraction level between DSP hardware features and task implementation, thus allowing a DSP developer to focus on the task rather than the hardware interface's design and coding. The DSP developer may have to just call different interfacing functions in case the code should be ported to a different DSP, hence easing code portability. A RTOS manages the task's execution hence the developer can cleanly structure the code, define appropriate priority levels for each task, and insure that their execution meets critical real-time deadlines. System debug and optimization can be improved, and memory protection can often be provided. There are, however, drawbacks to the use of RTOS. As an example, a RTOS uses DSP resources, such as processing time and DSP timers, for its own functioning. Additionally, the RTOS turnover is typically quite high and royalties are often required from developers.

Many RTOS are available at any time, typically targeted to a precise DSP family or processor. Examples are TALON RTOS from Blackhawk, targeted at TI DSPs, and INTEGRITY RTOS from Green Hills Software or NUCLEUS RTOS from Accelerated Technology, targeted at ADI Blackfin DSPs. It is worth mentioning Linux-based OS, such as RT-Linux, RTAI and uLinux. Both RT-Linux and RTAI use a small real-time kernel that runs Linux as a real-time task with lower priority. The last RTOS listed above, uLinux, is a soft-time OS adapted to ADI Blackfin DSPs. uLinux cannot always guarantee RTOS capabilities such as a deterministic interrupt latency; however, it can typically satisfy the needs of commercial products, where time constraints are often on the millisecond order as dictated by the ability of the user to recognise glitches in audio and video signals.

Other RTOS worth mentioning are those provided and maintained by DSP manufacturers. Both TI and ADI provide royalties-free RTOS with similar characteristics, such as a small memory footprint, multi-tasks and multi-priority levels support. They are called DSP/BIOS for TI and VisualDSP++ Kernel (VDK) for ADI, and can optionally be included in the DSP code. In particular, TI DSP/BIOS provides thirty priority levels and four classes of execution threads. The thread classes, listed in order of decreasing priority, are Hardware Interrupts (HWI), Software Interrupts (SWI), Tasks (TSK) and Background (IDL). Figure 5.30 shows how the processing time is shared between different threads in TI DSP/BIOS. In the vertical scale the different threads are ordered by priority, the higher up having more priority; in the horizontal scale the time is shown. Software interrupts can be pre-empted by a higher-priority software interrupt or by a hardware interrupt. Same-level interrupts are executed in a first-come, first-served way. Tasks are capable of suspension (see Task TSK2 in Fig.5.30) as well as of pre-emption.

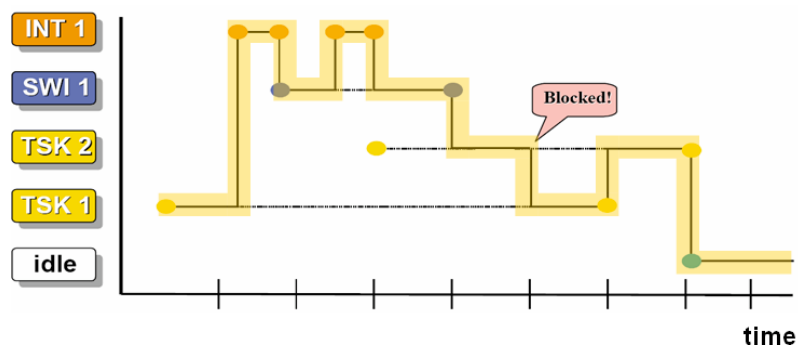


Fig. 5.30: DSP/BIOS prioritized thread execution example. Image courtesy of Texas Instruments.

Code-building process

The DSP code-building process relies on a set of software development tools, typically provided by DSP manufacturers. extensions for ADI and TI DSPs are shown at the bottom of the picture

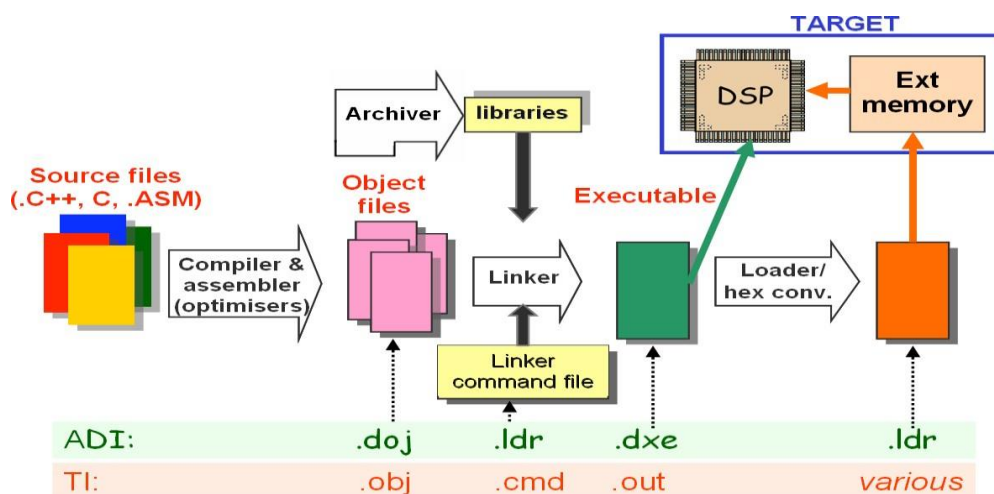


Fig.5. 31: Main elements of the code building process. Typical file.

Figure 5.31 shows the main elements and tools needed for the code-building process. Source files are converted to object files by the compiler and the assembler. Archiver tools allow the creation of libraries from object files; these libraries can then be linked to object files to create an executable. The executable can be directly downloaded from the IDE to the target DSP via a JTAG interface; as an alternative, the executable can be converted to a special form and loaded to a memory external to the DSP, from which the DSP itself will boot. The first approach is typically used during the DSP development phase, while the second approach is more convenient during system exploitation. Finally, the file extensions used at the different code-building process steps for ADI and TI DSPs are shown at the bottom of Fig.5.31.

Three tools, namely compiler, assembler, and linker, are used to generate executable code from C/C++ or assembly source code. Figure 5.32 shows their use in the code-building process on TI DSPs. The tools' main characteristics are summarized in Sub-sections 6.4.1 to 6.4.3.

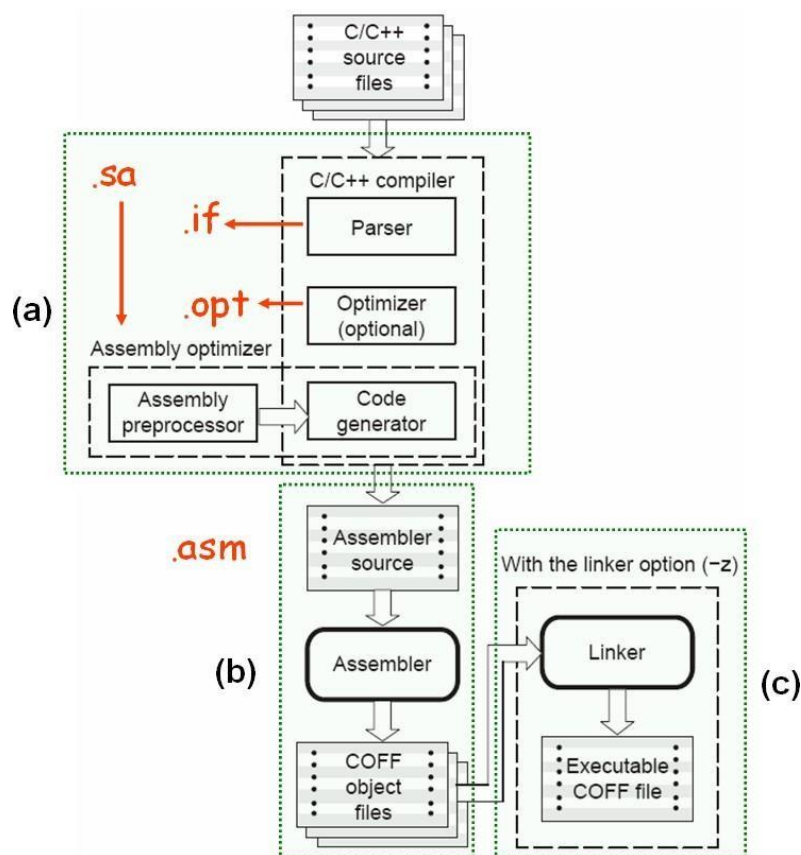


Fig.5.32: Generic code-building processing: (a) compiler; (b) assembler; (c) linker. The picture is courtesy of TI.

C / C++ compiler for TI C6xxx DSPs

The C/C++ compiler generates C6xxx assembler code (.asm extension) from C, C++ or linear assembly source files. The compiler can perform various levels of optimization: high-level optimization is carried out by the optimizer, while low-level, target-specific optimization occurs in the code generator. Finally, the compiler includes a real-time library which is non-target-specific.

Assembler for TI C6xxx DSPs

The assembler generates machine language object files from assembly files; the object files format is the Common Object File Format (COFF). The assembler supports macros both as inline functions and taken from a library; it also allows segmenting the code into sections, a section being the smaller unit of an object file. The COFF basic sections are

- a) text for the executable code;
- b) data for the initialized data;
- c) bss for the un-initialized variables.

Linker for TI C6xxx DSPs

The linker generates executable modules from COFF files as input. It resolves undefined external references and assigns the final addresses to symbols and to the various sections. A DSP system typically includes many types of memory and it is often up to the programmer to place the most critical program code and data into the on-chip memory. The linker allows allocating sections separately in different memory regions, so as to guarantee an efficient memory access. An example of this is shown in Fig.5.33.

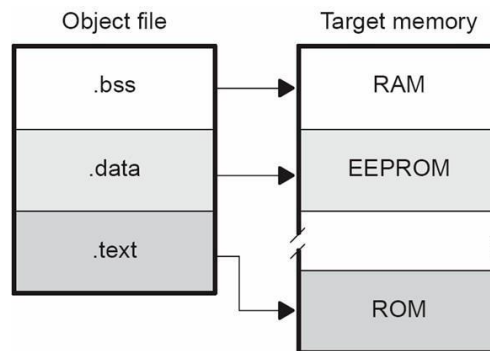


Fig.5.33: Example of sections allocation into different types of target memory

The linker also allows one to clearly implement a memory map shared between DSP and host processor; this is essential for instance to exchange data between them.

7 Real-time design flow: debugging

The debugging phase is the most critical and least predictable phase in the real-time design flow, especially for large systems. The debugging capabilities of the development environment tools can make the difference between creating a successful system and spiralling into an endless search for elusive bugs.

The starting point of this phase is an executable code, i.e., a code without

compilation and linker errors; the goal is to ascertain that the code behaves as expected. The debugging tools and techniques have a strong impact on the amount of time and effort needed to validate a DSP code.

There are many types of bugs: they can be repeatable or intermittent, the latter being much tougher to track down than the first ones. Bugs can be due to the code implementation, such as logical errors in the source code, or can derive from external problems, i.e., hardware misbehaviours. The approaches and the tools to debug a DSP code include simulation, emulation, and real-time debugging techniques. Simulation tools allow running the DSP code on a software simulator fitted with full visibility into DSP internal registers. Emulation tools embed debug components into the target to allow an information flow between target and host computer. Real-time debugging techniques allow a real-time data exchange between host and target without stopping the DSP. These techniques are described in detail in Sections 7.1. to 7.3.

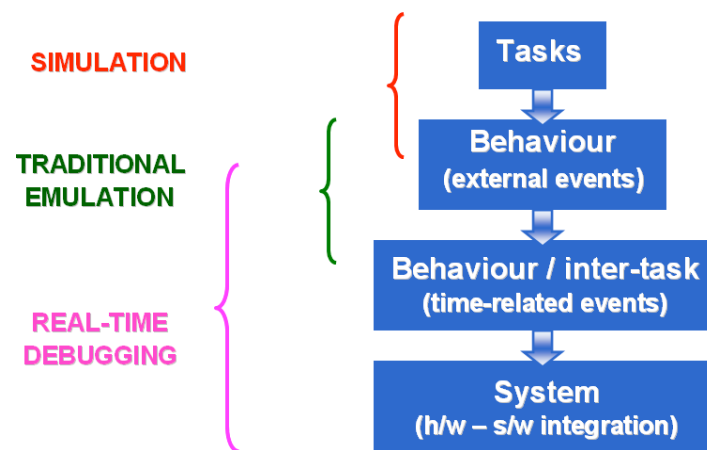


Fig.5.34: Debug steps and their suggested sequencing. The debug tools suited to different steps are also shown.

The developer should not attempt to debug the DSP code as a whole, unless the code itself is relatively short and simple. He is instead recommended to debug the code in several steps: Fig.5.34 shows an example of steps and of their sequencing, together with the appropriate debug tools and techniques. First, single tasks such as functions and routines should be validated; this step can be carried out via simulation only. Second, the behaviour of sub-systems or specific parts of the code can be tested with respect to external events, such as ISR triggering. This part can be carried out with the help of traditional emulation techniques. Third, the behaviour of many tasks can be validated with respect to real-time constraints, such as the proper frequency of ISR triggering. Once all system components have been validated, the whole system can be tested. These last two steps profit particularly from real-time debugging techniques.

Simulation

DSP software simulators have been available for more than fifteen years. They can

simulate CPU instruction sets as well as peripherals and interrupts, thus allowing DSP code validation at a reduced cost and even before the hardware the code should run on is available. Simulators provide a high visibility into the simulated target, in that the user can execute the code step by step and look at the intermediate values taken by internal DSP registers. Large amount of data can be collected and analysed; resource usage can be evaluated and used for an optimized hardware design.

Simulators are highly repeatable, since the same algorithm can be run in exactly the same way over and over. The reader should note that this kind of repeatability is difficult to obtain with other techniques, such as emulation, as external events (for instance interrupts) are almost impossible to be precisely repeated with hardware. Simulators may also allow measurement of the code execution time, with limitations due to the type of simulator chosen. A useful feature available with the TI C5x and C6x simulators is the ‘rewind’, which allows viewing the past history of the application being executed on the simulator.

The main limitation common to DSP simulators is their execution speed, several orders of magnitude slower than the target they simulate; in particular, the more accurate the modelling of the DSP chip and corresponding peripherals, the slower the simulation. DSP tool vendors have overcome this problem by providing different simulators for the same DSP, providing a different level of chip and peripherals modelling. Figure 5.35 shows some simulators available for TI DSPs. The reader should notice that TI provides up to three simulators for each DSP, namely:

- a) **CPU Cycle Accurate Simulator:** This simulator models the instruction set, timers, and external interrupts, allowing the debugging and optimization of the program for code size and CPU cycles.
- b) **Device Functional Simulator:** This simulator not only models instruction set, timers, and external interrupts, but also allows features such as DMA, Interrupt Selector, caches and McBSP to be programmed and used. However, the true cycles of a DMA data transfer are not simulated.
- c) **Device Cycle Accurate Simulator:** This simulator models all peripherals and caches in a cycle-accurate manner, thus allowing the user to measure the total device and stall cycles used by the application.

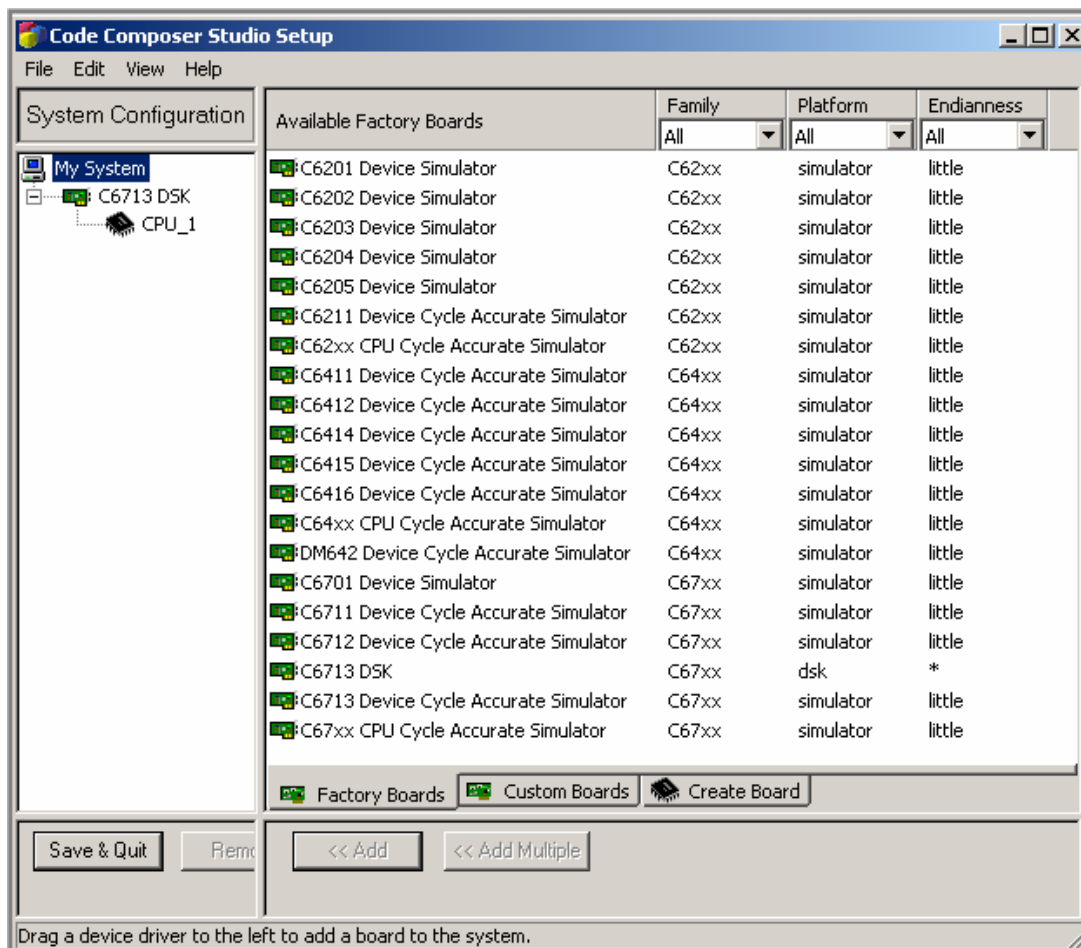


Fig. 5.35: Example of DSP simulators available with TI's Code Composer Studio development environment

Emulation

The integration of processor, memory, and peripherals in a single silicon chip is commonly referred to as System-On-a-Chip (SOC). This approach allows reducing the physical distance between components, hence devices become smaller in size, run faster, cost less to manufacture, and are typically more reliable. From a DSP code developer's viewpoint, the main disadvantage of this approach is the lack of access to embedded signals, often referred to as vanishing visibility. In fact, many chip packages (e.g., ball grid array) do not allow probing the chip pins; additionally, internal chip busses are often not even available at the chip pins. Emulation techniques restore the visibility needed for code debugging by embedding debug components into the chip itself.

There are three main kinds of emulation, namely:

- Monitor-based emulation:** A supervisor program (called monitor) runs on the DSP and uses one of the processor's input-output interfaces to communicate with the debugger program running on the host. The debugging capabilities of this approach are more limited than those provided by the two other approaches; additionally, the monitor presence changes the state of the processor, for instance regarding the instruction

pipeline. The advantage is that it does not require emulation hardware, hence its cost is lower.

- b) **Pod-based In Circuit Emulation (ICE):** The target processor is replaced by a device that acts like the original device, but is provided with additional pins to make accessible and visible internal structures such as internal busses. This emulation approach has the advantage of providing real-time traces of the program execution. However, replacing the target processor with a different and more complex device may create electrical loading problems. Additionally, this solution is quite costly, the hardware is different from the commercialized product and becomes quite difficult to implement at high processor speed.
- c) **Scan-based emulation:** Dedicated interfaces and debugging logic are incorporated into commercially-available DSP chips. This on-chip logic is responsible for monitoring the chip's real-time operations, for stopping the processor when for instance a breakpoint is reached, and for passing debugging information to the host computer. An emulation controller controls the flow of information to /from the target and can be located either on the DSP board or on an external pod. Many types of target–host interface exist. On the DSP board one can typically find a JTAG (IEEE standard 1149.1) connector. On the host computer, parallel or USB ports are often available.

The scan-based emulation technique has been widely preferred over the other two since the late 1980s and is nowadays available on the vast majority of DSPs. Figure 5.36 shows the TI XDS560 emulator, composed of a PC card, a cable with JTAG interface to the target, and an emulation controller pod. Many emulators are available on the market, with different interfaces and characteristics. As an example, it is worth mentioning Spectrum Digital's XDS510 USB galvanic JTAG emulator, which provides voltage isolation.



Fig.5. 36: TI XDS560 emulator, composed of a card to install on the host computer (PCI interface), a JTAG cable and an emulation controller pod

Capabilities of scan-based emulators include source-level debugging, i.e., the possibility to see the assembly instructions being executed and to access variables and memory locations either by name or by address.

Capabilities such as writing to the standard output are available. As an example, the *printf()* function allows printing DSP information on the debugger GUI; the reader should, however, be aware that this operation can be extremely time-consuming, and optimized functions (such as *LOG_printf()* for TI DSPs) should be preferred.

Another common capability supported by emulation technology is the breakpoint. A breakpoint freezes the DSP and allows the developer to examine DSP registers, to plot the content of memory regions, and to dump data to files. Two main forms of breakpoint exist, namely software and hardware. A software breakpoint replaces the instruction at the breakpoint location with one creating an exception condition that transfers the DSP control to the emulation controller. A hardware breakpoint is implemented by using custom hardware on the target device. The hardware logic can for instance monitor a set of addresses on the DSP and stop the DSP code execution when a code fetch is performed at a specific location. Breakpoints can be triggered also by a combination of addresses, data, and system status. This allows DSP developers to analyse the system when for instance it hangs, i.e., when the DSP program counter branches into an invalid memory address. Intermittent bugs can also be tracked down.

It is important to underline that the debugging capabilities provided by emulators allow mostly ‘stop-mode debugging’, in that the DSP is halted and information is sent to the host computer at that moment. This debugging technique is invasive and allows the developer to get isolated, although very useful, snapshots of the halted application. To improve the situation, DSP tool vendors have developed a more advanced debugging technology that allows real-time data exchange between target and host. This technique is described next.

Real-time techniques

Over the last ten years, DSP vendors have developed techniques for a real-time data exchange between target and host without stopping the DSP and with minimal interference on the DSP run. This provides a continuous visibility into the way the target operates. Additionally, it allows the simulation of data input to the target.

ADI’s real-time communication technology is called Background Telemetry Channel (BTC) . This is based upon a shared group of registers accessible by the DSP and by the host for reading and writing. It is currently supported on Blackfin and ADSP-219s DSPs only.

TI’s real-time communication technology is called Real Time Data eXchange (RTDX). Its main software and hardware components are shown in Fig. 37. A collection of channels, through which data is exchanged, are created between target and host. These channels are unidirectional and data can be sent across them asynchronously. TI provides two libraries, the RTDX target library and the RTDX host library, that have to be linked to target and host applications, respectively. As an example, the target application sends data to the host by calling functions in the

RTDX target library. These functions buffer the data to be sent and then give the program flow control back to the calling program; after this, the RTDX target library transmits the buffered data to the host without interfering in the target application. RTDX is also supported when running inside a DSP simulator; to that end, the DSP developer should link the target application with the RTDX simulator target library corresponding to the chosen target. On the host side, data can be visualized and treated from applications interfacing with the RTDX host library. On Windows platforms a Microsoft Component Object Module (COM) interface is available, allowing clients such as VisualBasic, VisualC++, Excel, LabView, MATLAB and others.

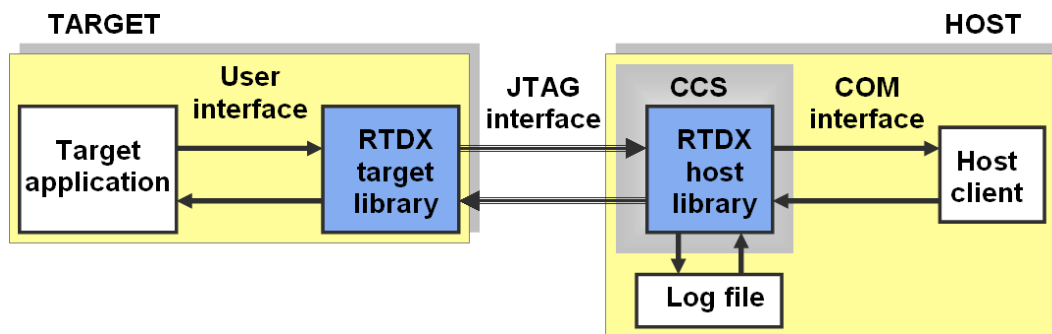


Fig.5. 37: TI's RTDX main components. The picture is courtesy of TI

In 1998 TI implemented the original RTDX technology, which runs on XDS510-class emulators. A high-speed RTDX version was developed later that relies on additional DSP chip hardware features and on improved emulators, namely the XDS560 class. These emulators make use of two non-JTAG pins in the standard TI JTAG connector to increase RTDX bandwidth. They are also backwards compatible and can support standard RTDX, thus allowing higher data transfer speed. The high-speed RTDX is supported in TI's highest performance DSPs, such as the TMS320C55x, TMS320C621x, TMS320C671x and TMS320C64x families. Table 11 shows the data transfer speeds available with different combinations of RTDX and emulators. RTDX offers a bandwidth of 10 to 20 kbytes/s, thus enabling real-time debugging of applications such as CD audio and audio telephony. The high-speed RTDX with XDS560-class emulators provides a data transfer speed higher than 2 Mbytes/s, thus allowing real-time visibility into applications such as ADSL, hard-disk drives and videoconferencing .

Table 11: Data transfer speed as a function of the emulator type for TI's RTDX

Emulation type	Speed
RTDX + XDS510	10–20 kbytes/s
RTDX + USB (ex: ‘C6713 DSK board)	10–20 kbytes/s
RTDX + XDS560	≤ 130 kbytes/s
High speed RTDX + XDS560	> 2 Mbytes/s

8 Code analysis and optimization

Most DSP applications are subject to real-time constraints and stress the available CPU and memory resources. As a consequence, code optimization might be required to satisfy the application requirements.

DSP code can be optimized according to one or more parameters such as execution speed, memory usage, input/output bandwidth, or power consumption. Different parts of the code can be optimized according to different parameters. A trade-off between code size and higher performance exists, hence some functions can be optimized for execution speed and others for code size.

Code development environments typically allow defining several code configuration releases, each characterized by different optimization levels. Figure 38 shows the project configurations available in TI Code Composer Studio. The ‘Release’ configuration comprises the higher optimization level, while the ‘Debug’ configuration enables debug features, which typically increase code size. Finally, the user can specify a ‘Custom’ configuration where user-selectable debug and optimization features are enabled.

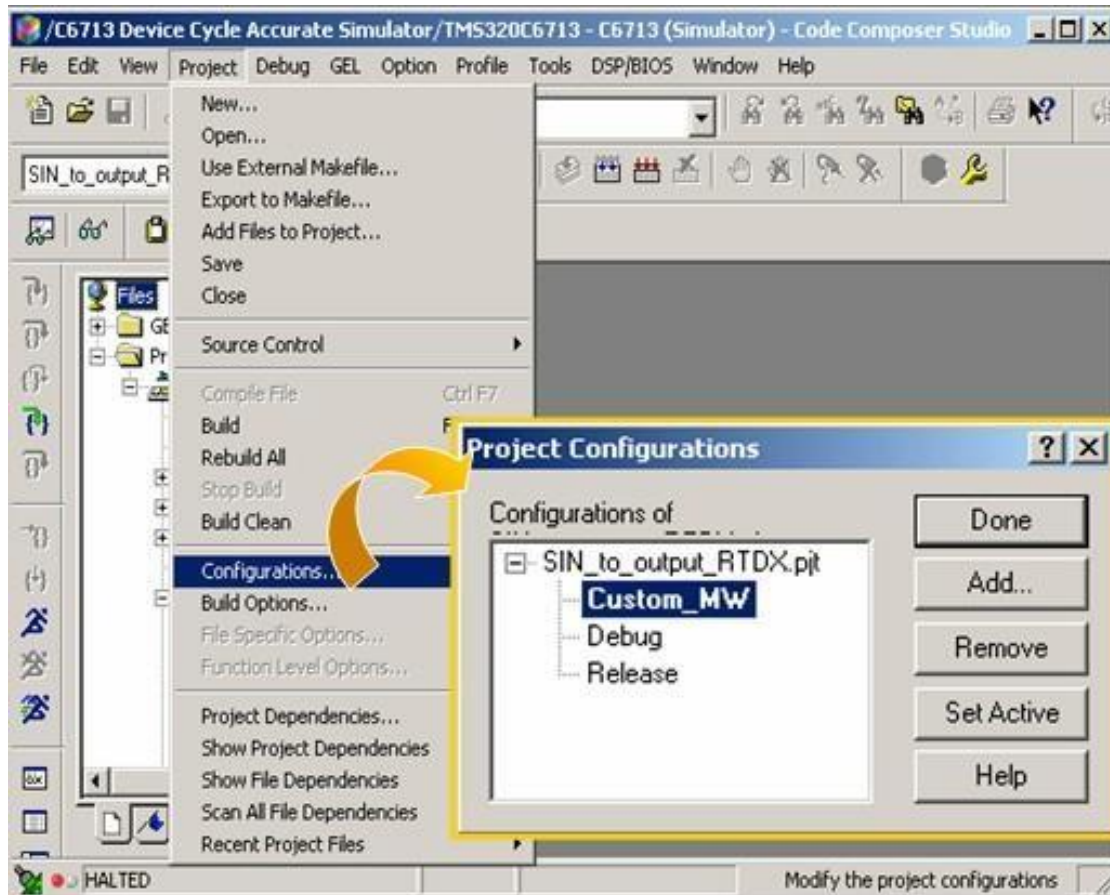


Fig. 5.38: Choice of the DSP code project configurations in TI Code Composer

It is important to underline that debug and optimization phases are different and often conflicting. In fact, an optimized code does not include any debug information; additionally, the optimizer can re-arrange the code so that the assembly code is not an immediate transposition of the original source code. The reader is strongly encouraged to avoid using the debug and the optimize options together; it is recommended instead to first debug the code and only then to enable the optimization.

Switching the code optimizer ON

Compilers are nowadays very efficient at code optimization, allowing DSP developers to write higher level code instead of assembly. To do this, compilers must be highly customized, i.e., tightly targeted to the hardware architecture the code will be running upon. However, current trends in software engineering include retargeting compilers to DSP specialized architectures .

As previously mentioned, many kinds of optimization can be required. An example is execution speed vs. executable size. Figure 5.39 shows how the user can select one or the other in the Code Composer Studio development environment.

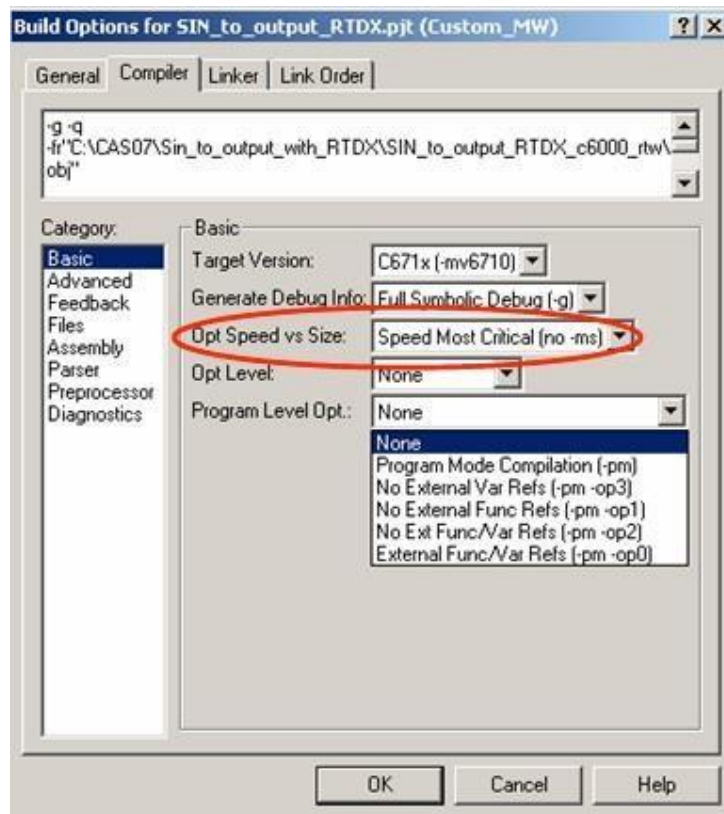


Fig. 5.39: Choice of optimization levels in TI Code Composer. The plot highlights execution speed vs. executable code size.

The reader should be aware that the optimizer can rearrange the code, hence the code must be written in a proper way. Failing this, the actions generated by the optimized code might be different from those desired and implemented by a non-optimized code. Figure 5.40 shows two code snippets where the value assumed by the memory location pointed to by *ctrl* determines the *while()* loop behaviour. In particular, the DSP exits the *while()* loop if the *ctrl* content takes the value 0xFF; the *ctrl* content can be modified by another processor or execution thread. Both code snippets will perform equally in case of non-optimization. However, in case of optimization the left-hand side code will not evaluate the *ctrl* content at every *while()* iteration, hence the DSP will remain forever in the loop. On the right-hand side snippet, the *volatile* keyword disables memory optimization locally, thus forcing the DSP to re-evaluate the *ctrl* content value at every *while()* loop iteration. This guarantees the desired behaviour even when the code is optimized. The number of *volatile* variables should be restricted to situations where they are strictly needed, as they limit the compiler's optimization.

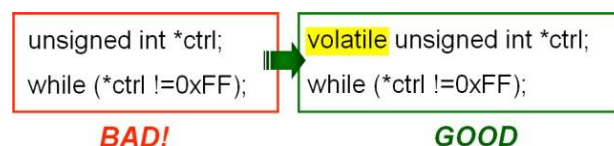


Fig.5.40: Example of good and bad programming techniques. The left-hand side code would likely result in a programming misbehaviour.

The recommended code development flow is to first write high-level code, such as C or C++. This code can then be debugged and optimized, to comply with the specified performance. In case the code runs still slower than desired, the time-critical areas can be re-coded in linear assembly. If the code is still too slow, then the DSP developer should turn to hand-optimized assembly code. Figure 5.41 shows a comparison of the different programming techniques, with corresponding execution efficiency and development effort.

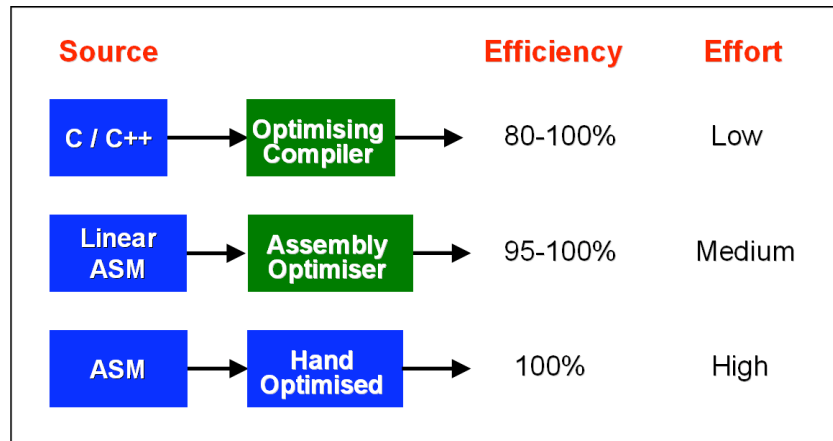


Fig. 5.41: Comparison of programming techniques with corresponding execution efficiency and estimated development effort. The picture is courtesy of TI.

Analysis tools

DSP code often follows the 20/80 rule, which states that 20% of the software in an application uses 80% of the processing time. As a consequence, the DSP developer should first concentrate efforts on determining where to optimize, i.e., on understanding where the execution cycles are mostly spent.

The best way to determine the parts of the code to optimize is to profile the application. Over the last ten years DSP development environments have considerably enlarged their offer of analysis tools. Some examples of TI's CCS analysis and tuning tools are:

- Compiler consultant.** It analyses the DSP code and provides recommendations on how to optimize its performance. This includes compiler optimization switches and programs, thus allowing a quick improvement in performance. Figure 42 shows how to enable the compiler consultant in CCS.
- Cache tune.** It provides a graphical visualization of memory reference patterns and memory accesses, thus allowing the identification of problem areas related for instance to memory access conflicts.
- Code size tune.** It profiles the application, collects data on individual functions and determines the best combinations of compiler options to optimize the trade-off between code size and execution speed.

d) **Analysis ToolKit (ATK).** It runs with DSP simulators only and allows one to analyse the DSP code robustness and efficiency. The DSP developer should not only know when to optimize, as described previously: he/she should also know when to stop. In fact, there is a law of diminishing returns in the code analysis and optimization process. It is thus important to take advantage of the improvements that come with relatively little effort, and leave as a last resort those that are difficult to implement and provide low- yield.

Finally, it is strongly recommended to make only one optimization change at the same time; this will allow the developer to exactly map the optimization to its result.

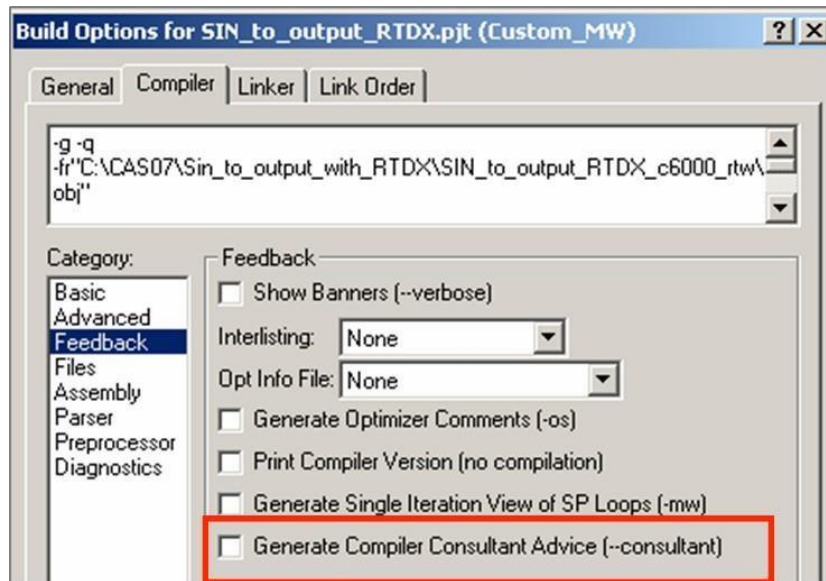


Fig.5. 42: How the ‘Compiler Consultant Advice’ can be enabled in TI’s CCS Development Environment

Programming optimization guidelines

This Section includes some general programming guidelines for writing efficient code; these guidelines are applicable to the vast majority of DSP compilers. DSP developers should, however, refer to the manuals of the development tools they are using for more precise information on how to write efficient code.

Finally, it is strongly recommended to make only one optimization change at the same time; this will allow the developer to exactly map the optimization to its result.

- **Guideline 1: Use the DMA when possible and allocate data in memory wisely**

DMA controllers (see Sub-section 3.2.3) must be used whenever possible so as to free the DSP core for other tasks. The linker (see Sub-section 6.4.3) should be used for allocating data in memory so as to guarantee an efficient memory access. Additionally, DSP developers should avoid placing arrays at the beginning or at the very end of memory blocks, as this creates problems for software pipelining. Software pipelining is a technique that optimizes tight loops by fetching a data set while the DSP is processing the previous one. However, the last iteration of a loop

would attempt to fetch data outside the memory space, in case an array is placed on the memory edge. Compilers must then execute the last iteration in a slower way ('loop epilogue') to prevent this address error from happening. Some compilers, such as the ADI Blackfin one, make available compiler options to specify that it is safe to load additional elements at the end of the array.

– **Guideline 2: Choose variable data types carefully**

DSP developers should know the internal architecture of the DSP they are working on, so as to be able to use native data type DSPs as opposed to emulated ones, whenever possible. In fact, operations on native data types are implemented by hardware, hence are typically very fast. On the contrary, operations on emulated data types are carried out by software functions, hence are slower and use more resources. An example of emulated data type is the double floating point format on ADI's TigerSHARC floating point DSPs. Another example is the floating point format on ADI's Blackfin family of fixed-point processors. In these DSPs the floating point format is implemented by software functions that use fixed-point multiply and ALU logic. In this last case a faster version of the same functions is available with non-IEEE-compliant data formats, i.e., formats implementing a 'relaxed' IEEE version so as to reduce the computational complexity. Table 12 shows a, execution times comparison of IEEE-compliant and non-IEEE-compliant functions in ADI's Blackfin BF533.

Table 12: Execution time of IEEE-compliant vs. non-IEEE-compliant library functions for ADI's Blackfin BF533

operation	fast-ft [cycles]	IEEE-ft [cycles]	ratio
multiply	93	241	0.4
add	127	264	0.5
subtract	161	329	0.5
divide	256	945	0.3
pow	8158	17037	0.5

– **Guideline 3: Functions and function calls**

Functions such as *max()*, *min()* and *abs()* are often single-cycle instruction and should be used whenever possible instead of manually coding them. Figure 43 shows on the right-hand side the *max()* function and on the left-hand side a manual implementation of the same function. The advantage in terms of code efficiency of using a single-cycle *max()* function is evident. Often more complex functions such as FFT, IIR, or FIR filters are available in vendor-provided libraries. The reader is strongly encouraged to use them, as their optimization is carried out at algorithm level.

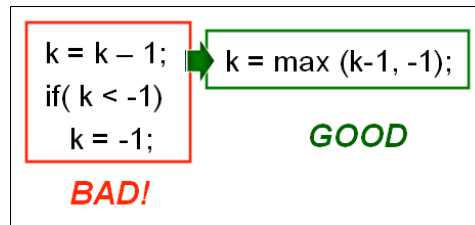


Fig. 5.43: Example of good and bad programming techniques

As few parameters as possible should be passed to a function. In fact, parameters are typically passed to functions by using registers. However, the stack is used when no more registers are available, thus slowing down the code execution considerably.

– **Guideline 4: Avoid data aliasing**

Aliasing occurs when multiple variables point to the same data. For example, two buffers overlap, two pointers point to the same software object or global variables used in a loop. This situation can disrupt optimization, as the compiler will analyse the code to determine when aliasing could occur. If it cannot work out if two or more pointers point to independent addresses or not, the compiler will typically behave conservatively, hence avoid optimization so as to preserve the program correctness.

– **Guideline 5: Write loops code carefully**

Loops are found very often in DSP algorithms, hence their coding can strongly influence the program execution performance. Function calls and control statements should be avoided inside a loop, so as to prevent pipeline flushes (see Sub-section 3.3.2). Figure 5.44 shows an example of good and bad programming techniques referred to control statements inside a *for()* loop: by moving the conditional expression *if...else* outside the loop, as shown in the right-hand side code snippet, one can reduce the number of times the conditional expression is executed.

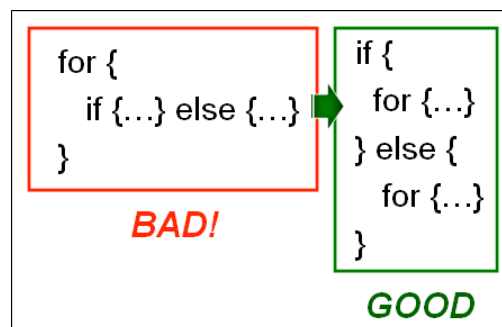


Fig. 5.44: Example of good and bad programming techniques

Loop code should be kept small, so as to fit entirely into the DSP cache memory and to allow a local repeat optimization. In case of many nested loops, the reader should be aware that compilers typically focus their optimization efforts on the inner loop. As a consequence, pulling operations from the outer to the inner loop can

improve performance. Finally, it is recommended to use *int* or *unsigned int* data types for loop counters instead of the larger-sized data type *long*.

– **Guideline 6: Be aware of time-consuming operations**

There are operations, such as the division, that do not have hardware support for a single-cycle implementation. They are instead implemented by functions implementing iterative approximations algorithms, such as the Newton–Raphson. The DSP developer should be aware of that and try to avoid them when possible. For example, the division by a power-of-two operation can be converted to the easier right shift on unsigned variables. DSP manufacturers often provide indications on techniques to implement the division instruction more efficiently .

Other operations are available from library functions. Examples are *sine*, *cosine* and *atan* functions, very often needed in the accelerator sector for the implementation of rotation matrixes and for rectangular to polar coordinates conversion. If needed, custom implementations can be developed to obtain a favourable ratio between precision and execution time. Table 13 shows the comparison of different implementations of the same functions; in particular, the second column shows a custom implementation used in CERN’s LEIR accelerator. In this implementation, the *sine*, *cosine* and *atan* calculation algorithm has been implemented by a polynomial expansion of the seventh order instead of the usual Taylor series expansion.

Table 13: Execution times vs. different implementations of the same functions

Function	Execution time [μs]		
	CERN single-precision implementation	VisualDSP++ single-precision implementation	VisualDSP++ double-precision implementation
cosine	0.25 (for a sine/cosine couple)	0.59	5.5
sine		0.59	5.3
atan	0.4125	1.4	5.6

Guideline 7: Be aware that DSP software heavily influences power optimization

DSP software can have a significant impact on power consumption: a software-efficient in terms of the required processor cycles to carry out a task is often also energy efficient. Software should be written so as to minimize the number of accesses to off-chip memory; in fact, the power required to access off-chip memory is usually much higher than that used for accessing on-chip memory. Power consumption can be further optimized in DSPs that support selective disabling of unused functional blocks (e.g., on-chip memories, peripherals, clocks, etc.). These

‘power down modes’ are available in ADI DSPs (such as Blackfin) as well as in TI DSPs (such as the TMS320C6xxx family). Making a good use of these modes and features can be difficult; however, APIs and specific software modules are available to help. An example is TI’s DSP/BIOS Power Manager (PWRM) module, providing a kernel-level API that interfaces directly to the DSP hardware by writing and reading configuration registers. Figure 5.45 shows how this module is integrated in a generic application architecture for DSPs belonging to TI’s TMS320C55x family.

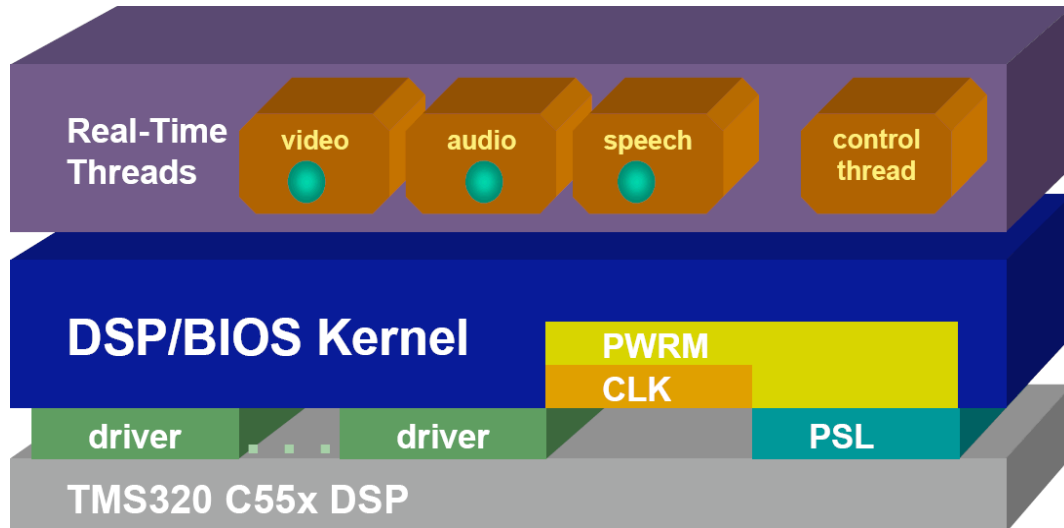


Fig. 5.45: TI’s DSP/BIOS Power Manager (PWRM) module in a general system architecture. Picture courtesy of Texas Instruments.

9 Real-time design flow: system design

This section deals with some aspects of digital systems design, particularly with software and hardware architectures. Here the assumption is that the system to be designed is based upon one or more DSPs. The reader should, however, be aware that in the accelerator sector there are currently three main real-time digital signal processing actors: DSPs, FPGAs and front-end computers. The front-end computers are typically implemented by embedded General Purpose Processors (GPPs) running a RTOS. Nowadays, the increase in clock speed allows GPPs to carry out real-time data processing and slow control actions; in addition, there is a tendency to integrate DSP hardware features and specialized instructions into GPPs, yielding GPP hybrids. One example of such processors is given in Fig.5.46, showing the PowerPC with Motorola’s AltiVec extension. The AltiVec 128-bit SIMD unit adds up to 16 operations per clock cycle, in parallel to the Integer and Floating Point units, and 162 instructions to the existing RISC architecture.

Fundamental choices to make when designing a new digital system are which digital signal processing actors should be used and how tasks should be shared between them. This choice requires detailed and up-to-date knowledge of the different possibilities.

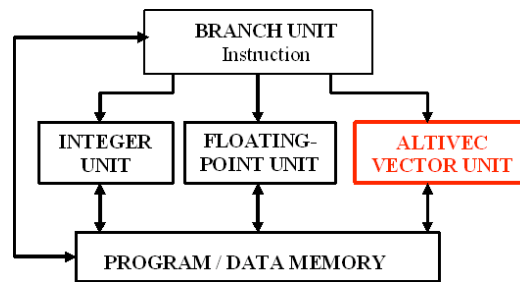


Fig. 5.46: AltiVec technology: SIMD expansion to Motorola PowerPC (G4 family)

In industry the choice of the DSP to use is often based on the ‘4P’ law: Performance, Power consumption, Price and Peripherals. In the accelerator sector, the power consumption factor is typically negligible. Other factors are instead decisive, such as standardization in the laboratory, synergies with existing systems, and possibilities of evolution to cover different machines. Last but not least, one should consider the existing know-how in terms of tools and of hardware, which can be directly translated to a shorter development time.

In this section three design aspects are considered and briefly discussed, namely:

- a) DSP choice in Sections 9.1 and 9.2.
- b) System architecture in Sections 9.3 to 9.6.
- c) DSP code design in Sections 9.7 and 9.8.

DSP choice: fixed vs. floating-point DSPs

The reader can find a basic description of fixed- and floating-point number formats in Section 3.4.

Fixed-point formats can typically be implemented in hardware in a cheaper way, with better energy efficiency and less silicon than floating-point formats. Very often fixed-point DSPs support a clock faster than floating-point DSPs; as an example, TI fixed-point DSPs can currently be clocked up to 1.2 GHz, while TI floating-point DSPs are clocked up to 300 MHz.

Floating-point formats are easier to use since the DSP programmer can mostly avoid carrying out number scaling prior to each arithmetic operation. In addition, floating-point numbers provide a higher dynamic range, which can be essential when dealing with large data sets and with data sets whose range cannot be easily predicted. The reader should be aware that floating-point numbers are not equispaced, i.e., the gap between adjacent numbers depends on their magnitude: large numbers have large gaps between them, and small numbers have small gaps. As an example, the gap between adjacent numbers is higher than 10 for numbers of the order of $2 \cdot 10^8$. Additionally, the error due to truncation and rounding during the floating-point number scaling inside the DSP depends on the number magnitude,

too. This introduces a noise floor modulation that can be detrimental for high-quality audio signal processing. For this reason, high-quality audio has been traditionally implemented by using fixed-point numbers. However, a migration of high-fidelity audio from fixed- to floating- point implementation is currently taking place, so as to benefit from the greater accuracy provided by floating point numbers.

The choice between fixed- and floating-point DSP is not always easy and depends on factors such as power consumption, price, and application type. As an example, military radars need floating- point implementations as they rely in finding the maximal absolute value of the cross-correlation between the sent signal and the received echo. This is expressed as the integral of a function against an exponential; the integral can be calculated by using FFT techniques that benefit from the floating point dynamic range and resolution. For radar systems, the power consumption is not a major issue. The floating-point DSP additional cost is not an issue either, as the processor represents only a fraction of the global system cost. Another example is the mobile TV. The core of this application is the decoder, which can be MPEG-2, MPEG-4 or JPEG-2000. The decoding algorithms are designed to be performed in fixed-point; the greater precision of floating-point numbers is not useful as the algorithms are in general bit-exact.

It should be underlined that many digital signal processing algorithms are often specified and designed with floating-point numbers, but are subsequently implemented in fixed-point architectures so as to satisfy cost and power efficiency requirements.

Finally, as mentioned in Section 8.3, some fixed-point DSPs make available floating-point numbers and operations by emulating them in software (hence they are slower than in a native floating-point DSP).

The fact that floating-point numbers are not equispaced has already been mentioned. The reader might be interested in looking at some consequences of this with an example from the LHC beam control implementation. Figure 47 shows a zoom onto the beam loops part of the LHC beam control. The ‘Low-level Loops Processor’ is a board including a TigerSHARC DSP and an FPGA. The FPGA carries out some simple pre-processing and data interfacing, while the DSP implements the low-level loops. In particular, the DSP calculates the frequency to be sent to the cavities from the beam phase, radial position, synchrotron frequency, and programmed frequency; these calculations are carried out in floating-point format. The frequency to be sent to the cavities, referred to as F_{out} in Fig.5.47, must be expressed as an unsigned, 16-bit integer. The desired frequency range to represent is 10 kHz, hence the needed resolution is 0.15 Hz. The LHC cavities work at a frequency of about 400.78 MHz but the spacing of a single-precision, floating-point number with magnitude of approximately $400 \cdot 10^6$ is higher than one. To avoid the use of slower, double-precision, floating-point format, the beam loop calculations are carried out as offset from 400.7819 MHz.

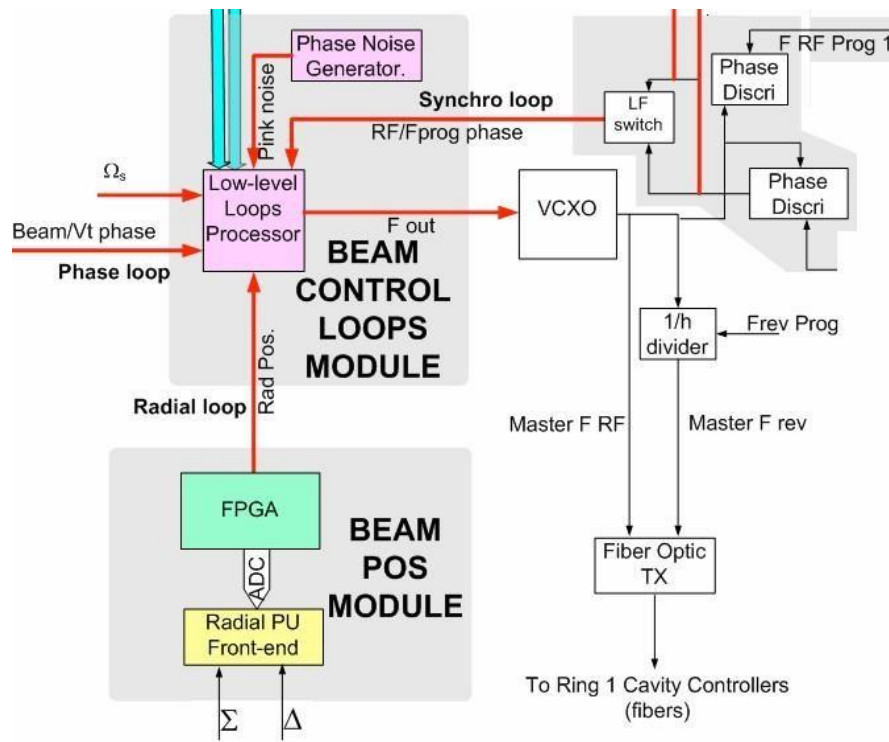


Fig. 5.47: LHC beam control – zoom onto the beam loops part

DSP choice: benchmarking

Benchmarking a DSP means evaluating it on a number of different metrics. Table 14 gives an example of some common metrics and corresponding units.

Table 14: Examples of DSP performance metric sets and corresponding units

Metric	Unit
Maximum clock frequency	MHz
Execution speed	Millions of Instructions Per Second (MIPS) Millions of Operations Per Second (MOPS) Number of Multiply-and-Accumulate operations per second
Memory bandwidth	Mbytes/s
Memory latency	Number of clock cycles
Power consumption	W or W/MIPS

Good benchmarks are important for comparing DSPs and allow critical business or technical decisions to be made. It should be underlined that benchmarks can be misleading, thus should be considered in a critical way. As an example, the maximum clock frequency of a DSP can be different from the instruction rates; hence this parameter might not be indicative of the real DSP processing power. Another example is the execution speed measured in MIPS: this metric is easy to

measure but it is often too simple to provide useful information about how a processor would perform in a real application. VLIW architectures issue and execute multiple instructions per instruction cycle. These processors usually use simpler instructions that perform less work than the instructions typical of conventional DSPs. As a consequence, MIPS comparison between VLIW-based DSP and conventional ones is misleading.

More complex benchmarks are available; examples are the execution of application tasks (typically called kernel functions) such as IIR filters, FIR filters, or FFTs. Kernel function benchmarking is typically more reliable and is available from DSP manufactures as well as from independent companies.

It is difficult to provide general guidelines to measure the efficacy of DSP benchmarks for DSP selection. Two general rules should be followed: first, the benchmark should perform the type of work the DSP will be expected to carry out in the targeted application. Second, the benchmark should implement the work in a way similar to what will be used in the targeted application.

System architecture: multiprocessor architectures

Multiprocessor architectures are those where two or more processors interact in real-time to carry out a task. Right from their early days, many DSP families have been designed to be compatible with multiprocessing operation; an example is the TI TMS320C40 family. Multiprocessing architectures are particularly suited for applications with a high degree of parallelism, such as voice processing. In fact, processing ten voice channels can be carried out by implementing a one-voice channel, then repeating the process ten times in parallel. Applications requiring multiprocessing computing to support processing of greater data flow include high-end audio treatment, 3D graphics acceleration, and wireless communication infrastructure, just to mention a few of.

There is another reason to move to multiprocessing systems. For many years developers have been taking advantage of the steady progress in DSP performance. New and faster processors would be available, allowing more powerful applications to be implemented sometimes only for the price of porting existing code to the new DSP. This favourable situation was driven by the steady progress of the semiconductor industry that managed to pack more transistors into smaller packages and at higher clock frequencies. The increased performance was enabled by architectural innovations, such as VLIW, as well as added resources, such as on-chip memories. In recent years, however, progress in single-chip performance has been slowing down. The semiconductor industry has turned to parallelism to increase performance. This is true not only for the DSP sector, but in general for business computing. One example is the Intel Core Duo processors, including two execution cores in a single processor, now the established platform for personal computers and laptops.

Finally, the reader should be aware that development environments have evolved to provide support for debugging multiple processor cores connected in the same JTAG path. An example is TI's Parallel Debug Manager [68], which is integrated within the Code Composer Studio IDE.

Of the many possible multiprocessing forms, the multi-DSP and multi-core approaches are considered and discussed in Sub-sections 9.3.1 and 9.3.2, respectively.

Multi-DSP architecture

Many separate DSP chips can co-operate to carry out a task providing an increased system performance. One advantage of this approach is the scalability, i.e., the ability to tune the system performance and cost to the required functionality and processing performance by varying the number of DSP chips used.

The reader should, however, be aware that multi-DSP designs involve different constraints than single-processing systems. Three key aspects must be taken into account.

- a) Tasks must be partitioned between processors. As an example, a single processor can handle a task from start to end; as an alternative, a processor can perform only a portion of the task, then pass the intermediate results to another processor.
- b) Resources such as memory and bus access must be shared between processors so as to avoid bottlenecks. As an example, additional memory may be added to store intermediate results. Organizing memory into segments or banks allows simultaneous memory accesses without contentions if different banks are accessed.
- c) A robust and fast inter-DSP communication means must be established. If the communication is too complex or takes too much time, the advantage of a multiprocessing can be lost.

Two examples of multi-DSP architectures based on ADI DSPs are shown in Fig.5.48.

On the left-hand side (plot a) the point-to-point architecture is depicted, based upon ADI linkport interconnect cable standard. Point-to-point interconnect provides a direct connection between processor elements. This is particularly useful when large blocks of intermediate results must be passed between two DSPs without involving the others. Read/write transactions to external memory are saved by passing data directly between two DSPs, thus allowing the use of slower memory devices. Additionally, the point-to-point interconnect can be used to scale a design: additional links can be added to have more DSPs interacting. This can be done either directly or by bridging across several links.

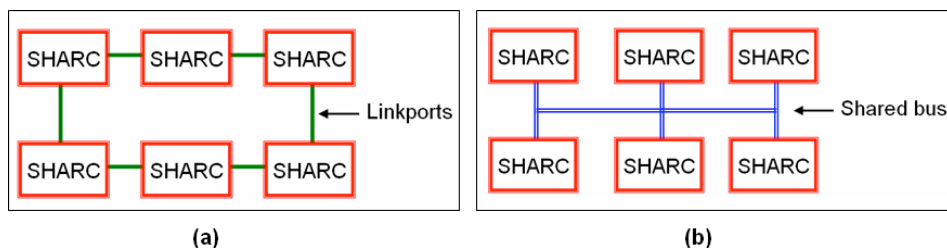


Fig. 5.48: Examples of multi-DSP configurations. (a) point-to-point, linkport-based and (b) cluster bus

On the right-hand side (plot b) the cluster bus architecture is depicted. A cluster bus maps internal memory resources, such as registers and processor memory addresses, directly onto the bus. This allows DSP code developers to exchange data between DSPs using addresses as if each processor possessed the memory for storing the data. Memory arbitration is managed by the bus master; this avoids the need for complex memory or data sharing schemes managed by software or by RTOS. The map includes also a common broadcast space for messages that need to reach all DSPs. As an example, Fig. 49 shows the TigerSHARC global memory map. The multiprocessing space maps the internal memory space of each TigerSHARC processor in the cluster into any other TigerSHARC processor. Each TigerSHARC processor in the cluster is identified by its ID; valid processor ID values are 0 to 7.

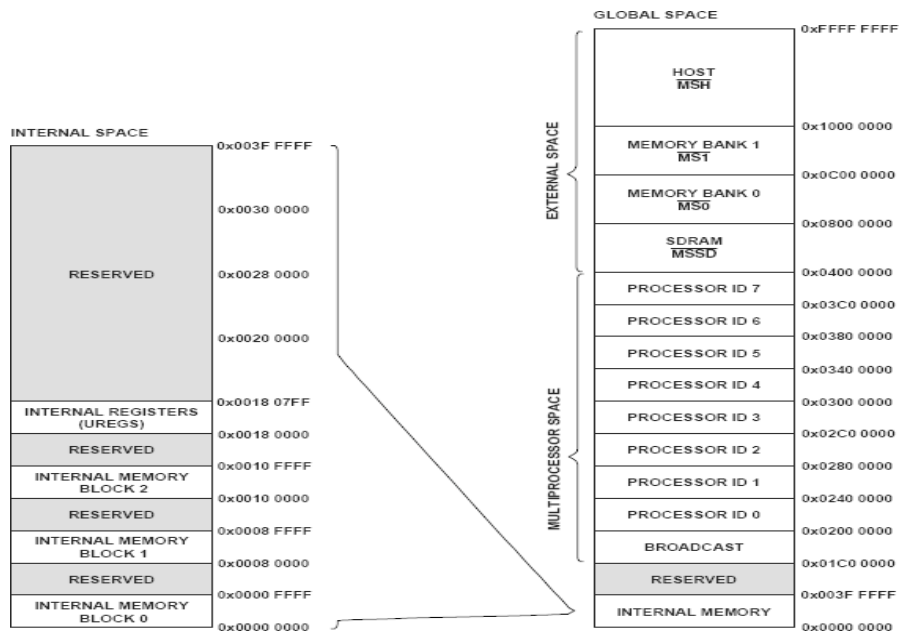


Fig.5.49: ADI TigerSHARC TS101 global memory map. Picture courtesy of Analog Devices

The reader should be aware that the two above-mentioned architectures, namely point-to-point and cluster bus, are not mutually exclusive; on the contrary, they can both be used in the same application as complementary solutions.

Multi-core architecture

In a multi-core architecture, multiple cores are integrated into the same chip. This provides a considerable increase of the performance per chip, even if the performance per core only increases slowly. Additionally, the power efficiency of multi-core implementations is much better than in traditional single-core implementations. This approach is a convenient alternative to DSP farms.

As the performance required by DSP systems keeps increasing, it is nowadays essential for DSP developers to devise a processing extension strategy. Multi-core architectures can provide it, in that the DSP performance is boosted without switching to a different core architecture. This has the advantage that applications can be based upon multiple instances of an already-proven core, rather than be adapted to new architectures.

DSP multi-core architectures have been commercialized only recently; however, the DSP market has relied for many years on co-processor technology (also called on-chip accelerators) to boost performance. Figure 5.50 shows the evolution of DSP architecture. From the initial single-core architecture (a), the single-core plus co-processor architecture soon emerged. The co-processor often runs at the same frequency as the DSP, therefore ‘doubling’ the performance for the targeted application. Co-processor examples are Turbo and Viterbi decoders for communication applications. Finally, over the last few years the multi-core architecture shown in plot (c) has emerged, which still includes co-processors.

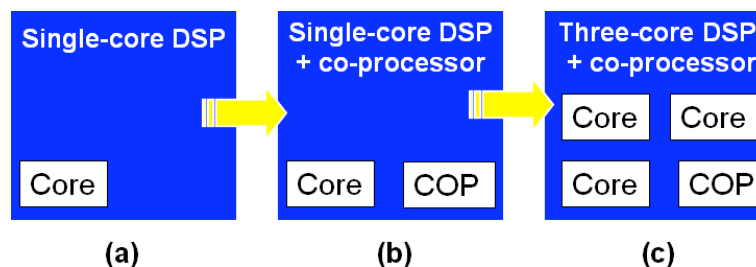


Fig.5. 50: Multi-core and co-processor DSP architectures evolution.
Single-core DSP (a), single- core DSP plus coprocessor (b) and multi-core DSP plus coprocessor (c).

Multi-core architectures are available in two different flavours, namely Symmetric Multi- Processing (SMP) and Asymmetric Multi-Processing (AMP). SMP architectures include two or more processors which are similar (or identical), connected thorough a high-speed path and sharing some peripherals as well as memory space. AMP architectures combine two different processors, typically a microcontroller and a DSP, into a hybrid architecture.

It is possible to use a multi-core device in different ways. The different cores can operate independently or they can cooperate for task completion. An efficient inter-core communication may be needed in both cases, but it is particularly

important when two or more cores work together to complete a task. As for the multi-DSP case discussed in Sub-section 9.3.1, it is important to decide how to share resources to avoid bottlenecks and deadlocks, and to ensure that one core does not corrupt the operation of another core. The resources must be partitioned not only at board level, like in the single-core case, but at device level, too, thus adding increase complexity. Figure 5.51 shows an example of multi-core bus and memory hierarchy architecture. L1 memories are typically dedicated to their own core as non-partitioned between cores, as it may be inefficient to access them from other cores. The L2 memory is an internal memory shared between the different cores, as opposed to the single-core case where the L2 memory can be either internal or external. The multi-core architecture must make sure that each core can access the L2 memory and the arbitration must be such that cores are not locked out from accessing this resource.

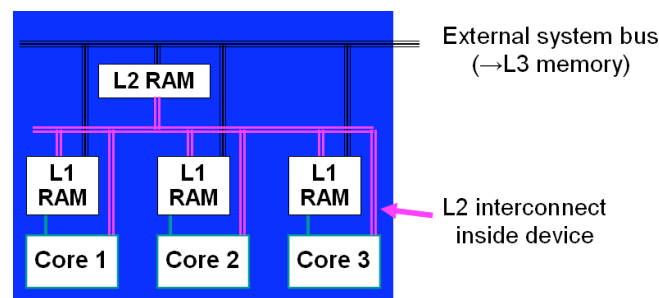


Fig. 5.51: Multi-core bus and memory hierarchy example

Figure 5.52 shows the TMS320C5421 DSP as an example of a multi-core, SMP DSP. The TMS320C5421 DSP is composed of two C54x DSP cores and is targeted at carrier-class voice and video end equipment. The cores are 16-bit fixed-point and the chip is provided with an integrated VITERBI accelerator. Four internal buses and dual address generators enable multiple program and data fetches and reduce memory bottlenecks.

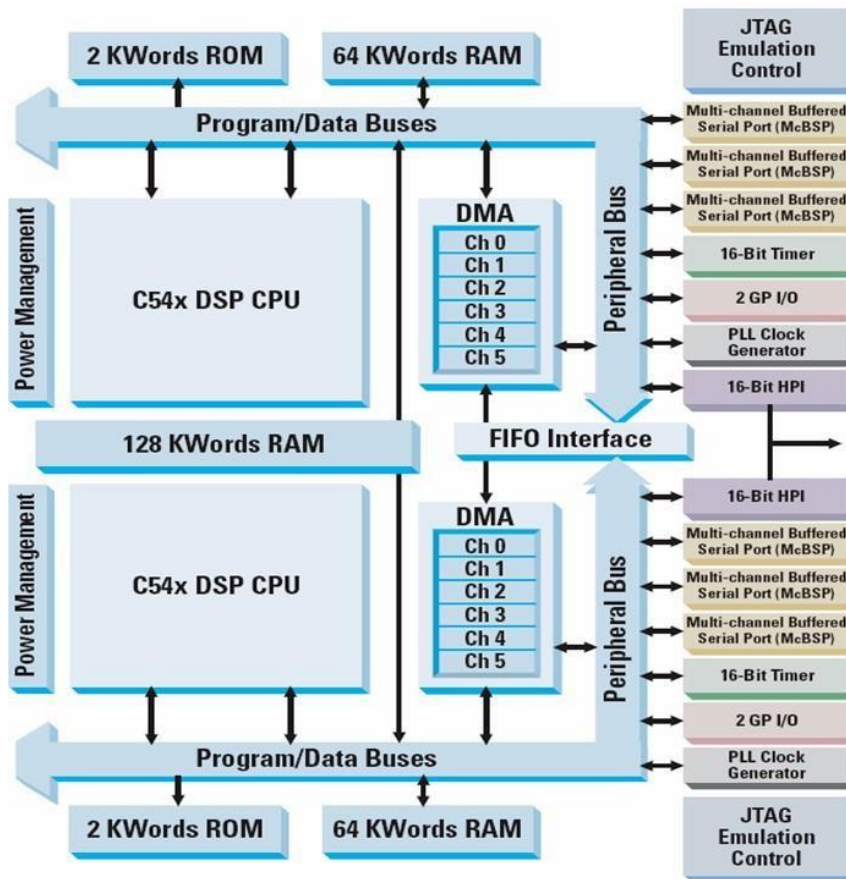


Fig. 5.52 TMS320C5421 multi-core DSP as an SMP example. Picture courtesy of Texas Instruments, DSP selection guide 2007, p. 48.

The programming of multi-core system is generally more complex than in the single-core case. In particular, the reader should be aware that multi-core code must follow the re-entrance rules, to make sure that one core's processing does not corrupt the data used by another core's processing. This approach is followed by single-core processors, too, when implementing multi-tasking operations.

System architecture: radiation effects

Single-Event Upset (SEU) events are alterations in the behaviour of electronic circuits induced by radiation. These alterations can be transient disruptions, such as changes of logic states, or permanent IC alterations.

Techniques to mitigate these effects in ICs can be carried out at different levels, namely:

- At device level, for instance by adding extra-doping layers to limit the substrate charge collection.
- At circuit level, for instance by adding decoupling resistors, diodes, or transistors in the SRAM hardening.
- At system level, with Error Detection And Correction (EDAC) circuitry or with algorithm-based fault tolerance. An example of the latter approach is the Triple Module Redundancy (TMR) algorithm or the newer Weighted

Checksum Code (WCC). The reader should, however, be aware that there are limitations to what these algorithms can achieve. For instance, the WCC method applied to floating-point systems may fail, as round off errors may not be distinguished from functional errors caused by radiation.

Neither ADI nor TI currently provide any radiation-hard DSP. Third-party companies have developed and marketed radiation-hard versions of ADI and TI DSPs. An example is Space Micro Inc., based in San Diego, California. This company devised the Proton 200k single-board computer based upon a TI C67xx DSP, fitted with EDAC circuitry and with a total dose tolerance higher than 100 krad.

The LHC power supply controllers are examples of mitigation techniques applied to DSP. They are based upon non-radiation-hard TI C32 DSPs and micro controllers. The memory is protected with EDAC circuitry and by avoiding the use of DSP internal memory, which cannot be protected. A watchdog system restarts the power supply controller in the event of a crash. Radiation tests have been carried out to check that the devised protection strategy is sufficient for normal operation.

System architecture: interfaces

An essential step in the digital system design is to clearly define the interfaces between the different parts of the system. Figure 53 shows some typical building blocks that can be found in a digital system, namely DSP(s), FPGA(s), daughtercards, Master VME, machine timings, and signals.

The DSP system designer must define the interfaces between DSP(s) and the other building blocks. It is strongly recommended to avoid hard-coding in the DSP code the address of memory regions shared with other processing elements. On the contrary, the linker should be used to allocate appropriately the software structures in the DSP memory, as mentioned in Sub-section 6.4.3. Additionally, the DSP developer should create data access libraries, so as to obtain a modular hence more easily upgradeable approach.

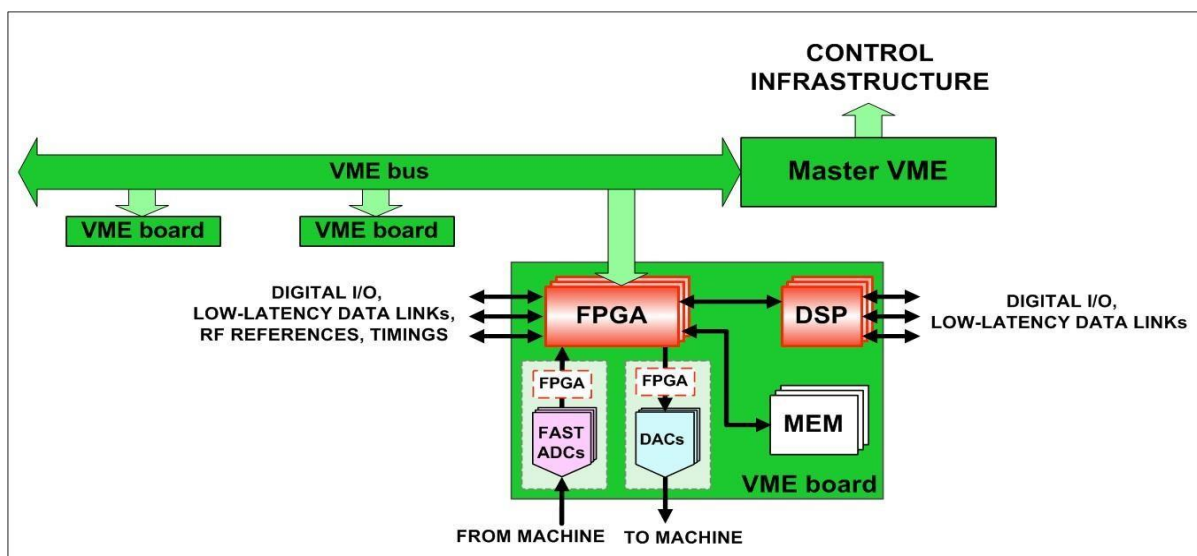
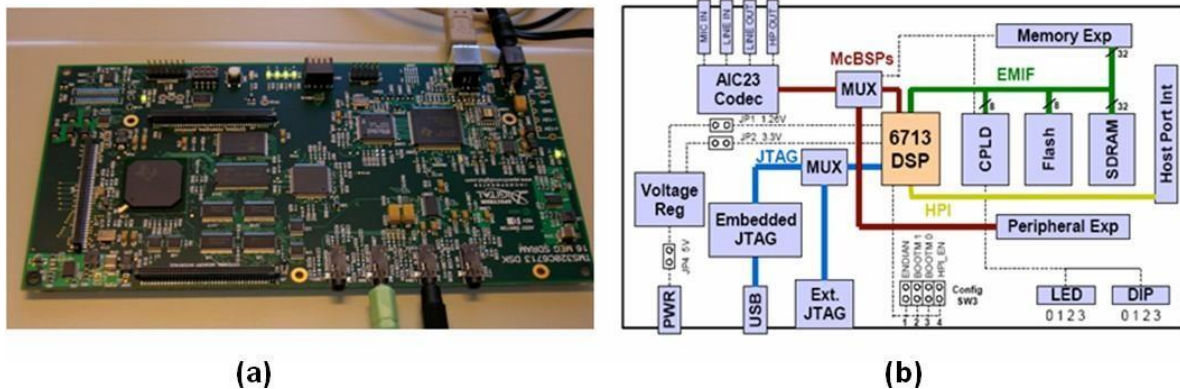


Fig. 5.53: Typical digital system building blocks and corresponding interfaces

System architecture: general recommendations

Basically all DSP chips present some anomalies on their expected behaviour. This is especially true for the first release of DSP chips, as discovered anomalies are typically solved on later releases. A list of all anomalies for a certain DSP release, which includes also workarounds when possible, is normally available on the



manufacturer's website. The reader is strongly encouraged to look at those lists, so as to avoid being delayed by already-known problems.

Fig.5. 54: TI C6713 DSK evaluation board – picture (a) and board layout (b)

A DSP system designer can gain useful software and hardware experience by using evaluation boards in the early stages of system design. Evaluation boards are typically provided by manufacturers for the most representative DSPs. They are relatively inexpensive and are typically fitted with ADCs and DACs; they come with the standard development environment and JTAG interface, too. The DSP designer can use them to solve technical uncertainties and sometimes can even modify them to quickly build a system prototype. Figure 5.54 shows TI's C6713 DSK evaluation board (a) and corresponding board layout (b); this evaluation board was that used in the DSP laboratory companion of the lectures summarized in this paper.

DSP code design: interrupt-driven vs. RTOS-based systems.

A fundamental choice that the DSP code developer must make is how to trigger the different DSP actions. The two main possibilities are via a RTOS or via interrupts.

An overview of RTOS is given in Section 6.3. RTOS can define different threads, each one performing a specific action, as well as the corresponding threads' priorities and triggers. RTOS-based systems have typically a clean design and many built-in checks. The disadvantage of using RTOS is a potentially slower response to external events (interrupts) and the use of DSP resources (such as some hardware timings and interrupts) for the internal RTOS functioning.

Interrupt-driven systems associate actions directly to interrupts. The resource use is therefore optimized. An example of interrupt-driven system is CERN's LEIR LLRF . Figure 5.55 shows some of its software components: a background task triggered every millisecond carries out housekeeping actions, while a control task triggered every 12.5 μ s implements the beam control actions. Driving a system through interrupts is very efficient with a limited number of interrupts. For a high

number of interrupts, the system can become very complex and its behaviour not easily predictable.

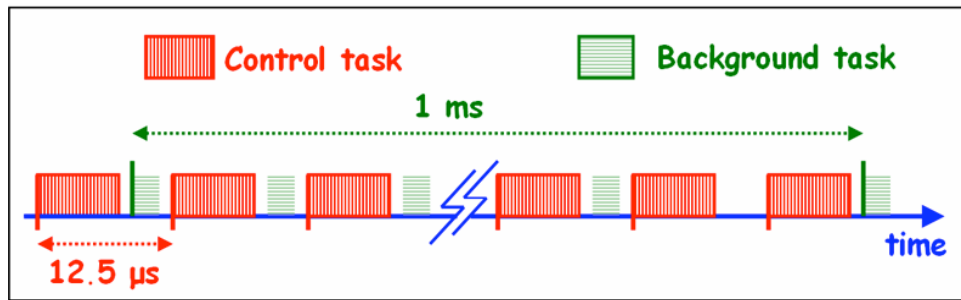


Fig. 5.55: Example of an interrupt-driven system. Control and background tasks are triggered by interrupts and are shown in red and green, respectively.

DSP code design: good practice

A vast amount of literature is available on code design good practice. Here just a few points are underlined, which are particularly relevant to embedded systems.

First, digital systems must not turn into tightly sealed black boxes. It is essential that designers embed many diagnostics buffers in the DSP code, so as to prevent this from happening. The diagnostics buffers could take many forms, such as post-mortem, circular or linear buffers. They might be user-configurable and must be visible from the application program.

Second, every new DSP code release should be characterized by a version number, visible from the application level. The functionality and interface map corresponding to a certain version number should be clearly documented, so as to avoid painful misunderstandings between the many system layers. Source code control is essential for managing complex software development projects, as large projects require more than one DSP code developer working on many source files. Source code control tools make it possible to keep track of the changes made to individual source files and prevent files from being accessed by more than one person at a time. DSP software development environments can often support many source control providers. Code Composer Studio, for example, supports any source control provider that implements the Microsoft SCC Interface.

Finally, DSP developers should also add checks on the execution duration, to make sure the code does not overrun. This is particularly important for interrupt-driven systems (mentioned in Section 9.7), where one or more interrupts may be missed if the actions corresponding to an interrupt are not finished by the time the next interrupt occurs. As an example, the minimum and maximum

number of clock cycles needed for executing a piece of code can be constantly measured and monitored by the user at high level. All DSPs provide means to measure the number of clock cycles required to execute a certain amount of code; the number of clock cycles can then be easily converted into absolute time. Figure 56 shows a possible implementation on ADI SHARC DSPS of the execution duration of a code called ‘critical action’. SHARC processors have a set of registers called *emuc1k* and *emuc1k2* which make up a 64-bit counter. This counter is unconditionally incremented during every instruction cycle on the DSP and is not affected by factors such as cache-misses or wait-states. Every time *emuc1k* wraps to zero, *emuc1k2* is incremented by one. By determining the difference in the *emuc1k* value between before and after the critical action, the DSP developer can determine the number of clock cycles — hence the time — to execute the code.

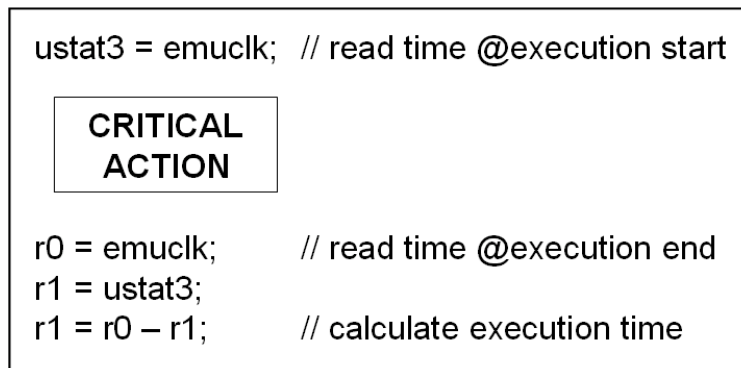


Fig.5. 56: Execution duration measurement with *emuc1k* registers in the ADI SHARC DSP

The system integration is one of the final parts in the system development process. This phase is extremely important as it can determine the success or the failure of a whole system. In fact, a system which is well integrated can become operational, while a system only partially integrated will often remain a ‘machine development’ tool, easily forgotten.

During the system integration phase, the system is commissioned with respect to data exchange with the control infrastructure and the application program(s). Two or more groups, such as Instrumentation, Controls and Operation, can be involved in this effort, depending on the laboratory’s organization. As a consequence, a coordination and specification work is required.

Good system integration practices will depend on the laboratory’s organization as well as on the system architecture. There are, however, some guidelines that can be applied to most cases.

- **Guideline 1: Work in parallel**

All software layers needed in a system should be planned in parallel. Waiting until the low-level part is completed before starting with the specification and/or with the development of the other layers may result in unacceptable delays.

– **Guideline 2: About interfaces**

Section 9.5 summarized the many interfaces that can exist in a system. For a successful system integration it is essential that the interfaces are specified clearly, are agreed upon with all different parties and are fully documented. Recipes on how to set up different software components of the system or on how to interact with them can be really useful and speed up considerably system development as well as debugging. It is recommended that all documents be kept updated and stored on servers accessible by all parties involved. Remember: good fences make good neighbours.

– **Guideline 3: Always include checks on the DSP inputs validity**

The validity of all control inputs to the DSP should be checked. Alarms or warnings should be raised if a control value falls outside the allowed range. This mechanism will help the system integration part and could even prevent serious malfunctioning from happening.

– **Guideline 4: Add spare parameters**

It is strongly recommended to map spare parameters between the DSP and application program; they should have different formats for maximum flexibility. These spare parameters allow adding debugging features or making some small update without modifications to the intermediate software layers.

– **Guideline 5: Code release and validation**

The source code (and if possible the corresponding executable, too) should be saved together with a description of its features and implemented interfaces. This will allow going back to previous working releases in case of problems. Procedure and data sets should also be defined for code validation.

Existing chip examples were often given and referenced to technical manuals or application notes. Examples of DSP use in existing accelerator systems were also given whenever possible.

The DSP field is of course very large and more information, as well as hands-on practice, is required to become proficient in it. However, the author hopes that this document and the references herein can be useful starting points for anyone wishing to work with DSPs.

- The Spartan-3 family architecture consists of five fundamental programmable functional elements:
Configurable Logic Blocks (CLBs) containing RAM-based Look-Up Tables (LUTs) to implement logic and storage elements that can be used as flip-flops or latches. CLBs can be programmed to perform a wide variety of logical functions as well as to store data.
- Input/Output Blocks (IOBs) controlling the flow of data between the I/O pins and the internal logic of the device.
- 18-Kbit dual-port RAM blocks providing data storage.
- Multiplier blocks which accept two 18-bit binary numbers as inputs and calculate the product.
- Digital Clock Manager (DCM) blocks provide self-calibrating, fully digital solutions for distributing, delaying, multiplying, dividing, and phase shifting clock signals.

These elements are organized as shown in Figure 5.57. A ring of IOBs surrounds a regular array of CLBs. The XC3S400 device contains 896 CLBs, 288kbit embedded RAM, 16 dedicated multipliers and 4 DCMs.

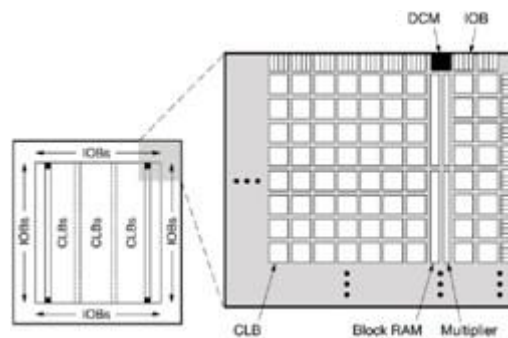


Fig.5.57 Spartan-3 family architecture

The main logic resource for implementing synchronous as well as combinatorial circuits is the Configurable Logic Block (CLB). Each CLB comprises four interconnected slices, as shown in Figure 5.58. These slices are grouped in pairs. Each pair is organized as a column with an independent fast carry chain. The carry chain supports implementing arithmetic functions such as addition.

All four slices have the following elements in common: two logic function generators (known as Look-Up Tables), two flip-flops, wide-function multiplexers, carry logic, and auxiliary arithmetic gates. The RAM-based Look-Up Table (LUT) is the main resource for implementing logic functions.

Each of the two LUTs in a slice have four logic inputs and a single output.

This permits any four-variable Boolean logic operation to be programmed into them. Wide-function multiplexers can be used to effectively combine

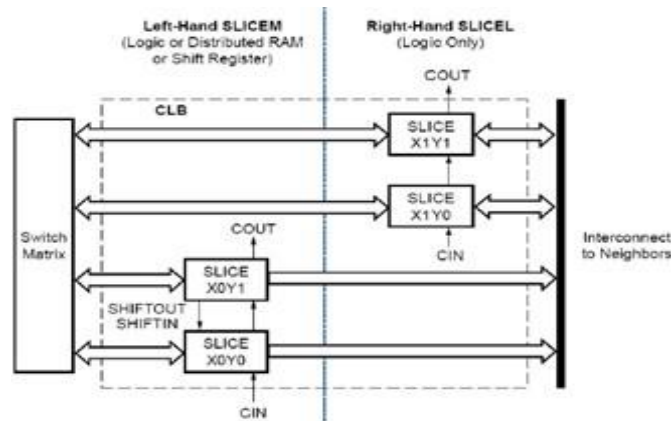


Fig. 5.58. Configurable logic block

LUTs within the same CLB or across different CLBs, making logic functions with many more input variables possible. Switch matrix (see Figure 2) allows programmable access into local and global routing resources. Clock signals are distributed by dedicated low- capacitance, low skew network well suited to carrying high-frequency signals.

Spartan-3 FPGAs are programmed by loading configuration data into static memory cells that control all functional elements and routing resources. Before powering on the FPGA, configuration data is stored externally in a PROM or some other nonvolatile medium either on or off the board. After applying power, the configuration data is written to the FPGA using one of the available modes, e.g. JTAG mode.

Memec Spartan-3 LC Development Board and P160 Analog Module

Memec Spartan-3 LC Development Board

The Spartan-3 LC Development Kit provides an easy-to-use evaluation platform for developing designs and applications based on the Xilinx Spartan-3 FPGA family. The development board utilizes the 400K-gate Xilinx Spartan- 3 device (XC3S400-4PQ208CES). The board includes a 50 MHz clock, a user clock socket, 29 user I/O header pins, an RS-232 port, a USB 2.0 slave port, LEDs, switches, and additional user support circuits. The P160 Analog Module containing A/D and D/A converters is connected to the main board by dedicated 160 pin socket.

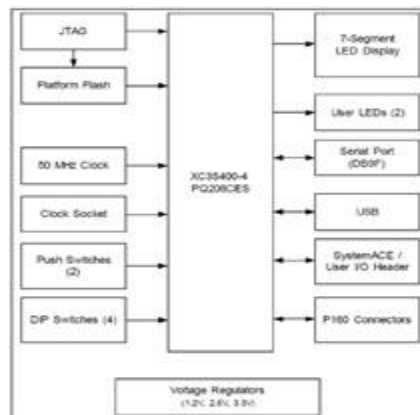


Fig.5.60. Spartan-3 LC block diagram

A simplified block diagram of the Spartan-3 LC development board is shown in Figure 3. Instructions for interfacing all included components are available in Memec Spartan-3 LC User's Guide. In digital signal processing purposes, the most emphasis shall be put on using ADC and DAC components.

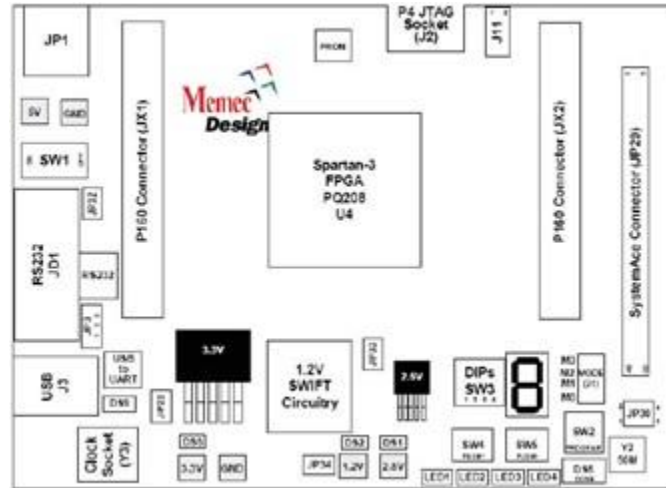


Fig.5.61. Spartan-3 LC Development board

A JTAG connector (J2, see Figure 4) provides interface to the board's JTAG chain. This chain can be used to program the on-board ISP PROM and configure the Spartan-3 FPGA. The JTAG chain consists of an XCF02S Platform Flash PROM followed by an XC3S400 FPGA. The XCF02S Platform Flash In System Programmable (ISP) PROM allows designers to store an FPGA configuration in nonvolatile memory. The JTAG port on the Platform Flash device is used to program the PROM with an .mcs file created by iMPACT in the Xilinx ISE software environment. Once the Flash has been programmed, the user can configure the Spartan-3 device by setting the Configuration Mode to Master Serial Mode (Jumper J1, see Figures 4 and 5). The Spartan-3 device configuration is initiated during power-up or by asserting the PROGAMn signal (by pressing the SW2 switch). The FPGA can be also configured directly through JTAG chain, without using of PROM. JTAG chain is connected into PC computer through dedicated Parallel cable.

P160 Analog Module

supporting analog outputs. Both channels are identical. The Texas Instruments ADS807 12-bit, 53Msps A/D converters are used to convert incoming analog signals into 12-bit data for the FPGA located on the baseboard. Analog outputs can be generated using the two DAC902 12-bit, 165Msps D/A converters from Texas Instruments. Gain and filtering is provided on the D/A outputs. Control of the ADCs and DACs is handled by the FPGA through the P160 digital interface.

D/A Converters

The FPGA interfaces to the DACs through 12-bit registers, which add a clock cycle delay between data out from the FPGA and the DAC analog outputs. Two independent data channels, one channel for each DAC, are driven from the FPGA. The DACs interface signals are:

- bits of input data. Output voltage values corresponding to input binary values are shown below.

111111111111	3.5V
100000000000	2.5V
000000000000	1.5V

As can be easily seen, the DAC output is normally DC coupled for a 2 V peak-to-peak (Vp-p) signal centered at 2.5 V.

- Two clock signals: DAC Clock (CLK) and Register Clock (CLK2), rising edge active. The CLK2 signal latches the digital DAC data from the FPGA into the register. The CLK signal latches the output data from the register into the DAC. On the falling edge of the CLK signal, the DAC output changes to the newly latched value.
- Reference Select (RefSel) control signal, which makes possible to disable internal reference voltage source and to use external reference input (Low = Internal, High = External Reference).
- Power Down (PD) control signal (Low = Normal, High = Power Down Mode).

A/D converters

Texas Instruments ADS807 converters provide 12-bit resolution at up to 53 Msps. The digital data out of the A/Ds is latched into external buffers and then passed to the FPGA through the P160 interface. The range of the input voltage is dependent on the Full Scale Select control signal to the

A/D. Before conversion the input signal is AC coupled, biased to 2.5 volts for unipolar operation, and buffered through the op amp. The ADCs interface signals are:

- bits of output data (binary range "000000000000" to "111111111111").
- Full Scale Select control signal (FsSel). Setting this signal to a logic high allows a 1.5 Vp-p input to the board. Setting the Full Scale Select to low, selects a 1 Vp-p input range to the board (i.e. -0:5V voltage corresponds to "000000000000" value and +0:5V corresponds to "111111111111" value).
- Reference Select (RefSel) control signal (Low = Internal Reference, High = External Reference).
- Output Enable (OE) control signal (Low = Output Enabled, High = Tri-Stated outputs).
- Convert clock (CLK) signal. The ADS807 samples the input signal on the rising edge of the CLK input. Output data values are valid at the outputs 6 clock cycles later, after the rising edge of the clock.

Audio Signal Processing

Our sense of hearing provides us rich information about our environment with respect to the locations and characteristics of sound producing objects. For example, we can effortlessly assimilate the sounds of birds twittering outside the window and traffic moving in the distance while following the lyrics of a song over the radio sung with multi-instrument accompaniment. The human auditory system is able to process the complex sound mixture reaching our ears and form high-level abstractions of the environment by the analysis and grouping of measured sensory inputs. The process of achieving the segregation and identification of sources from the received composite acoustic signal is known as auditory scene analysis. It is easy to imagine that the machine realization of this functionality (sound source separation and classification) would be very useful in applications such as speech recognition in noise, automatic music transcription and multimedia data search and retrieval. In all cases the audio signal must be processed based on signal models, which may be drawn from sound production as well as sound perception and cognition. While production models are an integral part of speech processing systems, general audio processing is still limited to rather basic signal models due to the diverse and wide-ranging nature of audio signals. Important technological applications of digital audio signal processing are audio data compression, synthesis of audio effects and audio classification. While audio compression has been the most prominent application of digital audio processing in the recent past, the burgeoning importance of multimedia content management is seeing growing applications of signal processing in audio segmentation and classification. Audio classification is a part of the larger problem of audiovisual data handling with important applications in digital libraries, professional media production, education, entertainment and surveillance. Speech and speaker recognition can be considered classic problems in audio retrieval and have received decades of research attention. On the other hand, the rapidly growing archives of digital music on the internet are now drawing attention to wider problems of nonlinear browsing and retrieval using more natural ways of interacting with multimedia data including, most prominently, music. Since audio records (unlike images) can be listened to only sequentially, good indexing is valuable for effective retrieval. Listening to audio clips can actually help to navigate audiovisual material more easily than the viewing of video scenes. Audio classification is also useful as a front end to audio compression systems where the efficiency of coding and transmission is facilitated by matching the compression method to the audio type, as for example, speech or music.

Audio Signal Characteristics

Audible sound arises from pressure variations in the air falling on the ear drum. The human auditory system is responsive to sounds in the frequency range of 20 Hz to 20 kHz as long as the intensity lies above the frequency dependent “threshold of hearing”. The audible intensity range is approximately 120 dB which represents the range between the rustle of leaves and boom of an aircraft take-off. Figure 1 displays the human auditory field in the frequency-intensity plane. The sound captured by a microphone is a time waveform of the air pressure variation at the location of the microphone in the sound field. A digital audio signal is obtained by the suitable sampling and quantization of the electrical output of the microphone. Although any sampling frequency above 40 kHz would be adequate to capture the full range of audible frequencies, a widely used sampling rate is 44,100 Hz, which arose from the historical need to synchronize audio with video data. “CD quality” refers to 44.1 kHz sampled audio digitized to 16-bit word length. Sound signals can be very broadly categorized into environmental sounds, artificial sounds, speech and music. A large class of interesting sounds is timevarying in nature with information coded in the form of temporal sequences of atomic sound events. For example, speech can be viewed as a sequence of phones, and music as the evolving pattern of notes. An atomic sound event, or a single gestalt, can be a complex acoustical signal described by a specific set of temporal and spectral properties. Examples of atomic sound events include short sounds such as a door slam, and longer uniform texture sounds such as the constant patter of rain. The temporal properties of an audio event refer to the duration of the sound and any amplitude modulations including the rise and fall of the waveform amplitude envelope. The spectral properties of the sound relate to its frequency components and their relative strengths

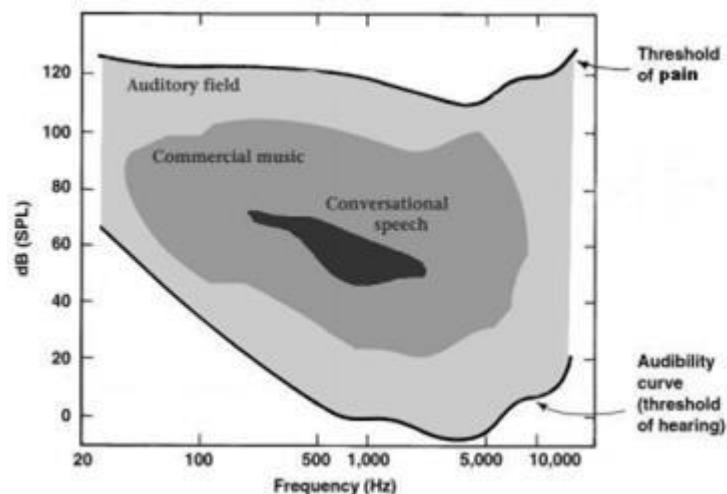


Fig.5.62. The auditory field in the frequency-intensity plane. The sound pressure level is measured in dB with respect to the standard reference pressure level of 20 micropascals.

Audio waveforms can be periodic or aperiodic. Except for the simple sinusoid, periodic audio waveforms are complex tones comprising of a fundamental frequency and a series of overtones or multiples of the fundamental frequency. The relative amplitudes and phases of the frequency components influence the sound “colour” or timbre. Aperiodic waveforms, on the other hand, can be made up of non-harmonically related sine tones or frequency shaped noise. In general, a sound can exhibit both tone-like and noise-like spectral properties and these influence its perceived quality. Speech is characterized by alternations of tonal and noisy regions with tone durations corresponding to vowel segments occurring at a more regular syllabic rate. Music, on the other hand, being a melodic sequence of notes is highly tonal for the most part with both fundamental frequency and duration varying over a wide range. Sound signals are basically physical stimuli that are processed by the auditory system to evoke psychological sensations in the brain. It is appropriate that the salient acoustical properties of a sound be the ones that are important to the human perception and recognition of the sound. Hearing perception has been studied since 1870, the time of Helmholtz. Sounds are described in terms of the perceptual attributes of pitch, loudness, subjective duration and timbre. The human auditory system is known to carry out the frequency analysis of sounds to feed the higher level cognitive functions. Each of the subjective sensations is correlated with more than one spectral property (e.g. tonal content) or temporal property (e.g. attack of a note struck on an instrument) of the sound. Since both spectral and temporal properties are relevant to the perception and cognition of sound, it is only appropriate to consider the representation of audio signals in terms of a joint description in time and frequency. While audio signals are non stationary by nature, audio signal analysis usually assumes that the signal properties change relatively slowly with time. Signal parameters, or features, are estimated from the analysis of short windowed segments of the signal, and the analysis is repeated at uniformly spaced intervals of time. The parameters so estimated generally represent the signal characteristics corresponding to the time center of the windowed segment. This method of estimating the parameters of a time-varying signal is known as “short-time analysis” and the parameters so obtained are referred to as the “short-time” parameters. Signal parameters may relate to an underlying signal model. Speech signals, for example, are approximated by the well-known source-filter model of speech production. The source-filter model is also applicable to the sound production mechanism of certain musical instruments where the source refers to a vibrating object, such as a string, and the filter to the resonating body of the instrument. Music due to its wide definition, however, is more generally modelled based on observed signal characteristics as the sum of elementary components such as continuous sinusoidal tracks, transients and noise.

Audio Signal Representations

The acoustic properties of sound events can be visualized in a time-frequency “image” of the acoustic signal so much so that the contributing sources can often be separated by applying gestalt grouping rules in the visual domain. Human auditory perception starts with the frequency analysis of the sound in the cochlea. The time-frequency representation of sound is therefore a natural starting point for machine-based segmentation and classification. In this section we review two important audio signal representations that help to visualize the spectro-temporal properties of sound, the spectrogram and an auditory representation. While the former is based on adapting the Fourier transform to time-varying signal analysis, the latter incorporates the knowledge of hearing perception to emphasize perceptually salient characteristics of the signal.

Spectrogram

The spectral analysis of an acoustical signal is obtained by its Fourier transform which produces a pair of real-valued functions of frequency, called the amplitude (or magnitude) spectrum and the phase spectrum. To track the time-varying characteristics of the signal, Fourier transform spectra of overlapping windowed segments are computed at short successive intervals. Time domain waveforms of real world signals perceived as similar sounding actually show a lot of variability due to the variable phase relations between frequency components. The short-time phase spectrum is not considered as perceptually significant as the corresponding magnitude or power spectrum and is omitted in the signal representation. From the running magnitude spectra, a graphic display of the time-frequency content of the signal, or spectrogram, is produced. Figure 1 shows the waveform of a typical music signal comprised of several distinct acoustical events as listed in Table 1. We note that some of the events overlap in time. The waveform gives an indication of the onset and the rate of decay of the amplitude envelope of the non-overlapping events. The spectrogram (computed with a 40 ms analysis window at intervals of 10 ms) provides a far more informative view of the signal. We observe uniformly spaced horizontal dark stripes indicative of the steady harmonic components of the piano notes. The frequency spacing of the harmonics is consistent with the relative pitches of the three piano notes. The piano notes’ higher harmonics are seen to decay fast while the low harmonics are more persistent even as the overall amplitude envelope decays. The percussive (low tom and cymbal crash) sounds are marked by a grainy and scattered spectral structure with a few weak inharmonic tones. The initial duration of the first piano strike is dominated by high frequency spectral content from the preceding cymbal crash as it decays. In the final portion of the spectrogram, we can now clearly detect the simultaneous presence of piano note and percussion sequence.

Table 2. A Description of the audio events corresponding to figure 5.62.

Time duration (seconds)	Nature of sound event
0.00 to 0.15	Low Tom (Percussive stroke)
0.15 to 1.4	Cymbal crash (percussive stroke)
1.4 to 2.4	Piano note (Low pitch)
2.4 to 3.4	Piano note (High pitch)
3.4 to 4.5	Piano note (Middle pitch) occurring simultaneously as the low tom and cymbal crash (from 0 to 1.4 sec)

The spectrogram by means of its time-frequency analysis displays the spectro-temporal properties of acoustic events that may overlap in time and frequency. The choice of the analysis window duration dictates the trade-off between the frequency resolution of steady-state content versus the time resolution of rapidly time-varying events or transients.

Audio Features for Classification

While the spectrogram and auditory signal representations discussed in the previous section are good for visualization of audio content, they have a high dimensionality which makes them unsuitable for direct application to classification. Ideally, we would like to extract low-dimensional features from these representations (or even directly from the acoustical signal) which retain only the important distinctive characteristics of the intended audio classes. Reduced-dimension, decorrelated spectral vectors obtained using a linear transformation of a spectrogram have been proposed in MPEG-7, the audiovisual content description standard.

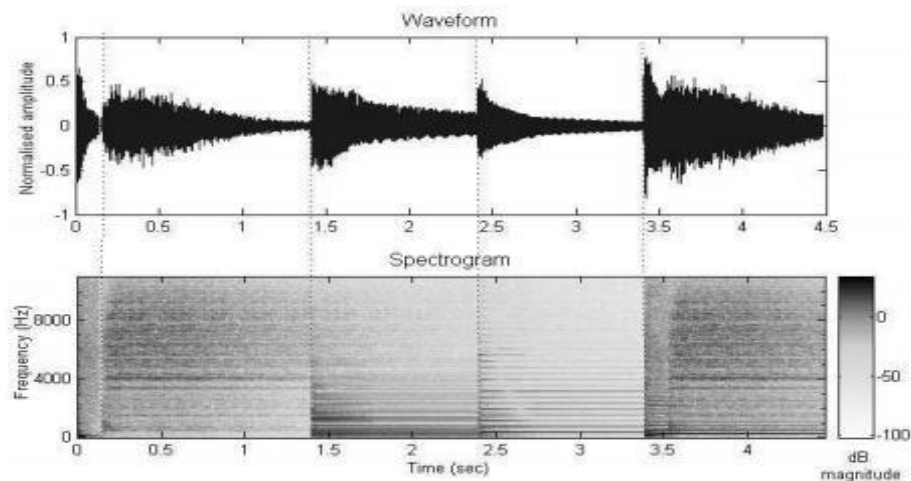


Fig 5.63. (a) Waveform, and (b) spectrogram of the audio segment described in table 2

The vertical dotted lines indicate the starting instants of new events. The spectrogram relative intensity scale appears at lower right.

A more popular approach to feature design is to use explicit knowledge about the salient signal characteristics either in terms of signal production or perception. The goal is to find features that are invariant to irrelevant transformations and have good discriminative power across the classes. Feature extraction, an important signal processing task, is the process of computing the numerical representation from the acoustical signal that can be used to characterize the audio segment. Classification algorithms typically use labeled training examples to partition the feature space into regions so that feature vectors falling in the same region come from the same class. A well-designed set of features for a given audio categorization task would make for robust classification with reasonable amounts of training data. Audio signal classification is a subset of the larger problem of auditory scene analysis. When the audio stream contains many different, but non-simultaneous, events from different classes, segmentation of the stream to separate class-specific events can be achieved by observing the transitions in feature values as expected at segment boundaries. However when signals from the different classes (sources) overlap in time, stream segregation is a considerably more difficult task. Research on audio classification over the years has given rise to a rich library of computational features which may be broadly categorized into physical features and perceptual features. Physical features are directly related to the measurable properties of the acoustical signal and are not linked with human perception. Perceptual features, on the other hand, relate to the subjective perception of the sound, and therefore must be computed using auditory models. Features may further be classified as static or dynamic features. Static features provide a snapshot of the characteristics of the audio signal at an instant in time as obtained from a short-time analysis of a data segment. The longer-term temporal variation of the static features is represented by the dynamic features and provides for improved classification. Figure 3 shows the structure of such a feature extraction framework. At the lowest level are the analysis frames, each representing windowed data of typical duration 10 ms to 40 ms. The windows overlap so that frame durations can be significantly smaller, usually corresponding to a frame rate of 100 frames per second. Each audio frame is processed to obtain one or more static features. The features may be a homogenous set, like spectral components, or a more heterogenous set. That is, the frame-level feature vector corresponds to a set of features extracted from a single windowed audio segment centered at the frame instant. Next the temporal evolution of frame-level features is observed across a larger segment known as a texture window to extract suitable dynamic features or feature summaries. It has been shown that the grouping of frames to form a texture window improves classification due to the availability of important statistical variation information. However increasing the texture window length beyond 1 sec does not improve classification any further. Texture

window durations typically range from 500 ms to 1 sec. This implies a latency or delay of up to 1 sec in the audio classification task.

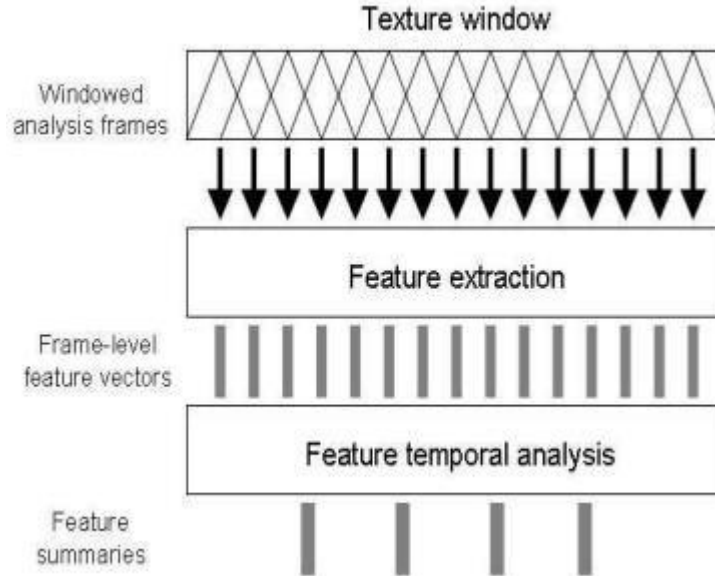


Fig.5.64. Audio feature extraction procedure Short-Time Energy

It is the mean squared value of the waveform values in the data frame and represents the temporal envelope of the signal. More than its actual magnitude, its variation over time can be a strong indicator of underlying signal content. It is computed as

$$E_r = \frac{1}{N} \sum_{n=1}^N |x_r(n)|^2 \quad \text{-----}>(2)$$

Band-Level Energy

It refers to the energy within a specified frequency region of the signal spectrum. It can be computed by the appropriately weighted summation of the power spectrum as given by

$$E_r = \frac{1}{N} \sum_{k=1}^{\frac{N}{T}} (X_r[k]W[k])^2 \quad \text{-----}>(3)$$

$W[k]$ is a weighting function with non-zero values over only a finite range of bin indices “k” corresponding to the frequency band of interest. Sudden transitions in the band-level energy indicate a change in the spectral energy distribution, or timbre, of the signal, and aid in audio segmentation. Generally log transformations of energy are used to improve the spread and represent (the perceptually more relevant) relative differences.

Spectral Centroid

It is the center of gravity of the magnitude spectrum. It is a gross indicator of spectral shape. The spectral centroid frequency location is high when the high frequency content is greater

$$C_r = \frac{\sum_{k=1}^{\frac{N}{2}} f[k] |X_r[k]|}{\sum_{k=1}^{\frac{N}{2}} |X_r[k]|} \text{----->(4)}$$

Since moving the major energy concentration of a signal towards higher frequencies makes it sound brighter, the spectral centroid has a strong correlation to the subjective sensation of brightness of a sound.

Spectral Roll-off

It is another common descriptor of gross spectral shape. The roll-off is given by

$$R_r = f[K] \text{----->(5)}$$

where K is the largest bin that fulfills

$$\sum_{k=1}^K |X_r[k]| \leq 0.85 \sum_{k=1}^{\frac{N}{2}} |X_r[k]| \text{-----> (6)}$$

That is, the roll-off is the frequency below which 85% of accumulated spectral magnitude is concentrated. Like the centroid, it takes on higher values for right-skewed spectra.

Spectral Flux

It is given by the frame-to-frame squared difference of the spectral magnitude vector summed across frequency as

$$F_r = \sum_{k=1}^{\frac{N}{2}} (|X_r[k]| - |X_{r-1}[k]|)^2 \text{-----> (7)}$$

It provides a measure of the local spectral rate of change. A high value of spectral flux indicates a sudden change in spectral magnitudes and therefore a possible segment boundary at the r th frame.

Audio Classification Systems

We review a few prominent examples of audio classification systems. Speech and music dominate multimedia applications and form the major classes of interest. As mentioned

earlier, the proper design of the feature set considering the intended audio categories is crucial to the classification task. Features are chosen based on the knowledge of the salient signal characteristics either in terms of production or perception. It is also possible to select features from a large set of possible features based on exhaustive comparative evaluations in classification experiments. Once the features are extracted, standard machine learning techniques to design the classifier. Widely used classifiers include statistical pattern recognition algorithms such as the k nearest neighbours, Gaussian classifier, Gaussian Mixture Model (GMM) classifiers and neural networks [14]. Much of the effort in designing a classifier is spent collecting and preparing the training data. The range of sounds in the training set should reflect the scope of the sound category. For example, car horn sounds would include a variety of car horns held continuously and also as short hits in quick succession. The model extraction algorithm adapts to the scope of the data and thus a narrower range of examples produces a more specialized classifier.

Speech-Music Discrimination

Speech-music discrimination is considered a particularly important task for intelligent multimedia information processing. Mixed speech/music audio streams, typical of entertainment audio, are partitioned into homogenous segments from which non-speech segments are separated. The separation would be useful for purposes such as automatic speech recognition and text alignment in soundtracks, or even simply to automatically search for specific content such as news reports among radio broadcast channels. Several studies have addressed the problem of robustly distinguishing speech from music based on features computed from the acoustic signals in a pattern recognition framework. Some of the efforts have applied well-known features from statistical speech recognition such as LSFs and MFCC based on the expectation that their potential for the accurate characterization of speech sounds would help distinguish speech from music. Taking the speech recognition approach further, Williams and Ellis use a hybrid connectionist-HMM speech recogniser to obtain the posterior probabilities of 50 phone classes from a temporal window of 100 ms of feature vectors. Viewing the recogniser as a system of highly tuned detectors for speech-like signal events, we see that the phone posterior probabilities will behave differently for speech and music signals. Various features summarizing the posterior phone probability array are shown to be suitable for the speech-music discrimination task.

A knowledge-based approach to feature selection was adopted by Scheirer and Slaney , who evaluated a set of 13 features in various trained-classifier paradigms. The training data, with about 20 minutes of audio corresponding to each category, was designed to represent as broad a class of signals as possible. Thus the speech data consisted of several male and female speakers in various background noise and channel conditions, and the music data contained various styles (pop, jazz, classical, country, etc.) including vocal music. Scheirer and Slaney evaluated several of the physical features, gether with

the corresponding feature variances over a one-second texture window. Prominent among the features used were the spectral shape measures and the 4 Hz modulation energy. Also included were the cepstral residual energy and, a new feature, the pulse metric. Feature variances were found to be particularly important in distinguishing music from speech. Speech is marked by strongly contrasting acoustic properties arising from the voiced and unvoiced phone classes. In contrast to unvoiced segments and speech pauses, voiced frames are of high energy and have predominantly low frequency content. This leads to large variations in ZCR, as well as in spectral shape measures such as centroid and roll-off, as voiced and unvoiced regions alternate within speech segments. The cepstral residual energy too takes on relatively high values for voiced regions due to the presence of pitch pulses. Further the spectral flux varies between near-zero values during steady vowel regions to high values during phone transitions while that for music is more steady. Speech segments also have a number of quiet or low energy frames which makes the short-time energy distribution across the segment more left-skewed for speech as compared to that for music. The pulse metric (or “rhythmicness”) feature is designed to detect music marked by strong beats (e.g. techno, rock). A strong beat leads to broadband rhythmic modulation in the signal as a whole. Rhythmicness is computed by observing the onsets in different frequency channels of the signal spectrum through bandpass filtered envelopes. There were no perceptual features in the evaluated feature set. The system performed well (with about 4% error rate), but not nearly as well as a human listener. Classifiers such as k-nearest neighbours and GMM were tested and performed similarly on the same set of features suggesting that the type of classifier and corresponding parameter settings was not crucial for the given topology of the feature space. Later work [18] noted that music dominated by vocals posed a problem to conventional speech-music discrimination due to its strong speech-like characteristics. For instance, MFCC and ZCR show no significant differences between speech and singing. Dynamic features prove more useful. The 4 Hz modulation rate, being related to the syllabic rate of normal speaking, does well but is not sufficient by itself. The coefficient of harmonicity together with its 4 Hz modulation energy better captures the strong voiced-unvoiced temporal variations of speech and helps to distinguish it from singing. Zhang and Kuo use the shape of the harmonic trajectories (“spectral peak tracks”) to distinguish singing from speech. Singing is marked by relatively long durations of continuous harmonic tracks with prominent ripples in the higher harmonics due to pitch modulations by the singer. In speech, harmonic tracks are steady or slowly sloping during the course of voiced segments, interrupted by unvoiced consonants and by silence. Speech utterances have language-specific basic intonation patterns or pitch movements for sentence clauses.

Audio Segmentation and Classification

Audiovisual data, such as movies or television broadcasts, are more easily navigated using the accompanying audio rather than by observing visual clips. Audio clips provide easily interpretable information on the nature of the associated scene such as for instance, explosions and shots during scenes of violence where the associated video itself may be fairly varied. Spoken dialogues can help to demarcate semantically similar material in the video while a continuous background music would help hold a group of seemingly disparate visual scenes together. Zhang and Kuo proposed a method for the automatic segmentation and annotation of audiovisual data based on audio content analysis. The audio record is assumed to comprise of the following nonsimultaneously occurring sound classes: silence, sounds with and without music background including the sub-categories of harmonic and inharmonic environmental sounds (e.g. touch tones, doorbell, footsteps, explosions). Abrupt changes in the short-time physical features of energy, zero-crossing rate and fundamental frequency are used to locate segment boundaries between the distinct sound classes. The same short-time features, combined with their temporal trajectories over longer texture windows, are subsequently used to identify the class of each segment. To improve the speech-music distinction, spectral peaks detected in each frame are linked to obtain continuous spectral peak tracks. While both speech and music are characterized by continuous harmonic tracks, those of speech correspond to lower fundamental frequencies and are shorter in duration due to the interruptions from the occurrence of unvoiced phones and silences. Wold et al in a pioneering work addressed the task of finding similar sounds in a database with a large variety of sounds coarsely categorized as musical instruments, machines, animals, speech and sounds in nature. The individual sounds ranged in duration from 1 to 15 seconds. Temporal trajectories of short-time perceptual features such as loudness, pitch and brightness were examined for sudden transitions to detect class boundaries and achieve the temporal segmentation of the audio into distinct classes. The classification itself was based on the salient perceptual features of each class. For instance, tones from the same instrument share the same quality of sound, or timbre. Therefore the similarity of such sounds must be judged by descriptors of temporal and spectral envelope while ignoring pitch, duration and loudness level. The overall system uses the short-time features of pitch, amplitude, brightness and bandwidth, and their statistics (mean, variance and autocorrelation coefficients) over the whole duration of the sound.

Audio Coding Techniques and Comparison Analysis

Portable electronic devices such as smart mobile phones, digital cameras and digital audio devices with audio players and recorders have been attractive now a days particularly due to prevalence of MP3 audio files. MP3 is the popular name of MPEG-1 layer-3 audio. Moreover, the so-called MP3's successor, MPEG-2 Advanced Audio Coding (AAC), finalized as an international standard in 1997 which was developed to

achieve a higher quality than that of previous coder that is MP3. AAC reaches the same sound quality as MP3 at about 70% of the bit rate. This way more compression is done in AAC as compare with MP3. High quality audio compression has found its way in many applications. Early research on audio has translated into standardization efforts of ISO/IEC and ITU-R 10 years ago. In the last couple of years, Internet audio broadcasting has come in powerful category of this type of high quality applications. These techniques become more and more popular in many parts of the world because of the business for the music industry.

Audio signal is the signal with frequency range of 20 Hz to 20 KHz. Human speeches and other musical component's sounds are merged together and it is called as audio signal. Broadcast of audio used 16-bit PCM encoding at 44.1 kHz, such an application would require a 1.4 Mbps channel for a stereo signal ($44.1\text{KHz} \times 16\text{bit} = 705.6\text{ Kbps}$ for Mono audio signal). Since the beginning of the twentieth century, the art of sound coding, transmission of audio signal, recording of audio signal, and also the mixing and reproduction of it has been constantly evolving. Starting from the mono-ponic technology, technologies on multichannel audio have been extended to include stereophonic, quadraphonic, 5.1 channels, 7.1 channels etc. Compared with the traditional mono or stereo audio signal the multichannel audio provides end users with a better experience and becomes more and more appealing to music producers. So, an efficient coding scheme is needed for the storage and transmission of multichannel audio and this subject has attracted a lot of attention now a days.

There are several multichannel audio compression algorithms. Dolby AC-3 and MPEG Advanced Audio Coding (AAC) are the two most prevalent perceptual digital audio coding systems. Dolby AC-3 is the 3rd generation of digital audio compression systems from Dolby Laboratories and has been used as the audio standard for High Definition Television systems. It is capable of providing transparent audio quality at 384 kb/sec for 5.1 channels. This 5.1 channel technology is come in the categories of multichannel audio. In MPEG family there are lots of different algorithms which can be used for compression of audio file. MP3 is most popular technique used for audio compression which supports to only up to two channels (stereo coding). There are also multichannel audio compression algorithms. Among that AAC is currently the most powerful multichannel audio coding algorithm of the MPEG family. It can support up to 96 audio channels. These low bit rate multichannel audio compression algorithms are utilized transform coding techniques to remove statistical redundancy within each channel of multichannel audio file. The audio compression is possible with different audio coding techniques. The audio file must be compressed without reducing the quality. Table 1 summarized brief history of MPEG family.

Table 3. Brief history of MPEG audio standards

Year	Standards	Sampling rate KHz	Bit rate Kbits/sec	Channels
1992	MPEG-1 Layer I	32, 44.1, 48	32 – 448	1-2
1992	MPEG-1 Layer II	32, 44.1, 48	32-384	1-2

Different audio techniques

1) **Sum-Difference Stereo Transform Coding:** The coder architecture for this technique was explained by Johnston and Ferreira [2] and it is shown in Fig. 1. There are four basic blocks for all perceptual coders. In the case of Perceptual Audio Coder (PAC), the filter bank block is implemented by an MDCT (Modified Discrete Cosine Transform) with the optional window switching abilities. The psychoacoustic analysis provides a noise threshold for the L (left), R (Right), M (Sum), and S (Difference) channels, as may be appropriate, for both the normal MDCT window and the optional shorter windows. The thresholds for the left and right channels THRL and THRR are calculated. This two thresholds are compared where the thresholds vary between left and right by less than 2dB, then the coder is switched into M/S mode, i.e. the left signal for that given band of frequencies is replaced by $M = (L+R)/2$ and the right signal replaced by $S = (L-R)/2$.

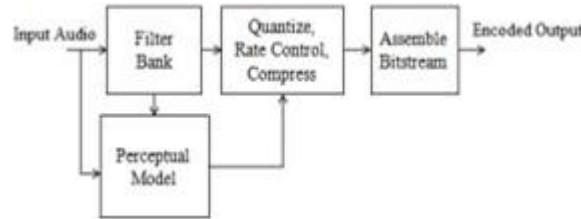


Fig.5.65. Block diagram of perceptual Audio coder

The method gives a substantial improvement over the dual monophonic case. The use of stereo redundancy in a time and frequency varying way results in a substantial increase in encoded signal quality.

2) **Improving Joint Stereo Audio Coding by Adaptive Inter-channel Prediction:** The above MS-stereo coding does not achieve any improvement for the class of most critical test sequences. In MS-stereo coding only the statistical dependencies between two samples of the left and right channels of signals are considered. A stereophonic sound signal is characterized by level differences as well as phase or time delay between the left and right channel signals. So, an appropriate and efficient stereo redundancy reduction technique has to be taken into account. So, for the improvement of it adaptive inter-channel predictor (AICP) is used, which compensates a possible phase or time delay and exploits more than one value of the cross-correlation function between the left and right channels of a sound signal [3]. From successive samples of input signal $x(n)$ in one channel the estimate of the actual sample of signal $y(n)$ in the other channel is calculated,

$$\hat{y}(n) = \sum_{k=0}^K a_k \cdot x(n-d-k) \text{----->(8)}$$

Where, k is the predictor order a_k is the predictor coefficients and d is a delay for compensation of phase or time delay between the two signals. The prediction error is then,

$$e(n) = y(n) - \hat{y}(n) \text{-----> (9)}$$

Compared to the variance of $y(n)$, the variance of the prediction error $e(n)$ is reduced. Therefore, a bit rate reduction is achieved by coding and transmitting $e(n)$ instead of $y(n)$. And the prediction gain is given by the ratio of the variances.

$$G = \frac{E[y^2(n)]}{E[e^2(n)]} = \frac{\sigma_y^2}{\sigma_e^2} \text{-----> (10)}$$

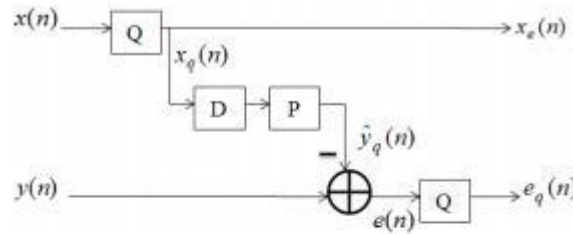


Fig.5.66.(a)Encoder block diagram of AICP

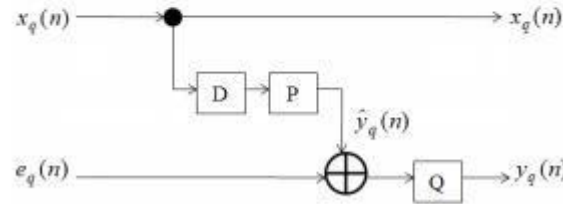


Fig.5.67(b) .Decoder Block diagram of AICP

Fig.5.66(a) and Fig.5.67 (b) show the encoder and decoder block diagram of AICP with quantization. Where, $x(n)$ and $y(n)$ represent the samples of the left and right channel of a stereophonic sound signal respectively and Q is the Quantizer, D is the Delay and P is the Predictor. The Sum difference method does not achieve any improvement of some critical audio file as told earlier, this improvement is done by AICP.

3. Scalable Audio Coder Based on Quantizer Units of MDCT Coefficients: A scalable codec has been constructed by using transform coding and the basic modules of scalable coder (encoder and decoder). The basic module is a quantizer that can quantize MDCT (Modified DCT) coefficients transformed from a variety of frequency regions. This module works at bitrates of more than 8 kb/s. Also the scalable structure can be changed according to the input signals. In the scalable codec described here, the input-output signals are monaural and the sampling frequency is 24 kHz. The total bit rate of this scalable codec is more than 8 kb/s [4]. The basic module is mainly constructed from a Twin VQ (Transform-domain Weighted Interleave Vector

Quantization) codec. It is type of transform coding. Transform coder is used in audio coding. This module is a quantizer for the MDCT coefficients. Here, 4-layer scalable codec as shown in Fig. 3. This codec uses four basic modules with input sampling frequency of 24 kHz . This is a 4-layered scalable codec, but it is possible to make any number of layers using these basic modules. The basic module for first layer (#1) has a fixed range of input or output frequency and other basic modules (#2, #3, #4) have a variable range of input or output frequency with each frequency band width fixed. The frequency point information of each basic module is added to the coded bit stream.

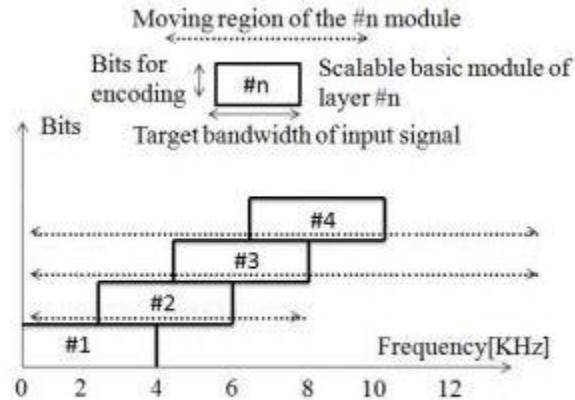


Fig.5.68.Hierarchical Structure of Scalable Codec

In Fig. 5.68, for example, the input 4-kHz signal is quantized in #1 module first and then its quantization error is again quantized in #2 and #3 modules. Here also one can change the width (frequency band width), height (number of bits for each frequency), and position (target frequency) of the each module.

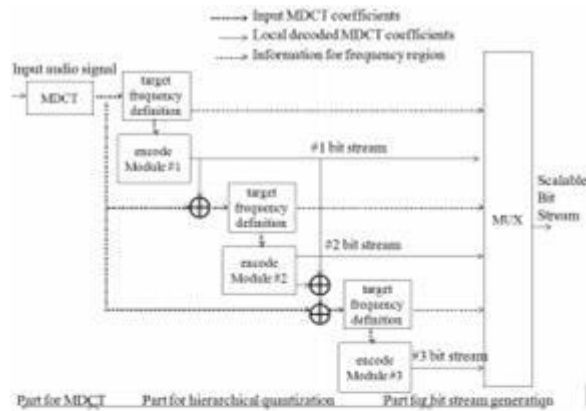


Fig.5.69 Structure of Scalable Encoder

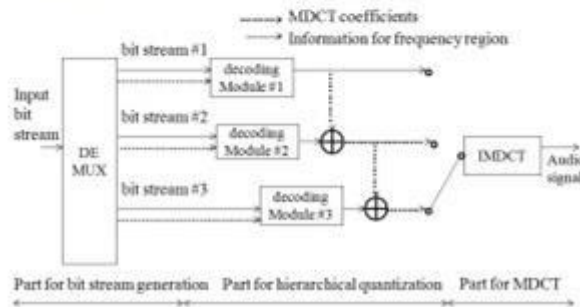


Fig.5.60 Structure of Scalable Decoder

Fig. 45.69 and 5.60 shows the Structure of Scalable Encoder. It has parts for MDCT, hierarchical quantization and bit stream generation. In the MDCT, the input time-series signals are transformed to MDCT coefficients according to its nature by particular transform points. And the scalable decoder has 3 parts for analyzing the bit stream, hierarchical requantization and Inverse MDCT. The Decoder is totally reversing then Encoder. The Decoder is shown in Fig. 5. Subjective quality evaluation tests, for musical sound sources, showed that its sound quality is better than MPEG-layer3 codec at 8, 16 and 24 kb/s when scalable codec is constructed of 8-kb/s basic modules.

4. A Study of Why Cross Channel Prediction is not Applicable to Perceptual Audio Coding: There has been a question that whether the compression rate of a multichannel perceptual audio coder can be increased by applying the cross channel linear prediction (LP) in the time domain or not. There exists correlation between two channels which can be removed by using cross channel linear prediction. Hence, theoretically, by coding the prediction residual instead of the original signal, the coder should achieve a higher compression ratio, at least without considering the requirements of human perception. There is considerable correlation between the channel pairs C-L, C-R, L-R, Ls-Rs, L-Ls, and R-Rs. The M/S stereo coding and intensity stereo coding in the MPEG AAC can highly remove the correlation between the pairs L-R and LsRs. But the rest of the channel pairs, it seems that one can take good advantage of the correlation between C and L channel and between C and R channel. Means one can get some coding gain by using C to predict L and R.

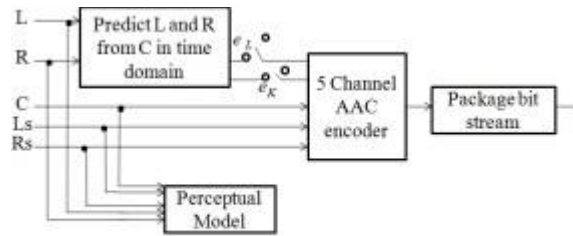


Fig.5.61 Prediction-incorporated AAC coder

the signal energy and coding bits in low frequency bands is also reducing, but increase the coding bits in high frequency bands. So, the increase in high frequency bits exceeds the bit reduction at low frequencies, which results in a net increase of coding bits required. So, overall improvement is not done and this is the reason why cross channel prediction is not applicable to perceptual audio coding.

5. MPEG-1 Layer-III (MP3): MP3 is a lossy compression technique it means some audio information is certainly lost by using this compression technique. This loss can be noticed because the compression technique tries to control it. MPEG-1 audio describes three layers of audio coding with the following properties: - Mono or stereo audio channels. - Sample rate 32 kHz, 44.1 kHz or 48 kHz. - Bit rates from 32 kbps to 448 kbps .

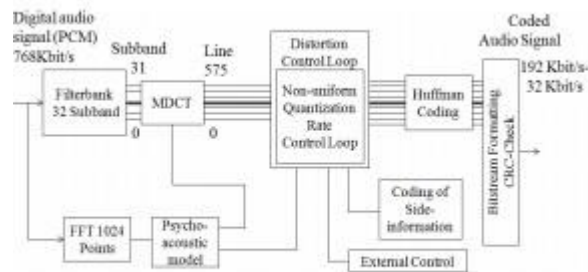


Fig.5.62 Block diagram of MP3 Encoder

Fig.5.62 shows the block diagram of MP3 encoder. There are two filter banks in a MPEG audio algorithm, namely a filter bank and a hybrid polyphase/MDCT filter bank. The input PCM samples are simultaneously given to a filter bank and a psychoacoustic model. This Filter bank splits the signal into 32 equal sub bands in frequency domain and psychoacoustic model takes the signal spectrum as input and it determines the ratio of signal energy to masking threshold for each sub band. For better frequency resolution the 32 sub bands are further divided into 576 frequency lines by the MDCT. Here MDCT used is 12 point (short) or 36 point (long) with 50 % overlap and the type of MDCT (long or short window) is determined by the window switching algorithm [6]. In Layer-3, these coder partitions are roughly equivalent to the critical bands of human hearing system. If the quantization noise can be below the masking threshold for each coder partition, then the compression result should be indistinguishable from the original signal. The signal to masking ratio (SMR) which is calculated by the psychoacoustic model is used by the quantizer to determine the number of bits that should be allocated for the quantization of the sub band coefficients. Here the quantization is done by the power-law

quantizer. The quantized values are coded by Huffman coding. Then Finally the Huffman coded values are formed into a bit stream. A bit stream formatter is used to assemble the whole bit stream. The encoded bit stream consists of quantized and coded spectral coefficients with some side information like bit allocation information and quantizer step size information.

Reference

1. P. Rao, , Chapter in Speech, Audio, Image and Biomedical Signal Processing using Neural Networks, (Eds.) Bhanu Prasad and S. R. Mahadeva Prasanna, Springer-Verlag, 2007.
2. Digital Signal processor fundamentals and system design.
M.E.Angollette, CERN, Geneva,Switzerland.