INTERNATIONAL ATOMIC ENERGY AGENCY
UNITED NATIONS EDUCATIONAL, SCIENTIFIC AND CULTURAL ORGANIZATION
INTERNATIONAL CENTRE FOR THEORETICAL PHYSICS
I.C.T.P., P.O. BOX 586, 34100 TRIESTE, ITALY, CABLE: CENTRATOM TRIESTE

H4.SMR/841-12

FOURTH ICTP-URSI-ITU(BDT) COLLEGE ON RADIOPROPAGATION:
Propagation, Informatics and Radiocommunication System Planning

30 January - 3 March 1995

Miramare - Trieste, Italy

Digital Signal Processing in Radiocommunications

J.G. Lucas
University of Western Sydney, Nepean
Kingswood (Sydney) Australia

DIGITAL SIGNAL PROCESSING IN RADIO COMMUNICATIONS

GODFREY LUCAS, UNI. OF WESTERN SYDNEY, NEPEAN. AUSTRALIA.

CONTENTS.

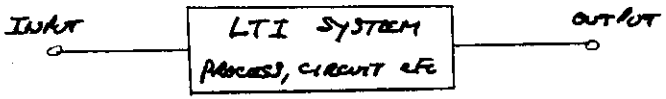
DIGITAL SIGNAL PROCESSING IN RADIO COMMUNICATIONS

GODFREY LUCAS, AUSTRALIA

TOOLS FOR DSP

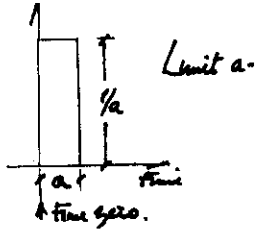
Start with analogue ideas and walk our way towards the digital.

Consider a relaxed, linear, time invariant (LTI) System

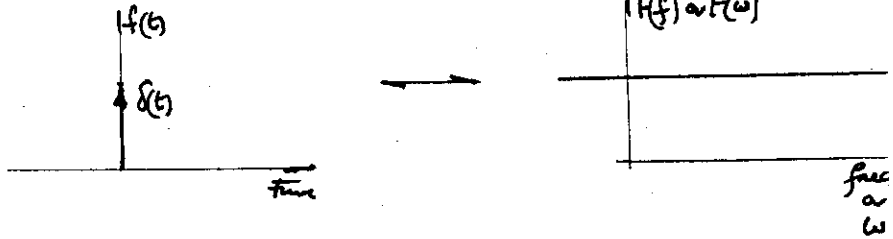


Argue that the system is completely specified by applying a "unit impulse" $\delta(t)$

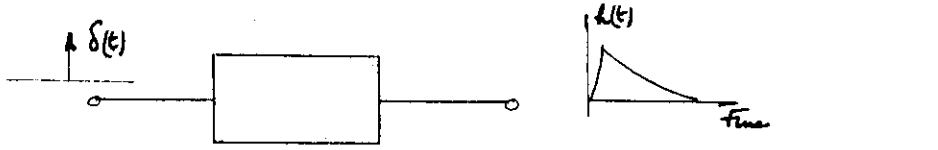
Defined by $\int_{-\infty}^{\infty} \delta(t)\,dt = 1$



Reasonable assumption because:

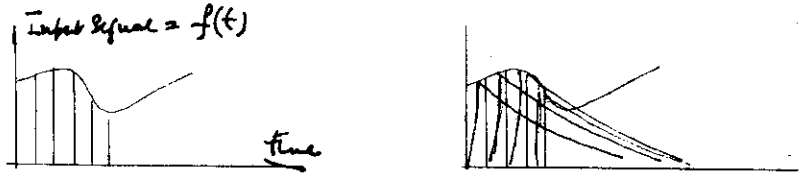


ie contains all frequencies



CAUSAL IMPULSE RESPONSE

We can then work out the response for any input signal by breaking that input signal up into its component impulses



& of course we actually evaluate this by using the convolution or folding integral.

$$\text{Output}(t) = \int_{-\infty}^{\infty} f(\tau)\, h(t-\tau)\, d\tau$$

$$\text{OR} \quad = \int_{-\infty}^{\infty} f(t-\tau)\, h(\tau)\, d\tau$$

Now suppose our input signal is an exponential e^{st} where $s = \sigma + j\omega$

Then $$\text{Output}(t) = \int_{-\infty}^{\infty} e^{s(t-\tau)}\, h(\tau)\, d\tau$$

$$= e^{st} \int_{-\infty}^{\infty} e^{-s\tau}\, h(\tau)\, d\tau$$

And this integral has the shape of the Laplace Transform

So recall the whole idea of these transform approaches is to move away from the time domain where we have to deal with a (often nasty) convolution integral into the transform domain where "simple" arithmetic applies

In fact it is always the case that:

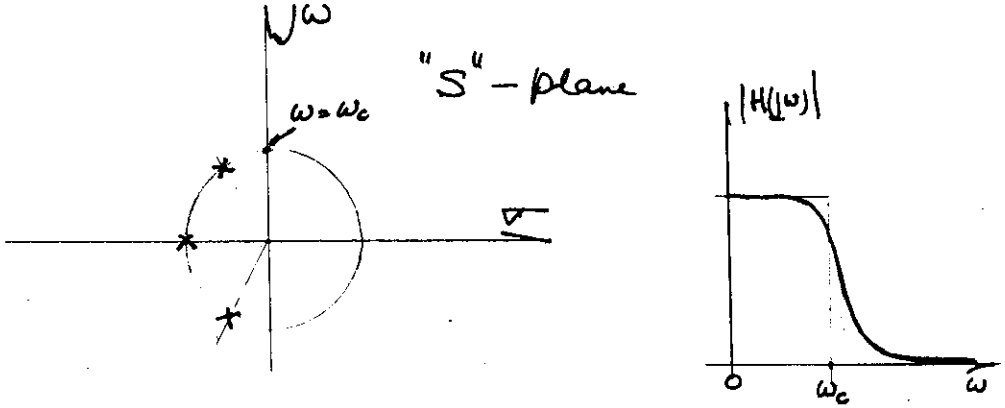
Multiplication in one domain ≡ Convolution in the other!

Laplace transform particularly useful because the "damping" term σ in $s = \sigma + j\omega$ neatly controls functions like a ramp & even an exponential — even allows us to deal with initial conditions.

Laplace transforms also lead to the development of a most useful "S" plane

For many design situations it is convenient to use rational polynomials to specify the system function

i.e. $$H(S) = \frac{P_1(S)}{P_2(S)}$$ $P_1(S)$ → ROOTS give ZEROES; $P_2(S)$ → ROOTS give POLES.



Example here of the specification of a 3rd order BUTTERWORTH FILTER SPECIFICATION.

Here the S-plane represents a filter CIRCUIT but it can just as easily represent a signal

Since $\frac{Y(S)}{X(S)} = H(S)$ $\left(= \frac{\text{Output}}{\text{Input}}\right)$

So $Y(S) = X(S) \cdot H(S)$ plot $Y(S)$!

we say that the Laplace transform is a slightly contrived approach because it has no practical real world equivalence.

But if we set the damping factor to zero (ie $\sigma = 0$) we then simply have the Fourier transform

$$F(f) = \int_{-\infty}^{\infty} f(t)\, e^{-j2\pi ft}\, dt$$

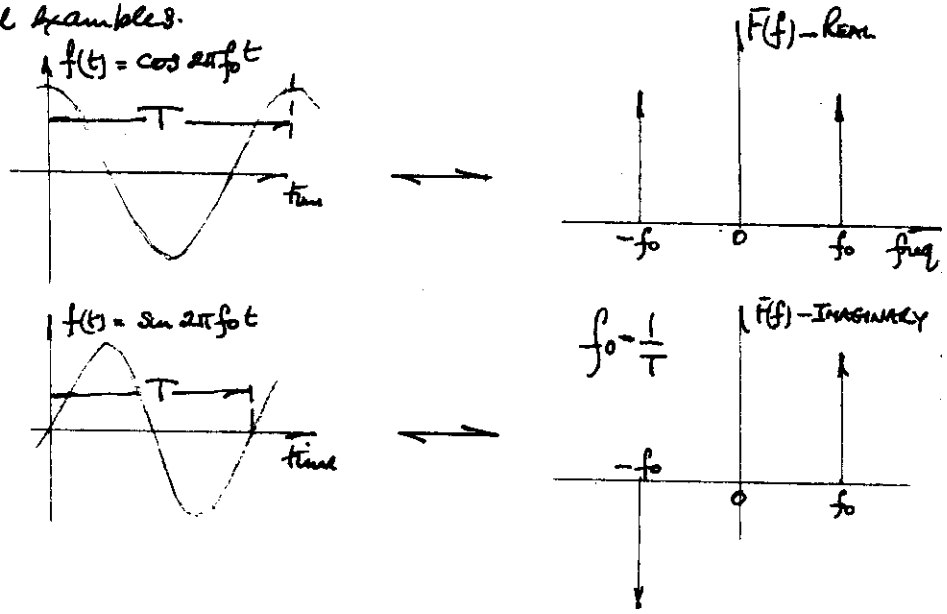
which does have exact real life parallels.

TIME	⟷	FREQUENCY
ANTENNA APERTURE DISTRIBUTION	⟷	RADIATION PATTERN
HOLOGRAM	⟷	OBJECTIVE PICTURE

etc

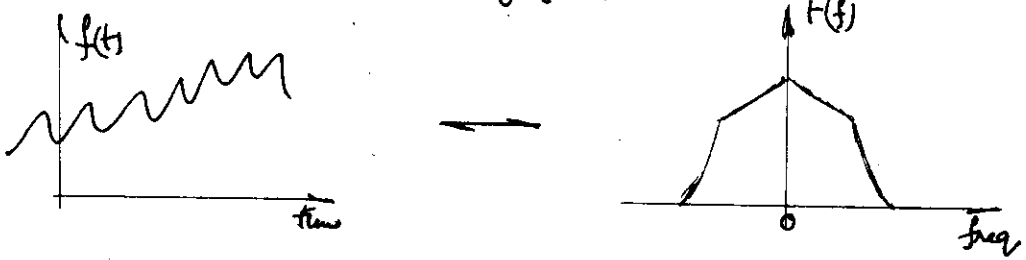
In radio communications we usually Fourier transforms to electrical signals

Typical examples.



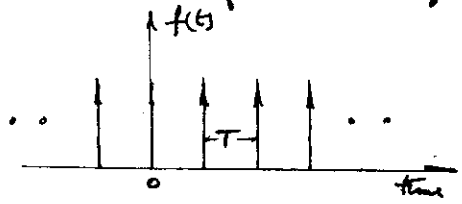
I am going to assume that you know Fourier well.

To transmit useful information of course we need to consider a bandwidth of frequencies:



Notice my spectra deliberately include "negative" frequencies

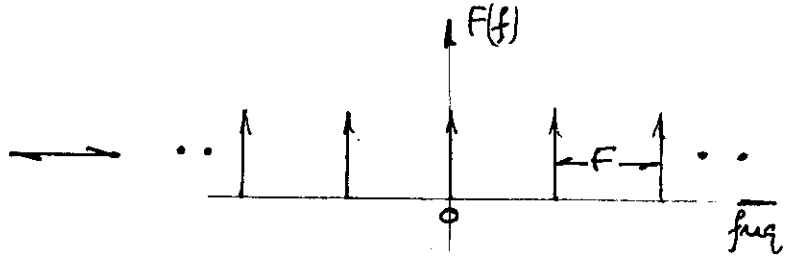
The first real step towards DIGITAL is to consider the process of sampling an analog signal.



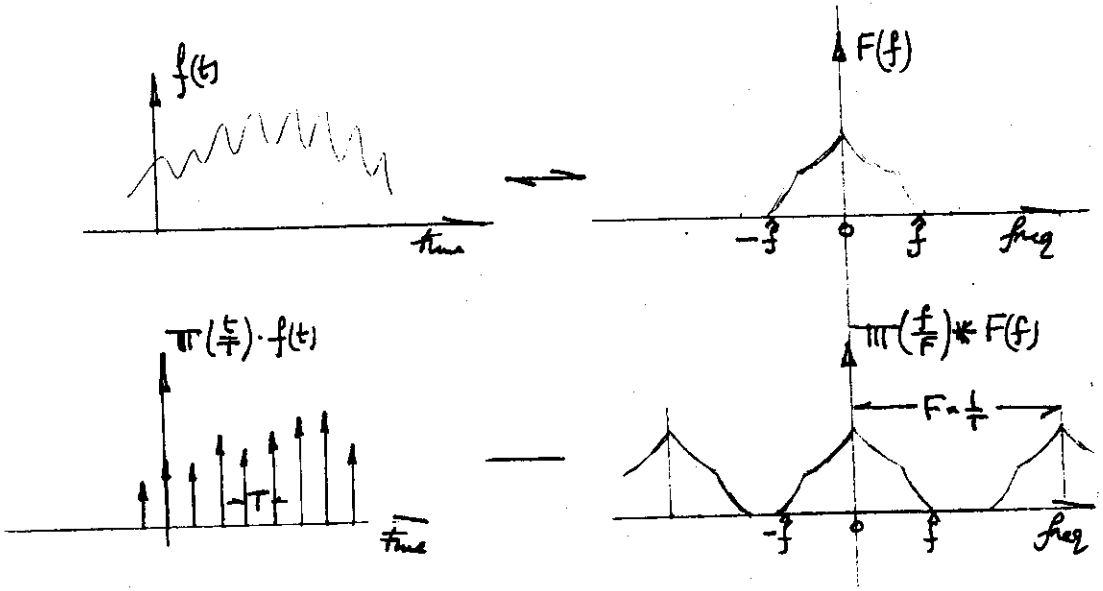
The sampling function is just an infinite set of equispaced impulses

$$f(t) = \delta(t \pm nT)$$
$$= \text{Ш}(t/T)$$

It is easy to prove that the Fourier transform of this is just a similar sampling function in the frequency domain



$$F(f) = \text{Ш}(f/F)$$
$$= \text{Ш}(Tf)$$



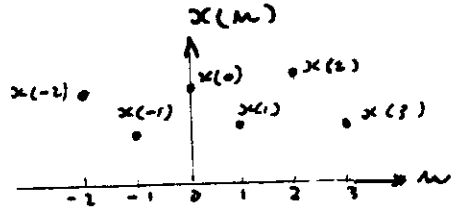
with insufficient sampling (ie T too big) these convolved spectra overlap = aliasing when the original signal cannot be retrieved.

Critical Sampling $F = \frac{1}{T} = 2f_m$

Discrete-time signals.

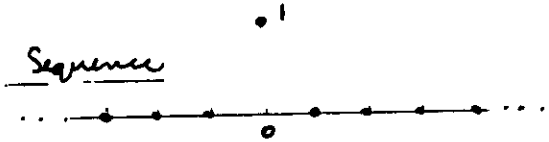
Defined at discrete times only.

Representation:



"Singularity" functions and special functions.

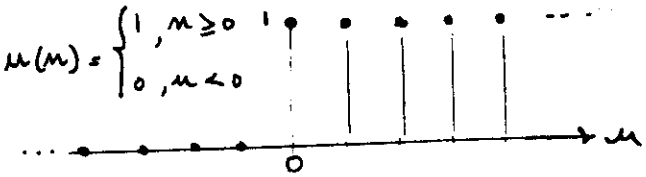
Unit Sample Sequence



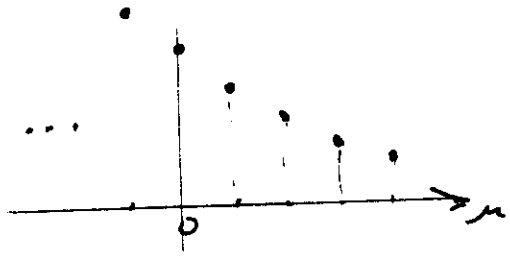
$$\delta(n) = \begin{cases} 0 & n \neq 0 \\ 1 & n = 0 \end{cases}$$

Unit Step

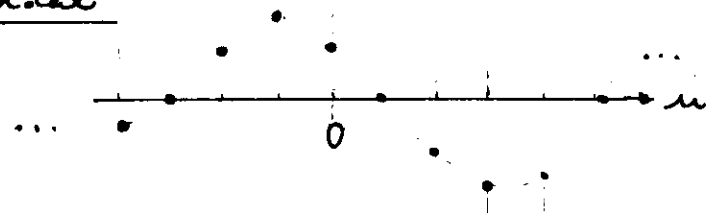
$$u(n) = \begin{cases} 1, & n \geq 0 \\ 0, & n < 0 \end{cases}$$



Real Exponential



Sinusoidal



Relation between unit sample and unit step.

$$\mu(n) = \sum_{k=-\infty}^{n} \delta(k)$$

$$\delta(n) = \mu(n) - \mu(n-1)$$

Real Exponential Sequence

Values of form a^n where a is a real number.

Sinusoidal Sequence

Values of form $A\cos(\omega_0 n + \phi)$

Complex Exponential Sequence

Values of form $e^{(\sigma + j\omega_0)n}$

Energy of a Sequence

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

Product $x \cdot y = \{x(n)\,y(n)\}$

Sum $x + y = \{x(n) + y(n)\}$

Multiplication by a number

$$\alpha \cdot x = \{\alpha\, x(n)\}$$

Delay Sequence y is said to be a delayed version (shifted) of a sequence x if

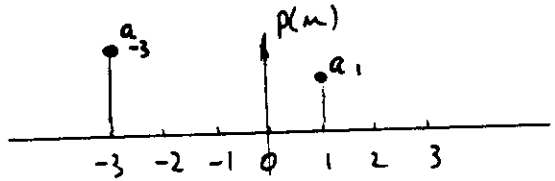
$$y(n) = x(n - n_0)$$

where n_0 is an integer.

(advance case should also be possible).

Representation of Scaled, delayed values

$p(n) = a_{-3}\delta(n+3) + a_1\delta(n-1)$ etc



Linear Shift Invariance

Linear → implies superposition

ie $x(n) \longrightarrow \boxed{T[-]} \longrightarrow y(n)$

$$T\left[a x_1(n) + b x_2(n)\right] = a T[x_1(n)] + b T[x_2(n)]$$

$= a y_1(n) + b y_2(n)$ for arbitrary constant a and b.

Description of Sequence by Unit Sample Response

$$y(n) = \sum_{k=-\infty}^{\infty} h(k)x(n-k) = h(n) * x(n)$$

Convolution Sum. Calculation of convolution sum!

Convolution Sum Of Two Sequences

$$y(n) = \sum_{k=-\infty}^{\infty} h(k)x(n-k)$$

1. Folding
2. Shifting
3. Multiplication
4. Summation.

EXAMPLE

Two sequences $\{1, 2, 1, -1\} = h(n)$

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

Fold $h(n)$ ⇒ $\{-1, 1, 2, 1\}$

Leaves to: $-1, 1, 2, 1$ | $1, 2, 3, 1$

Multiply 1

Then shift to right one step → 4

→ 8

→ 8

→ 3

→ −2

→ −1

etc

Stability & Causality.

Stable system BIBO.

Every bounded input produces a bounded output.

i.e. $S \triangleq \sum_{k=-\infty}^{\infty} |h(k)| < \infty$ Convergence Test.

Causal System

The output for any $n = n_0$ depends only on the input for $n \le n_0$.

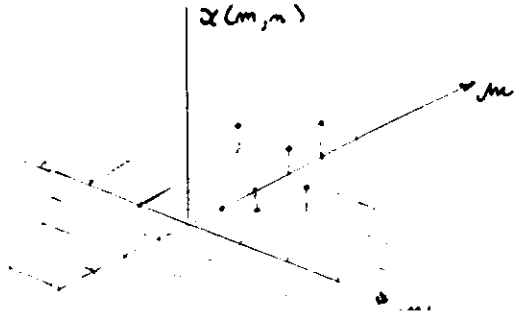
~~I~~ A linear shift invariant system is causal only if its unit sample response is $0, n < 0$.

Causal Sequence is a sequence which is zero, $n < 0$.

Two dimensional sequences and systems.

Three dimensional sequences and systems.

Diagram needed.



System representation.

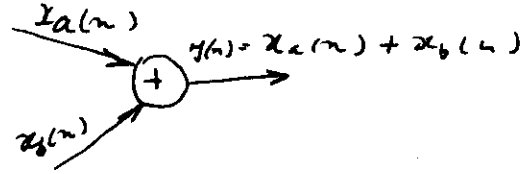
Block diagrams

Signal flow graphs.

Block diagrams and signal flow graphs are illustrative ways of describing systems.

Block diagrams: Symbols.

ADDER



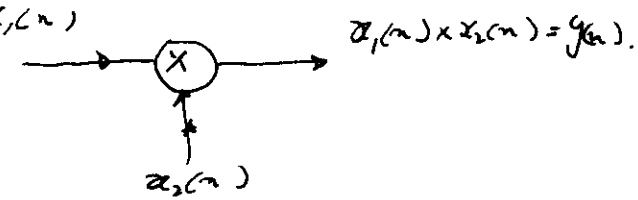
CONSTANT MULTIPLIER $x(n) \xrightarrow{\ a\ } ax(n) = y(n)$

Scaling factor shown alongside direction branch.

NB This must not be understood to mean that $x(n) = \frac{y(n)}{a}$ because one cannot go against the arrow. Signal flow is in direction of the arrow only.

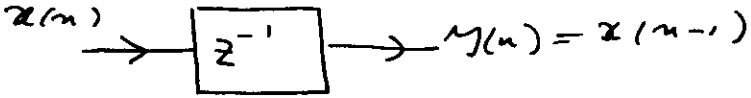
Also the branch scaling is without memory.

SIGNAL MULTIPLICATION / MULTIPLICATION BY A FUNCTION OF SOME VARIABLE



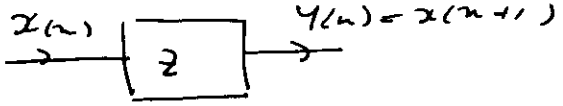
DELAY

Unit delay element.



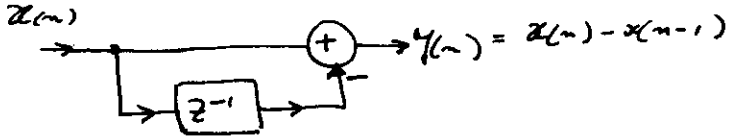
(Use of z^{-1} for delay will become clear later when we deal properly with the z transform).

ADVANCE

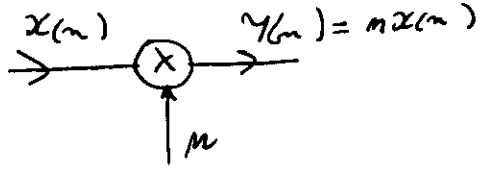


Use of these building blocks will be developed as we go along.

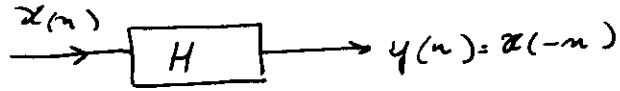
"DIFFERENTIATOR" note the inverted commas!



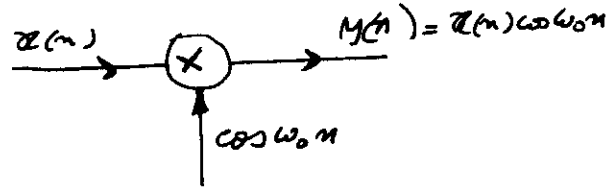
"TIME" MULTIPLIER



"FOLDER"



MODULATOR

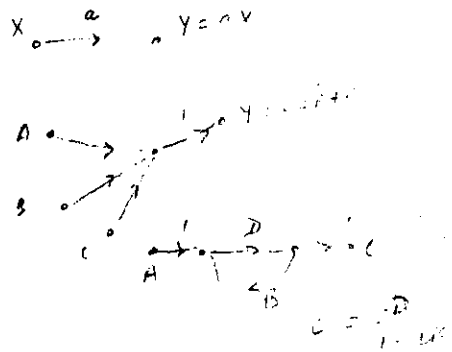


TYPICAL BLOCK DIAGRAMS TO BE ILLUSTRATED

Signal flow graphs.

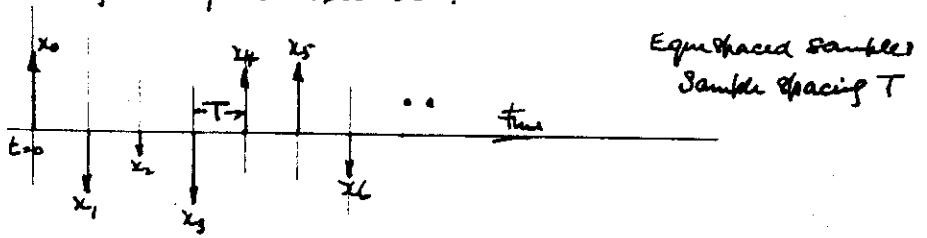
The use of signal-flow graphs is beneficial in many engineering problems. The original work on signal-flow graphs is due to MASON. (Copies of mason's original paper to be handed out). There is a lot of similarity between the signal flow graph representation and the block diagram of a system.

SIGNAL FLOW GRAPHS
Mason S.J.,
(Handout)



There is also an important transform alternative to handling these digital sample sets.

Consider the following sample set:



Equi-spaced samples
Sample spacing T

The time description is simple:

$$f(t) = x_0\,\delta(t) + x_1\,\delta(t-T) + x_2\,\delta(t-2T) + \ldots\ldots$$

we can immediately write down the Laplace Transform as

$$F(s) = x_0 + x_1 e^{-sT} + x_2 e^{-s\cdot 2T} + x_3 e^{-s\cdot 3T} + \ldots$$

& the Fourier transform is just as simple

$$F(\omega) = x_0 + x_1 e^{-j\omega T} + x_2 e^{-j\omega 2T} + \ldots$$

It is the Laplace form that we adapt to a "Z" transform by defining a new variable $Z = e^{sT}$

Leading to the Z transform form for this sample set

$$F(z) = x_0 + x_1 z^{-1} + x_2 z^{-2} + x_3 z^{-3} + \ldots \text{etc.}$$

Z is a new frequency variable which generally has both real and imaginary parts

ie. $Z = e^{sT} = e^{\sigma T}\cdot e^{j\omega T}$

$= e^{\sigma T}\cos\omega T + j\,e^{\sigma T}\sin\omega T$

now the term $e^{-j\omega T}$ implies a phase shift which is directly proportional to frequency which is the same as a constant time delay of T seconds.

In exactly the same way $e^{+j\omega T}$ implies a time advance of T seconds. Z is quite commonly known as a time shift operator.

The Z transform is generally written as

$$F(z) = \sum_{0}^{\infty} x(nT)\,z^{-n}$$

Generally the "T" is omitted & a general "two sided" form use is:

$$F(z) = \sum_{-\infty}^{\infty} x(n)\,z^{-n}.$$

This sequence is not always convergent and just as the condition for existence of the Fourier transform is important so is what is called the "REGION OF CONVERGENCE" of the Z transform.

Instructive to compare requirements with the F. transform.

Fourier Transform

$$F(j\omega) = \sum_{n=-\infty}^{\infty} x(n)\, e^{-j\omega n}$$

note: the two are identical when $r = 1$.

Convergence requires that the sequence be absolutely summable.

ie $\sum_{n=-\infty}^{\infty} |x(n)| \overset{\text{must be}}{<} \infty$

For example the simple constant sequence

$x(n) = u(n)$ fails

Z-Transform

$$F(z) = \sum_{n=-\infty}^{\infty} x(n)\, z^{-n}$$

or

$$F(re^{j\omega}) = \sum_{n=-\infty}^{\infty} x(n)\, r^{-n} e^{-j\omega n}$$

Here of course the factor r^{-n} really assists in achieving convergence eg cases with $x(n) = u(n)$ provided $|r| > 1$

Again we will be using rational polynomials with the Z transform which, as with S plane ideas, lead to poles and zeros. Obviously you cannot have a pole in a region of convergence and it can be shown that regions of convergence are bounded by poles.

In many cases of practical interest we shall be dealing with finite length sequences when these convergence ideas are simple.

Poles & Zeros in the "Z-plane" — relation to 'S-plane'

We know that the frequency spectrum of any Sampled signal is endlessly repetitive and repeats with a frequency period of $F = 1/T$ hertz

If we attempted to plot such a sampled signal on the S plane we would expect an endless by repetitive pole zero pattern!

In fact the elegance of the S plane was the ease of visualizing performance — a feature which becomes a nightmare with Sampled data. But the Z transform does allow us to deal easily with Sampled data by developing a "Z-plane" which is similar to the "S-plane" and the two representations are closely related.

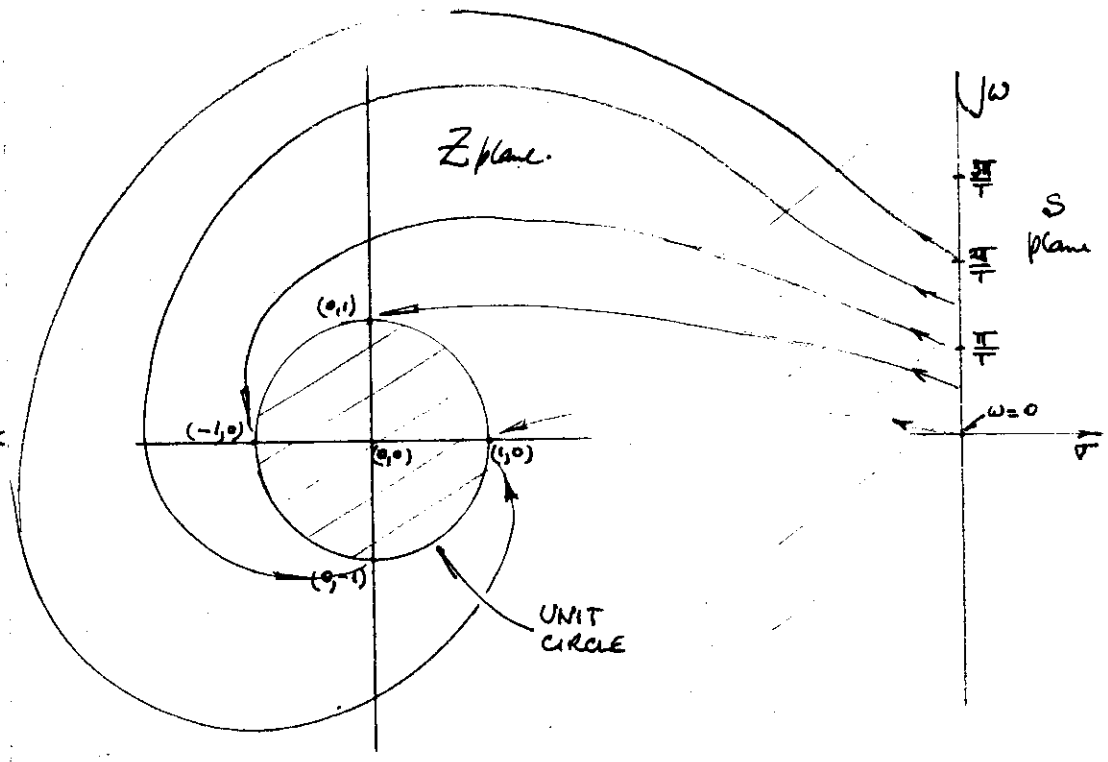
To find this relation we investigate the complex variable Z for various values of S. This process is called "mapping" the S-plane into the Z-plane.

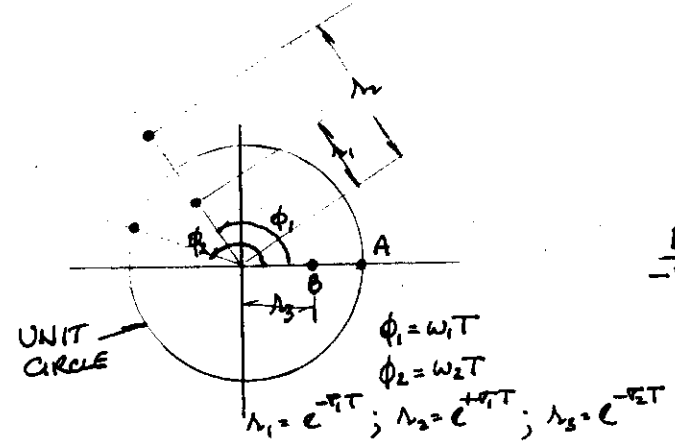
Now $S = \sigma + j\omega$. Consider $\sigma = 0$ & S is pure imaginary

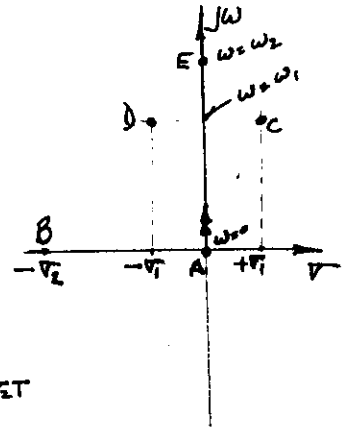
then $Z = e^{j\omega T}$ and as ω is varied the locus of Z will be a circle of radius 1.

When $\omega = 0$ $Z = +1$ & returns each $\frac{2\pi}{T}$

See next page diagram.







It is clear that the entire Left hand S plane (stable region) is mabbed INSIDE the unit circle on the Z plane. The Right hand S plane is mabbed outside. It is clear too that if a pole were placed at $\delta(\text{say})$ it would appear to be equivalent to an infinite set of poles in the S plane. each separated by $\frac{2\pi}{T}$ in the direction parallel to the imaginary axis.

An example here illustrates the power of the Z-transform technique:

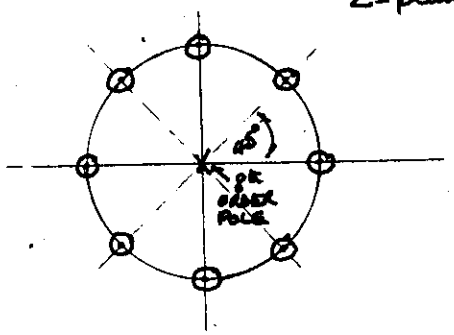
Consider the function $F_1(z) = (1 - z^{-8})$

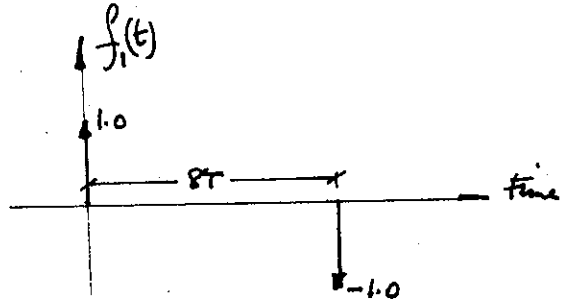
we can write this as:

$$F_1(z) = \frac{z^8 - 1}{z^8}$$

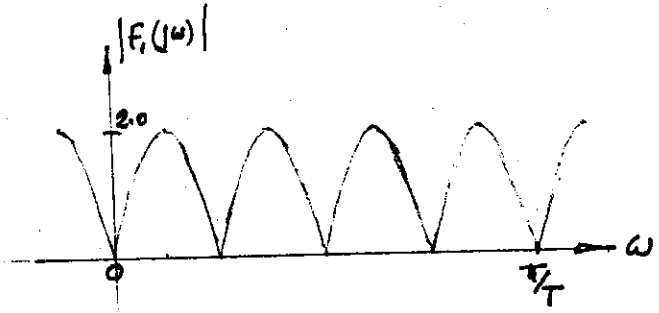
which appears as 8 zeros uniformly spaced around the unit circle and an eighth order pole at $z = 0$ as shown in the following figure together with Time function & spectrum.

Z-plane





CORRESPONDING TIME FUNCTION



CORRESPONDING SPECTRUM.

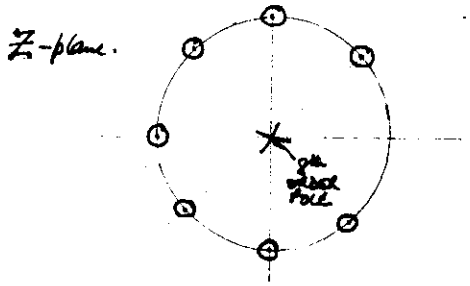
Suppose in the last example we were to remove the zero at $z=1$ to create:

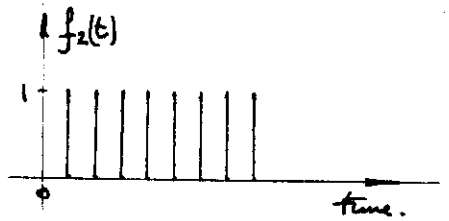
$$F_2(z) = \frac{1}{z^8}\left\{(z+j)(z+1)(z-j)\left(z-\frac{1}{\sqrt{2}}+j\frac{1}{\sqrt{2}}\right)\left(z-\frac{1}{\sqrt{2}}-j\frac{1}{\sqrt{2}}\right)\cdot\left(z+\frac{1}{\sqrt{2}}+j\frac{1}{\sqrt{2}}\right)\cdot\left(z+\frac{1}{\sqrt{2}}-j\frac{1}{\sqrt{2}}\right)\right\}$$

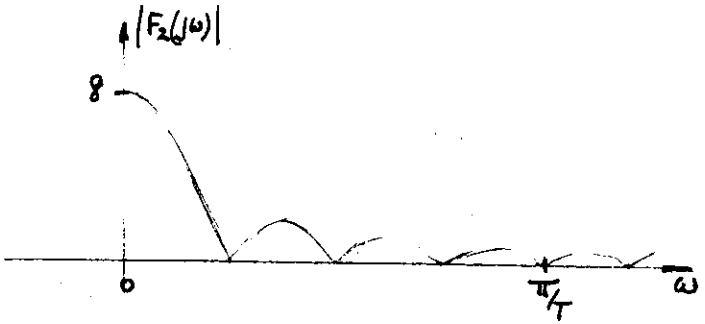
$$= z^{-1}+z^{-2}+z^{-3}+z^{-4}+z^{-5}+z^{-6}+z^{-7}+z^{-8}$$

& this must be the same as adding a pole at $z=+1$ (to cancel a zero) in the original function

ie $F_2(z) = \frac{z^8-1}{z^8} * \frac{1}{z-1}$







DIGITAL FILTERS.

Can be implemented with logic circuits but it is probably much more usual to program a computer and use an interface board. At the present time very fast digital signal processing chips are being developed which have their own program (PROM) & operate standalone.

Also particularly in the case where real time performance is not required use can be made of FFT type programs. — not going to be emphasised here at all.

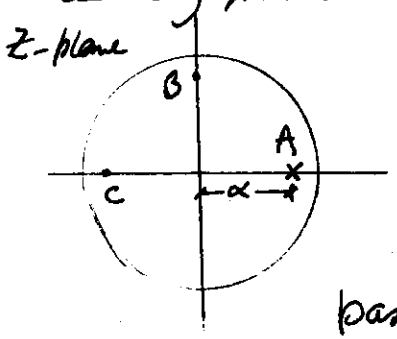
Although we learn to some extent on analogue designs but there are also a lot of brand new techniques which just cannot be applied to analogue designs

In essence you aim for a transfer function $H(z)$ which does the job that you want

From which the impulse response is obtained and this is then convolved with the incoming signal.

Many brand new approaches to design have been developed from the Z plane. There are whole new classes of digital filter which have no analogue counterpart at all

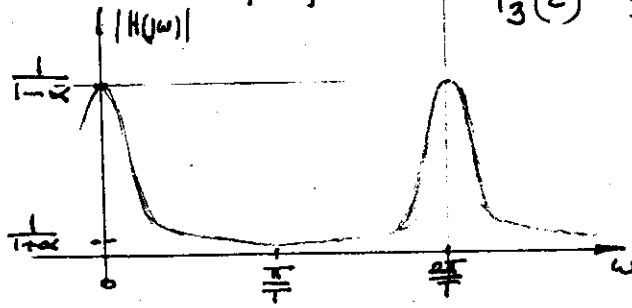
Causality is usually no longer a problem As we can simply arrange to delay the impulse response by the appropriate number of sample periods to achieve causality — this is simply equivalent to adding poles at the origin of the Z plane.



a typical design approach adds a pole close to the unit circle. at A it gives a low pass, at B a band pass & at C a high pass result

For the case shown (low pass).

The transfer function $F_3(z) = \frac{\text{output}}{\text{input}} = \frac{Y(z)}{X(z)} = \frac{1}{z-\alpha}$



The impulse response = $H(z) = \frac{1}{z-\alpha} = (z-\alpha)^{-1}$

$= z^{-1} + \alpha z^{-2} + \alpha^2 z^{-3} + \ldots\ldots$ to ∞

a big looking sequence ??

However if we cross multiply

So $Y(z) \cdot z - \alpha Y(z) = X(z)$

This corresponds to a recurrence formula:

$$y(n+1) - \alpha y(n) = x(n)$$

or advancing a single period delay

$$y(n) - \alpha y(n-1) = x(n-1)$$

i.e $y(n) = x(n-1) + \alpha y(n-1)$

So it uses a previous input AND a previous output. — but it is very simple indeed!

Known as RECURSIVE — always has an infinite impulse (II Response

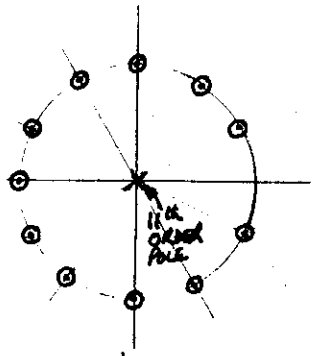
Clearly possible to add additional Zeros to what we have to get even better performance

The example considered on page 26 can also be extended which had the form:

$$H(z) = (z^k - 1)$$

k can now be a large number eg 12. For a low pass filter cancel the Zero at $z = 1$ by placing a pole there and add 11 poles at the origin to ensure realizability

Then $$H(z) = \frac{(z^{12} - 1)}{z^{11}(z-1)} = \frac{Y(z)}{X(z)}$$

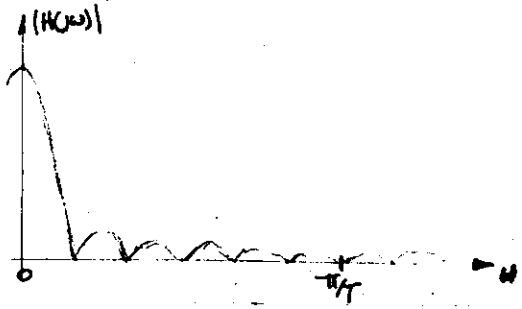


giving a recurrence formula:

$$y(n+12) - y(n+11) = x(n+12) - x(n)$$

or $$y(n) = y(n-1) + x(n) - x(n-12)$$

a very efficient algorithm!



IIR Filter Design By Bilinear Transformation

Proakis p. 626 et seq.

The previous mappings have some severe limits.

The bilinear transformation is a conformal mapping which transforms the $j\Omega$ axis onto the z-plane unit circle only once, and so avoids aliasing problems. All LHP points map to the interior of the z-plane unit circle, and all RHP points are mapped to the exterior of the unit circle.

To illustrate this, consider an analogue filter with transfer function

$$H(s) = \frac{b}{s+a}$$

This system has the differential equation:

$$\frac{dy(t)}{dt} + ay(t) = bx(t)$$

If we integrate the derivative we get

$$y(t) = \int_{t_0}^{t} y'(\tau)\, d\tau + y(t_0)$$

where $y'(t)$ denotes the derivative of $y(t)$.

Using trapezoidal rule at $t = nT$ and $t_0 = nT - T$ gives

$$y(nT) = \frac{T}{2}\left[y'(t) + y'(nT-T)\right] + y[nT-T]$$

The differential equation, evaluated at $t = nT$

gives $y'(nT) = -ay(nT) + bx(nT)$

This value is now substituted for $y'(nT)$ in the previous equation to obtain a difference equation.

Set $y(n) \equiv y(nT)$ $\quad x(n) \equiv x(nT)$

this leads to

$$\left(1 + \frac{aT}{2}\right)y(n) - \left(1 - \frac{aT}{2}\right)y(n-1) = \frac{bT}{2}\left[x(n) + x(n-1)\right]$$

Take the z-transform of this difference equation. leading to

$$\left(1 + \frac{aT}{2}\right)Y(z) - \left(1 - \frac{aT}{2}\right)z^{-1}.Y(z) = \frac{bT}{2}\left(1 + z^{-1}\right)X(z)$$

The filter transfer function is $H(z) = \dfrac{Y(z)}{X(z)}$

$$= \frac{\left(\frac{bT}{2}\right)\left(1 + z^{-1}\right)}{1 + \frac{aT}{2} - \left(1 - \frac{aT}{2}\right)z^{-1}}$$

This can be simplified somewhat to

$$H(z) = b \Big/ \left[\frac{2}{T}\left(\frac{1 - z^{-1}}{1 + z^{-1}}\right) + a\right]$$

The mapping is evidently given by setting

$$s = \frac{2}{T}\left(\frac{1 - z^{-1}}{1 + z^{-1}}\right)$$

This is the bilinear transformation.

[Although derived from 1st order diff. eqn. this holds for any order].

Relation between Ω and ω

When $T = 1$, $\Omega = 0$

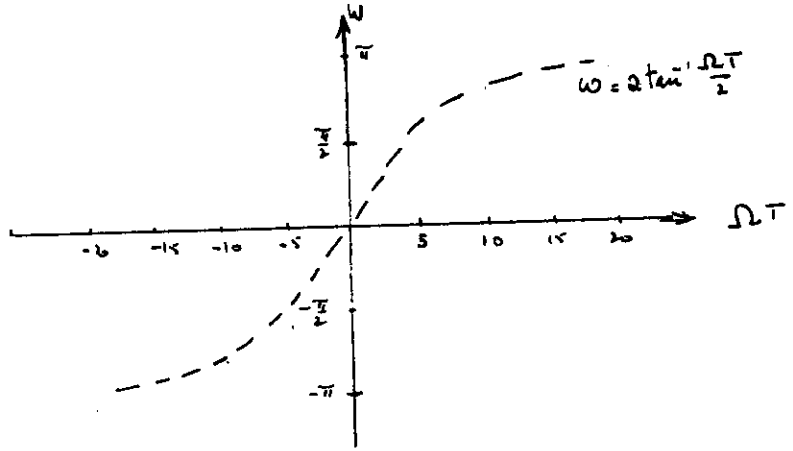
$$\Omega = \frac{2}{T} \cdot \frac{2T \sin\omega}{1 + T^2 + 2T\cos\omega} = \frac{2}{T} \cdot \frac{\sin\omega}{1 + \cos\omega}$$

(note $\tan\frac{A}{2} = \sin A / (1 + \cos A)$)

$$= \frac{2}{T} \cdot \tan\frac{\omega}{2}.$$

Now note :- (i) The entire range in Ω is mapped into range $-\pi \le \omega \le \pi$ once only.

(ii) This mapping is non-linear!



Relationship between the variables in the bilinear transformation.

Proakis Examples 8.2.4 and 8.2.5 apply.

Refer also to 8.2.4. Matched-z Transformation.

DIGITAL FILTERS — abbreviated overview.

Previous work on analogue filters should be reviewed, as it is the basis of the specification of digital filter forms.

References: PASSIVE AND ACTIVE FILTERS - WAI-KAI CHEN (Wiley) CHAPTER 2.
This reference covers filter types, responses, cost-locations;

Filters?

The function of a filter is to estimate the value of a signal, or the state of the system in the presence of noise / or other unwanted signals.

Consequently, filters form a part of almost all systems, and are of importance in (sonar, radar, communications, astronomy, navigation, medicine ...).

Design methods. Classical approach - we use the gross spectral content of the spectra of the signals and noise - in a kind of way, we are assuming a "steady state" system situation.

Modern approach considers the statistical properties of both the signals and the noise. In fact this approach is essential in many applications of communications, radar, sonar and control systems.

Discrete-Time Filters.

We will begin with an overview of D-T. filters and then later in the course deal with design methods for some important filter classes.

For stationary inputs, the filter solutions consist of appropriate interconnections of :-

(i) unit delay
(ii) multipliers
(iii) adders.

The use of signal-flow graphs is helpful in describing the behaviour of such systems.

Note that we are dealing with the time-domain behaviour of LINEAR, SHIFT-INVARIANT, DISCRETE-TIME filters.

LINEARITY means that the filter obeys the principle of superposition. If linear combinations of inputs map into linear combinations of outputs.

SHIFT-INVARIANCE implies that if the response of the filter to an input $x(n)$ is $y(n)$, then its response to $x(n \pm m)$ is $y(n \pm m)$.

Signal Flow Graphs.

It is important to read and understand the S. Mason paper on signal flow graphs. This was the first paper published on the subject, and it is the best.

The following comments on signal-flow graphs bring out a few important rules.

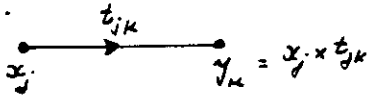
A graph consists of nodes and interconnecting directed branches.

A node j has an associated value x_j.

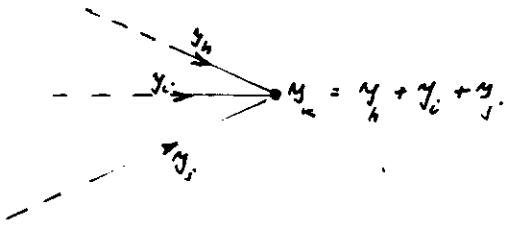
A directed branch jk begins at node j and ends at node k. It has a branch transmittance t_{jk}, specifying the way in which the signal y_k at node k depends on the signal x_j at node j.

RULES.

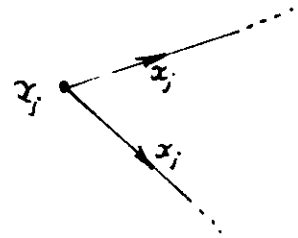
(i). A signal flows along a branch only in the direction of the arrow, and is multiplied by the branch transmittance.



(ii) A node signal value is equal to the algebraic sum of all signals entering the node via the incoming branches.

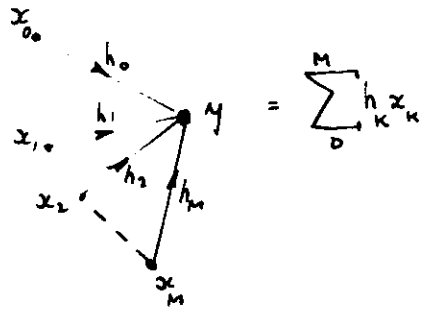


(iii) The signal at each node is applied to each outgoing branch originating at that node, and is entirely independent of the transmittance values of the outgoing branches.



Example.

Linear combination.

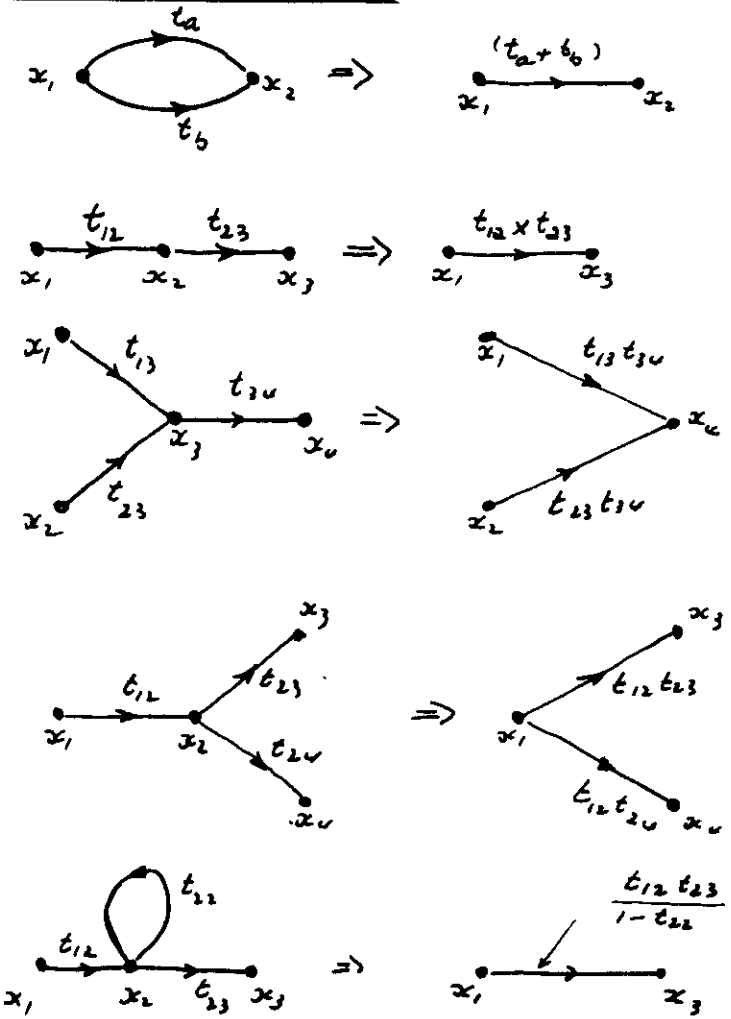


Complexity of Signal-Flow Graphs – Reduction.

The usual aim is to reduce the system graph to a single branch connecting input to output. (source to sink).

There are some elementary equivalences which facilitate this reduction process.

Signal-Flow Graph Reduction



$x_2 = t_{12} x_1 + t_{32} x_3$

$x_3 = \frac{t_{23} x_2}{1 - t_{33}}$

$x_4 = \frac{t_{23} t_{34}}{1 - t_{33}} \cdot x_1$

Using these basic equivalences, a signal-flow graph can be reduced to its elementary (single-branch) form.

Determination of Pulse Transfer Function: Mason's Rule

Given the signal-flow graph of a linear, shift-invariant, discrete-time filter in the z-domain, we can determine its transfer function in either one of two ways:-

(i) Reduce flowgraph to elementary form of source, sink and single branch. The branch transmittance is the required transfer function.

(ii) Using Mason's Rule, the $H(z)$ can be obtained directly from the flow graph (without graph reduction)

$$H(z) = \frac{1}{\Delta(z)} \sum_{k=0}^{M} P_k(z)\,\Delta_k(z)$$

where $P_k(z)$ = transmittance of the kth forward path from source to sink.

$\Delta(z)$ = 1 − (sum of all individual loop transmittances) + (sum of transmittance products of all possible sets of non-touching loops taken two at a time) − (sum of transmittance products of all possible sets of non touching loops taken three at a time) + ⋯

$\Delta_k(z)$ = value of $\Delta(z)$ for that portion of the graph not touching the kth forward path.

$\Delta(z)$ is called the determinant of the graph

$\Delta_k(z)$ is called the co-factor of forward path, k.

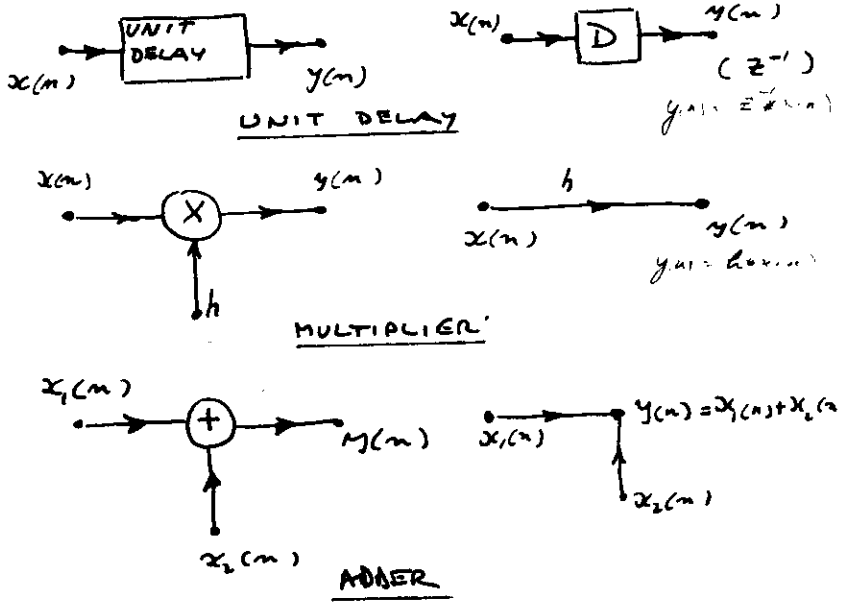
Time Domain Descriptions of Discrete-Time Filters

A discrete-time filter is some interconnection of the three basic elements

UNIT DELAY
MULTIPLIER
ADDER.

We represent these elements as follows:-

BLOCK DIAGRAM | SIGNAL-FLOW GRAPH



Finite-Impulse-Response (FIR) Filter.

The FIR filter has an output which consists of a linear combination of its present and past inputs.

Let $x(n)$ be the present input to the filter and $x(n-1), x(n-2), \ldots x(n-M)$ be its past inputs.

The input-output relationship can then be written in terms of the convolution sum

$$y(n) = \sum_{k=0}^{M} h(k)\, x(n-k)$$

where $h(k)$ is the filter impulse response.

The impulse response is the response to an input which consists of the unit pulse sequence.

The unit pulse sequence is $x(n) = \begin{cases} 1 & n = 0 \\ 0 & \text{otherwise.} \end{cases}$

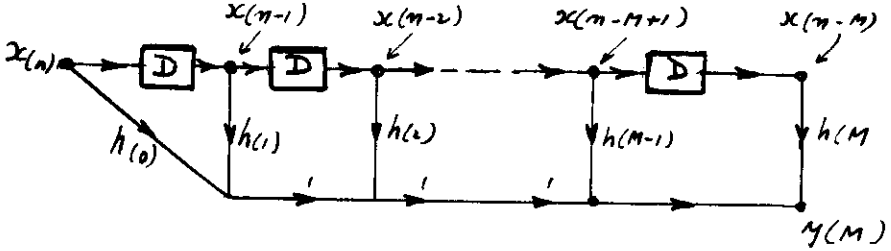
Hence the impulse response of the above filter is $h(0), h(1), \ldots, h(M)$. Because this sequence is of finite duration for a prescribed value of M, such a filter is called a FINITE-IMPULSE-RESPONSE filter.

FIR filters are also called TAPPED-DELAY-LINE filters.

ORDER.

The parameter M is called the order of the filter.

It equals the number of unit-delay elements in the filter.



FIR Filter.

$$y(n) = \sum_{k=0}^{M} h(k)\, x(n-k)$$

Infinite-Impulse-Response Filter. (I.I.R)

The output of an I.I.R. filter is made up of the present value of the input and past values of the output.

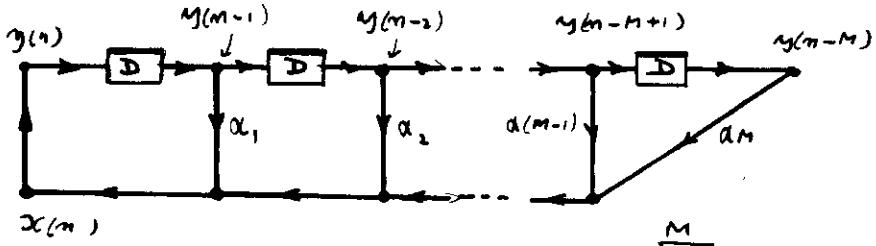
ie $$y(n) = x(n) + \sum_{k=1}^{M} \alpha_k\, y(n-k)$$

$\alpha_1, \cdots, \alpha_M$ are filter coefficients

$x(n)$ = present value of input.

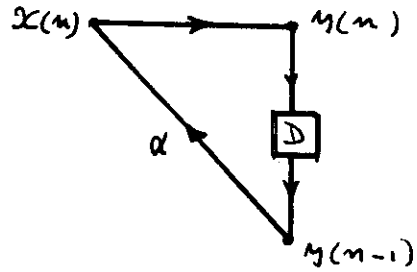
$y(n-1), \cdots y(n-M)$ are past values of the output.

The signal-flow graph for this case has FEEDBACK LOOPS.



I.I.R Filter. $$y(n) = x(n) + \sum_{k=1}^{M} \alpha_k\, y(n-k)$$

Example



The impulse response (ie the response to the unit sample sequence) is

$$h(0) = 1$$
$$h(1) = \alpha$$
$$h(2) = \alpha^2$$
$$h(3) = \alpha^3$$
$$\vdots$$

The usefulness of the signal-flow graph in these situations is obvious. It is well worth while taking the time to acquire facility with the signal-flow graph and its evaluation (examples).

Description of Discrete-time filters in the z-Domain

Consider a discrete-time F.I.R. filter whose time-domain response is given by

$$y(n) = \sum_{k=0}^{M} h(k)\, x(n-k)$$

If we take z-transforms of both sides of the above equation we get

$$Y(z) = \sum_{k=0}^{M} h(k).\, z^{-k} X(z)$$

where $X(z)$ is the z-transform of the input signal sequence, $\{x(n)\}$. This transform is usually called the EXCITATION TRANSFORM and $Y(z)$ which is the z-transform of the output sequence $\{y(n)\}$ is called the RESPONSE TRANSFORM.

The signal-flow graphs for the filter are same as before, with D replaced by z^{-1}.

z^{-1} has the role of UNIT DELAY VARIABLE.

D has the role of UNIT DELAY OPERATOR.

[PLEASE! NEVER DECIDE TO WRITE $D = z^{-1}$]

TRANSFER FUNCTION

$$H(z) = \frac{Y(z)}{X(z)} = \sum_{k=0}^{M} h(k)\, z^{-k}$$

the R.H.S of the above equation is the z-transform of $h(k)$, the filter impulse response.

IMPORTANT:

The transfer function of a linear shift-invariant discrete-time filter is the z-transform of the impulse-response of the filter.

I.I.R. Filter Time-domain response is given by

$$y(n) = x(n) + \sum_{k=1}^{M} a_k\, y(n-k)$$

Again, we take the z-transform of both sides of the equation and find

$$Y(z) = X(z) + \sum_{k=1}^{M} a_k z^{-k} Y(z)$$

and the signal-flow graph is the same as before with z^{-1} replacing D.

Now to find $H(z)$ we rearrange the above equation and reach

$$H(z) = 1 \Big/ \left(1 - \sum_{k=1}^{M} a_k z^{-k}\right)$$ which

is very much different from the response of the FIR filter.

All-pass filters and minimum phase filters.

These terms should be already familiar from previous work [Filter Course].

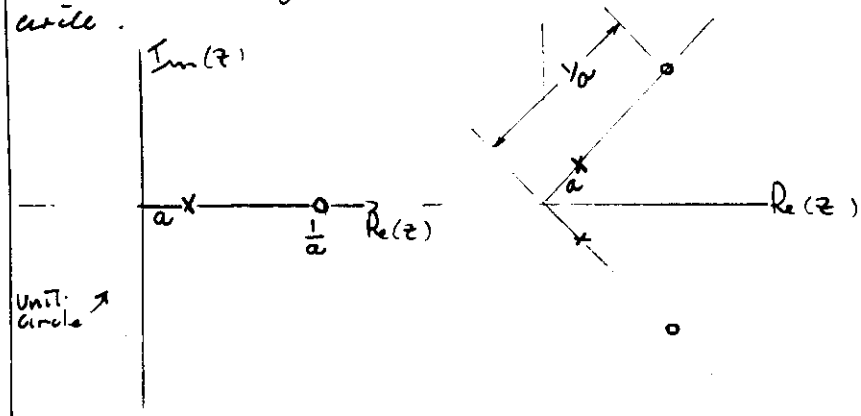
We can describe an all-pass filter (discrete-time) in terms of its time-domain or its complex z-domain behaviour.

A discrete-time filter characterized by an (time domain) impulse response $\{f(n)\}$ is said to be all-pass if it satisfies the following conditions:-

1. $f(n)$ is real for all n
2. $f(n) = 0$ for $n < -$ (causal)
3. $|f(0)| < 1$
4. $\sum_{n=0}^{\infty} f^2(n) = 1$

A discrete-time filter characterized in the complex z-domain by the transfer function $F(z)$ is all pass if it satisfies :-

1. The poles and zeros of $F(z)$ are real or else occur in complex-conjugate pairs.
2. The poles of $F(z)$ lie inside the unit circle
3. $F(z)$ has no poles or zeros at $z = 0$
4. The amplitude response $|F(e^{j\omega})|$ evaluated on the unit circle in the z-plane $= 1$ for all ω. This requires that $F(z)$ has an equal number of poles and zeros which are inversely related to each other and the unit circle.



ALL-PASS ORDER 1 ALL-PASS ORDER 2.

NOTE. Any all-pass filter can be realised as a cascade of order-1 and order-2 sections.

Minimum Phase Filters.

A discrete time filter characterized by the transfer function $H(z)$ is said to be minimum phase if.

(i) the poles of $H(z)$ lie neither inside the unit circle.
(ii) the zeros of $H(z)$ lie neither inside the unit circle or on the unit circle.

As a result, for a given $|H(e^{j\omega})|$ the phase response of the filter assumes the minimum possible value. — hence the term "MINIMUM PHASE" filter.

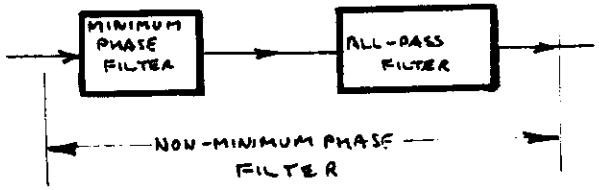
For minimum-phase filters, there is a unique relationship between the magnitude and phase functions. If either function is specified $0 < \omega < \infty$, the other function is uniquely determined.

This relationship is destroyed if any of the zeros of the transfer function lie outside the unit circle of the Z-plane. In such a case the filter is said to be NON-MINIMUM PHASE.

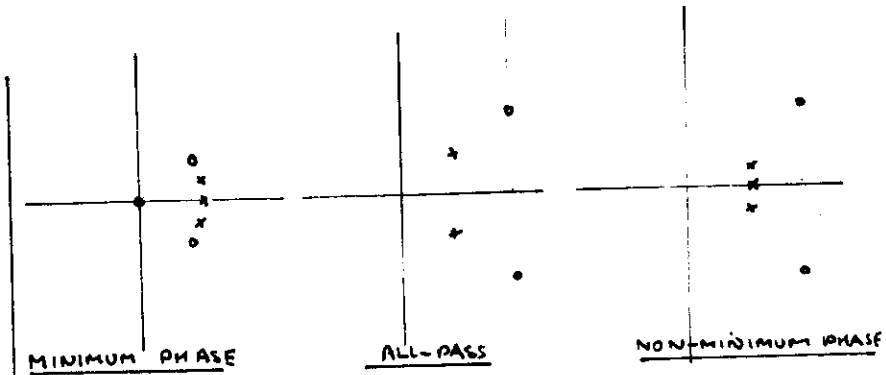
Factorization: We can factorize $H'(z)$ the transfer function of a non-minimum phase filter

$$H'(z) = F(z)\,H(z) \text{ where}$$

$F(z)$ is an all-pass component.
$H(z)$ is a minimum phase component.



This arrangement can be visualised simply from the Z-plane plots overleaf



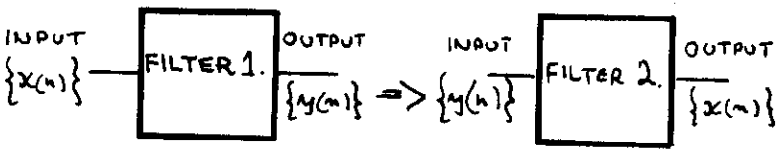
Showing production of a non-minimum phase response by cascade of an all-pass and a minimum-phase network.

Note that the multiplication of the transfer functions results in the addition of the pole-zero Z-plane patterns. Superposing a pole and a zero results in cancellation ie $\frac{(\checkmark)}{(\checkmark)}$.

Maximum Phase network.
When the zeros of the transfer function of a non-minimum phase network are ALL located outside the unit circle the filter is said to be maximum phase. Consequently, for a prescribed amplitude response $|H(z)|$, the phase of the filter assumes the maximum value possible.

Inverse Filters.

A pair of filters are said to be the inverse of each other, if, when connected in cascade their combined impulse response is equal to the unit sample sequence.



For linear, discrete-time filters which are shift-invariant the reproduction of the input $\{x(n)\}$ at the output of the second filter means that there is a one-to-one correspondence between the input and output signals of the filter.

Condition for Invertibility of a Filter.

Let $H(z)$ be the transfer function of FILTER 1 above. Then $Y(z) = H(z)X(z)$ where $X(z)$ is the z-transform of $\{x(n)\}$.

Similarly if $G(z)$ is the transfer function of FILTER 2.

$$X(z) = G(z)Y(z)$$

For the filters to be inversely related

$$H(z)G(z) = 1$$

i.e. $$G(z) = \frac{1}{H(z)}$$

This means that the poles of $G(z)$ must be the same as the zeros of $H(z)$ and vice versa. Based on this fact we can assume the following.

1. A minimum-phase filter has an inverse, provided that its transfer function does not have any zeros on the unit circle in the z-plane, and the inverse filter itself is minimum-phase.

2. Because the transfer function of a non-minimum phase filter has zeros outside the unit circle, such a filter does not have an inverse.

DIGITAL SIGNAL PROCESSING.

THE FOLLOWING TEXTS ARE RELEVANT TO THE COURSE IN DIGITAL SIGNALPROCESSING.

1.
COMMUNICATION SYSTEMS
A.Bruce Carlson
McGraw Hill ISBN 0-07-100560-9

2.
DIGITAL SIGNAL PROCESSING
Alan V. Oppenheim & Ronald W. Schafer
Prentice-Hall
ISBN 0-13-214107-8 01

3.
DIGITAL AND ANALOG COMMUNICATION SYSTEMS
Leon W. Couch II
Maxwell Macmillan
ISBN 0-02-946062-X

4.
MODERN FILTERS
Simon Haykin
Maxwell Macmillan
ISBN 0-02-946153-7

5.
PRINCIPLES OF ANALOG AND DIGITAL COMMUNICATION
Jerry D. Gibson
Maxwell Macmillan
ISBN 0-02-946300-9

6.
SIGNALS AND SYSTEMS: CDONTINUOUS AND DISCRETE
Rodger E. Ziemer
William H. Tranter
D. Ronald Fannin
Maxwell Macmillan
ISBN 0-02-946104-9

7.
INTRODUCTION TO DIGITAL SIGNAL PROCESSING
John G. Proakis
Dimitris G. Manolakis
Maxwell Macmillan
ISBN0-02-946253-3

8.
INTRODUCTION TO DIGITAL COMMUNICATION
Rodger E. Ziemer
Roger L.Peterson
Mazwell Macmillan
ISBN0-02-946431-5

DIGITAL SIGNAL PROCESSING Laboratories 1994
Course 84182

Lab 1
Response Magnitude, Phase and Group Delay, Pole-Zero Locations

Lab 2
Convolution Sums

Lab 3
DSP Hardware - ADSP-2101 Development Tools

Lab 4
Elementary DSP Operations with ADSP-2101

Lab 5
MATLAB and Signal Processing Toolbox
IIR Design Using Analogue Prototype
Direct IIR Filter Design

Lab 6
IIR Filter Implementations on ADSP-2101

Lab 7
MATLAB and Signal Processing Toolbox
Direct FIR Filter Design
Parkes-McClellan Optimal FIR Filter Design

Lab 8
FIR Filter Implementations on ADSP-2101

Lab 9
MATLAB FFT
Spectral Analysis and Windows

Lab 10
Fast Fourier Transform and Spectrum Analyser based on ADSP-2101

Lab 11 & 12
DSP Applications
Use of FFT Algorithms in Spectral Analysis and Other Calculations
Professional DSP Hardware and Software - demonstration

Time to complete each laboratory : 3 hours

Assignment:

Prepare a program in C language (DOS environment) performing a 256-point FFT using the Radix-2 Decimation-In-Time method.

The assignment is due before you start the laboratory 10.

Laboratory 3
DSP Hardware - ADSP-2101 Development Tools

INTRODUCTION:

The ADSP-2101 Development Tools appendix contains brief yet sufficient information on the ADSP-2101 system that will be used throughout the Digital Signal Processing Course.

Before you start the lab you have to be familiar with the Appendix - section 1 and 3, and System Builder - attached to this lab sheet.

Part A System Builder

Prepare a system specification file (name.SYS) for the system to be used during all DSP labs, i.e.:

- boot sequence enabled
- 32K EPROM (8-bit) containing boot memory is connected
- full internal Program Memory will be used
- full internal Data Memory will be used
- 8 Kwords of external Data Memory
- 4 D/A converters at data memory locations 1000h through 1003h
- load DAC register at data memory location 2000h

The above file is your prework and will be checked by the lab demonstrator at the beginning of the lab.

Full documentation about the target system can be found in the Appendix - section 4.

Generate an architecture file (name.ACH) by executing System Builder. This file will be required to run Linker and Simulator.

Part B Cross-Assembler and Emulator

Type in the below program. The comments can be skipped.
In this program, an audio signal from a microphone is passed to a speaker through the ADSP-2101 system.

```
{ ADSP-2101 Talk Through Program                    TALKTHRU.DSP

This program takes an input sample from serial port 0 receive register    and
outputs it to serial port 0 transmit register. It is intended for speech signals.
The serial clock is internally generated and 12.288 MHz processor rate provides
8000 Hz sampling rate. It can be used for testing the hardware.
}

.MODULE/RAM/ABS=0 TALKTHRU;          { Beginning of TALKTHRU Program }
      JUMP start; NOP; NOP; NOP;     { Start Interrupt }
      RTI; NOP; NOP; NOP;            { External Pin Interrupt IRQ2 }
      RTI; NOP; NOP; NOP;            { SPORT0 Transmit Interrupt }
      JUMP sample; NOP; NOP; NOP;    { SPORT0 Receive Interrupt }
      RTI; NOP; NOP; NOP; NOP;       { SPORT1 Transmit Interrupt }
      RTI; NOP; NOP; NOP; NOP;       { SPORT1 Receive Interrupt }
      RTI; NOP; NOP; NOP;            { TIMER Interrupt }

      { initializations }

start: toggle flag_out;              { Lights FLAG LED}
      AX0=0x00B0;
      DM(0x3FFE)=AX0;                { All DM 2 Wait States  }
      AX0=0x0000;
      DM(0x3FFB)=AX0;                { TIMER not used, cleared }
      DM(0x3FFC)=AX0;
      DM(0x3FFD)=AX0;
      DM(0x3FF9)=AX0;                { Receive Multichannels }
      DM(0x3FFA)=AX0;
      DM(0x3FF7)=AX0;                { Transmit Multichannels }
      DM(0x3FF8)=AX0;
      AX0=0x6B27;                    { Multichannel disabled }
      DM(0x3FF6)=AX0;                { Int. gen serial clock }
                                   { Receive frame sync required, width 0 }
                                   { Transmit frame sync required, width 0 }
                                   { Int trans, receive frame sync enabled }
                                   { u-law companding, 8 bit word length }

      AX0=0x0002;
      DM(0x3FF5)=AX0;                { Generate 2.048 MHz serial clock }
      AX0=255;
      DM(0x3FF4)=AX0;                { Divide by 256 for 8KHz sampling rate }
      Ax0=0x0000;
      DM(0x3FF3)=AX0;                { SPORT0 AUTOBUFF disabled }
      DM(0x3FF2)=AX0;                { SPORT1 CNTL disabled }
      DM(0x3FF1)=AX0;                { SPORT1 timer not used }
      DM(0x3FF0)=AX0;                { SPORT1 timer not used }
      DM(0x3FEF)=AX0;                { SPORT1 AUTOBUFF disabled }
      AX0=0x1000;
      DM(0x3FFF)=AX0;                { SPORT0 enabled, PM Wait State 0, }
                                   { BOOT Wait State 0, BOOT page 0  }
      IMASK=b#001000;                { Enable SPORT0 Interrupt }
      ICNTL=b#00111;                 { Enable Edge Sensitive Interrupt }

wait:  IDLE;                         { Wait until next sample appears at }
      JUMP wait;                     {                          SPORT0.}

sample: AX0=RX0;                     { Put received sample in AX0 }
      TX0=AX0;                       { Transmit sample value in AX0 }
      RTI;                           { Return from Interrupt }

.ENDMOD;                             { End of TALKTHRU Program }
```

This program is written for EZ-ICE (Emulator) and the EZ-LAB system with the architecture file you prepared in Part A.
Assemble the program using ASM21.EXE and link using LD21.EXE to produce TALKTHRU.EXE.

Apply +12, -12 and +5V to EZ-LAB; connect ANALOG and DIGITAL GROUND together.

NB: Wrong supply voltages may result in damaging the board.

Connect the microphone and speaker to EZ-LAB. Load TALKTHRU.EXE to EZ-ICE and execute.

Part C PROM Splitter

For TALKTHRU.EXE, generate PROM burn files with the boot loader option in Intel Hex data format. To program an EPROM (27256) use the file with the .BNM extension. This EPROM will replace the existing one on the board.
Turn power on and the program will be executed automatically.

Part D C Compiler

Type in the below program written in C language.

```
(*      conv.c                                  *)
(* calculates convolution : N points by M point *)
#include <adsp2101.h>

#define N 5
#define M 5
int x[N] = { 6, 6, 6, 6, 6 };
int h[M] = { 6, 5, 4, 3, 2 };
int y[N+M-1];

void main()
{
 int i,m,n;

 for ( i=0; i<N+M-1; i++ )
     y[i] = 0;

 for ( n=0; n<N; n++ )
      for ( m=0; m<M; m++ )
       {
          i=n+m-1;
          y[i] += x[n]*h[m];
       }
}
```

Compile the above program using CC21.EXE and link using LD21.EXE to produce an executable file.
Invoke the simulator SIM_2101.EXE, run CONV.EXE, and check the results.

System Builder 2

2.1 INTRODUCTION

The system builder is a software tool for describing your hardware environment. Each ADSP-21xx system can have a unique hardware configuration and may use different amounts of memory. The system builder output specifies your hardware configuration, including memory and parallel I/O ports, in a file format read by the linker and simulators. The linker requires this information in order to allocate code and data storage to the available memory space. The simulators must accurately model your system architecture and the ADSP-21xx processor it is based on.

The system builder may also be used to preset the memory map for an ADSP-21xx emulator. See the section of this chapter called "Segment Mapping For Emulator" for details.

Figure 2.1 shows the memory configurations available—the maximum number of addressable words in each memory space—for each processor. The system builder will only allow you to create system architecture descriptions within these limits.

	ADSP-2100 (all memory external)	ADSP-2105 ADSP-2115	ADSP-2101 ADSP-2111 ADSP-21msp50
Data Memory (16-bit data)	16K maximum	14.5K maximum (.5K internal, 14K external)	15K maximum (1K internal, 14K external)
Program Memory (24-bit code, 16/24-bit data)	16K mixed code & data or 32K (16K code, 16K data)	15K maximum (1K internal, 14K external)	16K maximum (2K internal, 14K external)
Boot Memory (24-bit code, 16/24-bit data, padded to 32-bit word width)	——	8K maximum (32K bytes organized in 32-bit words)	16K maximum (64K bytes organized in 32-bit words)

Figure 2.1 ADSP-2100 Family Memory Configurations

2 System Builder

Each memory space is addressed separately. Addresses in program memory are different from addresses in data memory. Boot memory addresses are unique and are used only by the processor during booting. (Note: Boot memory does not exist for the ADSP-2100.)

You must write a **system specification source file** as input to the system builder; this file describes your target hardware. The system builder directives described in this chapter are used to write the file.

The system builder processes the input file and generates an **architecture description file** with the filename extension **.ACH**. The architecture description file is interpreted by the linker in order to place relocatable code and data fragments in memory. The file is also read by the simulator to model the system memory configuration and by the emulator to set up target system memory-mapping. The system builder outputs error messages if any are detected, otherwise a summary of the architecture is displayed on the screen. You should use operating system commands to capture this output into a file if you need to refer to it for debugging or documentation purposes.

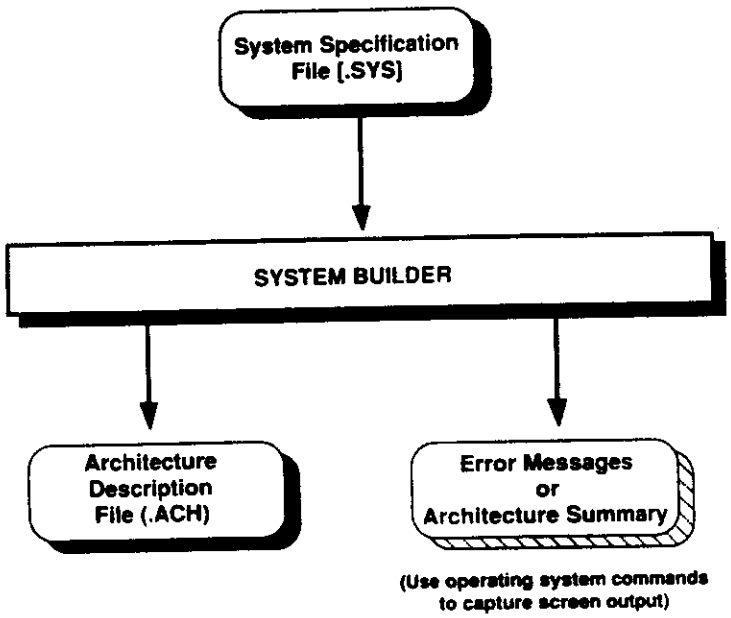


Figure 2.2 System Builder I/O

2.1.1 Memory-Mapped Control Registers

All ADSP-21xx processors except the ADSP-2100 have a set of memory-mapped control registers which configure various modes of processor operation. These registers are located in the reserved portion of internal data memory on each chip. This memory space is the top 1K of internal DM, addresses 0x3C00–0x3FFF.

You may not declare a memory segment in this space, and the control registers cannot be defined in the system builder input file. Instead, each register must be initialized in your assembly language programs by writing a data word to the appropriate address. (Writes and reads to the register locations are allowed even though this segment of memory cannot be declared with the system builder's .SEG directive.) The address and format of each control register is given in Appendix E of this manual and in the Control/Status Registers appendix of the *ADSP-2100 Family User's Manual*.

See the section "Setting the Memory-Mapped Control Registers" in Chapter 3 for further information on this topic.

2.2 RUNNING THE SYSTEM BUILDER

To invoke the system builder from your operating system, type:

BLD21 *filename*[.*ext*] [–c]

where *filename* is your system specification source file. The filename may have an extension; if none is present, the system builder appends the default extension .SYS to the filename. The system builder will generate an output architecture description file called *filename*.ACH, using the filename of your input.

There is one optional command line switch for the system builder. The –c switch makes the system builder case-sensitive, preserving your usage of upper and lower-case characters. This option is provided primarily for compatibility with the ADSP-2100 Family C Compiler, which is always case-sensitive.

If the –c switch is not used, the system builder output is in all uppercase. You must use this switch in order to preserve any lower-case characters entered. This is necessary if the assembler is to be run with its case-sensitive switch, as is required when assembling compiled C code. If you

refer (in assembly language code) to a memory segment declared for the system builder in lower-case, and the assembler is run in case-sensitive mode, the segment name will not be recognized unless it's case is preserved by the system builder.

If you forget the syntax for invoking the system builder, type:

BLD21 –help

This will show you how the command must be entered. The **–help** switch works with all of the development software tools.

2.3 SYMBOL USAGE & RESERVED KEYWORDS

In the system specification file you assign symbolic names to the target system itself, to memory segments, and to memory-mapped I/O ports. You can use the memory segment names in your program to assign code and data fragments to that segment. The assignments are passed from the assembler to linker, which determines the exact placement of your program in memory.

All symbolic names must be unique. A symbolic name is a string of letters, digits, and underscores with a letter or underscore as the first character. A small group of symbols are reserved for use as keywords by the system builder—you may not use these symbols in your system specification file. Table 2.1 lists the system builder keywords.

ABS	CODE	PM
ADSP2100	CONST	PORT
ADSP2101	DATA	RAM
ADSP2105	DM	ROM
ADSP2111	ENDSYS	SEG
ADSP2150	MMAP0	SYSTEM
BOOT	MMAP1	

Table 2.1 System Builder Keywords

Assembler keywords, listed in Chapter 3, are also reserved for use by the development software. Please consult this list also before you choose names for your system components. If you use a reserved keyword the linker will generate errors when you attempt to link your program.

2.4 SYSTEM SPECIFICATION FILE

The system specification source file specifies the amount of data and program memory in your system. For those processors with boot memory, the file also declares each page of boot memory which will be used. Comment fields are enclosed within braces, { }, and may be placed anywhere in the file. Nested comments are not allowed.

You can generate your file with any editor that produces plain text files. Do not use a word processor which embeds special control codes.

2.4.1 ADSP-2100 Example File

Figure 2.3 is an example of a system specification source file for an ADSP-2100 system. (The term "ADSP-2100" is used to denote both the ADSP-2100 and ADSP-2100A processors.)

```
.SYSTEM fir_system;                                  {system name}
.ADSP2100;                                           {specifies processor}
.SEG/PM/ROM/ABS=0/CODE  prog_mem[4096];              {code storage}
.SEG/PM/RAM/ABS=4096/DATA  coeff_table[15];          {coefficients table}
.SEG/DM/RAM/ABS=0/DATA  delay_line[15];              {data storage}
.PORT/DM/ABS=16382  ad_sample;                       {memory-mapped port}
.PORT/DM/ABS=16383  da_data;                         {memory-mapped port}
.ENDSYS;
```

Figure 2.3 ADSP-2100 System Specification File

The first directive in the file is the .SYSTEM directive. This directive assigns the name *fir_system* to the architecture description and marks the start of the file.

The .ADSP2100 statement identifies the processor type, here naming the ADSP-2100 microprocessor. This statement is required.

The .SEG directives declare the system memory segments and their characteristics. The memory segments can be declared in any order. In this example three segments are declared.

The first, *prog_mem*, is a 4K-word segment which will store program code. The *coeff_table* segment is 15-word block of program memory declared to store data; the data will be a set of FIR filter coefficients. The third segment, *delay_line*, is needed to store intermediate data of the filter algorithm. This segment exists in ADSP-2100 data memory.

The two .PORT directives declare memory-mapped I/O ports for system input and output. The port names *ad_sample* and *da_data* suggest external connections to analog-to-digital and digital-to-analog converters. The ports are located at the data memory addresses given with the ABS (absolute address) qualifier. The port names become program symbols which can be used in code to read or write to the ports.

The last statement in a system specification file is the .ENDSYS directive. The system builder stops processing when it encounters the .ENDSYS directive.

2.4.2 ADSP-2101 Example File

Figure 2.4 is an example of a system specification source file for an ADSP-2101 system.

The first directive in the file is the .SYSTEM directive. This directive assigns the name *fir_system* to the architecture description and marks the start of the file.

The .ADSP2101 statement identifies the processor type, here naming the ADSP-2101 microcomputer. This statement is required.

The .MMAP0 directive specifies the state of the MMAP pin on the ADSP-2101 device in this system. Defining MMAP as 0 indicates that boot memory is to be loaded into the chip's internal program memory, beginning at address 0x0000.

The .SEG directive declares the system memory segments and their characteristics. The memory segments can be declared in any order. In this example, the segments declared comprise the full on-chip and off-chip program and data memory configuration of the ADSP-2101.

```
.SYSTEM fir_system;                                  {system name}
.ADSP2101;                                           {specifies processor}
.MMAP0;                                              {boot loading enable}
.SEG/ROM/BOOT=0  boot_mem[2048];                     {boot page zero}
.SEG/PM/RAM/ABS=0/CODE/DATA  int_pm[2048];           {internal program mem}
.SEG/PM/RAM/ABS=2048/CODE/DATA  ext_pm[14336];       {external program mem}
.SEG/DM/RAM/ABS=0/DATA  ext_dm[14336];               {external data mem}
.SEG/DM/RAM/ABS=14336/DATA  int_dm[1024];            {internal data mem}
.ENDSYS;
```

Figure 2.4 ADSP-2101 System Specification File

(Note: Referring to program memory and data memory does not include boot memory, which should be thought of as a unique memory space in the system architecture.)

The *boot_mem* segment is a 2K-word segment for one page of boot memory. If this system was based on the ADSP-2105 rather than ADSP-2101, the boot page segment would be 1K (or less) in size.

The *int_pm* declaration identifies the 2K-word on-chip program memory space starting at absolute address 0. In the ADSP-2101 (as well as ADSP-2105, ADSP-2111, and ADSP-21msp50) this memory can store both code and data and should be explicitly declared in this way. The following statement line declares *ext_pm* as a 14K-word segment for off-chip program memory, starting at address 2048, which may also store code and data.

Next, *ext_dm* is declared as a 14K-word segment for off-chip data storage starting at address 0 (in data memory, as opposed to address 0 in program memory). The *int_dm* segment declares the 1K-word on-chip data memory space starting at address 14336. The 1K of on-chip data memory above this is reserved for memory-mapped control registers and may not be declared as a segment.

The last statement in a system specification file is the .ENDSYS directive. The system builder stops processing when it encounters the .ENDSYS directive.

2.5 SYSTEM BUILDER DIRECTIVES

This section describes each system builder directive and its syntax. The format of some directives requires arguments and qualifiers. Qualifiers immediately follow the directive and are separated by slashes; arguments follow the qualifiers. The general form of directives is:

.DIRECTIVE/qualifier/qualifier ... *argument;*

2.5.1 Naming Your System (.SYSTEM)

The .SYSTEM directive must be the first statement in the system specification source file. You name your ADSP-21xx system with the symbol given as an argument for this directive. The system name will be displayed in the simulator.

The .SYSTEM directive has the form:

.SYSTEM *system_name;*

The .ENDSYS directive must be the last statement in the file. System builder processing terminates at the .ENDSYS directive.

The .ENDSYS directive has the form:

.ENDSYS;

2.5.2 Identifying The Processor (.ADSP21XX)

This directive identifies which ADSP-2100 Family processor is used in your system. This information is passed to the linker and simulator via the .ACH output file. The linker is then able to allocate code and data storage according to the addressable memory limits of each processor. This directive takes one of the following forms:

.ADSP2100;	*used for ADSP-2100 & ADSP-2100A*
.ADSP2101;	
.ADSP2105;	
.ADSP2111;	
.ADSP2150;	*used for ADSP-21msp50 & ADSP-21msp55*
.ADSP2151;	*used for ADSP-21msp51 & ADSP-21msp56*
.ADSP2101MV;	*used for ADSP-2101 memory-variant processor (e.g. ADSP-2115)*
.ADSP2101P;	*used for ADSP-2101 paged memory system*

When the ADSP-21msp50 Simulator is invoked with an ADSP-21msp51 architecture file, the simulator automatically configures itself for the on-chip memory map of the ADSP-21msp51. If an ADSP-21msp51 architecture file is used and the ROMENABLE bit is set to 1, the simulator configures program memory locations PM[0x800] – PM[0x1000] as ROM. The ROMENABLE bit is displayed in the simulator's Control Registers window.

2.5.3 MMAP Pin (.MMAP)

This directive is used only for the ADSP-2100 Family processors which have on-chip memory, boot memory, and an MMAP pin (i.e. all except the ADSP-2100). The directive specifies the logic state of the processor's MMAP pin in the target system.

This directive takes one of two forms:

.MMAP0	*MMAP pin held low*
.MMAP1	*MMAP pin held high*

If .MMAP0 is used, boot loading takes place at reset and on-chip program memory starts at address 0x0000. If .MMAP1 is used, boot loading is disabled and on-chip program memory is mapped to the top (highest addresses) of program memory space.

When this directive is omitted, the simulator default is MMAP=1.

2.5.4 Memory Segment Declarations (.SEG)

The .SEG directive defines a specific section of system memory and describes its attributes. There is no default memory map—you must define all system memory with .SEG directives. This information is passed to the linker, simulator, and emulator via the .ACH output file.

The .SEG directive has the form:

.SEG/qualifier/qualifier ... *seg_name[length];*

The segment is assigned the symbolic name *seg_name*. Assigning a name to the segment allows you to explicitly place code and data fragments in it. This is accomplished in assembly language with the assembler's SEG qualifier.

You must specify the segment length inside brackets. This value is interpreted as the number of words (either 16-bit data or 24-bit instructions) in the segment.

Data memory segment size in bytes is 2x the word count while program memory segment size in bytes is 3x the word count. Boot memory segment size in bytes is 4x the word count, due the padding of boot memory with an extra byte per word in order to place the beginning of each word on an even byte boundary (see Chapter 5, PROM Splitter, for further details).

The following two qualifiers are required for the .SEG directive:

PM *or* DM *or* BOOT=0–7	*memory space*
RAM *or* ROM	*memory type*

Four others are optional:

ABS=address	*absolute start address*
DATA *or* CODE *or* DATA/CODE	*what is stored in segment*
EMULATOR *or* TARGET	*preset emulator memory map*
INTERNAL	*located in on-chip memory of ADSP-2101 memory-variant processor*

The PM/DM/BOOT qualifier indicates which memory space the segment is in: program memory, data memory, or boot memory. (Remember that boot memory space does not exist for the ADSP-2100.) The remaining qualifiers specify the memory type, the starting address of the segment, what is stored (DATA and/or CODE), and how the emulator's memory map is preset.

If you are using one of the ADSP-21xx emulators, see "Segment Mapping For Emulator" below.

Each memory space is addressed separately: address 16 in program memory is different from address 16 in data memory.

PM memory segments can store CODE only, DATA only, or both CODE and DATA. If you give neither of these qualifiers the default is to CODE. For a PM segment to contain code and data, both qualifiers must be used. The ADSP-21xx processors require that any data transfers to or from program memory must be made with segments which have the DATA attribute. If your system requires that executable code be written or read by the processor, the segments to be accessed should be declared with both the CODE and DATA qualifiers.

DM memory segments must be DATA only; this is the default if the DATA qualifier is omitted. An error is generated if a DM segment is assigned the CODE attribute.

BOOT memory segments default to both CODE and DATA since boot memory will store both in most systems, and the qualifiers may be omitted. The BOOT qualifier must specify one page number only: for example, BOOT=0. You must have a separate segment declaration for each boot page of your system.

A system may have up to 8 boot pages, with page numbers from 0 to 7. Each ADSP-2101, ADSP-2111, and ADSP-21msp50 boot page can store up to 2K words of code and data. Each ADSP-2105 and ADSP-2115 boot page stores up to 1K words. Do not use the ABS qualifier for boot pages—the system builder assigns appropriate boundary addresses.

2.5.4.1 .SEG Directive Examples

The example

```
.SEG/PM/RAM/ABS=0/CODE/DATA  restart[2048];
```

declares a program memory RAM segment called *restart* which is located at address 0. The segment may hold 2048 words of code and data.

The example

```
.SEG/ROM/BOOT=0   boot_mem[1536];
```

declares the boot memory segment *boot_mem* which is located on boot page 0 (automatically corresponding to address 0 in boot memory). The length of the segment is 1536 words. Boot memory segments should always be ROM.

2.5.4.2 Segment Mapping For Emulator

The system builder allows you to preset the memory map for an ADSP-21xx emulator via the .ACH file. Two optional qualifiers of the .SEG directive are used for this purpose:

/EMULATOR *maps segment to emulator overlay memory*

or

/TARGET *maps segment to target board memory*

These settings will configure the emulator's memory map when the emulator program is invoked. The segments must be mapped in blocks of 1K or larger. Once the emulator is running you can change the memory map with emulator commands.

If you are using the emulator in standalone mode (i.e. without a target board), you must map all memory segments to EMULATOR. You may also want to map all segments to EMULATOR during the initial stages of hardware/firmware integration. Consult your *ADSP-21xx Emulator Manual* for further information.

2.5.5 Memory-Mapped I/O Ports (.PORT)

The .PORT directive declares a memory-mapped I/O port. You must assign a unique name and address to each port in your system. Ports can be located in either data or program memory. For those processors with internal memory, ports may be assigned addresses in external memory only.

The .PORT directive takes one of two forms:

.PORT/DM/ABS=address *port_name;*

or

.PORT/PM/ABS=address *port_name;*

The DM qualifier indicates that the port is located in data memory; PM indicates placement in program memory. If neither qualifier is used, the default is to DM. The port is located at the absolute address you specify (with the ABS qualifier), and is assigned the symbol *port_name*. This symbol can be used in assembly language instructions to access the port.

For example,

```
.PORT/DM/ABS=0x0400    ad_sample;
```

declares a port named *ad_sample* which is located at data memory address 1024 (decimal). Assembly code references to this symbol are interpreted by the linker based on the contents of the .ACH file.

Data memory-mapped ports allow 16-bit read/writes while program memory-mapped ports allow either 16 or 24-bit transfers. (Refer to the *ADSP-2100 Family User's Manual* for a description of 24-bit data transfers in program memory.)

2.5.6 System Builder Constants (.CONST)

The .CONST directive defines system builder constants. Once you declare a symbolic constant you may use it in place of the actual number. This symbol definition is recognized only by the system builder, however—it is not carried over to the assembler or simulator.

The .CONST directive has the form:

.CONST *constant_name* = constant *or* expression, ... ;

Only an arithmetic or logical operation on two or more integer constants may be given as an expression; symbols are not allowed. See "Assembler Expressions" in Chapter 1 for an exact definition of allowed expressions.

A single .CONST directive may contain one or more constant declarations, separated by commas, on a single line. A list of multiple declarations may not be continued on the following line.

Laboratory 4
Elementary DSP operations

INTRODUCTION

Elementary DSP operations are: delay (data move), scale (constant multiplier), and add operations.
The delay operation provides a shift of one to several sample-intervals or even shifts of a few seconds. In the Delay section, we will simulate an acoustic delay-line by delaying the input signal up to 1 second.
The constant multiplier scales a sample while the adder adds two signal samples to form another sample. In fact almost all DSP algorithms can be done using these three elementary operations and the ADSP-2101 processor is optimised for these operations.

However there are operations other then the above which are required by DSP applications. Transcendental functions such as sines and logarithms are often approximated by polynomial expansions. The most widely used of these are the Taylor and Maclaurin series. They can be used to approximate almost any function whose derivative is defined over the specified input range.

The most important programming concept in this experiment is the circular buffer. The incoming data is stored in the buffer whose length is equal to the total amount of delay (in number of samples) we want to generate. The stored value in a buffer location is first transmitted to the serial port and then the incoming sample value is stored at the buffer location. The buffer location is advanced by one and the process continues. Due to the circular nature of the buffer. The stored sample is read out after one rotation(equal to the length) of the buffer thus providing necessary delay. The addressing of the circular buffer is provided by the index, modify and length registers.

Part A Delay

In this part, we will demonstrate the delay operation using the Emulator.
If x(n) is the input sample and y(n) is the output sample then the operation performed by this program is
y(n) = x(n-N) where N is the amount of delay in samples.

Type in the below program or modify TALKTHRU.DSP from lab 1.

```
{ ADSP-2101 Delay Program                    DELAY.DSP
This program takes an input sample from serial port 0 receive register,
outputs previous signal obtained from a circular buffer of 4000 elements, to
serial port 0 transmit register, stored the current sample in the same
location in the circular buffer. The serial clock is internally generated
```

```
and 12.288 MHz processor rate provides 8000 Hz sampling rate. The total
delay time = ( 1 / 8000 ) * 8000 second = 1.0 second.
}
.MODULE/RAM/ABS=0 DELAY;                { Beginning of DELAY Program }
.VAR/DM/CIRC    dm_buff[4000];          { Circular buffer with 4000 elements }

        JUMP start; NOP; NOP; NOP;      { Start Interrupt }
        RTI; NOP; NOP; NOP;             { External Pin Interrupt IRQ2 }
        RTI; NOP; NOP; NOP;             { SPORT0 Transmit Interrupt }
        JUMP sample; NOP; NOP; NOP;     { SPORT0 Receive Interrupt }
        RTI; NOP; NOP; NOP; NOP;        { SPORT1 Transmit Interrupt }
        RTI; NOP; NOP; NOP; NOP;        { SPORT1 Receive Interrupt }
        RTI; NOP; NOP; NOP;             { TIMER Interrupt }

        { initializations }

start:  AX0=0x00B0;
        DM(0x3FFE)=AX0;                 { All DM 2 Wait States  }
        AX0=0x0000;
        DM(0x3FFB)=AX0;                 { TIMER not used, cleared }
        DM(0x3FFC)=AX0;
        DM(0x3FFD)=AX0;
        DM(0x3FF9)=AX0;                 { Receive Multichannels }
        DM(0x3FFA)=AX0;
        DM(0x3FF7)=AX0;                 { Transmit Multichannels }
        DM(0x3FF8)=AX0;
        AX0=0x6B27;                     { Multichannel disabled }
        DM(0x3FF6)=AX0;                 { Int. gen serial clock }
                                { Receive frame sync required, width 0 }
                                { Transmit frame sync required, width 0 }
                                { Int trans, receive frame sync enabled }
                                { u-law companding, 8 bit word length }

        AX0=0x0002;
        DM(0x3FF5)=AX0;                 { Generate 2.048 MHz serial clock }
        AX0=255;
        DM(0x3FF4)=AX0;                 { Divide by 256 for 8KHz sampling rate }
        Ax0=0x0000;
        DM(0x3FF3)=AX0;                 { SPORT0 AUTOBUFF disabled }
        DM(0x3FF2)=AX0;                 { SPORT1 CNTL disabled }
        DM(0x3FF1)=AX0;                 { SPORT1 timer not used }
        DM(0x3FF0)=AX0;                 { SPORT1 timer not used }
        DM(0x3FEF)=AX0;                 { SPORT1 AUTOBUFF disabled }
        AX0=0x1000;
        DM(0x3FFF)=AX0;                 { SPORT0 enabled, PM Wait State 0, }
                                { BOOT Wait State 0, BOOT page 0  }
        IMASK=0x08;                     { Enable SPORT0 Interrupt }
        ICNTL=0x07;                     { Enable Edge Sensitive Interrupt }

        I0=^dm_buff;                    { Set up index for indirect addressing }
        L0=%dm_buff;                    { Set up size of circular buffer }
        M0=0;                           { No post-increment }
        M1=1;                           { Post-increment by 1 }
```

```
        CNTR=%DM_BUFF;                  { Set up counter }
        DO init_b UNTIL CE;             { Initialize ALL circ buff data to 0 }
init_b: DM(I0,M1)=0;

wait:   IDLE;                           { Wait until next sample appears at }
        JUMP wait;                      { SPORT0 }

sample: AX0=DM(I0,M0);                  { Get delayed signal from circ buffer }
        AX1=RX0;                        { Put received sample in AX0 }
        TX0=AX0;                        { Transmit sample value in AX0 }
        DM(I0,M1)=AX1;                  { Move current sample to circ buffer }
        RTI;                            { Return from Interrupt }

.ENDMOD;                                { End of DELAY Program }
```

Assemble the program and link to produce DELAY.EXE.

The ADSP-2101 microcomputer has 1K 16-bit words of data memory which can be used to store samples in a circular buffer. This can simulate delays up to 128 ms at an 8 KHz sampling rate. To generate delays up to 1 sec, we need a data memory of size 8 K which then must be an external memory.
The EZ-ICE probe has 8K 24-bit words of high speed static memory available as overlay memory. We will use this memory as data memory.
Overlay Memory begins at address 0x0000 and continues through address 0x1FFF.
Connect a microphone and speaker to EZ-LAB.
Load DELAY.EXE in EZ-ICE, enable Overlay Memory and execute the program.

Prepare the same program which can be stored in the boot memory.

Part B Echo

In this part, we will simulate the echo effect using EZ-LAB.
For one delayed reflection, the output sample y(n) is given by
y(n) = a*x(n)+b*x(n-N); a+b <=1.

Type in the below program.

```
{ ADSP-2101 Echo Program                                        ECHO.DSP

    This program takes an input sample from serial port 0 receive register,
    scale the signal down by 50%, obtained previous signal from a circular
    buffer of 4000 elements, sum up the scaled current sample and the scaled
    delayed sample, outputs to serial port 0 transmit register, stored the
    scaled current sample in the same location in the circular buffer. The
    serial clock is internally generated and 12.288 MHz processor rate
    provides 8000 Hz sampling rate.
```

```
}
.MODULE/RAM/ABS=0 ECHO;                    { Beginning of ECHO Program }
.VAR/DM/CIRC    dm_buff[4000];             { Circular buffer with 4000 element }

        JUMP start; NOP; NOP; NOP;      { Start Interrupt }
        RTI; NOP; NOP; NOP;             { External Pin Interrupt IRQ2 }
        RTI; NOP; NOP; NOP;             { SPORT0 Transmit Interrupt }
        JUMP sample; NOP; NOP; NOP;     { SPORT0 Receive Interrupt }
        RTI; NOP; NOP; NOP; NOP;        { SPORT1 Transmit Interrupt }
        RTI; NOP; NOP; NOP; NOP;        { SPORT1 Receive Interrupt }
        RTI; NOP; NOP; NOP;             { TIMER Interrupt }

        { initializations }

start:  AX0=0x00B0;
        DM(0x3FFE)=AX0;                 { All DM 2 Wait States }
        AX0=0x0000;
        DM(0x3FFB)=AX0;                 { TIMER not used, cleared }
        DM(0x3FFC)=AX0;
        DM(0x3FFD)=AX0;
        DM(0x3FF9)=AX0;                 { Receive Multichannels }
        DM(0x3FFA)=AX0;
        DM(0x3FF7)=AX0;                 { Transmit Multichannels }
        DM(0x3FF8)=AX0;
        AX0=0x6B27;                     { Multichannel disabled }
        DM(0x3FF6)=AX0;                 { Int. gen serial clock }
                                { Receive frame sync required, width 0 }
                                { Transmit frame sync required, width 0 }
                                { Int trans, receive frame sync enabled }
                                { u-law companding, 8 bit word length }

        AX0=0x0002;
        DM(0x3FF5)=AX0;                 { Generate 2.048 MHz serial clock }
        AX0=255;
        DM(0x3FF4)=AX0;                 { Divide by 256 for 8KHz sampling rate }
        Ax0=0x0000;
        DM(0x3FF3)=AX0;                 { SPORT0 AUTOBUFF disabled }
        DM(0x3FF2)=AX0;                 { SPORT1 CNTL disabled }
        DM(0x3FF1)=AX0;                 { SPORT1 timer not used }
        DM(0x3FF0)=AX0;                 { SPORT1 timer not used }
        DM(0x3FEF)=AX0;                 { SPORT1 AUTOBUFF disabled }
        AX0=0x1000;
        DM(0x3FFF)=AX0;                 { SPORT0 enabled, PM Wait State 0, }
                                { BOOT Wait State 0, BOOT page 0  }
        IMASK=0x08;                     { Enable SPORT0 Interrupt }
        ICNTL=0x07;                     { Enable Edge Sensitive Interrupt }

        I0=^dm_buff;                    { Set up index for indirect addressing }
        L0=%dm_buff;                    { Set up size of circular buffer }
        M0=0;                           { No post-increment }
        M1=1;                           { Post-increment by 1 }

        CNTR=%DM_BUFF;                  { Set up counter }
        DO init_b UNTIL CE;             { Initialize ALL circ buff data to 0 }
```

```
init_b:  DM(I0,M1)=0;

wait:    IDLE;                          { Wait until next sample appears at }
         JUMP wait;                     {                                SPORT0 }

sample:  AX0=DM(I0,M0);                 { Get delayed signal from circ buffer }
         SI=RX0;                        { Put received sample in SI }
         SR=ASHIFT SI by -1 ( HI );     { Scale down current sample by 50% }
         AY0=SR1;                       { Copy the scaled sample to AY0 }
         AR=AX0 + AY0;                  { Sum up scaled current and delay signal }
         TX0=AR;                        { Transmit sample value in AR }
         DM(I0,M1)=AY0;                 { Move current sample to circ buffer }
         RTI;                           { Return from Interrupt }

.ENDMOD;                                { End of ECHO Program }
```

The only difference between this and the previous program is in the interrupt service routine. The input sample is shifted to the right by one bit to obtain 50% scaling and added to the delayed sample (which was previously scaled by 50%) to produce the output sample. Execute this program using the Emulator. Make sure that you are using Overlay Memory and a corresponding .ACH file.

Modify this file to simulate the echo of duration 50 ms. Prepare this program to be run from the boot memory.

Part C Digital Sinewave Generation

In this experiment we will develop subroutines to implement sine function approximation from the polynomial expansion:
$\sin(x) = 3.140625x + 0.02026367x^2 - 5.325196x^3 + 0.5446778x^4 + 1.800293x^5$
The approximation is accurate for any value of x from 0 to 90 . However because $\sin(-x) = -\sin(x)$ and $\sin(x) = \sin(180 - x)$, the sine of any angle can be obtained from the sine of an angle in the first quadrant.

The subroutine, SINE.DSP, that implements this sine approximation is listed below.

```
{ ADSP-2101 Sine Program                                        SINE.DSP

This subroutine calculates sine value, ( Y = sin( X ), by using a polynomial
approximation:
sin(x)=3.140625x+0.02026367x^2-5.325196x^3+0.5446778x^4+1.800293x^5

    Calling Parameters
      AX0 = X in scaled 1.15 format
      M3  = 1
```

```
      L3  = 0

    Return Value
      AR = Y in 1.15 format


}

.MODULE/RAM SINE_FUNCTION;                  { Beginning of SINE Program }

.VAR/DM sin_coeff[5];                       { Sine Polynomial Coefficients }

.INIT   sin_coeff : H#3240, H#0053, H#AACC, H#08B7, H#1CCE;
                                            { Initialize coefficients }
.ENTRY  sin;                                { Entry point of sine function }

sin:   I3=^sin_coeff;                       { Index to coefficients }
       M3=1;                                { Post-increment by 1 }
       L3=0;                                { Linear buffer }
       AY0=H#4000;
       AR=AX0, AF=AX0 AND AY0;              { Check 2nd or 4th quad. }
       IF NE AR= -AX0;                      { If yes, negate input }
       AY0 = H#7FFF;
       AR = AR and AY0;                     { Remove sign bit }
       MY1=AR;
       MF=AR*MY1 ( RND ), MX1=DM( I3, M3 );     { MF= X * X }
       MR=MX1*MY1 ( SS ), MX1=DM( I3, M3 );     { MR= C_1 * X }
       CNTR=3;
       DO approx UNTIL CE;                  { Calculate sine value }
          MR=MR+MX1*MF (SS);
approx:    MF=AR*MF (RND ), MX1=DM( I3, M3 );
       MR=MR+MX1*MF (SS);
       SR=ASHIFT MR1 BY 3 ( HI );           { Convert to 1.15 format }
       SR = SR OR LSHIFT MR0 BY 3 ( LO );
       AR=PASS SR1;
       IF LT AR=PASS AY0;                   { Saturate if needed }
       AF=PASS AX0;
       IF LT AR=-AR;                        { Negate if needed }
       RTS;

.ENDMOD;                                    { End of SINE Program }
```

The routine accepts input values in 1.15 format; the coefficients are in 4.12 format to achieve maximum range allowed by this format. On this scale, 180 equals the maximum positive value, 0x7FFF, and -180 - the maximum negative value, 0x8000, as shown in Figure 1.

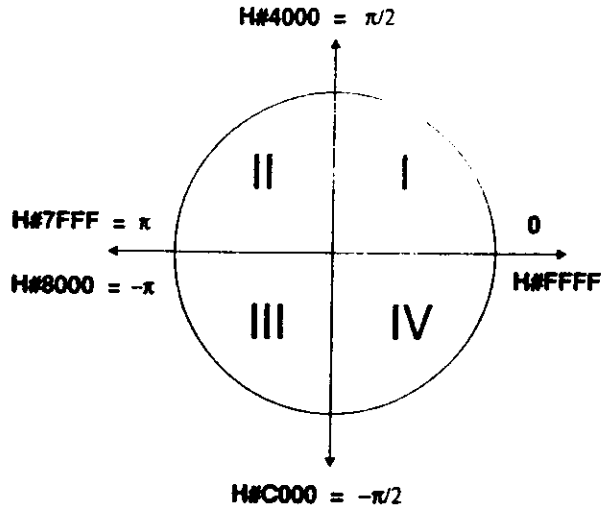


Figure 1

To produce a sine waveform we have to compute several sine values over the 0 to 2*PI interval and generate these samples at periodic intervals.
In this experiment we will generate a sinusoidal waveform at 3KHz frequency with 16 samples per cycle and display it on the scope.

Prepare the executable code from the following program.

```
{ ADSP-2101 Sine wave generator for emulator                GENER.DSP

    This program generates sine wave taking 16 angle values which are
    initialized from disk file SINE.DAT . Timer registers are used to
    get sine wave for desired frequency. Output can be displayed on scope
    from port DAC0.

}
.MODULE/RAM/ABS=0 MAIN;                    { Beginning of GENERATOR program }
.EXTERNAL sin;                             { External sine wave generator }
.port   write_dac0;                        { Memory Mapped DAC port at 0x1000 }
.port   load_dac;                          { Memory Mapped port at 0x2000 }
.VAR/DM/CIRC    sine_data[16];             { Define sine data }
.INIT   sine_data: <sine.dat>;             { Initialize sine data }

{ Interrupt Vectors }

      JUMP start; NOP; NOP; NOP;       { Start Interrupt }
      RTI; NOP; NOP; NOP;
      RTI; NOP; NOP; NOP;
      RTI; NOP; NOP; NOP;
      RTI; NOP; NOP; NOP;
      RTI; NOP; NOP; NOP;
```

```
      JUMP serv; NOP; NOP; NOP;        { Timer Interrupt }

{ Initialization }

start: I0=0x3fef;
      M0=1;
      L0=0;
      CNTR=17;                         { Clear all the control reg }
      DO clr_reg UNTIL CE;
clr_reg:DM(I0,M0)=0;

      AX0=255;
      DM(0X3FF4)=AX0;                  { Divide by 256 for 8KHz samp rate}
      AX0=0X0002;
      DM(0X3FF5)=AX0;                  { Generate 2.048 MHz serial clock}
      AX0=0X6B27;
      DM(0X3FF6)=AX0;                  { Int. gen serial clock}
                                 { Receive frame sync regd, width 0}
                                 { Transmit frame sync regd, width 0}
                                 { Int trans, receive frame sync ena }
                                 { u-law companding, 8 bit word len}
      AX0=260;
      DM(0X3FFC)=AX0;                  { set Tperiod to 260}
      DM(0X3FFD)=AX0;                  { set TCOUNT to 260}

      I1=^sine_data;                   { Index to circular buffer}
      L1=%sine_data;                   { Length of circular buffer}
      M1=1;                            { Post increment by 1}

      IMASK=h#01;                      { Timer interrupt enable}
      ICNTL=h#07;                      { Enable edge sensitive interrupt}
      ENA TIMER;

wait:   IDLE;                            { Wait until next interrupt}
      JUMP wait;

serv:   AX0=DM(I1,M1);                   { Get sine data from circular buff}
      call sin;                        { Compute sine value}
      DM(write_dac0)=AR;               { Output to DAC}
      DM(load_dac)=AR;
      TX0=AR;
      RTI;

.ENDMOD;                                 { End of Program}
```

Part D Uniform Random Number Generator

Although the generation of a random number is not a function, it is a useful operation for many applications.
The implementation presented here is based on the linear congruence method, which uses the following equation:

x(n+1) = (ax(n)+c) mod m

Prepare the executable code from the following program.

```
{ ADSP-2101 Random Program                                   RANDOM.DSP

    This subroutine calculates uniformly distributed random numbers using a
    linear congruence method.

    Calling Parameters
      SR1 = MSW of the seed value
      SR0 = LSW of the seed value

    Return Value
      AR  = The Random Number
      SR1 = MSW of the new seed value
      SR0 = LSW of the new seed value

    Altered registers
       MY0, MY1, MR ,SI,SR, AR

   Computaton time
       10 *N +4 cycles


}

.MODULE/RAM URAND_SUB;                          { Beginning of RANDOM Subprogram }
.ENTRY  urand;                                  { Entry point of random function }

urand:  MY1=25;                                 { Upper half of a }
      MY0=26125;                               { Lower half of a }
      AR=SR1, MR=SR0*MY1(UU);                  { a(hi) * x(lo) }
      MR=MR+SR1*MY0(UU);                       { a(hi) * x(lo) + a(lo) * x(hi) }
      SI=MR1;
      MR1=MR0;
      MR2=SI;
      MR0=H#FFFE;                              { c-32767, left-shifted by 1 }
      MR=MR+SR0*MY0(UU);                       { ( above ) + a(lo) X x(lo) + c }
      SR=ASHIFT MR2 BY 15 ( HI );
      SR=SR OR LSHIFT MR1 BY -1 ( HI );        { right shift by 1 }
      SR=SR OR LSHIFT MR0 BY -1 ( LO );
      RTS;
.ENDMOD;                                        { End of RANDOM Program }


{ ADSP-2101 Main Program                                     MAIN.DSP

    This program setup timer interrupt and call a uniform random number
    generator program. The output is sent to a speaker via the SPORT0
```

Transmit Register. A 8 KHz random noise will be heard.The output can also be displayed on scope through DAC0.

```
}

.MODULE/RAM/ABS=0 MAIN;                    { Beginning of MAIN Prorgam }
.PORT WRITE_DAC0;
.PORT LOAD_DAC;
.EXTERNAL urand;                           { External sine wave generator }

.VAR/DM seed_msw, seed_lsw;                { MSW, LSW of the seed value }

      JUMP start; NOP; NOP; NOP;           { Jump to START label }
      RTI; NOP; NOP; NOP;                  { Interrupt vectors ( not use ) }
      RTI; NOP; NOP; NOP;
      RTI; NOP; NOP; NOP;
      RTI; NOP; NOP; NOP;
      RTI; NOP; NOP; NOP;
      JUMP rand; RTI; NOP; NOP;            { Timer interrupt }

start:   AX0=0x1000;
       DM(0x3FFF)=AX0;                     { SPORT0 enabled, PM Wait State 0, }
                                  { BOOT Wait State 0, BOOT page 0  }
       AX0=0x00B0;
       DM(0x3FFE)=AX0;                     { All DM 2 Wait States 0 }
       AX0=0x0000;
       DM(0x3FFB)=AX0;
       DM(0x3FFC)=AX0;
       DM(0x3FFD)=AX0;
       DM(0x3FF9)=AX0;                     { Receive Multichannels }
       DM(0x3FFA)=AX0;
       DM(0x3FF7)=AX0;                     { Transmit Multichannels }
       DM(0x3FF8)=AX0;
       AX0=0x6B27;                         { Multichannel disabled }
       DM(0x3FF6)=AX0;                     { Int. gen serial clock }
                                  { Receive frame sync required, width 0 }
                                  { Transmit frame sync required, width 0 }
                                  { Int trans, receive frame sync enabled }
                                  { u-law companding, 8 bit word length }

                                  { Set up timer to interrupt every }
                                  { 125 KHz for a 12.288 MHz clock }
      AX0 = 0;
      DM(0x3ffb) = AX0;                    {set TSCALE to 0 }
      AX0 = 1562;
      DM(0x3ffc) = AX0;                    {set TPeriod to 1562}
      DM(0x3ffd) = AX0;                    {set TCOUNT to 1562 }


      SR1 = h#1234;                        { Initialize seed values }
      dm(seed_lsw)=SI;
      dm(seed_msw)=SI;
```

```
        IMASK=H#11;                     { Enable Timer and SPORT0 receive }
        ICNTL=H#07;
        ENA TIMER;

wait:     IDLE;                           { Wait for timer interrupt }
        JUMP wait;

rand:     SR1 = DM( seed_msw );           { Get seed values }
        SR0 = DM( seed_lsw );
        CALL urand;                     { Compute random number }
        TX0=AR;                         { Send to speaker }
        DM(WRITE_DAC0)=AR;              { Displaying on the scope }
        DM(LOAD_DAC)=AR;
        DM( seed_msw ) = SR1;           { Store new seed values }
        DM( seed_lsw ) = SR0;
        rti;                            { Return from interrupt }

.ENDMOD;                                { End of MAIN Program }
```

Laboratory 6
IIR Filter Implementation

INTRODUCTION

IIR filters, compared to FIR filters, can be much more efficient in terms of attaining better magnitude response with a given filter order. This is because IIR filters incorporate feedback and are capable of realising both poles and zeros of a system function, whereas FIR filters are only capable of realising the zeros. This means that IIR filters can run faster and hence must be considered in applications where speed is important. They, however, have stability problems and have nonlinear phase characteristics which might make them unsuitable for some applications.

This lab deals with the realisation and design IIR filter based on two most popular methods: Bilinear Transformation and Impulse Invariant Transformation.

Part A Lowpass Filter

Specification of the filter to be designed:

Passband frequency	800 Hz
Stopband frequency	1000 Hz
Max attenuation in passband	0.5 dB
Min attenuation in stopband	30 dB
Sampling frequency	8000 Hz

The above filter must be designed using QEDesign software from Momentum Data (student version).

QEDesign is entirely menu-driven for ease of use. There are no commands to memorise. The menus are largely self-explanatory thus allowing the system to be implemented with the minimum of difficulty. The user needs select the option(s) required by following the menu prompts. It should be remembered that individual selections or fields are accepted by striking ENTER, whereas entire screens by striking either F1 or F10.
Most fields have pop-up help screens. To obtain help on a specific field, place the cursor in that field and type the ALT and H keys simultaneously.

Before you start designing your filter ask your lab demonstrator to explain each section of the main menu.

For the above specifications you have to design the following types filters:
- Butterworth
- Tschebyscheff

- Elliptic
- Bessel

using the Bilinear Transformation method.
For all filters, make a hardcopy of each characteristic.
Generate the assemble code for each filter using the ADSP-2100 Family Code Generator. The output of the code generator results in the creation of two files: a .DAT file with the coefficients and .DSP file containing the assembly code.
These files will be input to the ASM21 Assembler and LD21 Linker to generate an executable file.

To assemble and link the program use the batch file MAKEEXE.BAT as follows:
MAKEEXE file_name (without extension).
Program an EPROM (27256) with the executable file and replace the existing one in EZ-LAB.
Measure the magnitude characteristic, step response and impulse response.
Compare results with the printouts from QEDesign.

For the same specification design one of the above filters using the Impulse Invariant Transformation method.
Repeat all step leading to the creation of an EPROM with the executable file and measure the same characteristics.

Part B Bandpass Filter

Specification of the filter to be designed:

Passband frequency	800 - 1000 Hz
Stopband frequency	0 - 600, 1000 - 1100 Hz
Max attenuation in passband	0.5 dB
Min attenuation in stopband	30 dB
Sampling frequency	8000 Hz

Repeat all step from part A leading to the creation of an EPROM with the executable file and measure the same characteristics.
In this part you have to design any two types of filters only.

Laboratory 8
FIR Filter Implementation

INTRODUCTION

Digital signal processing comprises of two important areas: signal filtering in the time domain and signal representation in the frequency domain.
There are two types of digital filters namely FIR and IIR.
In practice, FIR filters are employed in filtering problems where there is a requirement for a linear phase characteristic within the passband of the filters. Using linear phase FIR filters it is possible to implement such system as differentiators and Hilbert transformers, which can not be implemented using IIR filters. However, if phase distortion is unimportant then IIR filters are generally preferable.

This lab deals with the design and implementation of FIR filters using two basic methods: window design technique and the Parks-McClellan algorithm.

Part A Lowpass Filter

Specification of the filter to be designed:

Passband frequency	800 Hz
Stopband frequency	1000 Hz
Max attenuation in passband	0.5 dB
Min attenuation in stopband	30 dB
Sampling frequency	8000 Hz

The above filter must be designed using QEDesign software from Momentum Data (student version).

To invoke the system, type FILTER (a batch file in the FDAS2K1 directory).
For the above specifications you have to design your filter using any three out of six windows (Rectangular, Triangular, Hanning, Hamming, Blackman, Kaiser).
For the chosen filters, make a hardcopy of each characteristic.

Generate the assemble code for each filter using the ADSP-2100 Family Code Generator. The output of the code generator results in the creation of two files:
- a .DAT file with the coefficients
- a .DSP file containing the assembly code.

These files will be input to the ASM21 Assembler and LD21 Linker to generate an executable file.
To create such a file use the batch MAKEEXE.BAT file as follows: MAKEEXE file_name (without extension).

Program an EPROM (27256) with the executable file and replace the existing one in EZ-LAB.
Measure the magnitude characteristic, step response and impulse response.
Compare the results with the printouts from QEDesign.

For the same specification, design a filter using the Parks-McClellen algorithm.
Repeat all step leading to the creation of an EPROM with the executable file and measure the same characteristics.

Part B Bandpass Filter

Specification of the filter to be designed:

Passband frequency	800 - 1000 Hz
Stopband frequency	0 - 600, 1000 - 1100 Hz
Max attenuation in passband	0.5 dB
Min attenuation in stopband	30 dB
Sampling frequency	8000 Hz

Repeat all step from part A, using the same window functions and the Parks-McClellan algorithm, leading to the creation of an EPROM with the executable file and measure the characteristics.

Laboratory 10
Fast Fourier Transform

INTRODUCTION

Signal representation and analysis in the frequency domain is an important area in digital signal processing. It is performed using the discrete Fourier transform (DFT) on the data sequence. The DFT is widely used in applications to determine spectral contents, to perform correlation analysis and to implement linear filtering in the frequency domain. This widespread use is due to the existence of efficient algorithms for computing the DFT.

The architecture and the instruction set of the ADSP-2101 processor is well suited for implementation of the FFT algorithms. These algorithms repeatedly perform a core multiply/add operation, called a butterfly, on ordered pairs of data points and either require a scramble order of the input data or result in a scramble order of the output data. This requires a very efficient address generation which can be accomplished by two data address generators (DAG0 and DAG1) of ADSP-2101. Further more, DAG1 contains a bit-reversal hardware which can be used to scramble or unscramble the data on the fly.

Part A DFT and Window Functions

Using MatLab run the DFTDEMO.M file.
This program has two pop-up menus. The Signal and Window menu allows you to change the waveform and window functions respectively. The fundamental frequency of the waveform is given in the editing box. You can change the fundamental frequency by editing the value in the box or by clicking and dragging the waveform. The sampling rate,used in the program, is 200 Hz.

For each waveform at three different frequencies , within the range 0 - 50 Hz (e.g. 5, 10 and 20 Hz), perform FFT.
Compare the results for different window functions.

Part B Spectrum Analyser based on ADSP-2101

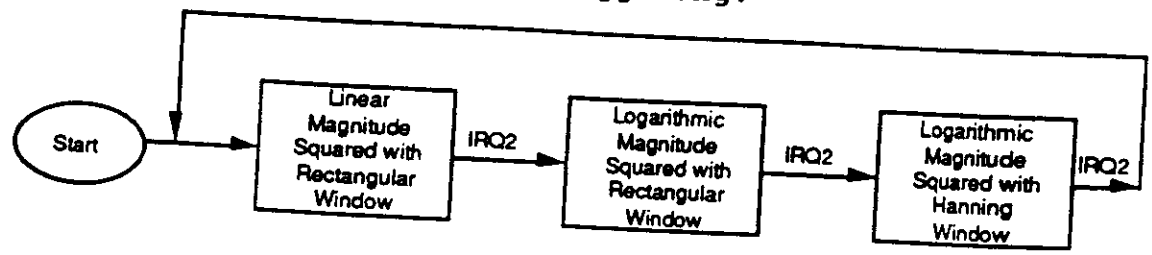
Generate the machine code of the file FFT.DSP (see the listing attached).
This program performs a 256-point FFT (radix-4 complex algorithm) on the data from the function generator or the microphone and then displays its spectrum on an oscilloscope (DAC0).

Input data can be displayed in three ways:

- Linear magnitude squared with rectangular window
- Logarithmic magnitude with rectangular window
- Logarithmic magnitude with Hanning window

Sequence through these different modes by pressing the IRQ2 switch on EZ-LAB. An input signal is sampled at 8 KHz by the EZ-LAB codec. A frame of 256 samples is then windowed by a Hanning window or a Rectangular window depending on the mode setting. A 256-point FFT is then performed on this frame.

The resultant complex spectrum is magnitude squared. Depending on the mode setting, these magnitude squared results can also be passed through a logarithm function stage. The final results are then stored in a result buffer. This result buffer is scanned and displayed on the screen at 40 frames/s. The built in timer is used to continuously scan and display the results independently of all other operations. The display driver also generates a negative sync pulse to allow proper scope triggering.



Demo Flow

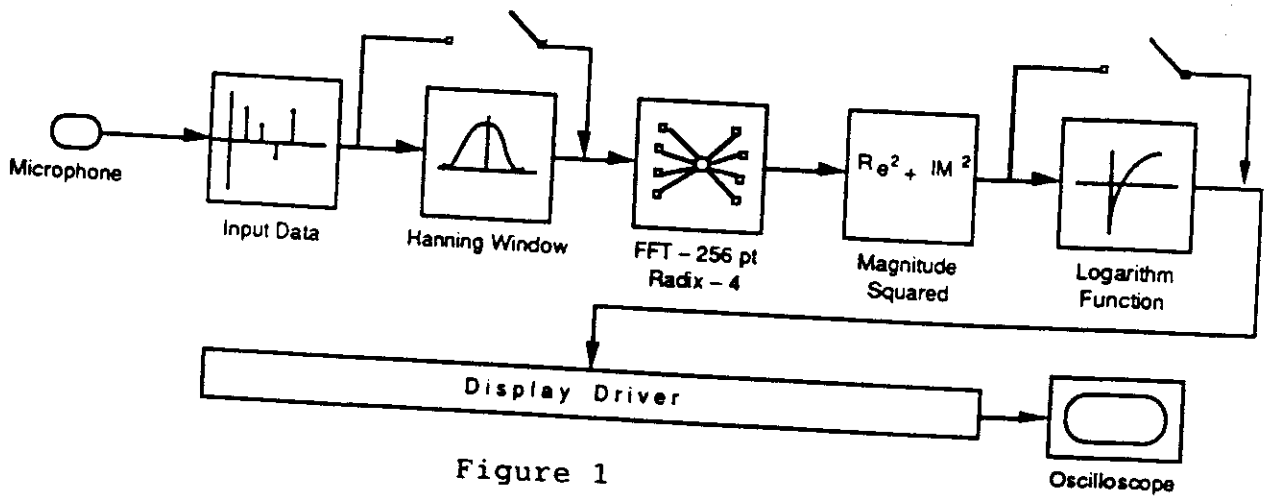


Figure 1

Prepare an EPROM (27256) with the executable file FFT.EXE and replace the original EZ-LAB EPROM.
Measure spectrum of some signals from the function generator: sine, triangular and rectangular waveforms. Compare the spectra with theoretical ones.

Plug in the microphone and measure the spectra of some phonemes. Write down the results in the lab book.

Part C FFT

Adapt your program to be compiled using CC21.EXE

Take advantage from the following hints:
- place the ADSP2101.H header file at the beginning of your program and before other preprocessor directives
- define your own functions performing complex numbers operations
- don't use I/O operations

Link using LD21.EXE to produce an executable file.
Invoke the simulator SIM_2101.EXE, run your program, write down the results and compare them with the results produced by your program working under DOS.

Laboratory 11&12
DSP Applications

Part A Pulse Code Modulation

The purpose of this experiment is to gain an understanding of the PCM compression (linear-to-logarithmic) and the PCM expansion (logarithmic-to-linear) algorithms.

The source code of the program performing the u-low companding is included in the PCM.DSP file and the input samples in the *.DAT files.

The input signals for the program are :
a) saw-tooth waveform
b) exponential waveform
c) user-defined (option)
which are included in the files SAW.DAT ,EXP.DAT and USER.DAT respectively.

Generate the executable file with the input samples from SAW.DAT (see the .INIT linear_input directive at the beginning of PCM.DSP).
Using the EZ-ICE emulator and EZ-LAB system, execute PCM.EXE. Display the input (DAT1) and output (DAT0) waveforms and comment on the results.

Repeat the same operations for other input samples.

Part B Dual-Tone MultiFrequency DTMF

Using MatLab run the PHONE.M program to understand the DTMF tone generation and the DTMF decoding algorithm.

Part C GSM Speech Processing

The GSM.EXE demonstration program performs the digital mobile radio (GSM) speech processing functions which make transmitted speech more intelligible in noisy environments. This program takes digitised speech, processes it, and outputs it to the speaker. The EZ-LAB FLAG_OUT output is configured to show the state of the program's voice activity detector flag. When the light is on, voice activity has been detected.
Press the IRQ2 button to change between states:

State 0 Input is directly in a talk-through mode, with no intermediary processing; i.e. no encoding or decoding. The voice activity flag is disabled.

State 1 Speech is encoded and decoded in a talk-trough mode. This mode demonstrates the need for comfort noise insertion. The intelligibility of speech in a noisy background is reduced. Frames are encoded as speech or comfort noise, depending on the state of the voice activity flag. If a frame is encoded as speech, it is decoded as speech. If a frame is encoded as comfort noise, the output is muted. The voice activity flag is determined for each frame.

State 2 Speech is encoded and decoded in a talk-through mode. This mode is the normal mode of the GSM system. Frames are encoded and decoded as speech or comfort noise, depending on the state of the voice activity flag. The voice activity flag is determined by each frame.

State 3 Speech is encoded and decoded an example mode. Each frame is encoded and decoded as comfort noise. The voice activity flag is forced inactive.

State 4 No input or encoding. The last valid silence discriptor frame (comfort noise insertion) is continuously decoded. The voice activity flag is forced inactive.

To show the difference between State 1 and State 2, mix a random noise source with the microphone input. This will demonstrate the adaptation of the noise threshold for voice activity detection and the loss of intelligibility in State 1 compared to State 2 in a, noise environment.

Part D Computer Graphics

This part presents an example of graphics application.
The GRAPH.DSP file contains the actual ADSP-2101 source code. The graphics routine uses two DAC channels (DAC0 and DAC1) to generate an X-Y offset on the oscilloscope.

Generate the executable file and run it using the EZ-LAB emulator. Set the oscilloscope to X-Y mode. Set both channels to 0.5 V/div (X - DAC0, Y - DAC1). Connect the probe ground wires to the analog ground terminal (J3). The demonstration works best if the signals are AC coupled.

The screen image is a 3-D projection of "2101" rotating in real time. The main loop of this program includes all the functions necessary to rotate the image and display it on the oscilloscope.

Part E Professional DSP Hardware and Software - Demonstration

Presentation of TMS320C40 and DSPower

UNIVERSITY OF WESTERN SYDNEY Nepean
Faculty of Engineering

DIGITAL SIGNAL PROCESSING
Laboratories

ADSP-2101 DEVELOPMENT TOOLS
Appendix

ANALOG DEVICES

DSP Microcomputer

ADSP-2101

FEATURES

Complete DSP Microcomputer
60 ns Instruction Cycle Time from 16.67 MHz Crystal
ADSP-2100 Family Code & Function Compatible
2K Words of On-Chip Program Memory RAM
1K Word of On-Chip Data Memory RAM
Separate Program and Data Buses On-Chip
Dual Purpose Program Memory for Both Instruction and Data Storage
Three Independent Computational Units:
ALU, Multiplier/Accumulator and Barrel Shifter
Two Independent Data Address Generators
Powerful Program Sequencer Provides:
Zero Overhead Looping
Conditional Arithmetic Instruction Execution
Two Double-Buffered Serial Ports with Companding Hardware and Automatic Data Buffering
Programmable 16-Bit Interval Timer with Prescaler
Programmable Wait State Generation
Automatic Booting of Internal Program Memory from Byte-Wide External Memory, e.g., EPROM
Provisions for Multiprecision Computation and Saturation Logic
Single-Cycle Instruction Execution
Single-Cycle Context Switch
Multifunction Instructions
Three Edge- or Level-Sensitive External Interrupts
Low Power Dissipation in Standby Mode
68-Pin PGA, 68-Lead PLCC and 80-Lead PQFP
MIL-STD-883 Compliant Versions Available

GENERAL DESCRIPTION

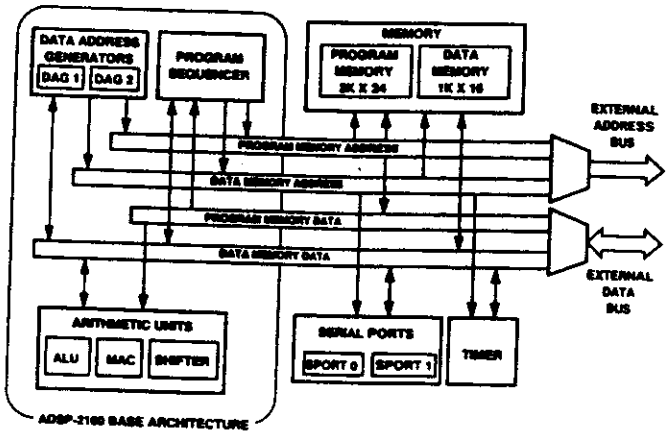
The ADSP-2101 is a single-chip microcomputer optimized for digital signal processing (DSP) and other high-speed numeric processing applications. It combines the complete ADSP-2100 architecture (three computational units, data address generators and a program sequencer) with two serial ports, a programmable timer, extensive interrupt capabilities and on-chip program and data memory RAM. The ADSP-2101 has 1K words of (16-bit) data memory RAM and 2K words of (24-bit) program memory RAM on chip.

Fabricated in a high-speed, 1.0 micron, double-layer metal CMOS process, the ADSP-2101 operates with a 60 ns instruction cycle time. Every instruction executes in a single cycle. Fabrication in CMOS results in low power operation.

EZ-ICE and EZ-LAB are registered trademarks of Analog Devices, Inc.

REV. C

FUNCTIONAL BLOCK DIAGRAM



The ADSP-2101's flexible architecture and comprehensive instruction set support a high degree of operational parallelism. In one cycle the ADSP-2101 can:

- generate the next program address
- fetch the next instruction
- perform one or two data moves
- update one or two data address pointers
- perform a computational operation
- receive and transmit data via the two serial ports

Development System

The ADSP-2101 is supported by a complete set of tools for software and hardware system development. The Development Software is a set of modules that supports all ADSP-2100 family processors. The System Builder provides a high-level method for defining the architecture of systems under development. The Assembler produces object code and the Linker combines object modules and library calls into an executable file. The Simulator provides an interactive instruction-level simulation with a reconfigurable user interface. A PROM Splitter generates PROM programmer compatible files. The C Compiler generates ADSP-21xx assembly source code.

Emulators aid in the hardware debugging of ADSP-2101 systems. The full-featured emulator performs a full range of emulation functions including trace and triggering. EZ-Tools are low cost, easy-to-use hardware tools. The EZ-ICE® emulator provides basic functions like changing register values and setting breakpoints. The EZ-LAB® demonstration board is a complete ADSP-2101 system that executes EPROM-based programs. The EZ-Kit package is a DSP design kit that contains an EZ-LAB board, development software, DSP textbooks, and example programs.

One Technology Way, P.O. Box 9106, Norwood, MA 02062-9106, U.S.A.
Tel: 617/329-4700 **Fax: 617/326-8703**

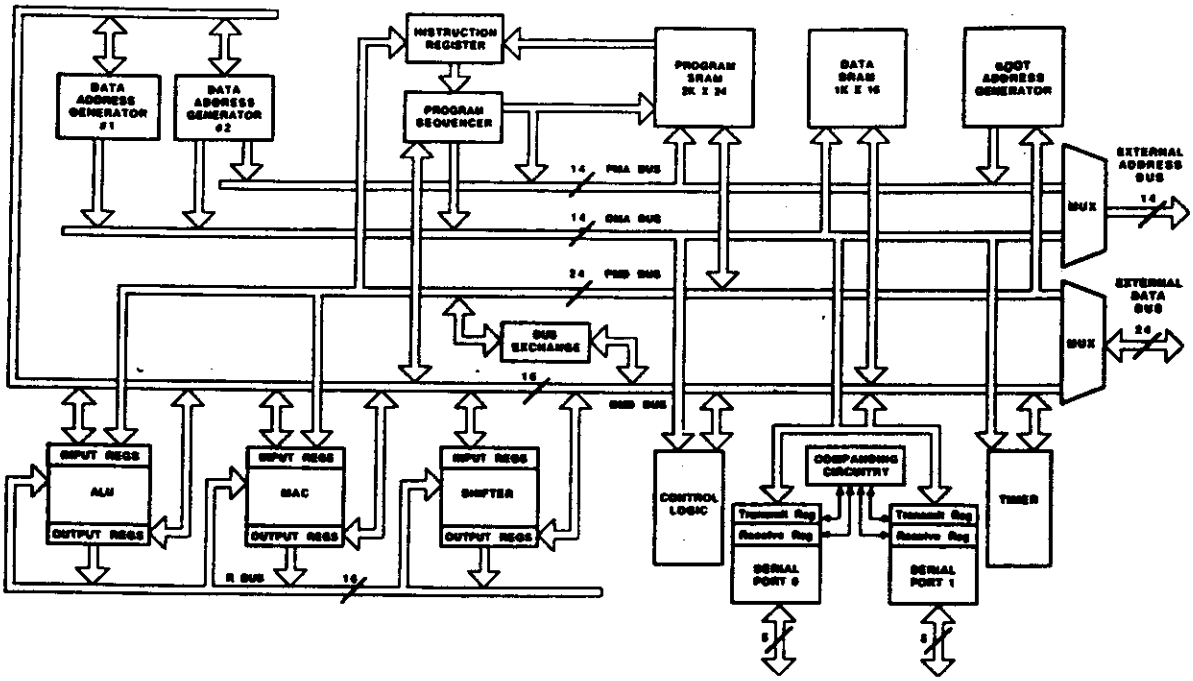


Figure 1. ADSP-2101 Block Diagram

Additional Information

This data sheet provides a general overview of ADSP-2101 functionality. For additional information on the architecture and instruction set of the processor, refer to the *ADSP-2100 Family User's Manual*. For more information about the Development System and ADSP-2101 programmer's reference information, refer to the *ADSP-2100 Family Development Software Manuals*.

ARCHITECTURE OVERVIEW

Figure 1 is an overall block diagram of the ADSP-2101. The processor contains three independent computational units: the ALU, the multiplier/accumulator (MAC) and the shifter. The computational units process 16-bit data directly and have provisions to support multiprecision computations. The ALU performs a standard set of arithmetic and logic operations; division primitives are also supported. The MAC performs single-cycle multiply, multiply/add and multiply/subtract operations. The shifter performs logical and arithmetic shifts, normalization, denormalization, and derive exponent operations. The shifter can be used to efficiently implement numeric format control including multiword floating-point representations.

The internal result (R) bus directly connects the computational units so that the output of any unit may be the input of any unit on the next cycle.

A powerful program sequencer and two dedicated data address generators ensure efficient use of these computational units. The sequencer supports conditional jumps, subroutine calls and returns in a single cycle. With internal loop counters and loop stacks, the ADSP-2101 executes looped code with zero overhead; no explicit jump instructions are required to maintain the loop.

Two data address generators (DAGs) provide addresses for simultaneous dual operand fetches (from data memory and program memory). Each DAG maintains and updates four address pointers. Whenever the pointer is used to access data (indirect addressing), it is post-modified by the value of one of four possible modify registers. A length value may be associated with each pointer to implement automatic modulo addressing for circular buffers. The circular buffering feature is also used by the serial ports for automatic data transfers; these are described on the next page in "Serial Ports."

Efficient data transfer is achieved with the use of five internal buses.

- Program Memory Address (PMA) Bus
- Program Memory Data (PMD) Bus
- Data Memory Address (DMA) Bus
- Data Memory Data (DMD) Bus
- Result (R) Bus

The two address buses (PMA and DMA) share a single external address bus, allowing memory to be expanded off-chip, and the two data buses (PMD and DMD) share a single external data bus. The $\overline{\text{BMS}}$, $\overline{\text{DMS}}$ and $\overline{\text{PMS}}$ signals indicate which memory space the external buses are being used for.

Program memory can store both instructions and data, permitting the ADSP-2101 to fetch two operands in a single cycle, one from program memory and one from data memory. The ADSP-2101 can fetch an operand from on-board program memory and the next instruction in the same cycle.

The memory interface supports slow memories and memory-mapped peripherals with programmable wait state generation. External devices can gain control of buses with bus request/grant signals ($\overline{\text{BR}}$ and $\overline{\text{BG}}$). One execution mode allows the ADSP-2101 to continue running from internal memory. A second execution mode requires the processor to halt while buses are granted.

The ADSP-2101 can respond to six interrupts. There can be up to three external interrupts, configured as edge or level sensi-

tive. Internal interrupts can be generated by the Timer and the Serial Ports, "SPORTs." There is also a master $\overline{\text{RESET}}$ signal.

The two serial ports provide a complete synchronous serial interface with optional companding in hardware and a wide variety of framed or frameless data transmit and receive modes of operation. Each port can generate an internal programmable serial clock or accept an external serial clock.

Boot circuitry provides for loading on-chip program memory automatically from byte-wide external memory. After reset three wait states are automatically generated. This allows, for example, a 60 ns ADSP-2101 to use an external 200 ns EPROM as boot memory. Multiple programs can be selected and loaded from the EPROM with no additional hardware.

A programmable interval timer can generate periodic interrupts. A 16-bit count register (TCOUNT) is decremented every n cycles, where $n-1$ is a scaling value stored in an 8-bit register (TSCALE). When the value of the count register reaches zero, an interrupt is generated and the count register is reloaded from a 16-bit period register (TPERIOD).

The ADSP-2101 instruction set provides flexible data moves and multifunction (one or two data moves with a computation) instructions. Every instruction can be executed in a single processor cycle. The ADSP-2101 assembly language uses an algebraic syntax for ease of coding and readability. A comprehensive set of development tools supports program development.

Serial Ports

The ADSP-2101 incorporates two complete synchronous serial ports (SPORT0 and SPORT1) for serial communications and multiprocessor communication.

Each serial port has a 5-pin interface consisting of the following signals.

Signal Name	Function
SCLK	Serial clock (I/O)
RFS	Receive frame synchronization (I/O)
TFS	Transmit frame synchronization (I/O)
DR	Serial data receive
DT	Serial data transmit

Here is a brief list of the capabilities of the ADSP-2101 SPORTs. (Refer to User's Manual for further details.)

- Bidirectional: each SPORT has a separate transmit and receive section.
- Double-buffered: each SPORT section (both receive and transmit) has a data register accessible to the user and an internal transfer register. The double-buffering provides additional time to service the SPORT.
- Flexible clocking: each SPORT can use an external serial clock (from 0 Hz to 12.5 MHz) or generate its own (up to 1/2 the processor frequency).
- Flexible framing: framings for the receive and transmit sections on each SPORT are independent. Each section can run in a frameless mode, with internally generated or externally generated frame synchronization signals, with active high or inverted frame signals, with either of two pulse widths/timings. The receive and transmit sections share the same serial clock.
- Flexible word length: each SPORT supports serial data word lengths from three to sixteen bits.
- Companding in hardware: each SPORT provides optional A-law and μ-law companding according to CCITT recommendation G.711. Different companding can be used for each SPORT, for example, A-law for SPORT0 and μ-law for SPORT1.
- Flexible interrupt scheme: each SPORT section (receive and transmit) can generate a unique interrupt upon completing a data word transfer or after transferring an entire buffer (see next item).
- Auto-buffering with single-cycle overhead: using the ADSP-2101 DAGs, each SPORT can receive and/or transmit an entire circular buffer of data with an overhead of only one cycle per data word. Transfers to and from the SPORT and the circular buffer are automatic in this mode and do not require additional programming. An interrupt is generated only when the receive buffer is full or the transmit buffer is empty.
- Multichannel capability: SPORT0 provides a multichannel interface for selective receipt and transmission of arbitrary data channels from a twenty-four or thirty-two word, time-division multiplexed, serial bitstream. This is especially useful for T1 or CEPT interfaces or as a network communication scheme for multiple processors.
- Alternate configuration: SPORT1 can be configured as two external interrupt inputs ($\overline{\text{IRQ0}}$ and $\overline{\text{IRQ1}}$) and the Flag In and Flag Out signals. The internally generated serial clock may still be used in this configuration.

Pin Description

The ADSP-2101 is available in a 68-pin PGA, a 68-lead PLCC and an 80-lead PQFP.

Table I. ADSP-2101 Pin List

Pin Group Name	# of Pins	Input/ Output	Function
Address	14	O	Address Output for Program, Data and Boot Memory Spaces.
Data	24	I/O	Data I/O pins for program and data memories. Input only for Boot memory space, with two MSBs used as Boot space addresses.
$\overline{\text{RESET}}$	1	I	Processor Reset Input.
$\overline{\text{IRQ2}}$	1	I	External Interrupt Request #2.
$\overline{\text{BR}}$	1	I	External Bus Request Input.
$\overline{\text{BG}}$	1	O	External Bus Grant Output.
$\overline{\text{PMS}}$	1	O	External Program Memory Select.
$\overline{\text{DMS}}$	1	O	External Data Memory Select.
$\overline{\text{BMS}}$	1	O	Boot Memory Select.
$\overline{\text{RD}}$	1	O	External Memory Read Enable.
$\overline{\text{WR}}$	1	O	External Memory Write Enable.
MMAP	1	I	Memory Map Select.
CLKIN, XTAL	2	I	External Clock or Quartz Crystal Input.
CLKOUT	1	O	Processor Clock Output.
SPORT0	5	I/O	Serial Port 0 I/O Pins. (TFS0, RFS0, DT0, DR0, SCLK0).
SPORT1 or	5	I/O	Serial Port 1 I/O Pins.
$\overline{\text{IRQ1}}$ (TFS1)	1	I	External Interrupt Request #1.
$\overline{\text{IRQ0}}$ (RFS1)	1	I	External Interrupt Request #0.
SCLK1	1	O	Programmable Clock Output.
FO (DT1)	1	O	Flag Output Pin.
FI (DR1)	1	I	Flag Input Pin.
GND	4		Ground Pins (8 on PQFP).
V_{DD}	3		Power Supply (5 on PQFP).

Interrupts

The interrupt controller allows the processor to respond to the six possible interrupts with a minimum of overhead. The ADSP-2101 provides up to three external interrupt input pins, $\overline{IRQ0}$, $\overline{IRQ1}$ and $\overline{IRQ2}$. $\overline{IRQ2}$ is always available as a dedicated pin; $\overline{IRQ1}$ and $\overline{IRQ0}$ may be alternately configured as part of serial port 1. The ADSP-2101 also supports internal interrupts from the timer and the two serial ports. The interrupt levels are internally prioritized and individually maskable. The input pins can be programmed to be either level- or edge-sensitive. The priorities of all six interrupts are shown in Table II.

Table II. Interrupt Priority & Interrupt Vector Addresses

Source of Interrupt	Interrupt Vector Address (Hex)
$\overline{IRQ2}$ (external pin)	0004 (*highest priority*)
SPORT0 Transmit (internal)	0008
SPORT0 Receive (internal)	000C
SPORT1 Transmit (internal) or $\overline{IRQ1}$ (external)	0010
SPORT1 Receive (internal) or $\overline{IRQ0}$ (external)	0014
Timer (internal)	0018 (*lowest priority*)

The ADSP-2101 supports a vectored interrupt scheme: when an interrupt is acknowledged, the processor shifts program control to the interrupt vector address corresponding to the interrupt level. Interrupts can optionally be nested so that a higher priority interrupt can preempt the currently executing interrupt service routine. Each interrupt vector location is four instructions in length, so that simple service routines can be coded entirely in this space. Longer routines require an additional JUMP or CALL instruction.

Individual interrupt requests are logically ANDed with the bits in IMASK; the highest priority unmasked interrupt is then selected.

The interrupt control register, ICNTL, allows the external interrupts to be set as either edge- or level-sensitive. Depending on Bit 4 in ICNTL, interrupt routines can either be nested with higher priority interrupts taking precedence or processed sequentially with only one interrupt service active at a time.

The 12-bit interrupt force and clear register, IFC, is a write-only register that contains a force bit and a clear bit for each of the six possible interrupts.

When responding to an interrupt, the status registers ASTAT, MSTAT, IMASK are pushed onto the status stack, and the PC counter is loaded with the appropriate vector address. The status stack is seven levels deep to allow interrupt nesting. The stack is automatically popped when a return from the interrupt is executed.

SYSTEM INTERFACE

Figure 4 shows a basic system configuration with the ADSP-2101, two serial devices, a boot EPROM and optional external program and data memories. Up to 15K words of data memory and 16K words of program memory can be supported. Programmable wait state generation allows the processor to interface easily to slow memories.

The ADSP-2101 also provides one external interrupt and two serial ports or three external interrupts and one serial port.

Clock Signals

The ADSP-2101 may be clocked by either a crystal or by a TTL-compatible clock signal.

The CLKIN input may not be halted, changed during operation, or operated below the specified frequency.

If an external clock is used, it should be a TTL-compatible signal running at the instruction rate. The signal is connected to the processors's CLKIN input. When an external clock is used, the XTAL input *must* be left unconnected.

Because the ADSP-2101 includes an on-chip oscillator circuit, an external crystal may be used. The crystal should be connected across the CLKIN and XTAL pins, with two capacitors connected as shown in Figure 2. A parallel-resonant, fundamental frequency microprocessor grade crystal should be used.

A clock output (CLKOUT) signal is generated by the processor, synchronized to the processor's internal cycles.

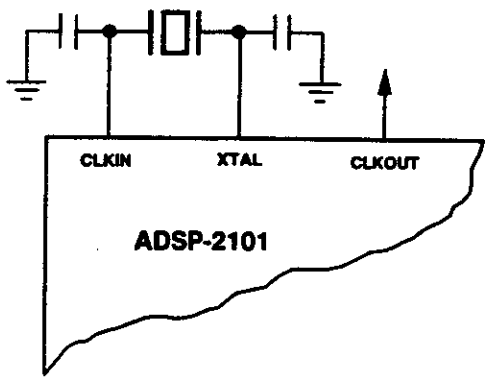


Figure 2. External Crystal Connections

Reset

The $\overline{RESET}$ signal initiates a master reset of the ADSP-2101. The $\overline{RESET}$ signal must be asserted when the chip is powered up to assure proper initialization. $\overline{RESET}$ during initial power-up must be held long enough to allow the internal clock to stabilize. If $\overline{RESET}$ is activated at any time after power-up, the clock continues and does not require this stabilization time.

The power-up sequence is defined as the total time required for the crystal oscillator circuit to stabilize after a valid V_{DD} is applied to the processor and for the internal phase-locked loop (PLL) to lock onto the specific crystal frequency. A minimum of 1000 t_{CK} cycles will ensure that the PLL has locked but does not include the crystal oscillator start-up time. During this power-up sequence the $\overline{RESET}$ signal should be held low. On any subsequent resets, the $\overline{RESET}$ signal must meet the minimum pulse width specification, t_{RSP}.

The $\overline{RESET}$ input contains some hysteresis; however, if you use an RC circuit to generate your $\overline{RESET}$ signal, the use of an external Schmidt trigger is recommended.

The master reset sets all internal stack pointers to the empty stack condition, masks all interrupts and clears the MSTAT register. When $\overline{RESET}$ is released, if there is no pending bus request and the chip is configured for booting (MMAP = 0), the boot-loading sequence is performed. Then the first instruction is fetched from internal program memory location 0x0000.

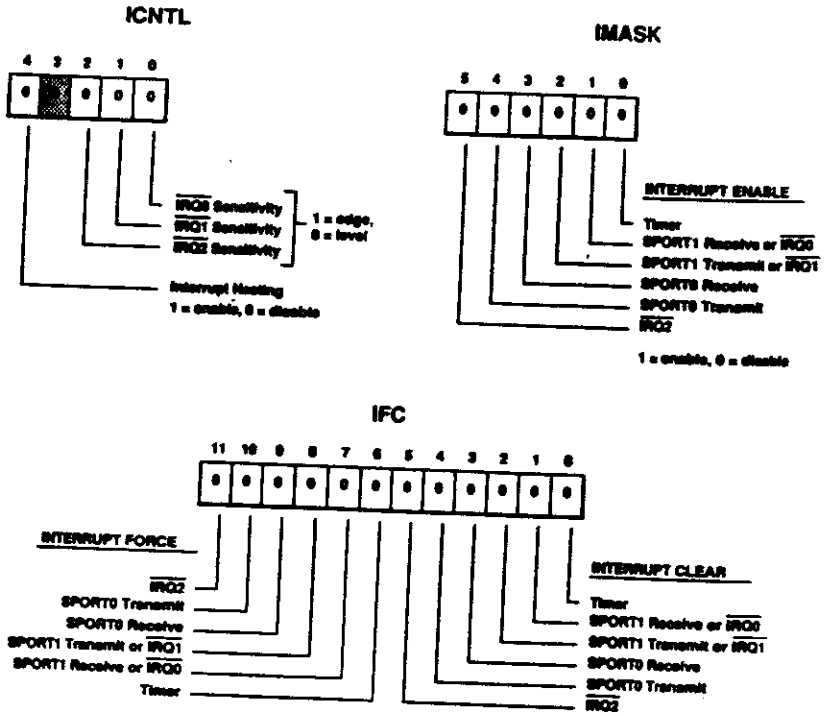
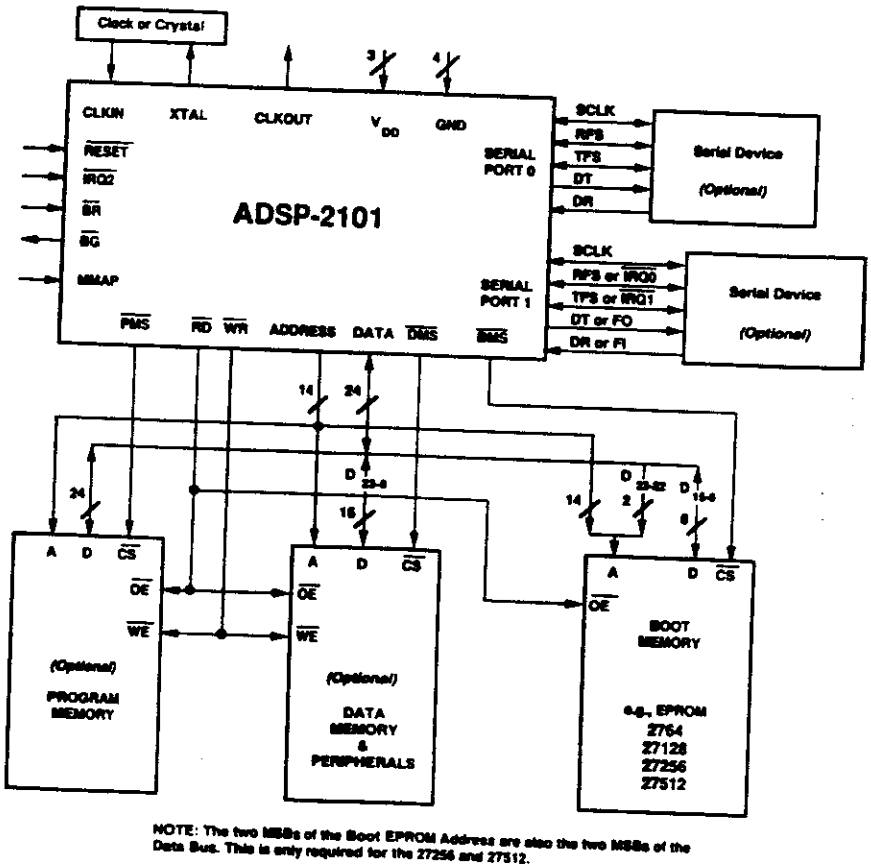


Figure 3. Interrupt Registers



NOTE: The two MSBs of the Boot EPROM Address are also the two MSBs of the Data Bus. This is only required for the 27256 and 27512.

Figure 4. ADSP-2101 Basic System Configuration

Program Memory Interface

The on-chip program memory address bus (PMA) and the on-chip program memory data bus (PMD) are multiplexed with on-chip DMA and DMD buses, creating a single external data bus and a single external address bus. The 14-bit address bus directly addresses up to 16K words, of which 2K are on-chip. The data bus is bidirectional and 24 bits wide to external program memory. Program memory may contain code and data.

The program memory data lines are bidirectional. The program memory select ($\overline{PMS}$) signal indicates access to the program memory and can be used as a chip select signal. The write ($\overline{WR}$) signal indicates a write operation and is used as a write strobe. The read ($\overline{RD}$) signal indicates a read operation and is used as a read strobe or output enable signal.

The ADSP-2101 writes data from its 16-bit registers to the 24-bit program memory using the PX register to provide the lower eight bits. When it reads data (not instructions) from 24-bit program memory to a 16-bit data register, the lower eight bits are placed in the PX register.

Program Memory Maps

Program memory can be mapped in two ways, depending on the state of the MMAP pin. Figure 5 shows the two configurations. When MMAP = 0, internal RAM occupies 2K words beginning at address 0x0000; external program memory uses the remaining 14K words beginning at address 0x0800. In this configuration, the boot loading sequence (described in "Boot Memory Interface") is automatically initiated when $\overline{RESET}$ is released.

When MMAP = 1, 14K words of external program memory begin at address 0x0000 and internal RAM is located in the upper 2K words, beginning at address 0x3800. In this configuration, program memory is not loaded although it can be written to and read from under program control.

The program memory interface can generate 0 to 7 wait states for external memory devices; default is to 7 wait states after $\overline{RESET}$.

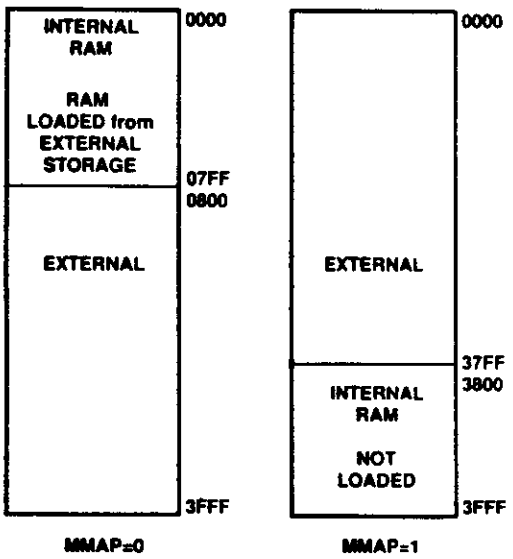


Figure 5. ADSP-2101 Program Memory Maps

Data Memory Interface

The data memory address (DMA) bus is 14 bits wide. The bidirectional external data bus is 24 bits wide, with the upper 16 bits used for data memory data (DMD) transfers.

The data memory select ($\overline{DMS}$) signal indicates access to the data memory and can be used as a chip select signal. The write ($\overline{WR}$) signal indicates a write operation and can be used as a write strobe. The read ($\overline{RD}$) signal indicates a read operation and can be used as a read strobe or output enable signal.

The ADSP-2101 supports memory-mapped I/O, with the peripherals memory mapped into the data memory address space and accessed by the processor in the same manner as data memory.

Data Memory Map

The on-chip data memory RAM resides in the 1K words of data memory beginning at address 0x3800, as shown in Figure 6. In addition, data memory locations from 0x3C00 to the end of data memory at 0x3FFF are reserved. Control registers for the system, timer, wait state configuration and serial port operations are located in this region of memory.

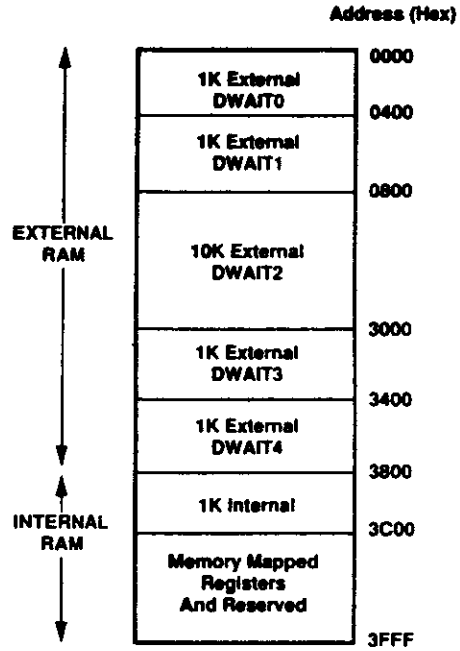


Figure 6. ADSP-2101 Data Memory Map

The remaining 14K of data memory is external. External data memory is divided into five zones, each associated with its own wait state generator. This allows slower peripherals to be memory mapped into data memory for which wait states are specified. By mapping peripherals into different zones, you can accommodate peripherals with different wait state requirements. All zones default to 7 wait states after $\overline{RESET}$.

Boot Memory Interface

The boot memory space consists of an external 64K by 8 space, divided into eight separate 8K by 8 pages. Three bits in the system control register select which page is loaded by the boot memory interface. Another bit in the system control register allows the user to force a boot loading sequence under software control. Boot loading from page 0 after $\overline{RESET}$ is initiated automatically if MMAP = 0.

The boot memory interface can generate 0 to 7 wait states; it defaults to 3 wait states after $\overline{RESET}$. This allows the ADSP-2101 to boot from a single, low cost EPROM such as a 27256. Program memory is booted one byte at a time and converted to 24-bit program memory words.

The $\overline{BMS}$ and $\overline{RD}$ signals are used to select and strobe the boot memory interface. Only 8-bit data is read over the data bus, on pins D8–D15. To accommodate up to eight pages of boot memory, the two MSBs of the data bus are used in the boot memory interface as the two MSBs of the boot space address.

$\overline{BR}$ is recognized during the booting sequence. The bus is granted after the completion of loading the current byte. $\overline{BR}$ during booting may be used to implement booting under the control of a host processor.

The ADSP-2100 Family Assembler and Linker support the creation of programs and data structures requiring multiple boot pages during execution.

Bus Interface

The ADSP-2101 can relinquish control of the data and address buses to an external device. When the external device requires access to memory, it asserts the bus request ($\overline{BR}$) signal. If the ADSP-2101 is not performing an external memory access, then it responds to the active $\overline{BR}$ input in the same cycle by:

- tristating the data and address buses and the $\overline{PMS}$, $\overline{DMS}$, $\overline{BMS}$, $\overline{RD}$, $\overline{WR}$ output drivers,
- asserting the bus grant ($\overline{BG}$) signal, and
- halting program execution.

If the Go mode is set, however, the ADSP-2101 will not halt program execution until it encounters an instruction that requires an external memory access.

If the ADSP-2101 is performing an external memory access when the external device asserts the $\overline{BR}$ signal, then it will not tristate the memory interfaces or assert the $\overline{BG}$ signal until the cycle after the access completes, up to eight cycles later depending on the number of wait states. The instruction does not need to be completed when the bus is granted; the ADSP-2101 will grant the bus in between two memory accesses if an instruction requires more than one external memory access.

When the $\overline{BR}$ signal is released, the processor releases the $\overline{BG}$ signal, re-enables the output drivers and continues program execution from the point where it stopped.

The bus request feature operates at all times, including when the processor is booting and when $\overline{RESET}$ is active.

ADSP-2101 REGISTERS

Figure 7 summarizes all the registers in the ADSP-2101. Some registers store values. For example, AX0 stores an ALU operand; I4 stores a DAG2 pointer. Other registers consist of control bits and fields, or status flags. For example, ASTAT contains status flags from arithmetic operations, and fields in DWAIT control the numbers of wait states for different zones of data memory.

The bit and field definitions for control and status registers are given in the rest of this section, except for IMASK, ICNTL and IFC, which are defined earlier in this data sheet. The system control register, DWAIT register, timer registers and SPORT control registers are all mapped into data memory; that is, you access these registers by reading and writing data memory locations rather than register names. The particular data memory address is shown with each memory-mapped register.

Register bit values shown on the following pages are the default bit values after reset. If no values are shown, the bits are indeterminate at reset. Reserved bits are shown in gray; these bits should always be written with zeros.

A secondary set of registers in all computational units allows a single-cycle context switch.

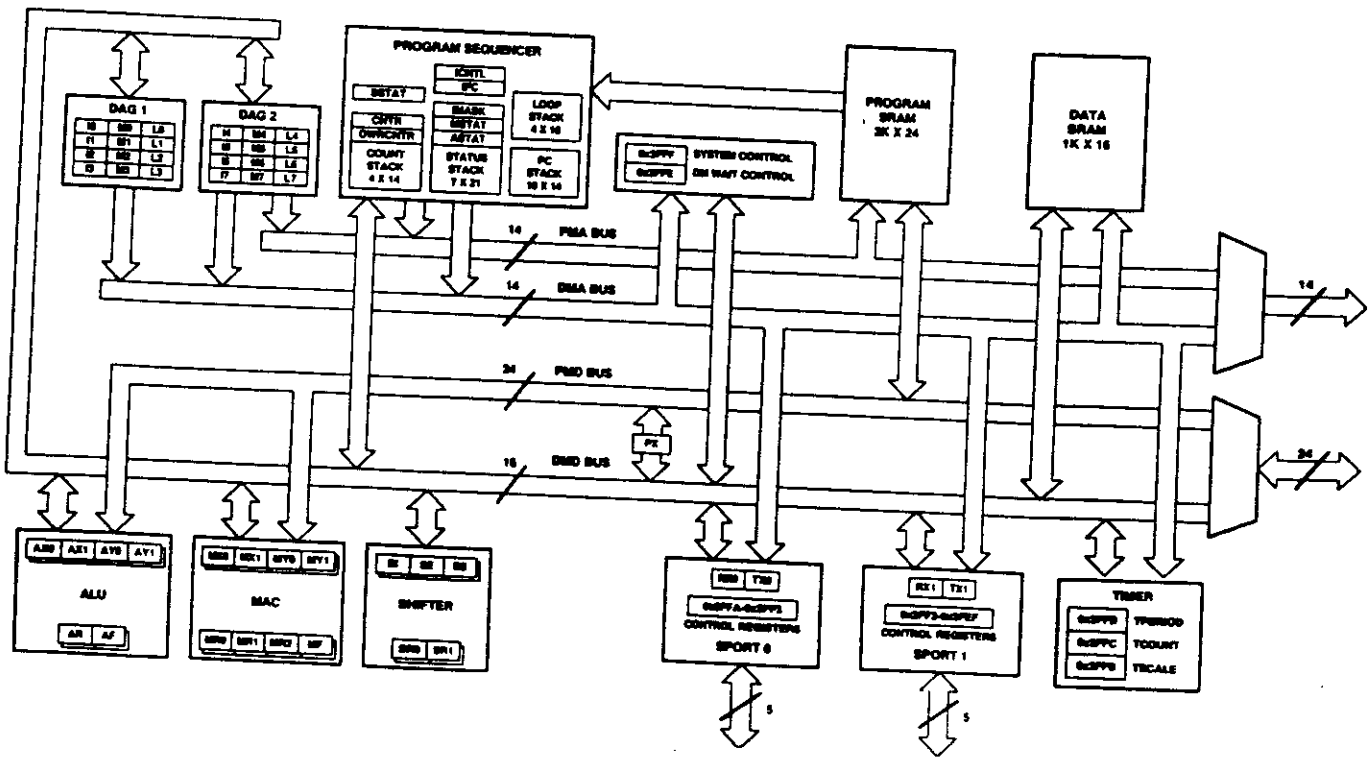
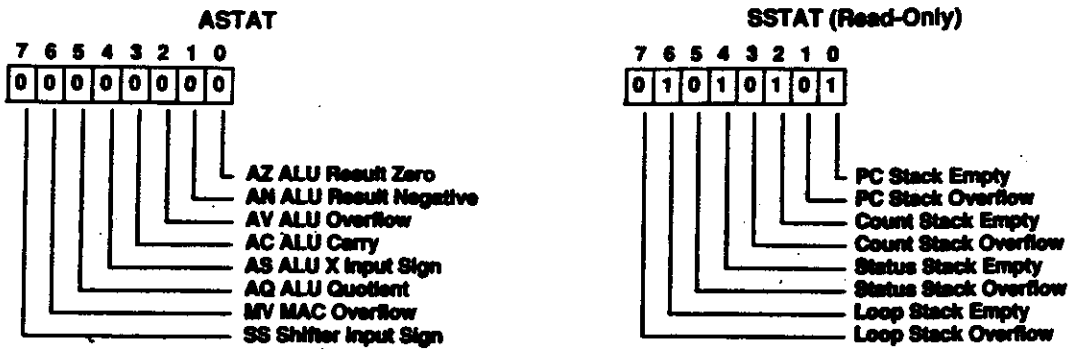
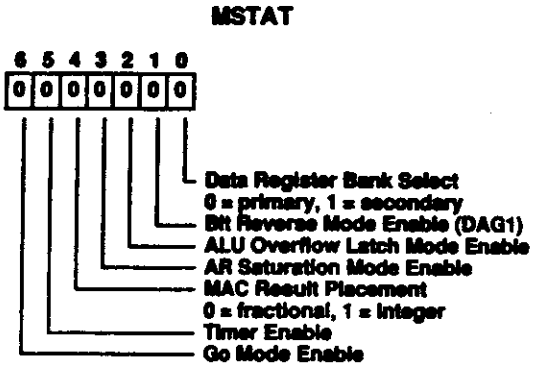
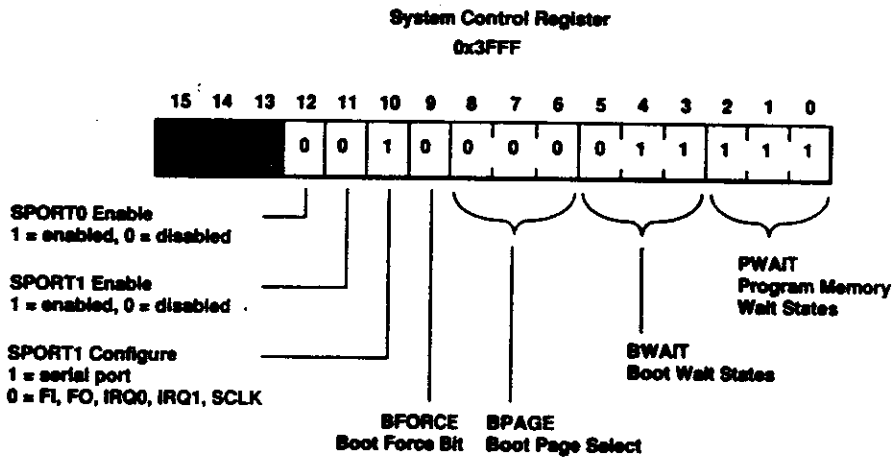
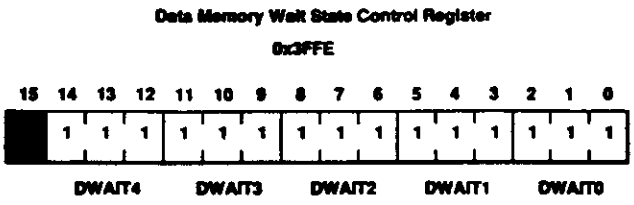


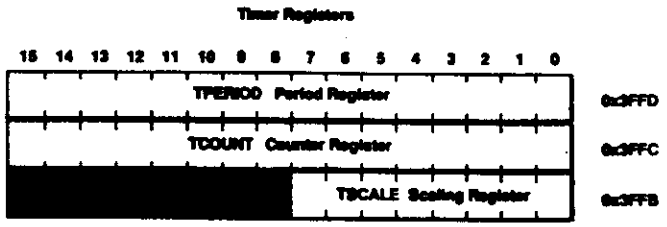
Figure 7. ADSP-2101 Registers


ASTAT
7 6 5 4 3 2 1 0
0 0 0 0 0 0 0 0
AZ ALU Result Zero
AN ALU Result Negative
AV ALU Overflow
AC ALU Carry
AS ALU X Input Sign
AQ ALU Quotient
MV MAC Overflow
SS Shifter Input Sign
SSTAT (Read-Only)
7 6 5 4 3 2 1 0
0 1 0 1 0 1 0 1
PC Stack Empty
PC Stack Overflow
Count Stack Empty
Count Stack Overflow
Status Stack Empty
Status Stack Overflow
Loop Stack Empty
Loop Stack Overflow


MSTAT
6 5 4 3 2 1 0
0 0 0 0 0 0 0
Data Register Bank Select
0 = primary, 1 = secondary
Bit Reverse Mode Enable (DAG1)
ALU Overflow Latch Mode Enable
AR Saturation Mode Enable
MAC Result Placement
0 = fractional, 1 = integer
Timer Enable
Go Mode Enable

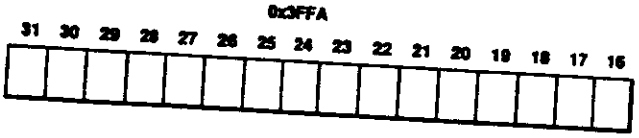

System Control Register
0x3FFF
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 0 1 0 0 0 0 0 1 1 1 1 1 1
SPORT0 Enable
1 = enabled, 0 = disabled
SPORT1 Enable
1 = enabled, 0 = disabled
SPORT1 Configure
1 = serial port
0 = FI, FO, IRQ0, IRQ1, SCLK
BFORCE
Boot Force Bit
BPAGE
Boot Page Select
BWAIT
Boot Wait States
PWAIT
Program Memory
Wait States


Data Memory Wait State Control Register
0x3FFE
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
DWAIT4
DWAIT3
DWAIT2
DWAIT1
DWAIT0

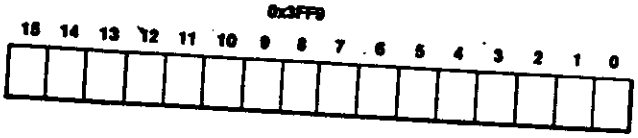

Timer Registers
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
TPERIOD Period Register
TCOUNT Counter Register
TSCALE Scaling Register
0x3FFD
0x3FFC
0x3FFB

SPORT0 Multichannel Receive Word Enable Registers
1 = Channel Enabled
0 = Channel Ignored

0x3FFA



0x3FF9

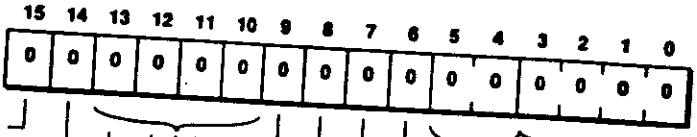


SPORT0 Multichannel Transmit Word Enable Registers
1 = Channel Enabled
0 = Channel Ignored

0x3FF8

0x3FF7

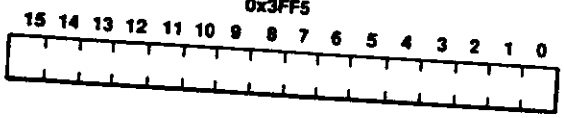
SPORT0 Control Register
0x3FF6



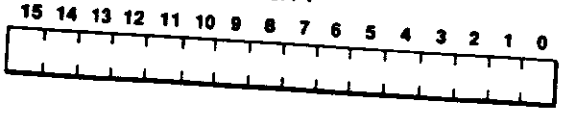
Multichannel Enable MCE
Internal Serial Clock Generation ISCLK
Receive Frame Sync Required RFSR
Receive Frame Sync Width RFSW
Multichannel Frame Delay MFD
Only If Multichannel Mode Enabled
Transmit Frame Sync Required TFSR
Transmit Frame Sync Width TFSW
ITFS Internal Transmit Frame Sync Enable
(or MCL Multichannel Length; 1 = 32 words, 0 = 24 words
Only If Multichannel Mode Enabled)

SLEN Serial Word Length
DTYPE Data Format
00 = right justify, zero-fill unused MSBs
01 = right justify, sign extend into unused MSBs
10 = compand using μ-law
11 = compand using A-law
INVRFS Invert Receive Frame Sync
INVTFS Invert Transmit Frame Sync
(or INVTDV Invert Transmit Data Valid
Only If Multichannel Mode Enabled)
IRFS Internal Receive Frame Sync Enable

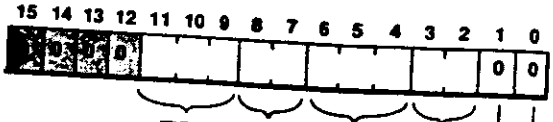
SPORT0 SCLKDIV
Serial Clock Divide Modulus
0x3FF5



SPORT0 RFSDIV
Receive Frame Sync Divide Modulus
0x3FF4

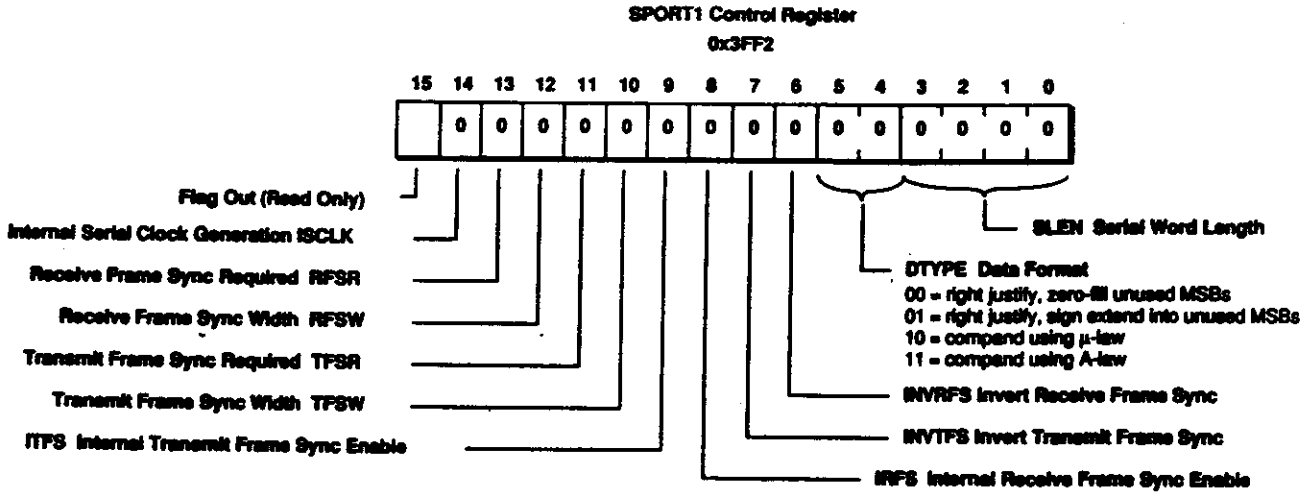


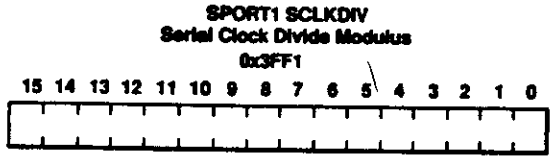
SPORT0 Autobuffer Control Register
0x3FF3

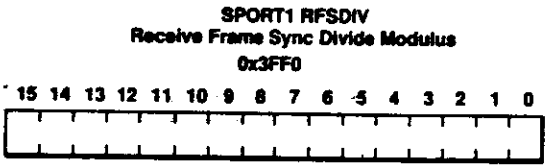


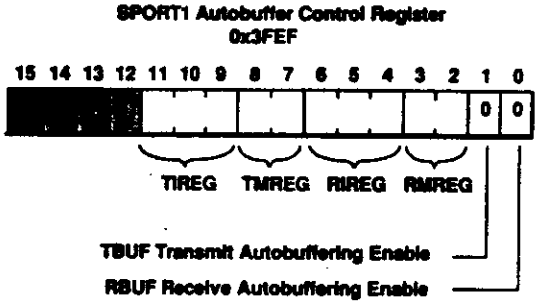
TIREG TMREG RIREG RMREG
TBUF Transmit Autobuffering Enable
RBUF Receive Autobuffering Enable

REV. C









INSTRUCTION SET DESCRIPTION

The ADSP-2100 family assembly language uses an algebraic syntax for ease of coding and readability. The sources and destinations of computations and data movements are written explicitly in each assembly statement, eliminating cryptic assembler mnemonics. Every instruction assembles into a single 24-bit word and executes in a single cycle. The instructions encompass a wide variety of instruction types along with a high degree of operational parallelism. There are five basic categories of instructions: data move instructions, computational instructions, multifunction instructions, program flow control instructions and miscellaneous instructions. Each of these instruction types is described briefly. The complete instruction set is summarized on the following page. The *ADSP-2100 Family User's Manual* and the *ADSP-2100 Family Assembler Manual* contain a complete reference to the instruction set.

ADSP-2100 Family Compatibility

The ADSP-2101 instruction set is a superset of the ADSP-2100 instruction set. The ADSP-2101 is source and object code compatible with all processors of the ADSP-2100 family. An ADSP-2100 program may need to be relocated to utilize internal memory and conform to the ADSP-2101's interrupt vector and reset vector map.

The TRAP instruction of the ADSP-2100, however, is not supported since the ADSP-2101 does not have the TRAP/$\overline{\text{HALT}}$ signals. The TRAP instruction executes as a NOP on the ADSP-2101.

Condition Codes

The condition codes are used to determine whether a conditional instruction, such as a jump, trap, call, return, MAC saturation, or arithmetic operation, is performed. The sixteen basic composite status conditions and their derivations are shown in Table III. Since arithmetic status is latched into ASTAT at the end of a processor cycle, the condition logic represents conditions generated on the previous cycle.

Table III. Condition Codes

Code	Status Condition	True If:
EQ	ALU Equal Zero	AZ = 1
NE	ALU Not Equal Zero	AZ = 0
LT	ALU Less Than Zero	AN .XOR. AV = 1
GE	ALU Greater Than or Equal Zero	AN .XOR. AV = 0
LE	ALU Less Than or Equal Zero	(AN .XOR. AV) .OR. AZ = 1
GT	ALU Greater Than Zero	(AN .XOR. AV) .OR. AZ = 0
AC	ALU Carry	AC = 1
NOT AC	Not ALU Carry	AC = 0
AV	ALU Overflow	AV = 1
NOT AV	Not ALU Overflow	AV = 0
MV	MAC Overflow	MV = 1
NOT MV	Not MAC Overflow	MV = 0
NEG	ALU X Input Sign Negative	AS = 1
POS	ALU X Input Sign Positive	AS = 0
NOT CE	Not Counter Expired	CE = 0
FOREVER	Always	Always True

In addition to the basic sixteen conditions, the JUMP and CALL instructions also support the use of the FI (Flag In) pin as a conditional flag. This pin is one of the five dual-function pins used for serial port 1. The state of this pin and its complement are available as conditions for JUMP and CALL instructions if the pin is configured as FI rather than DR1.

Table IV. Additional Condition Codes For JUMP and CALL

FLAG_IN	FI pin last sampled 1
NOT FLAG_IN	FI pin last sampled 0

Example Code

The following example is a code fragment that performs the filter tap update for an adaptive (least-mean-squared algorithm) filter. Notice that the computations in the instructions are written like algebraic equations.

```
        MF = MX0*MY1 (RND), MX0 = DM (I2,M1) ; {MF = error*beta}
        MR = MX0*MF (RND), AY0 = PM (I6,M5) ;
        DO adapt UNTIL CE;
          AR = MR1 + AY0, MX0 = DM (I2,M1), AY0 = PM (I6,M7) ;
adapt:  PM(I6,M6) = AR, MR = MX0*MF (RND) ;
        MODIFY (I2, M3) ;                    {Point to oldest data}
        MODIFY (I6, M7) ;                    {Point to start of data}
```

INSTRUCTION SET SUMMARY

Key

UPPERCASE	Assembler keyword; exact syntax of instruction
[text]	Parts of the instruction in brackets are optional
x | y | z	Choose x, y or z
[,...]	Any of the operations allowed by this instruction can be combined in any order, separated by commas
Ia, Mb or Ic, Md	Index and modify registers for indirect addressing
x	X input; permissible registers depend on instruction
y	Y input; permissible registers depend on instruction
<data>	Immediate data value
<address>	Immediate address value
condition	Condition from Table x
dreg	Computation unit data register
reg	Any register (except memory-mapped registers)
ALU	Any ALU instruction (except division)
MAC	Any multiply/accumulate instruction
SHIFT	Any shifter instruction (except shift immediate)

ALU Instructions

```
[IF condition]   AR | AF  =  x + y  [+C]      ;
                            x + C              ;
                            x - y  [+C - 1]    ;
                            y - x  [+C - 1]    ;
                            y + 1              ;
                            y - 1              ;
                            x AND | OR | XOR y ;
                            PASS x | y | 0 | 1 ;
                            -x | y             ;
                            NOT x | y          ;
                            ABS x              ;
DIVS y, x;
DIVQ x;
```

MAC Instructions

```
[IF condition]  MR | MF  =  x * y          (SS | SU | US | UU | RND) ;
                            MR + x * y     (SS | SU | US | UU | RND) ;
                            MR - x * y     (SS | SU | US | UU | RND) ;
                            MR             [(RND)] ;
                            0 ;
IF MV SAT MR;
```

Shifter Instructions

```
[IF condition]  SR = [SR OR] ASHIFT | LSHIFT | NORM x (HI | LO) ;
[IF condition]  SE = EXP x  (HI | LO | HIX) ;
[IF condition]  SB = EXPADJ x ;
SR = [SR OR]  ASHIFT | LSHIFT x  BY <data>  (HI | LO) ;
```

Move Instructions

```
reg                 = reg | <data> | DM (<address>);
DM (<address>)      = reg;
dreg                = DM (Ia, Mb);
DM (Ia, Mb)         = dreg | <data>;
dreg                = PM (Ic, Md);
PM (Ic, Md)         = dreg;
```

Multifunction Instructions

```
ALU | MAC*,                x = DM (Ia, Mb) ,        y = PM (Ic, Md);
x = DM (Ia, Mb),           y = PM (Ic, Md);
ALU | MAC | SHIFT*,        dreg = DM | PM (Ia, Mb);
DM | PM (Ia, Mb) = dreg ,  ALU | MAC | SHIFT*;
ALU | MAC | SHIFT*,        dreg = dreg;
```

*All computation is unconditional; Division and Shift Immediate operations prohibited.

Program Flow Control Instructions

```
[IF condition]          JUMP | CALL        (Ic) | <address> ;
IF [NOT] FLAG_IN        JUMP | CALL        <address>       ;
[IF condition]          RTS | RTI ;
DO <address>            [UNTIL termination];
IDLE ;
```

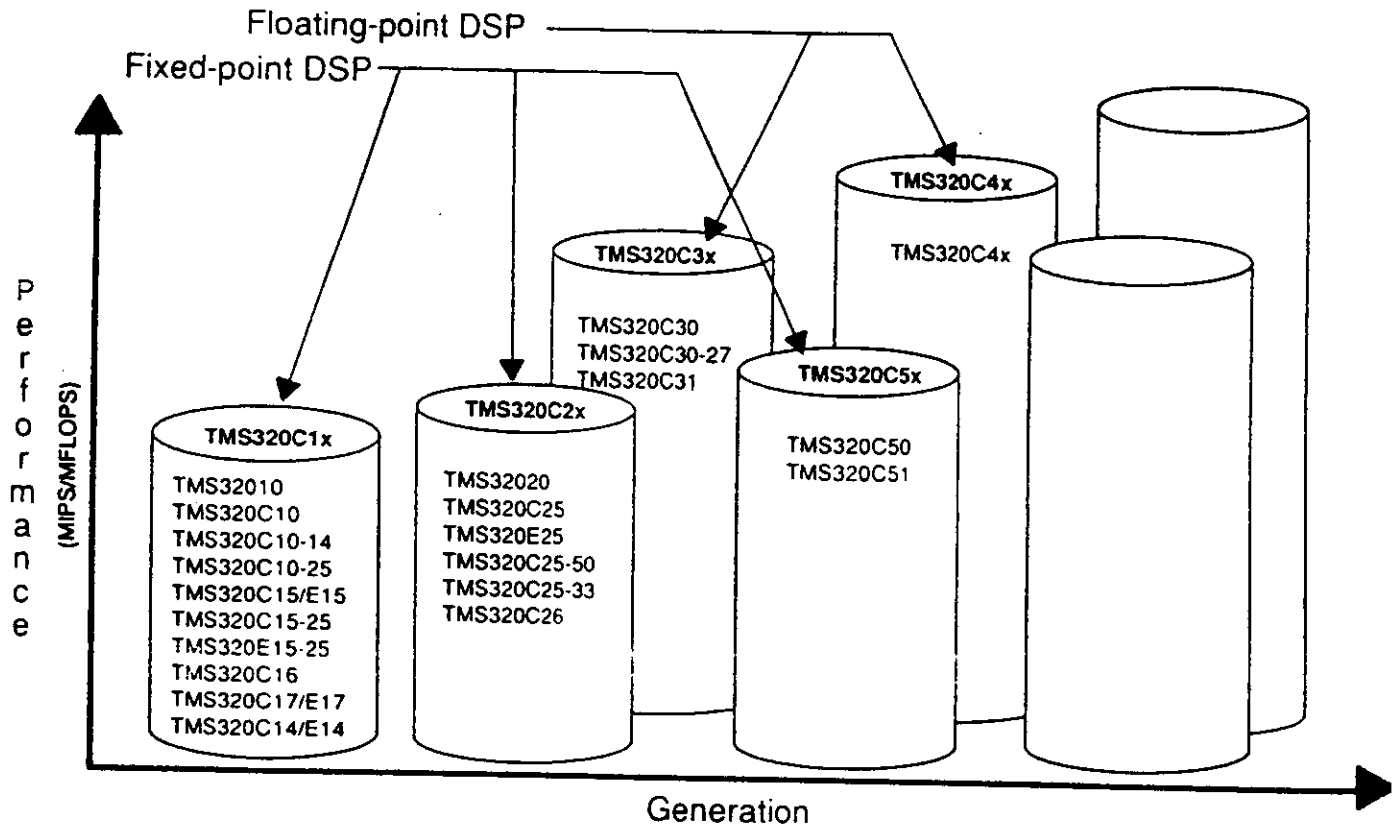
Miscellaneous Instructions

```
[IF condition]          SET | RESET | TOGGLE      FLAG_OUT ;
ENA | DIS               BIT_REV  [,...] ;
                        AV_LATCH
                        AR_SAT
                        SEC_REG
                        TIMER
                        G_MODE
                        M_MODE
[PUSH | POP STS]        [, POP  CNTR | PC | LOOP] ;  [,...]
MODIFY (Ia, Mb) ;
NOP;
```

TMS320 family

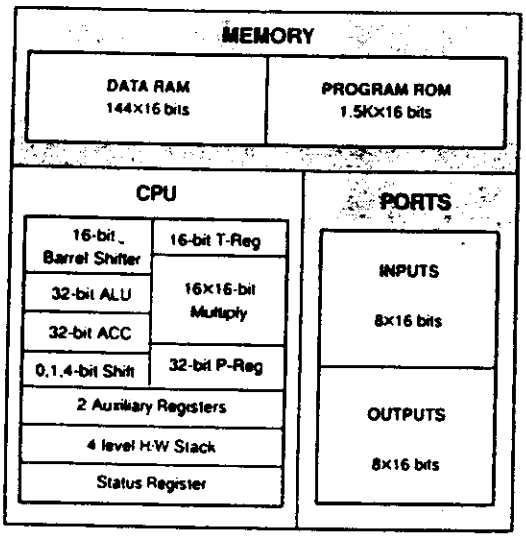
The TMS320 family consists of five generations: TMS320C1 TMS320C2x, TMS320C3x, TMS320C4x, and TMS320C5x (se Figure 1–1). The expansion includes enhancements of earlier generatior and more powerful new generations of digital signal processors.

TMS320 Device Evolution

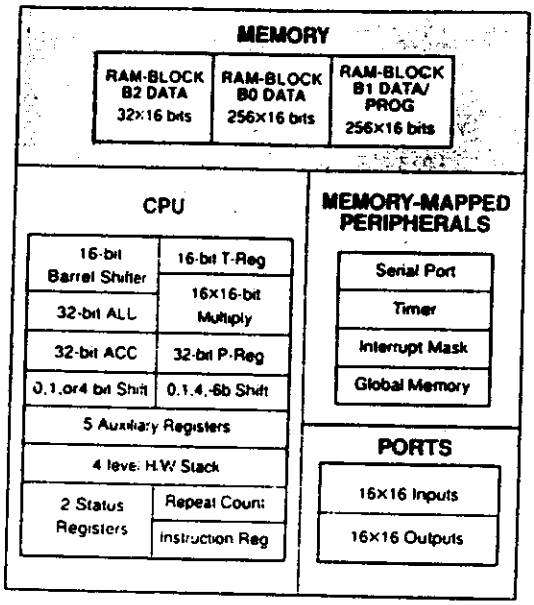


Typical Applications of the TMS320 Family

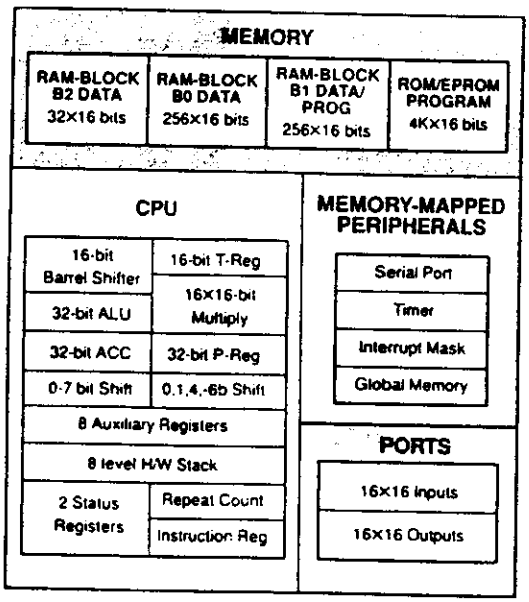
GENERAL-PURPOSE DSP	GRAPHICS/IMAGING	INSTRUMENTATION
Digital Filtering Convolution Correlation Hilbert Transforms Fast Fourier Transforms Adaptive Filtering Windowing Waveform Generation Discrete Cosine Transforms Hartley Transforms	3-D Rotation Robot Vision Image Transmission/ Compression Pattern Recognition Image Enhancement Homomorphic Processing Workstations Animation/Digital Map	Spectrum Analysis Function Generation Pattern Matching Seismic Processing Transient Analysis Digital Filtering Phase-Locked Loops
VOICE/SPEECH	**CONTROL**	**MILITARY**
Voice Mail Speech Vocoding Speech Recognition Speaker Verification Speech Enhancement Speech Synthesis Text-to-Speech	Disk Control Servo Control Robot Control Laser Printer Control Engine Control Motor Control	Secure Communications Radar Processing Sonar Processing Image Processing Navigation Missile Guidance Radio Frequency Modems
TELECOMMUNICATIONS		**AUTOMOTIVE**
Echo Cancellation ADPCM Transcoders Digital PBXs Line Repeaters Channel Multiplexing 1200 to 19200-bps Modems Adaptive Equalizers DTMF Encoding/Decoding Data Encryption Low Speed Transcoders/Vocoders ISDN Basic/Primary Rate Interfaces	FAX Cellular Telephones Speaker Phones Digital Speech Interpolation (DSI) X.25 Packet Switching Video Conferencing Spread Spectrum Communications Answering Machines	Engine Control Vibration Analysis Antiskid Brakes Adaptive Ride Control Global Positioning Navigation Voice Commands Digital Radio Cellular Telephones Active Suspension Noise Suppression
CONSUMER	**INDUSTRIAL**	**MEDICAL**
Radar Detectors Power Tools Digital Audio/TV Music Synthesizer Educational Toys Answering Machines	Robotics Numeric Control Security Access Power Line Monitors	Hearing Aids Patient Monitoring Ultrasound Equipment Diagnostic Tools Prosthetics Fetal Monitors



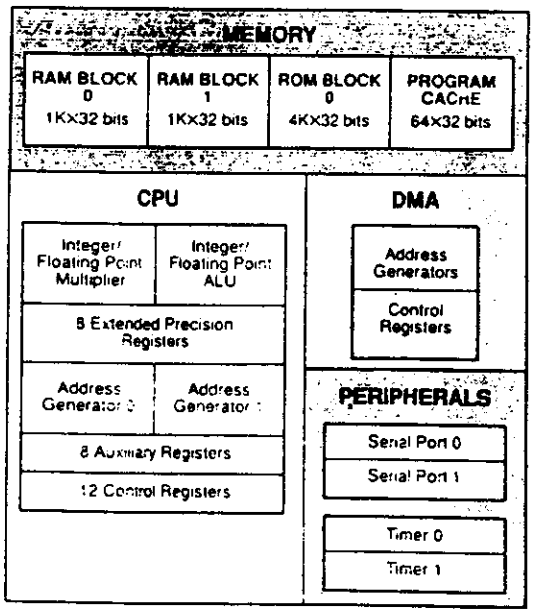
TMS320C10 Block Diagram



TMS32020 Block Diagram



TMS320C25/E25 Block Diagram



TMS320C30 Block Diagram

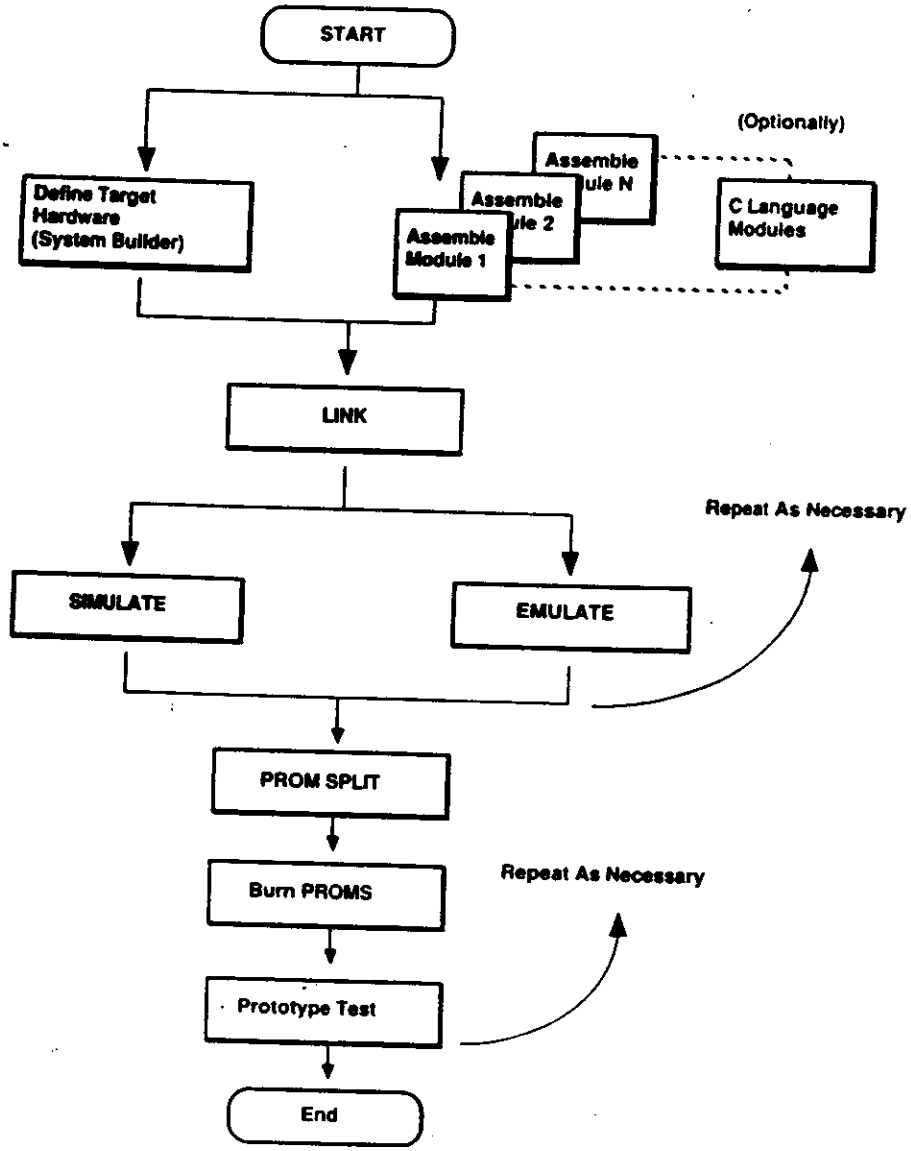
TMS320 DSP Family Benchmarks

BENCHMARK	TMS320C1x CYCLES	@160ns	TMS320C2x CYCLES	@80ns	TMS320C3x CYCLES	@60ns
FIR Filter						
20 Tap	49	127.5 KHz	29	431.03 KHz	25	667 KHz
64 Tap	133	47.0 KHz	73	171.73 KHx	73	228 KHz
67 Tap	139	45.0 KHz	76	164.47 KHz	76	219 KHz
IIR Filter						
4X Biquad	44	142.0 KHz	36	347.22 KHz	23	725 KHz
5X Biquad	56	111.6 KHz	43	284.09 KHz	27	617 KHz
Transpose Biquad	69	90.6 KHz	54	231.48 KHz	37	450 KHz
Dot Product	6	0.96 µs	6	0.48 µs	4	0.240 µs
Matrix Multiply						
2 x 2 Times 2 x 2	24	3.84 µs	21	1.7 µs	12	0.720 µs
3 x 3 Times 3 x 1	24	3.84 µs	22	1.8 µs	13	0.780 µs
Memory to Memory FFT						
64 Point Radix 2	3687	590 µs	3088	247 µs	2603	156 µs
256 Point Radix 2	41478	6.64 ms	17602	1.408 ms	12857	0.771 ms
1024 Point Radix 2	331237	53.0 ms	109755	8.784 ms	61511	3.67 ms
Port to Memory FFT						
64 Point Radix 2	2954	473 µs	1621	130 µs	1370	75 µs
256 Point Radix 2	41478	6.64 ms	8520	0.682 ms	6734	0.354 ms
1024 Point Radix 2	331237	53.0 ms	56286	4.503 ms	32354	1.67 ms

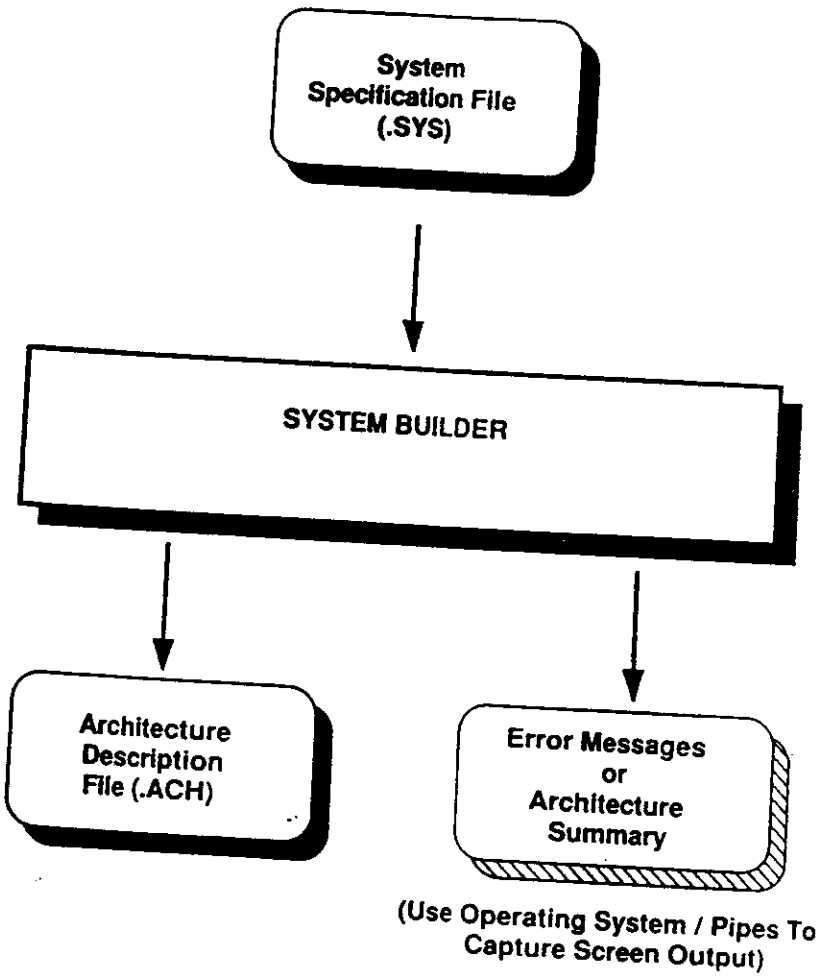
TMS320 DSP System Benchmarks

Application	TMS320C1x	TMS320C2x	TMS320C3x†
	CPU Loading		
Echo cancellation (CCITT G.165) Echo length 16 mS	–	50%	22%
Data encryption (ANSI X3.92-1981) Data rate 42 kbps	100%	52%	<25%
Split-band modem (CCITT V.22/212A) full-duplex	64%	30%	<14%
32-kbps ADPCM (CCITT G.721) half-duplex	100%	50%	<25%
2400-bps LPC-10 (DOD 43) half-duplex	76%‡·	40%‡	<17%
2400-bps LPC-10 (DOD 52) half-duplex	–	87%‡·	<35%
16-kbps subband coder full-duplex	64%‡	35%	<16%

† Conservative benchmarks based on extrapolated data.
‡ Requires external data memory
· Requires external program memory



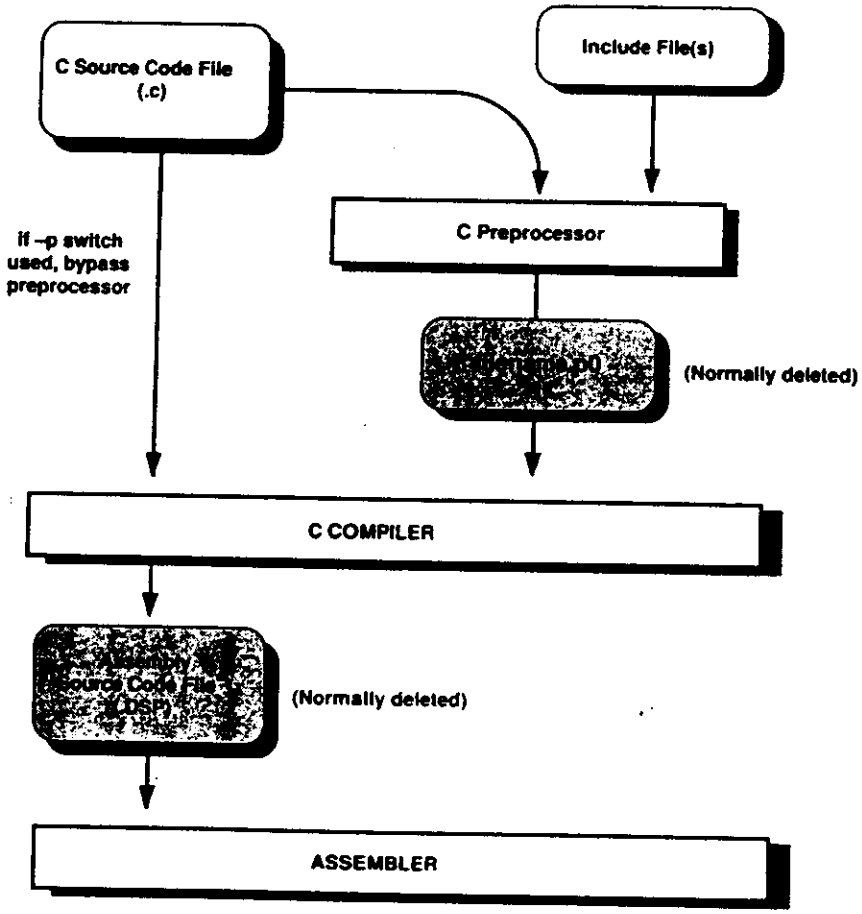
ADSP-2101 System Development Flow



The System Builder is invoked by typing

```
BLD21 filename[.ext]  [-switch]
```

where *filename.ext* is the system specification source file. The filename extension is optional and defaults to .SYS. There is one switch for invoking the System Builder. The -c switch makes the System Builder case-sensitive. This is provided primarily for compatibility with the C Compiler, which is always case-sensitive.



The Assembler List file (.LST) will contain C source in comment lines if an interlist merge file was requested.

C Compiler

Invoking The C Compiler

After you have written a C program and saved it as an ASCII text file on your software development system, you may invoke the C compiler to begin the process that creates an executable image for your selected target processor. Specify differences between programs written for the various processors of the ADSP-2100 Family with an *#include* file. Depending on which microprocessor you are using, you must place one of the following preprocessor directives in each of your C source files:

```
#include <ADSP2100.h> (for ADSP-2100 & ADSP-2100A systems)
#include <ADSP2101.h> (for ADSP-2101 systems)
#include <ADSP2105.h> (for ADSP-2105 systems)
#include <ADSP2111.h> (for ADSP-2111 systems)
#include <ADSP2150.h> (for ADSP-21msp50 systems)
```

Generally, you must place one of these directives before other preprocessor directives, declarations, and statements in your C source files. See Section 2.3.1, "#include," for the format of the *#include* preprocessor directive.

Invoke the compiler by entering the command **CC21** with the name of your input source filename, any desired switches, and an output filename as arguments. The square brackets below show that switches and *outfile* are optional.

```
CC21 infile [-switch...] [outfile]
```

The output files normally have a filename based on the same filename as the input file. If you give a second filename (shown above as *outfile*) when you invoke the compiler, the output file(s) use this name, instead of the input filename. Switches may come anywhere after **CC21**.

If you invoke the compiler with no argument, it displays a brief summary of the switches as explained below. This is the same as invoking the compiler with the **–help** switch.

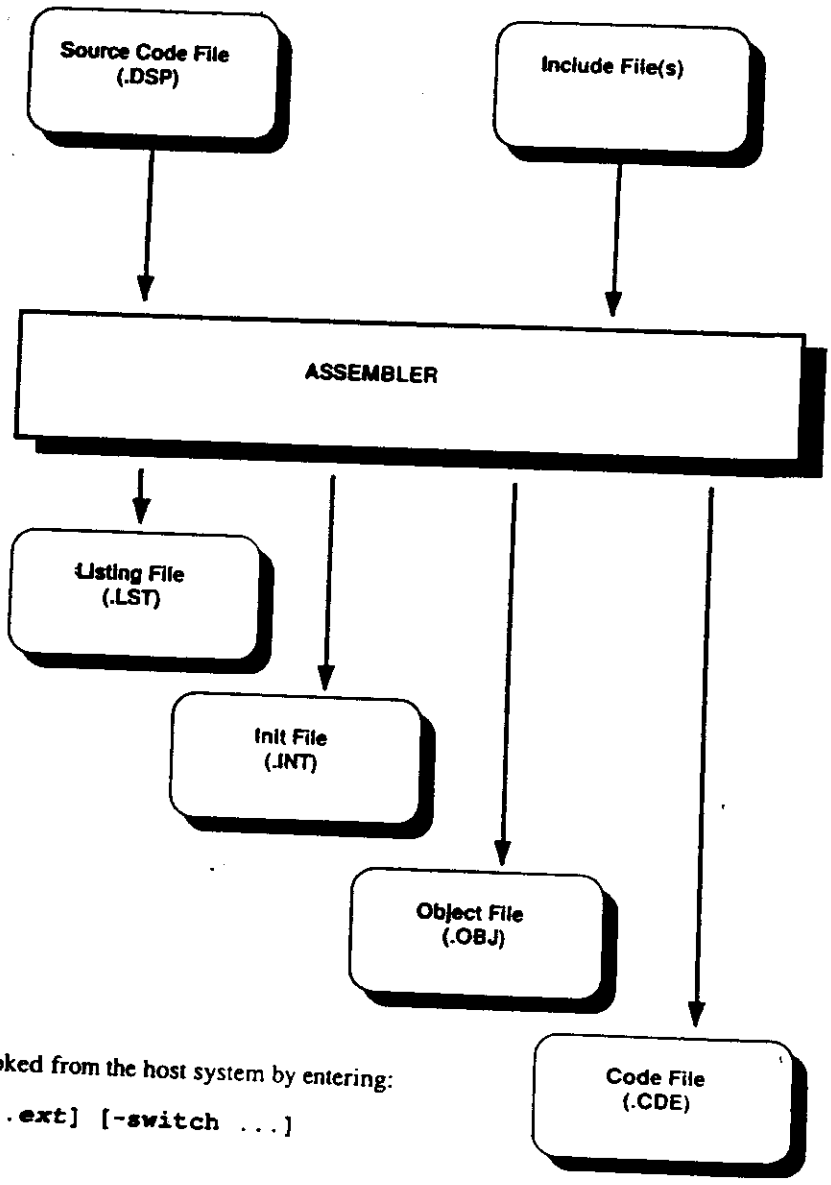
You may use any number of switches from the group shown in Table 2.1. Switches not self-explanatory are detailed in the following sections. Note that the compiler can generate a ROMable system.

C Compiler

Switch	*Action*	*Default*
–0	Save preprocessor output file (.P0)	Delete .P0 file
–1	Save compiler output file (.DSP)	Delete .DSP file
–a	Do not invoke assembler	Assembler invoked
–abs=*addr*	Specify absolute address for module	Location determined by linker
–b#[#...] *	Specify boot page(s) destination	No boot page information generated in assembly module
–crom	Code module placed in ROM	Code module in RAM
–D*macro*[=*exp*]	Define a macro	No such definition
–dmo	Modify **–gpm** switch action	**–gpm** switch forces all globals into PM, despite initial declaration
–dsppa	Invoke assembler's C preprocessor	Assembler invoked witho its C preprocessor
–e	Call floating-point emulation routines	Floating-point routines placed as inline code
–gpm	Force all global variables into program memory (PM)	Globals placed in data memory (DM), or as specif in initial declaration
–help	Display list of switches	Compile program
–I=*path*	Specify a unique search path for #includes	Only current directory & ADII path is searched
–lpm	String literals & switch tables in PM	Placement in DM
–lrom	String literals & switch tables in ROM	Placement in RAM
–m	Produce a merged listing file	No merged listing file
–noopt	Optimizing features disabled	Optimizing features enabl
–p	Invoke compiler without preprocessor	Preprocessor and compiler invoked
–pmstack	Stack placed in PM	Stack placed in DM
–pp	Invoke preprocessor without compiler	Preprocessor and compiler invoked
–w	Suppress warning messages	Display warning messages

* *Do not use the –b switch with the ADSP-2100*

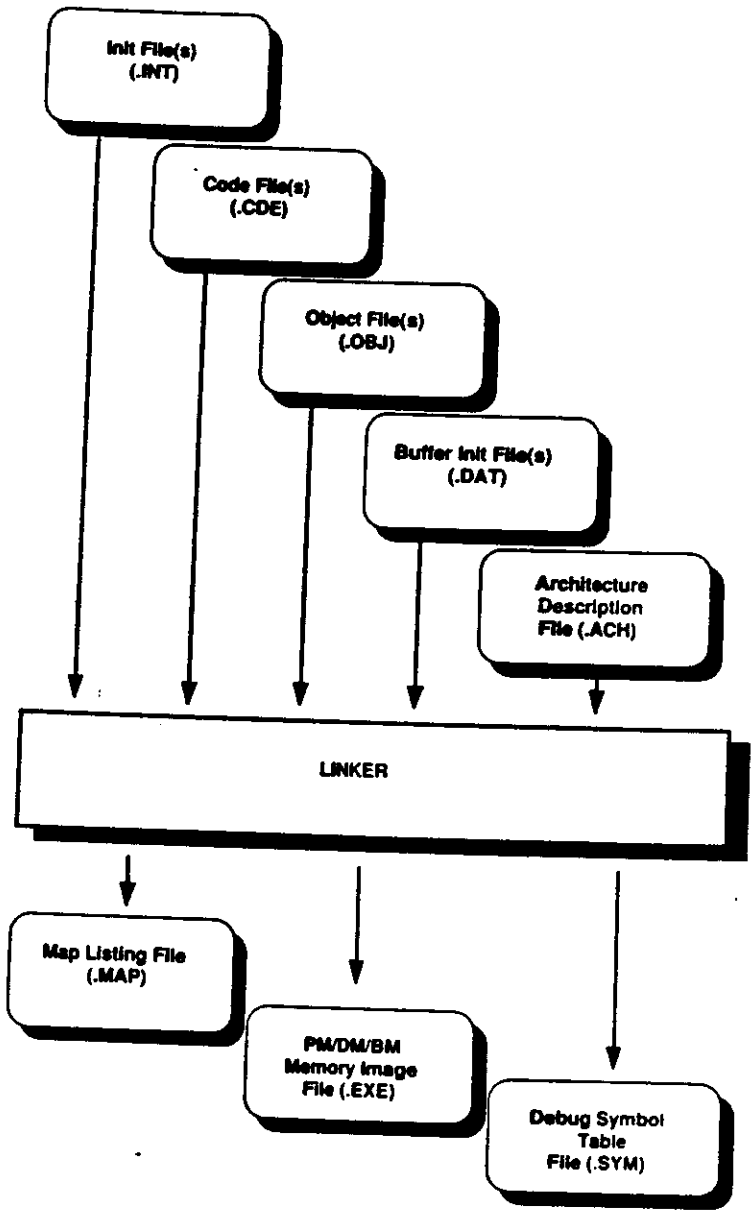
Table 2.1 Compiler Switches



The Assembler is invoked from the host system by entering:

ASM21 *filename*[.ext] [-switch ...]

Switch	Result
-cp	Runs C language preprocessor
-p	Runs standard preprocessor without core assembler
-d*variable*[=value]	Define *variable* for C preprocessor
-l	Creates .LST file
-m [number]	Macros expanded in .LST file, to depth of [number]
-i [number]	INCLUDE files expanded in .LST file, to depth of [number]
-s	No semantics checking
-c	Makes the Assembler case-sensitive


Init File(s)
(.INT)
Code File(s)
(.CDE)
Object File(s)
(.OBJ)
Buffer Init File(s)
(.DAT)
Architecture
Description
File (.ACH)
LINKER
Map Listing File
(.MAP)
PM/DM/BM
Memory Image
File (.EXE)
Debug Symbol
Table
File (.SYM)

Linker I/O

Running the Linker

To invoke the Linker from the host system, the command form is:

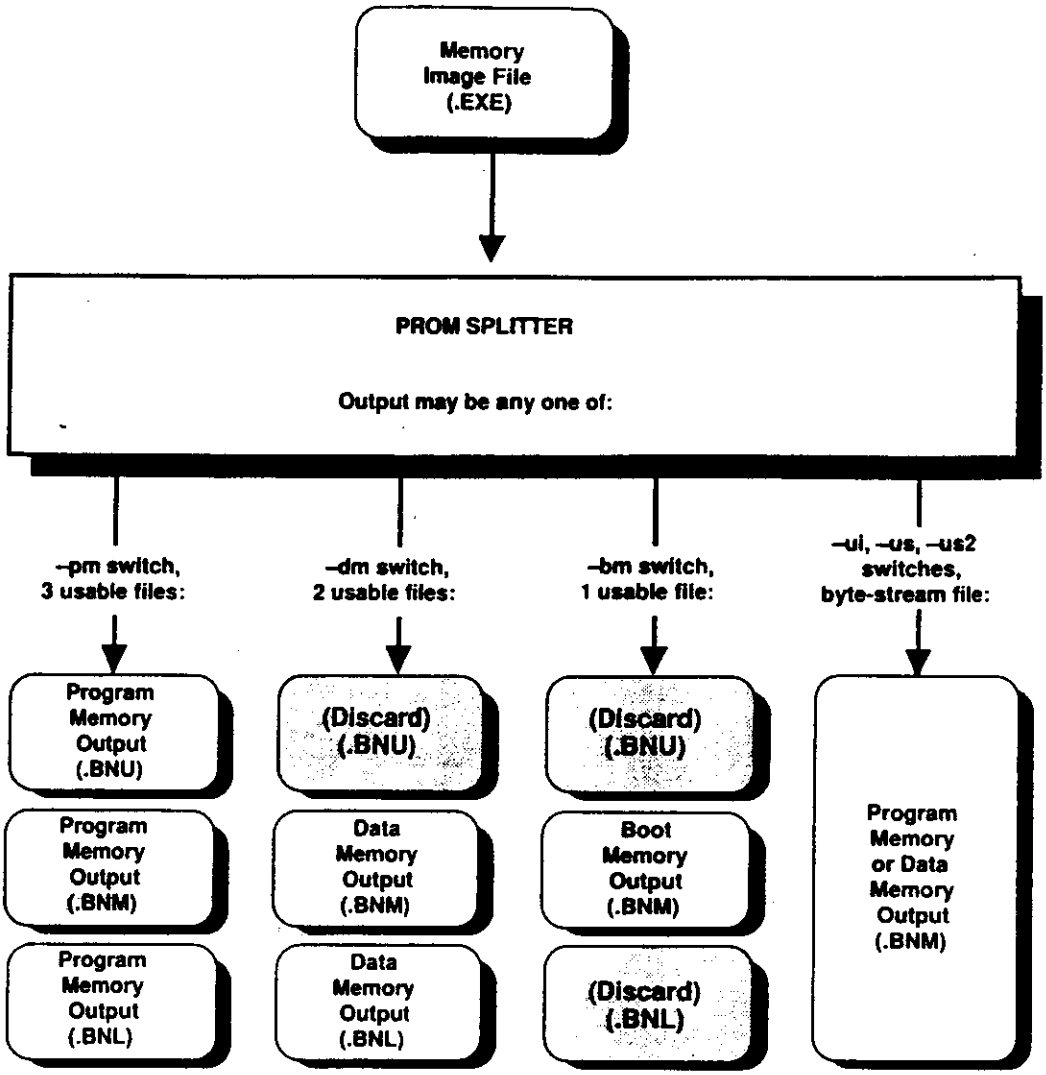
```
LD21 file1 [file2 .] [-switch .]
```

or

```
LD21 -i file_all [-switch .]
```

The -i switch causes the Linker to read the file *file_all* for a list of files to link. The file containing the list of files to link must be a simple text file with one pathname/file per line.

Switch	Result
-a *archname*	Use *archname.ACH* Architecture Description file instead of default 210x.ACH
-c	Linker creates "top of RAM" symbol to locate the stack; this symbol is used by programs generated with the ADSP-2101 C Compiler (See Chapter 7)
-dryrun	Linker does not generate an .EXE file; quick test to check for link errors
-e *target*	Output files named *target.EXE*, instead of default 210x.EXE
-g	Linker generates a debugger symbol table, .SYM file
-i *file_all*	Links all files listed in text file *file_all*
-lib *directory; ...*	Directories listed are added to those found in ADIL environment setting for locating libraries; multiple directories are separated by commas in Unix systems or by semicolons in PC-DOS systems
-old	Not used (ADSP-2100 feature)
-p	Library subroutines are assigned to the boot pages that call them
-pmstack	Used with -c switch; moves "top of RAM" symbol to program memory
-s *stack_size*	Used with -c switch; specify a maximum size for stack
-x	Linker generates a .MAP file



PROM Splitter I/O

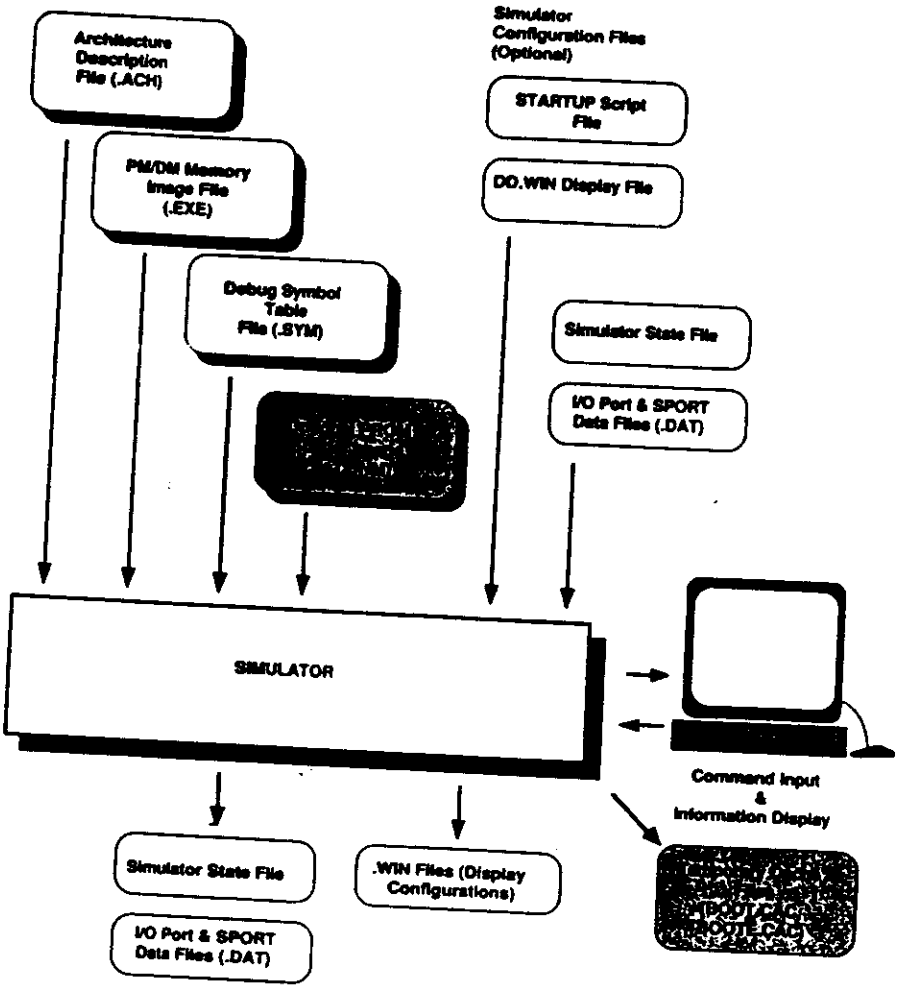
Five command line switches are used with the PROM splitter, one required and four optional:

switch	*possible forms*	
Memory space	**–pm, –dm, –bm**	*required*
PROM file format	**–s, –i, –us, –us2, –ui**	*optional: defaults to –s*
Boot page size	**–bs** *pagesize*	*optional: default=2K*
Boot page boundaries	**–bb** *boundary*	*optional: default=2K*
Boot loader	**–loader**	*optional*

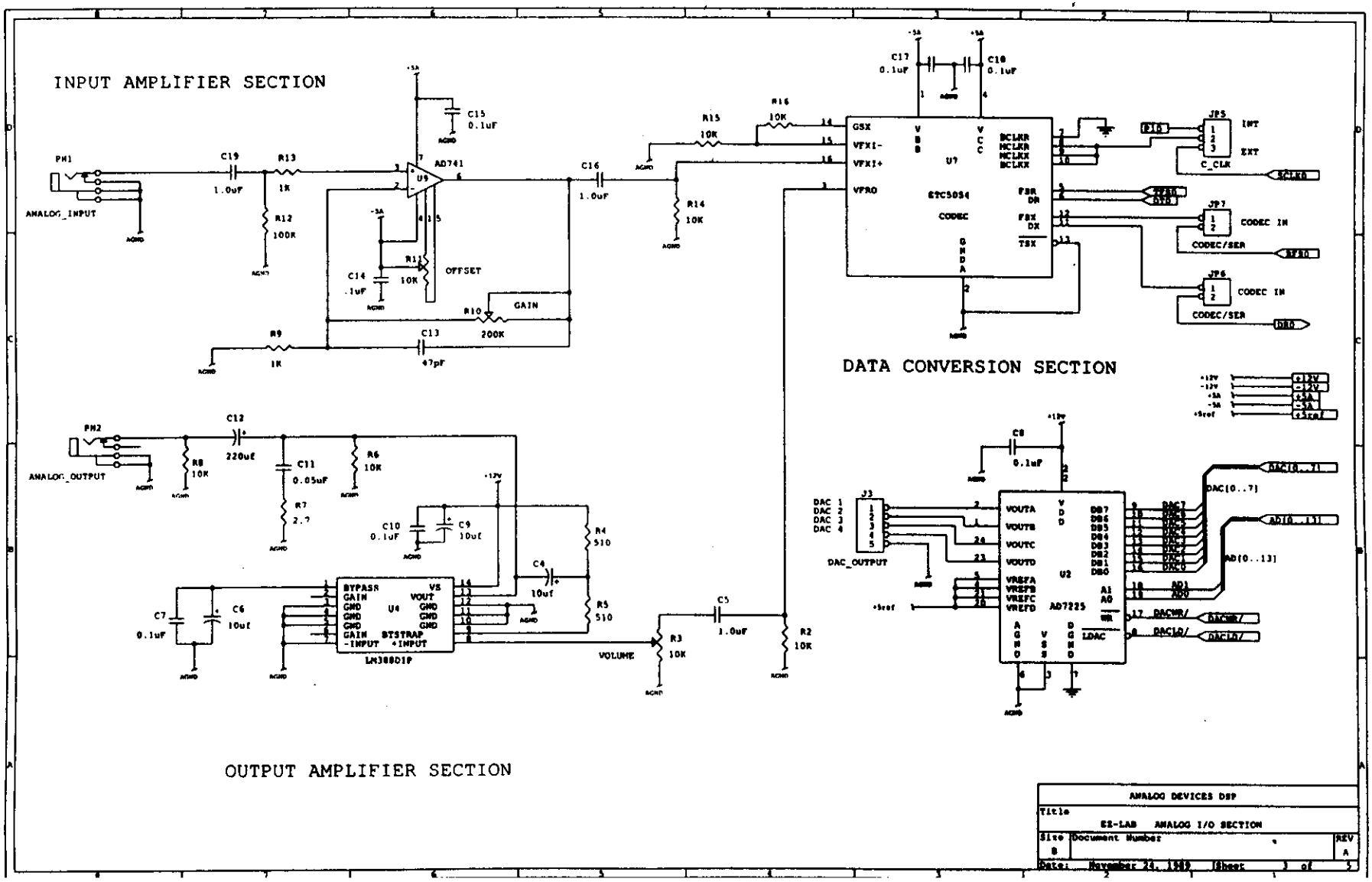
The memory space and PROM file format switches are defined as follows:

–pm	program memory	
–dm	data memory	
–bm	boot memory	*not used in ADSP-2100 systems*

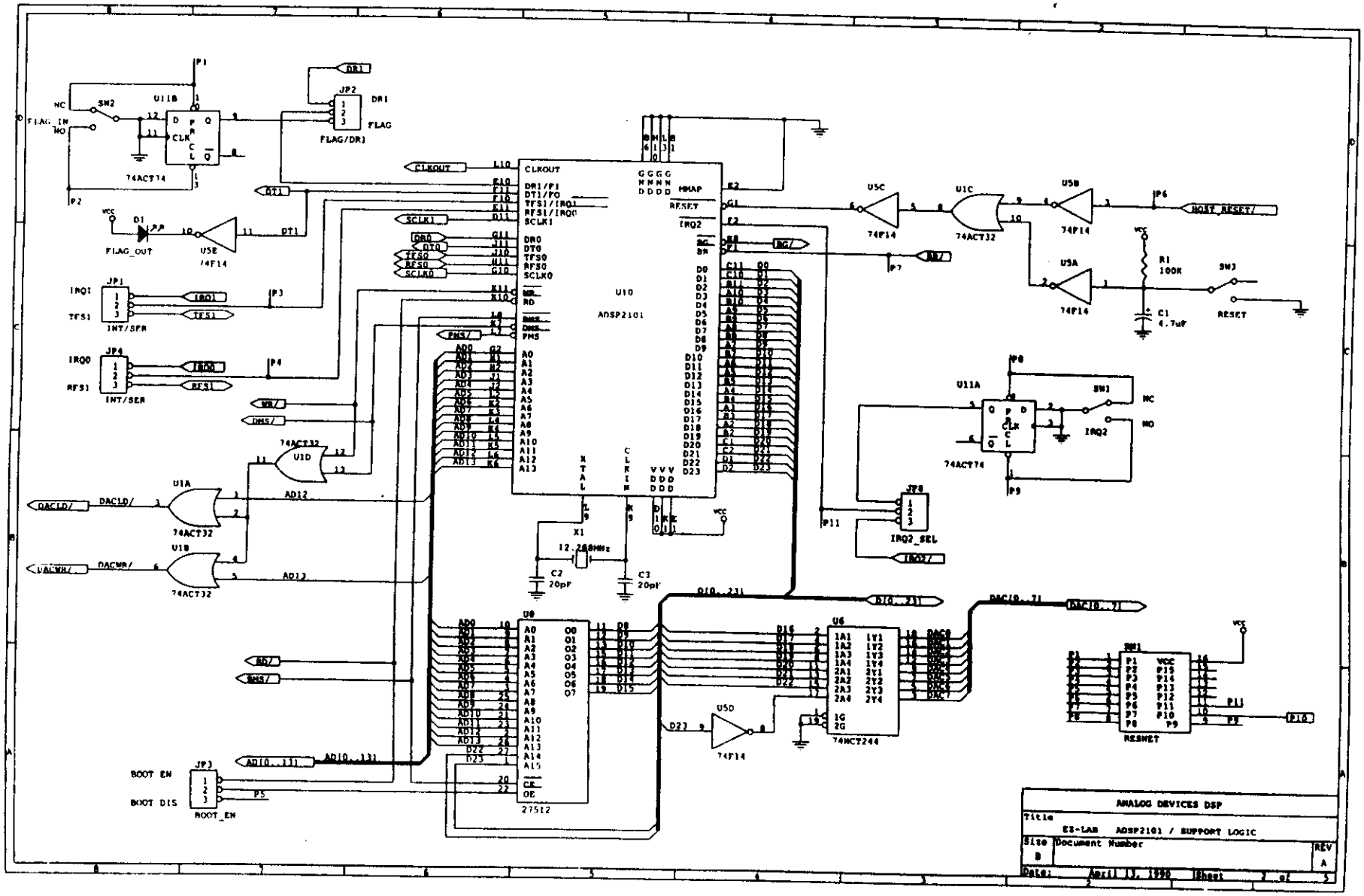
–s	Motorola S record	*default*
–i	Intel Hex record	
–us	Motorola S record byte-stream	*used with –pm or –dm only*
–us2	Motorola S2 record byte-stream	*used with –pm or –dm only*
–ui	Intel Hex record byte-stream	*used with –pm or –dm only*

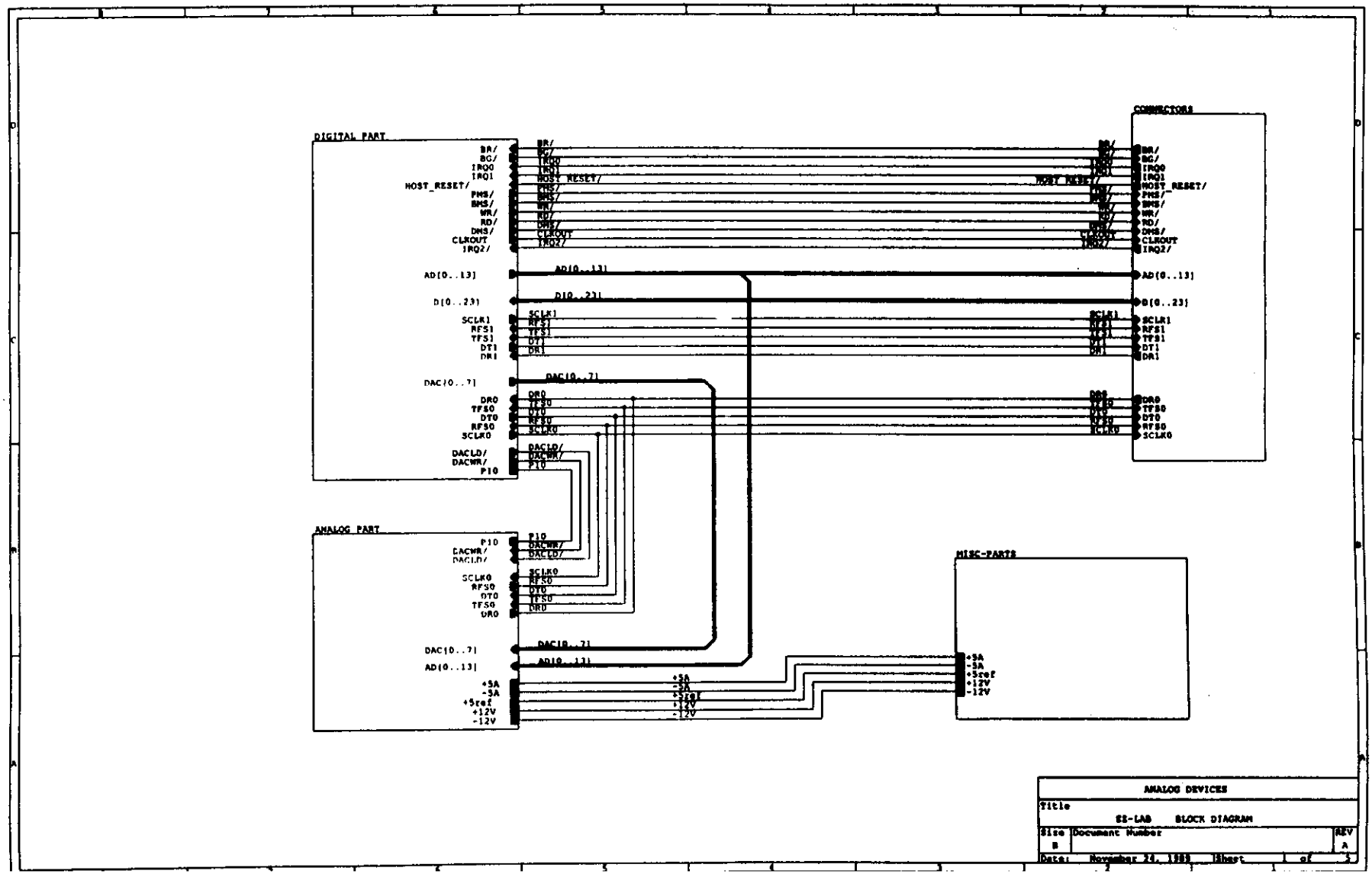


Simulator I/O


INPUT AMPLIFIER SECTION
DATA CONVERSION SECTION
OUTPUT AMPLIFIER SECTION
ANALOG_INPUT
ANALOG_OUTPUT
DAC_OUTPUT
AD741
OFFSET
GAIN
VOLUME
AD7225
ETC5054
LM388DIP
CODEC
CODEC IN
CODEC/SER
EXT
INT
C_CLK
SCLKR
DR0
DX0
AD[0..13]
DAC[0..7]
ANALOG DEVICES DSP
EZ-LAB ANALOG I/O SECTION

Schematics


ANALOG DEVICES DSP
Title
EZ-LAB ADSP2101 / SUPPORT LOGIC
Size B
Document Number
REV A
Date: April 13, 1990 Sheet 2 of 5
U10
ADSP2101
U11B
74ACT74
U5E
74F14
U5C
74F14
U1C
74ACT32
U5B
74F14
U5A
74F14
U11A
74ACT74
U1A
74ACT32
U1B
74ACT32
U1D
74ACT32
U5D
74F14
U6
74HCT244
U8
27512
JP1
INT/SER
JP4
INT/SER
JP2
FLAG/DR1
JP3
BOOT_EN
JP8
IRQ2_SEL
SW1
SW2
SW3
RESET
RESNET
X1
12.288MHz
C1
4.7uF
C2
20pF
C3
20pF
R1
100K
FLAG_OUT
FLAG_IN
BOOT EN
BOOT DIS
HOST_RESET/
IRQ2
RESET


DIGITAL PART
BR/
BG/
IRQ0
IRQ1
HOST_RESET/
PHS/
BHS/
WR/
RD/
DHS/
CLKOUT
IRQ2/
AD[0..13]
D[0..23]
SCLK1
RFS1
TFS1
DT1
DR1
DAC[0..7]
DR0
TFS0
DT0
RFS0
SCLK0
DACLD/
DACWR/
P10
ANALOG PART
MISC-PARTS
+5A
-5A
+5ref
+12V
-12V
CONNECTORS
AMALOG DEVICES
Title
ES-LAB BLOCK DIAGRAM
Size
B
Document Number
REV
A
Date: November 24, 1989 Sheet 1 of 5

National Semiconductor

TP3052, TP3053, TP3054 TP3054-1, TP3057, TP3057-1 "Ruggedized" Serial Interface CODEC/Filter COMBO® Family

General Description

The TP3052, TP3053, TP3054, TP3057 family consists of μ-law and A-law monolithic PCM CODEC/filters utilizing the A/D and D/A conversion architecture shown in *Figure 1*, and a serial PCM interface. The devices are fabricated using National's advanced double-poly CMOS process (micro-CMOS).

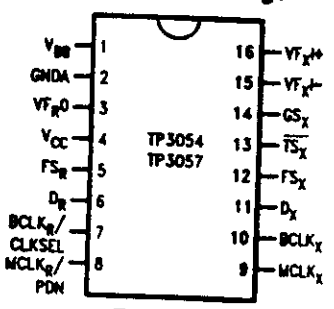
The encode portion of each device consists of an input gain adjust amplifier, an active RC pre-filter which eliminates very high frequency noise prior to entering a switched-capacitor band-pass filter that rejects signals below 200 Hz and above 3400 Hz. Also included are auto-zero circuitry and a companding coder which samples the filtered signal and encodes it in the companded μ-law or A-law PCM format. The decode portion of each device consists of an expanding decoder, which reconstructs the analog signal from the companded μ-law or A-law code, a low-pass filter which corrects for the sin x/x response of the decoder output and rejects signals above 3400 Hz followed by a single-ended power amplifier capable of driving low impedance loads. The devices require two 1.536 MHz, 1.544 MHz or 2.048 MHz transmit and receive master clocks, which may be asynchronous; transmit and receive bit clocks, which may vary from 64 kHz to 2.048 MHz; and transmit and receive frame sync pulses. The timing of the frame sync pulses and PCM data is compatible with both industry standard formats.

Features

- Complete CODEC and filtering system (COMBO) including:
  - — Transmit high-pass and low-pass filtering
  - — Receive low-pass filter with sin x/x correction
  - — Active RC noise filters
  - — μ-law or A-law compatible COder and DECoder
  - — Internal precision voltage reference
  - — Serial I/O interface
  - — Internal auto-zero circuitry
- μ-law with signaling, TP3020 or TP5116A timing—TP3052
- μ-law with signaling, TP5116A family timing—TP3053
- μ-law without signaling, 16-pin—TP3054
- A-law, 16-pin—TP3057
- Meets or exceeds all D3/D4 and CCITT specifications
- ±5V operation
- Low operating power—typically 60 mW
- Power-down standby mode—typically 3 mW
- Automatic power-down
- TTL or CMOS compatible digital interfaces
- Maximizes line interface card circuit density
- Dual-In-Line or PCC surface mount packages

Connection Diagrams

Dual-In-Line Package

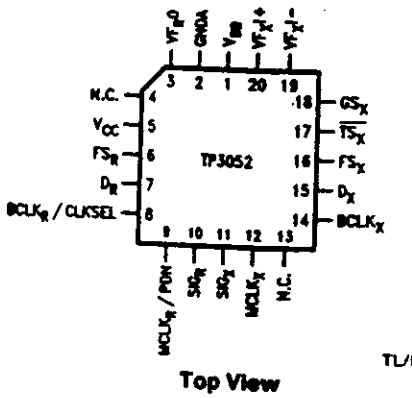


TL/H/5510-1

Top View

Order Number TP3054J, TP3054J-1, TP3057J or TP3057J-1
See NS Package Number J16A

Plastic Chip Carriers



TL/H/5510-10

Top View

Order Number TP3052V*
See NS Package Number V20A

*Available mid 1990

Block Diagram

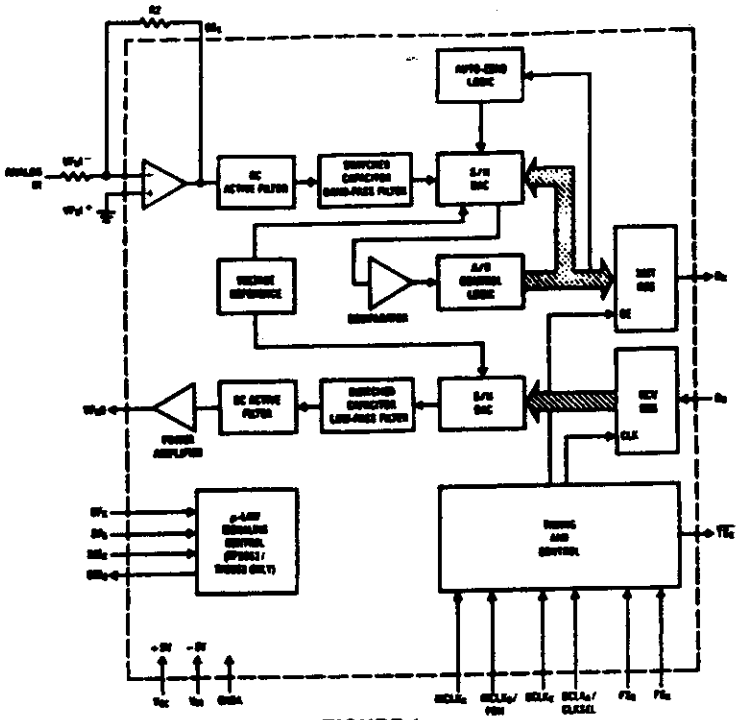


FIGURE 1 TL/H/5510-2

Pin Description

Symbol	Function
V_{BB}	Negative power supply pin. $V_{BB} = -5V \pm 5\%$.
GNDA	Analog ground. All signals are referenced to this pin.
VF_RO	Analog output of the receive power amplifier.
V_{CC}	Positive power supply pin. $V_{CC} = +5V \pm 5\%$.
FS_R	Receive frame sync pulse which enables $BCLK_R$ to shift PCM data into D_R. FS_R is an 8 kHz pulse train. See *Figures 2* and *3* for timing details.
D_R	Receive data input. PCM data is shifted into D_R following the FS_R leading edge.
$BCLK_R$/CLKSEL	The bit clock which shifts data into D_R after the FS_R leading edge. May vary from 64 kHz to 2.048 MHz. Alternatively, may be a logic input which selects either 1.536 MHz/1.544 MHz or 2.048 MHz for master clock in synchronous mode and $BCLK_X$ is used for both transmit and receive directions (see Table 1).
$MCLK_R$/PDN	Receive master clock. Must be 1.536 MHz, 1.544 MHz or 2.048 MHz. May be asynchronous with $MCLK_X$, but should be synchronous with $MCLK_X$ for best performance. When $MCLK_R$ is connected continuously low, $MCLK_X$ is selected for all internal timing. When $MCLK_R$ is connected continuously high, the device is powered down.

Symbol	Function
SF_R	When high during FS_R, this input indicates a receive signal frame.
SIG_R	The eighth bit of the PCM data appears at this output after each receive signalling frame.
SIG_X	Signal data input. Data at this input is inserted into the 8th bit of the PCM word during transmit signaling frames.
SF_X	When high during FS_X, this input indicates a transmit signaling frame.
$MCLK_X$	Transmit master clock. Must be 1.536 MHz, 1.544 MHz or 2.048 MHz. May be asynchronous with $MCLK_R$. Best performance is realized from synchronous operation.
FS_X	Transmit frame sync pulse input which enables $BCLK_X$ to shift out the PCM data on D_X. FS_X is an 8 kHz pulse train, see *Figures 2* and *3* for timing details.
$BCLK_X$	The bit clock which shifts out the PCM data on D_X. May vary from 64 kHz to 2.048 MHz, but must be synchronous with $MCLK_X$.
D_X	The TRI-STATE® PCM data output which is enabled by FS_X.
$\overline{TS_X}$	Open drain output which pulses low during the encoder time slot.
GS_X	Analog output of the transmit input amplifier. Used to externally set gain.
VF_XI^-	Inverting input of the transmit input amplifier.
VF_XI^+	Non-inverting input of the transmit input amplifier.

Functional Description

POWER-UP

When power is first applied, power-on reset circuitry initializes the COMBO and places it into a power-down state. All non-essential circuits are deactivated and the D_X and VF_RO outputs are put in high impedance states. To power-up the device, a logical low level or clock must be applied to the $MCLK_R$/PDN pin *and* FS_X and/or FS_R pulses must be present. Thus, 2 power-down control modes are available. The first is to pull the $MCLK_R$/PDN pin high; the alternative is to hold both FS_X and FS_R inputs continuously low—the device will power-down approximately 2 ms after the last FS_X or FS_R pulse. Power-up will occur on the first FS_X or FS_R pulse. The TRI-STATE PCM data output, D_X, will remain in the high impedance state until the second FS_X pulse.

SYNCHRONOUS OPERATION

For synchronous operation, the same master clock and bit clock should be used for both the transmit and receive directions. In this mode, a clock must be applied to $MCLK_X$ and the $MCLK_R$/PDN pin can be used as a power-down control. A low level on $MCLK_R$/PDN powers up the device and a high level powers down the device. In either case, $MCLK_X$ will be selected as the master clock for both the transmit and receive circuits. A bit clock must also be applied to $BCLK_X$ and the $BCLK_R$/CLKSEL can be used to select the proper internal divider for a master clock of 1.536 MHz, 1.544 MHz or 2.048 MHz. For 1.544 MHz operation, the device automatically compensates for the 193rd clock pulse each frame.

With a fixed level on the $BCLK_R$/CLKSEL pin, $BCLK_X$ will be selected as the bit clock for both the transmit and receive directions. Table 1 indicates the frequencies of operation which can be selected, depending on the state of $BCLK_R$/CLKSEL. In this synchronous mode, the bit clock, $BCLK_X$ may be from 64 kHz to 2.048 MHz, but must be synchronous with $MCLK_X$.

Each FS_X pulse begins the encoding cycle and the PCM data from the previous encode cycle is shifted out of the enabled D_X output on the positive edge of $BCLK_X$. After 8 bit clock periods, the TRI-STATE D_X output is returned to a high impedance state. With an FS_R pulse, PCM data is latched via the D_R input on the negative edge of $BCLK_X$ (or $BCLK_R$ if running). FS_X and FS_R must be synchronous with $MCLK_{X/R}$.

TABLE I. Selection of Master Clock Frequencies

$BCLK_R$/CLKSEL	Master Clock Frequency Selected	
	TP3057	TP3052 TP3053 TP3054
Clocked	2.048 MHz	1.536 MHz or 1.544 MHz
0	1.536 MHz or 1.544 MHz	2.048 MHz
1 (or Open Circuit)	2.048 MHz	1.536 MHz or 1.544 MHz

ASYNCHRONOUS OPERATION

For asynchronous operation, separate transmit and receive clocks may be applied. $MCLK_X$ and $MCLK_R$ must be 2.048 MHz for the TP3057, or 1.536 MHz, 1.544 MHz for the TP3052, 53, 54, and need not be synchronous. For best transmission performance, however, $MCLK_R$ should be synchronous with $MCLK_X$, which is easily achieved by applying only static logic levels to the $MCLK_R$/PDN pin. This will automatically connect $MCLK_X$ to all internal $MCLK_R$ functions (see Pin Description). For 1.544 MHz operation, the device automatically compensates for the 193rd clock pulse each frame. FS_X starts each encoding cycle and must be synchronous with $MCLK_X$ and $BCLK_X$. FS_R starts each decoding cycle and must be synchronous with $BCLK_R$. $BCLK_R$ must be a clock, the logic levels shown in Table 1 are not valid in asynchronous mode. $BCLK_X$ and $BCLK_R$ may operate from 64 kHz to 2.048 MHz.

SHORT FRAME SYNC OPERATION

The COMBO can utilize either a short frame sync pulse (the same as the TP3020/21 CODECs) or a long frame sync pulse. Upon power initialization, the device assumes a short frame mode. In this mode, both frame sync pulses, FS_X and FS_R, must be one bit clock period long, with timing relationships specified in *Figure 2*. With FS_X high during a falling edge of $BCLK_X$, the next rising edge of $BCLK_X$ enables the D_X TRI-STATE output buffer, which will output the sign bit. The following seven rising edges clock out the remaining seven bits, and the next falling edge disables the D_X output. With FS_R high during a falling edge of $BCLK_R$ ($BCLK_X$ in synchronous mode), the next falling edge of $BCLK_R$ latches in the sign bit. The following seven falling edges latch in the seven remaining bits. All four devices may utilize the short frame sync pulse in synchronous or asynchronous operating mode.

LONG FRAME SYNC OPERATION

To use the TP5116A/56A long frame mode, both the frame sync pulses, FS_X and FS_R, must be three or more bit clock periods long, with timing relationships specified in *Figure 3*. Based on the transmit frame sync, FS_X, the COMBO will sense whether short or long frame sync pulses are being used. For 64 kHz operation, the frame sync pulse must be kept low for a minimum of 160 ns. The D_X TRI-STATE output buffer is enabled with the rising edge of FS_X or the rising edge of $BCLK_X$, whichever comes later, and the first bit clocked out is the sign bit. The following seven $BCLK_X$ rising edges clock out the remaining seven bits. The D_X output is disabled by the falling $BCLK_X$ edge following the eighth rising edge, or by FS_X going low, whichever comes later. A rising edge on the receive frame sync pulse, FS_R, will cause the PCM data at D_R to be latched in on the next eight falling edges of $BCLK_R$ ($BCLK_X$ in synchronous mode). All four devices may utilize the long frame sync pulse in synchronous or asynchronous mode.

SIGNALING

The TP3052 and TP3053 μ-law COMBOs contain circuitry to insert and extract signaling information in the PCM data stream. The TP3052 is intended for short frame sync applications, and the TP3053 for long frame sync applications, although the TP3053 may also be used in short frame sync applications. The TP3054 and TP3057 have no provision for signaling.

Functional Description (Continued)

Signaling for the TP3052 is accomplished by applying a frame sync pulse two bit clock periods long, as shown in *Figure 2*. With FS_X two bit clock periods long, the data present at SIG_X input will be inserted as the LSB in the PCM data transmitted during that frame. With FS_R two bit clock periods long, the LSB of the PCM data read into the D_R input will be latched and appear on the SIG_R output pin until updated following the next signaling frame. The decoder will then interpret the lost LSB as "½" to minimize noise and distortion. This short frame signaling may also be implemented using the TP3053, providing SF_R and SF_X are left open circuit or tied low. The TP3052 is not capable of inserting or extracting signaling information in the long frame mode.

Signaling for the TP3053 may be accomplished in either short or long frame sync mode. The short mode signaling is the same as the TP3052. For long frame signaling, two additional frame sync pulses are required, SF_X and SF_R, which indicate transmit and receive signaling frames, respectively. With an SF_X signaling frame sync, the data present at the SIG_X input will be inserted as the LSB in the PCM data transmitted during that frame. With an SF_R signaling frame sync, the LSB of the PCM data at D_R will be latched and appear on the SIG_R output pin until the next signaling frame. The decoder will also do the "½" step interpretation to compensate for the loss of the LSB.

TRANSMIT SECTION

The transmit section input is an operational amplifier with provision for gain adjustment using two external resistors, see *Figure 4*. The low noise and wide bandwidth allow gains in excess of 20 dB across the audio passband to be realized. The op amp drives a unity-gain filter consisting of RC active pre-filter, followed by an eighth order switched-capacitor bandpass filter clocked at 256 kHz. The output of this filter directly drives the encoder sample-and-hold circuit. The A/D is of companding type according to μ-law (TP3052, TP3053, TP3054) or A-law (TP3057) coding conventions. A precision voltage reference is trimmed in manufacturing to provide an input overload (V_{MAX}) of nominally 2.5V peak (see table of Transmission Characteristics). The FS_X frame sync pulse controls the sampling of the filter output, and then the successive-approximation encoding cycle begins. The 8-bit code is then loaded into a buffer and shifted out through D_X at the next FS_X pulse. The total encoding delay will be approximately 165 μs (due to the transmit filter) plus 125 μs (due to encoding delay), which totals 290 μs. Any offset voltage due to the filters or comparator is cancelled by sign bit integration.

RECEIVE SECTION

The receive section consists of an expanding DAC which drives a fifth order switched-capacitor low pass filter clocked at 256 kHz. The decoder is A-law (TP3057) or μ-law (TP3052, TP3053, TP3054) and the 5th order low pass filter corrects for the sin x/x attenuation due to the 8 kHz sample/hold. The filter is then followed by a 2nd order RC active post-filter/power amplifer capable of driving a 600Ω load to a level of 7.2 dBm. The receive section is unity-gain. Upon the occurrence of FS_R, the data at the D_R input is clocked in on the falling edge of the next eight $BCLK_R$ ($BCLK_X$) periods. At the end of the decoder time slot, the decoding cycle begins, and 10 μs later the decoder DAC output is updated. The total decoder delay is ~ 10 μs (decoder update) plus 110 μs (filter delay) plus 62.5 μs (½ frame), which gives approximately 180 μs.