**E**

**S**

**I**

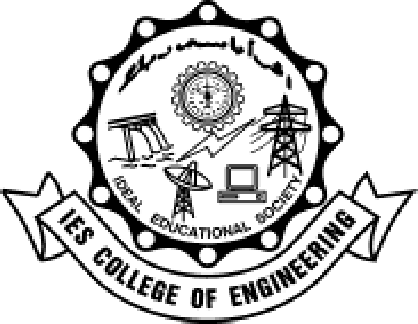
**COLLEGE OF ENGINEERIN****G CHITTILAPPILLY, Thrissur**. **Kerala – 680 551**

PH: (91) 0487 2309966, 2309967

[www.iesce.org](http://www.iesce.org/)

E-mail: [mail@iesce.org](mailto:mail@iesce.org)

(Approved by AICTE & Affiliated to APJ Abdul Kalam TechnologicalUniversity.)



**Department of Electronics & Communication Engineering ECL333**

# DIGITAL SIGNAL PROCESSING LABORATORY

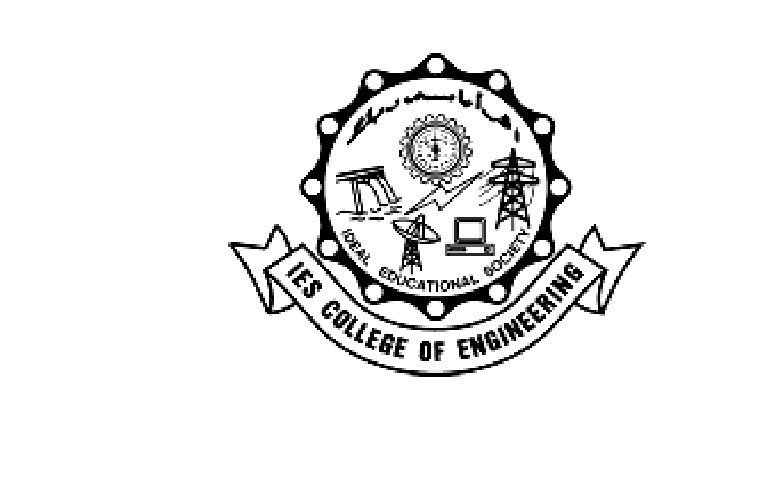
**COLLEGE OF ENGINEERING CHITTILAPPILLY**

**I I**

**E**

**S**

**Thrissur, Kerala – 680 551**



Name of the laboratory/ Workshop :

Name of the student :

Branch :……………………semester…………………

Roll No :…………………..Year……………………..

-------------------------------------------------------------------------------------------------------------------------------------

**CERTIFICATE**

*Certified that this is the bonafide record of practical work done by……………………………………………in the ……………………………………..of this institution during the year 2024-2025*

**Staff –In-Charge**

Date:…………………… Dept. Seal

**Head of the Department**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **SL. NO.** | **LIST OF EXPERIMENTS** | **CO** | **BTL** | **CYCLE** |
| **1** | **Simulation of Signals** | **CO1** | **3** | **CYCLE 1** |
| **2** | **Verification of the Properties of DFT** | **CO2** | **3** |
| **3** | **Familarization of DSP Hardware** | **CO3** | **3** | **CYCLE 2** |
| **4** | **Linear convolution** | **CO4** | **2** |
| **5** | **FFT of signals** | **CO5** | **3** |
| **6** | **IFFT with FFT** | **3** |
| **7** | **FIR low pass filter** | **CO6** | **2** |
| **8** | **Overlap Save Block Convolution** | **CO7** | **2** |
| **9** | **Overlap Add Block Convolution** | **3** |
| **10** | **Content beyond syllabus** | | **2** |

# INTRODUCTION TO MATLAB

‘MATLAB’ is a high performance for technical computing. It integrates computation visualization and programming in an easy to use environment where problem and solution are expresses in mathematical notation.

Typical uses includes

* Math and computation
* Algorithm development
* Modeling and proto typing
* Data analysis, exploration and visualization
* Scientific and engineering graphics
* Application development, including graphical user interface building

MATLAB is an interactive system whose basic data element is an array that doesn’t require dimensioning. This helps in solving many technical computing problems, especially those with matrix and vector formulation in a fraction of the time it would take to write a program in a scalar interactive language such as ’c’ or for tan.

The name ‘MATLAB’ stands for matrix laboratory.MATLAB was originally written to provide easy access to matrix software developed by the LINPACK and EISPACK projects, which together represents the state of art in software for matrix computation.

The ‘MATLAB’ system consists of five main parts

# THE MATLAB LANGUAGE:-.

This is high level matrix /array language with control flow statements functions, data structurers, IO features.

# MATLAB WORKING ENVIRONMENT:-

It includes facilities for managing the variables in your workspace and exporting data. It also includes tools for developing, managing and profiling M- files MATLAB’s applications.

# HANDLE GRAPHICS:-

This is the MATLAB graphic system. It includes high level commands for two dimensional and three dimensional data visualization, image processing, animation and presentation graphics.

# MATLAB MATHEMATICAL FUNCTION LIBRARY:-

This is a vast collection of ‘n’ computational algorithm ranging from elementary functions like sum, sine, and cosine to more sophisticated functions like matrix inverse, matrix Eigen value, Bessel function and FFT functions

# MATRIX APPLICTION PROGRAM INTERFACE (API)

This is a library that allows the uses to write ‘c’ and FORTAN programs that interact with MATLAB.

# TOOL BOX IN MATLAB

* + Communication tool box.
  + Control system tool box
  + Signal processing toolbox.
  + Data Acquisition tool box.
  + Filter design toolbox
  + Financial derivates toolbox.
  + Image processing
  + Instrument control toolbox.
  + Neutral network toolbox.
  + Wavelet toolbox.

# SIGNAL PROCESSING TOOL BOX:-

The signal processing toolbox in ‘MATLAB’facilites in analysis of signal and design of systems.

This contains a list of functions pertaining to the various areas of signal processing.

* Wave generation and plotting Eg.sine,cosine,
* Filter analysis and implementation eg.conv,filter
* Linear system and transformation
* IIR filter design
* FIR filter design eg.cremez
* IIR filter order selection.eg.butt ord.
* Transforms eg.DFT,FFT
* Statisti8cal signal processing:eg.csd,Pcov
* Windows eg.boxcar, triang.

**EXP NO: 1**

# GENERATION OF SIGNALS

**AIM**

Program to generate the following signals using MATLAB.

1. Unit impulse signal
2. Unit pulse signal
3. Unit ramp signal
4. Bipolar pulse
5. Triangular signal

# TOOLS REQUIRED:-

MATLAB

# PROGRAM

%% Impulse signal

clc;

clear all;

close all; p=[0];

q=[1];

subplot(5,1,1);

stem(p,q);

xlabel('n');

ylabel('x(n)'); title('impulse'); t=1:0.1:100;

%% Bipolar SQUARE WAVE a=0.7\*square(t); subplot(5,1,2);

plot(t,a);

title('square');

xlabel('time');

ylabel('amplitude');

%%RAMP WAVE

t=-10:10;

b=(t>=0).\*t;

subplot(5,1,3);

plot(t,b);

title('ramp');

xlabel('time');

ylabel('amplitude');

% PULSE WAVE

fs = 100E9; % sample freq

D = [2.5 10 17.5]' \* 1e-9; % pulse delay times t = 0 : 1/fs : 2500/fs; % signal evaluation time w = 1e-9; % width of each pulse(in nano seconds) yp = 1\*pulstran(t,D,@rectpuls,w);

subplot(5,1,4); plot(t\*1e9,yp); axis([0 25 -0.2 2]);

xlabel('Time (ns)');

ylabel('Amplitude');

%%The first pulse occur at 2.5ns with a width 1ns

% Sawtooth wave T = 10\*(1/50);

fs = 1000;

t = 0:1/fs:T-1/fs;

x = sawtooth(2\*pi\*50\*t,1/2); subplot(5,1,5);

plot(t,x); title('triangular'); xlabel('time'); ylabel('amplitude'); grid on;

**RESULT**

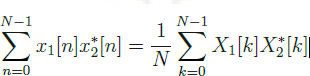
Basic continuous signals Unit impulse signal, Unit pulse signal, Unit ramp signal, Bipolar pulse, Triangular signal were generated.

**EXP NO: 2**

# VERIFICATION OF THE PROPERTIES OF DFT

**AIM**

1. Generate and appreciate a DFT matrix.
   * Write a function that returns the N point DFT matrix **VN** for a given N.
   * Plot its real and imaginary parts of **VN** as images using *matshow*or *imshow*commands (in Python) for *N* = 16, *N* = 64 and *N* = 1024
   * Compute the DFTs of 16 point, 64 point and 1024 point random sequences using the above matrices.
   * Observe the time of computations for *N* = 2*γ* for 2 *γ* 18
   * Use some iterations to plot the times of computation against *γ*. Plot and understand this curve. Plot the times of computation for the *fft*function over this curve and appreciate the computational saving with FFT.
2. Write a python function *circcon.py* that returns the circular convolution of an *N*1 point sequence and an *N*2 point sequence given at the input. The easiest way is to convert a linear convolution into circular convolution with *N* = *max*(*N*1*, N*2).
3. For the complex random sequences *x*1[*n*] and *x*2[*n*],



* + Generate two random complex sequences of say 5000 values.
  + Prove the theorem for these signals.

# TOOLS REQUIRED:-

GOOGLE COLAB

# THEORY DFT

The DFT X[k] of a finite length sequence x[n] defined for n=0… N-1 can be obtained by sampling its DTFT  on the  axis between 

at .



i.e.

Usin th common use notatio ,

.

It is possible to view the DFT equation as a linear transformation on the sequence

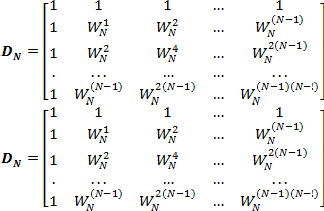
as:

 where is the vector composed of N DFT samples

 **, **is the vector of N

input samples and 

is the  DFT matrix given by



To understand how this is correct, consider a general linear transformation of the form .

We can write this matrix equation in scalar form as

 . If we rearrange the DFT equation as



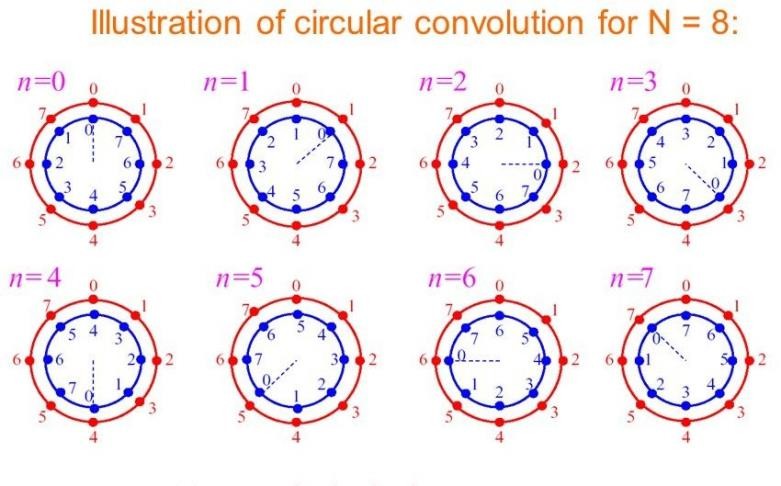
, we see that the two are similar in form. Just as the th element of is, the th element

(counting from 0) of is . This is exactly how the matrix  is arranged.

# CIRCULAR CONVOLUTION

The circular convolution of two N-point sequences g[n] and h[n] is another N point sequence y[n] defined as 

. To perform circular convolution graphically, N samples of g(n) are equally spaced around the outer circle in the clockwise direction, and N samples of h(n) are displayed on the inner circle in the counterclockwise direction starting at the same point. Corresponding samples on the two circles are multiplied, and the products are summed to form an output. The successive value of the circular convolution is obtained by rotating the inner circle



of one sample in the clockwise direction, and repeating the operation of computing the sum of corresponding products. This process is repeated until the first sample of inner circle lines up with the first sample of the exterior circle again.

# PARSEVAL’S RELATION

Parseval’s theorem states that the energy or power of a periodic signal in the time domain is equal to the energy or power in the frequency domain.

If G[k] denotes the N-point DFT of the length N sequence g[n], then:

.

# PROGRAM

**DFT**

#GENERATE AND APPRECIATE DFT MATRIX

import numpy as np from scipy import fft

import matplotlib.pyplot as plt import time

import random import math

N=int(input('how many point dft:'))

#program for direct computation of DFT start1=time.time()

V\_N=np.empty((N, N), dtype=np.cdouble); W=np.exp(-1j\*2\*np.pi/N)

k= np.arange(N)

for n in np.arange(N):

V\_N[:, n]= W\*\*(k\*n) np.round(V\_N)

xn = random.sample(range(0, 1500), N)

X=V\_N@xn; # @ is the matrix multiplication operator np.round(X)

end1=time.time()

#program for calculation of DFT using FFT function start2=time.time()

y=fft.fft(xn, axis=0) end2=time.time() print("Input sequence=",xn) print("Direct DFT of xn=",X) print("FFT of xn=",y) t1=end1 - start1

print("Runtime of the direct computation=",t1) t2=end2 - start2

print(f"Runtime of the fft computation=",t2) eff=100-((t2/t1)\*100)

print("Computational saving of FFT as compared to direct DFT=",eff, "%")

#to plot real and imaginary parts of V\_N plt.subplot(1, 3, 1) plt.title('$\mathrm{Re}(\mathrm{DFT}\_N)$')

plt.imshow(V\_N.real) plt.xlabel('Time (sample, index $n$)') plt.ylabel('Frequency (index $k$)') plt.subplot(1, 3, 2)

plt.title('$\mathrm{Im}(\mathrm{DFT}\_N)$') plt.imshow(V\_N.imag)

plt.xlabel('Time (samples, index $n$)') plt.ylabel('Frequency (index $k$)')

#to find value of gamma(no. of stages) gamma=math. log2(N) print("\u03B3=",gamma)

# CIRCULAR CONVOLUTION

#CIRCULAR CONVOLUTION

import numpy as np from scipy import signal g=np.array([1, 5, 0])

h=np.array([2, 3, 6, 7, 9, 10]) def circonv(g, h):

N1=g.size N2=h.size N=max(N1,N2)

y=np.zeros(N) if N1>N2:

h=np.append(h,np.zeros(N1-N2)) elif N2>N1: g=np.append(g,np.zeros(N2-N1))

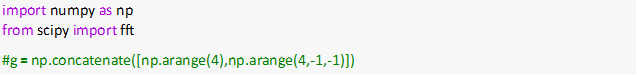
htr=np.concatenate([[h[0]], h[:0:-1]])#circular time-reversal for n in np.arange(N):

y[n] =np.sum(g\*htr) htr=np.roll(htr,1)#circular shift by 1 unit return y

print(circonv(g,h))

**PARSEVAL'S THEOROM**

# Verify Parseval’s relation for a sequence g[n]



print(g)

LHS = np.sum(g\*\*2) G = fft.fft(g)

RHS = 1/G.size \* np.sum(np.abs(G)\*\*2) print(LHS, RHS)

**EXPERIMENT 3**

**FAMILIARISATION OF DSP KIT-TMS320C6748**

**AIM:-**

To familarize with DSP kit -TMS320 C6748

# COMPONENTS USED:-

Computer, DSP kit, Function Generator Software used :- CCStudio V10.1.0

**THEORY**:-

The DSP kit VSK 6748 is based on TMS 320 C6748 processor which is used for low power dual core applications. TMS 320 C6748 is based on Texas instruments very long instruction word (VLIW) architecture. This processor is well suited for numerically intensive algorithms. The internal program memory is structured so that a total of eight instructions can be fetched every cycle. The internal C6748 has a clock rate of 375 MHz and is capable of fetching eight 32 bit instructions every 1/375 MHz or 2.67ns. The processor includes both floating point and fixed point architectures in one core.The C6748 includes 326KB of internal memory (32KB of LIP program RAM/ cache, 32 KB of LID data RAM/cache and 256 KB of L2 RAM/cache), eight functional units composed of six, ALUs and two multiplies units, an external memory interface addressing 256MB of 16 bit MDDR SDRAM and 64 32-bit general purpose registers. The C6748 CPU consists of eight functional units, two register files, and two data paths as shown in figure.

# PROGRAM COMPILATION PROCEDURE

**Step 1:**Open the Code Composer Studio10.1 Software.

**Step 2:**Click **Browse** and creat new folder for workspace and click **Launch** button

**Step 3:**Goto **File****New****Project.**

**Step 4:**Select **CSS project** and click **Next** button

**Step 5:**Select following details

|  |  |
| --- | --- |
| **Target** | **:** C674x Floating-point DSP /TMS320C6748 |
| **Connection** | **:** Texas Instruments XDS100v2 USB Debug Probe |

**Project name :** [Name] Press “**Finish**” Button

**Step 6:**Type the Program in Editor window and save it.

## Step 7:Click Project Show Build Settings

**Step 8:**Add **vsk-6748 common** files in **Include Options. Step 9:**Click **Project** **Build All.**

After finishing compilation OUT file is generated in project directory(**inside Debug folder**) If any case of error in program it will show in problem window (**View-->problem window**).

# DOWNLOADING PROCEDURE

## Step 1:Click view Target Configurations

**Step 2:**Right click on **User Defined**  **New Target Configurations Step 3:**Click **“Finish”** button

**Step 4:Connections** : Texas Instruments XDS100v2 USB Debug Probe.

**Board or Device** : TMS320C6748 Click **Save** Button.

**Step 5:**Right click on **User defined** and select **Launch Selected Configuration. Step 6:**In **Debug** window Right click on “**Texas Instuments XDS100v2...”**

and Click **Connect Target.**

**Step 7:**Now the device connected to PC properly means **Debug** window will appear as follows.

**Step 8:**Now click **Run** **Load**  **Load Program** and select [**.out**] file**. Step 9:**To Run a program Goto **Run**  **Resume (F8).**

# RESULT:-

Familiarized DSP kit.

# LED GLOW AND MIC AUDIO

**LED GLOW**

**Objective:** To glow the LED depending upon the SWITCH position in VSK-6748 kit.

## Procedure:

Load the (.out) file into vsk 6748 kit. Press Resume button to Run a program.

**Result:** When change is made in the switch position, the LED will glow correspondingly.

# MIC AUDIO

**Objective:** To interface Audio CODEC with TMS320C6748 Processor where the input is taken through Mic and the output is driven by the Speaker.

## Procedure:

Connect the input audio signal to MIC IN and Output Signal to SKR OUT Run the code.

**Result:** Interfaced Audio CODEC with TMS320C6748 Processor and output is driven by the Speaker.

# SWITCH LED

## Program:

**#include** "types.h"

**#include** "evmc6748.h"

**#include** "evmc6748\_gpio.h" **#include** "vi6748.h"

## int main(void)

{

uint8\_t \*XinSeq,i; XinSeq=(uint8\_t\*)0x80010000;

**EVMC6748\_lpscTransition**(PSC1, DOMAIN0, LPSC\_GPIO, PSC\_ENABLE); EVMC6748\_pinmuxConfig(PINMUX\_MCASP\_REG\_18, PINMUX\_MCASP\_MASK\_18,

PINMUX\_MCASP\_VAL\_18);

EVMC6748\_pinmuxConfig(PINMUX\_MCASP\_REG\_19, PINMUX\_MCASP\_MASK\_19, PINMUX\_MCASP\_VAL\_19);

EVMC6748\_pinmuxConfig(PINMUX\_MCASP\_REG\_1, PINMUX\_MCASP\_MASK\_1, PINMUX\_MCASP\_VAL\_1);

**for**(i=8;i<=15;i++)

{

VSK\_GPIO\_setDir(8, i, GPIO\_OUTPUT); VSK\_GPIO\_setDir(0, (i-8), GPIO\_INPUT);

}

**while**(1) {

**for**(i=8;i<=15;i++)

{

**GPIO\_getInput**(0,(i-8), XinSeq);

**GPIO\_setOutput**(8, i, OUTPUT\_HIGH );

}

}

}

## MIC AUDIO Program:

//

// \file evmc6748.c

// \brief implementation of initialization functions for C6748.

//

//

**#include** "VSK\_6748.h"

**#ifndef** NULL **#define** NULL 0 **#endif #ifdef** DEBUG **#include** "stdio.h"

## #endif

//

// Private Defines and Macros

//

// pinmux defines.

**#define** PINMUX\_MCASP\_REG\_0 (0)

**#define** PINMUX\_MCASP\_MASK\_0 (0x00FFFFFF)

**#define** PINMUX\_MCASP\_VAL\_0 (0x00111111)

**#define** PINMUX\_MCASP\_REG\_2 (2)

**#define** PINMUX\_MCASP\_MASK\_2 (0xFFFFFFFF)

**#define** PINMUX\_MCASP\_VAL\_2 (0x11111111)

//

// Private Defines and Macros

//

**#define** TIMER\_DIV (12)

**#define** TICKS\_PER\_US (2)

**#define** I2C\_PORT\_AIC3106 (I2C0)

//

// Private Defines and Macros

//

**#define** I2C\_PORT\_GPIO (I2C0)

**#define** I2C\_GPIO\_PIN\_MAX (16)

// config input/output pins (1 -> input, 0 -> output).

**#define** I2C\_GPIO\_CONFIG0\_EX (0x3F) **#define** I2C\_GPIO\_CONFIG1\_EX (0xFF) **#define**

I2C\_GPIO\_CONFIG0\_UI (0x0F) **#define** I2C\_GPIO\_CONFIG1\_UI (0xFF)

// TCA6416 command byte defines. **#define** CMD\_BYTE\_INPUT0 (0x00) **#define** CMD\_BYTE\_INPUT1 (0x01) **#define** CMD\_BYTE\_OUTPUT0 (0x02) **#define** CMD\_BYTE\_OUTPUT1 (0x03) **#define** CMD\_BYTE\_POLARITY0 (0x04) **#define** CMD\_BYTE\_POLARITY1 (0x05)

**#define** CMD\_BYTE\_CONFIG0 (0x06) **#define** CMD\_BYTE\_CONFIG1 (0x07)

**#define** PINMUX\_GPIO\_UI\_IO\_EXP\_REG (6)

**#define** PINMUX\_GPIO\_UI\_IO\_EXP\_MASK (0x0000000F)

**#define** PINMUX\_GPIO\_UI\_IO\_EXP\_VAL (0x00000008)

**#define** GPIO\_UI\_IO\_EXP\_BANK (2)

**#define** GPIO\_UI\_IO\_EXP\_PIN (7)

**#define** PINMUX\_I2C0\_REG (4)

**#define** PINMUX\_I2C0\_MASK (0x0000FF00)

**#define** PINMUX\_I2C0\_VAL (0x00002200)

**#define** PINMUX\_I2C1\_REG (4)

**#define** PINMUX\_I2C1\_MASK (0x00FF0000)

**#define** PINMUX\_I2C1\_VAL (0x00440000)

// i2c bus timeout.

**#define** I2C\_TIMEOUT (500000)

//

// Global Variable Initializations

//

//

// Static Variable Declarations

//

//

// Private Function Prototypes

//

//static uint32\_t init\_psc(void);

//static uint32\_t init\_clocks(void);

**static** i2c\_clk\_e g\_clock\_rate;

//static uint32\_t testAudioLineOut(void);

**static** uint32\_t **testAudioLineIn**(**void**);

//

// prints a chunk of flash data in a readable format.

//

**#ifdef** DEBUG

**void** UTIL\_printMem(uint32\_t begin\_addr, uint8\_t \*buffer, uint32\_t

length, uint8\_t continuation)

{

**#define** BYTES\_PER\_LINE 16

uint32\_t i, j, line\_end;

**if** (!continuation)

{

printf("\r\n\r\nPrint Data\r\n");

printf(" \r\n");

// print idices across the top.

printf("address ");

**for** (i = 0; i < BYTES\_PER\_LINE; i++)

{

printf("%02X ", i);

}

printf("\r\n");

}

// print data.

**for** (i = 0; i < length; i += BYTES\_PER\_LINE)

{

**if** (length > (i + BYTES\_PER\_LINE))

{

line\_end = (i + BYTES\_PER\_LINE);

}

**else**

{

line\_end = length;

}

**for** (j = i; j < line\_end; j++)

printf("\r\n"); 02X ", buffer[j]);

", (begin\_addr + i));

} printf("\n%08X

}

## #endif

printf("%

//

// looks for the UI gpio expander to see if UI board is attached.

//

uint8\_t **UTIL\_isUIBoardAttached**(**void**)

{

**if** (I2CGPIO\_init(I2C\_ADDR\_GPIO\_UI) == ERR\_NO\_ERROR)

**return** (1);

## else

}

**return** (0);

uint32\_t **MCASP\_init**(**void**)

{

// enable the psc and config pinmux for mcasp. EVMC6748\_lpscTransition(PSC1, DOMAIN0, LPSC\_MCASP0, PSC\_ENABLE); EVMC6748\_pinmuxConfig(PINMUX\_MCASP\_REG\_0, PINMUX\_MCASP\_MASK\_0,

PINMUX\_MCASP\_VAL\_0);

EVMC6748\_pinmuxConfig(PINMUX\_MCASP\_REG\_2, PINMUX\_MCASP\_MASK\_2, PINMUX\_MCASP\_VAL\_2);

// reset mcasp. MCASP->GBLCTL = 0;

// configure receive registers.

MCASP->RMASK = 0xFFFFFFFF; MCASP->RFMT = 0x00008078; MCASP->AFSRCTL

= 0x00000112; MCASP->ACLKRCTL = 0x000000AF; MCASP->AHCLKRCTL = 0x00000000; MCASP->RTDM = 0x00000003; MCASP->RINTCTL = 0x00000000; MCASP->RCLKCHK = 0x00FF0008;

// configure transmit registers. MCASP->XMASK = 0xFFFFFFFF; MCASP->XFMT = 0x00008078; MCASP->AFSXCTL = 0x00000112; MCASP->ACLKXCTL =

0x000000AF; MCASP->AHCLKXCTL = 0x00000000; MCASP->XTDM = 0x00000003; MCASP->XINTCTL = 0x00000000; MCASP->XCLKCHK = 0x00FF0008;

// config serializers (11 = xmit, 12 = rcv).

// MCASP->SRCTL11 = 0x000D;

// MCASP->SRCTL12 = 0x000E; MCASP->SRCTL3 = 0x000D; MCASP->SRCTL4

= 0x000E;

// config pin function and direction. MCASP->PFUNC = 0;

// MCASP->PDIR = 0x14000800; MCASP->PDIR= 0x14000008;

//

MCASP->DITCTL = 0x00000000; MCASP->DLBCTL = 0x00000000; MCASP->AMUTE

= 0x00000000;

MCASP->XSTAT = 0x0000FFFF; // Clear all MCASP->RSTAT = 0x0000FFFF;

// Clear all

**return** (ERR\_NO\_ERROR);

}

//

// Private Function Definitions

//

uint32\_t **AIC3106\_init**(**void**)

{

// select page 0 and reset codec. AIC3106\_writeRegister(AIC3106\_REG\_PAGESELECT, 0); AIC3106\_writeRegister(AIC3106\_REG\_RESET, 0x80);

// config codec regs. please see AIC3106 documentation for explination.

// Document Num: TLV320AIC3106 AIC3106\_writeRegister(3, 0x22); AIC3106\_writeRegister(4, 0x20); AIC3106\_writeRegister(5, 0x6E); AIC3106\_writeRegister(6, 0x23); AIC3106\_writeRegister(7, 0x0A); AIC3106\_writeRegister(8, 0x00); AIC3106\_writeRegister(9, 0x00); AIC3106\_writeRegister(10, 0x00); AIC3106\_writeRegister(15, 0x17); AIC3106\_writeRegister(16, 0x17); AIC3106\_writeRegister(17, 0x0F); AIC3106\_writeRegister(18, 0xF0); AIC3106\_writeRegister(19, 0x7C); AIC3106\_writeRegister(22, 0x7C); AIC3106\_writeRegister(25, 0x40); AIC3106\_writeRegister(27, 0);

AIC3106\_writeRegister(30, 0); AIC3106\_writeRegister(37, 0xE0); AIC3106\_writeRegister(38, 0x10); AIC3106\_writeRegister(43, 0); AIC3106\_writeRegister(44, 0); AIC3106\_writeRegister(47, 0x80); AIC3106\_writeRegister(51, 0x09); // 51 HPLOUT Output

<- [Mute=OFF][Power=ON] AIC3106\_writeRegister(58, 0); AIC3106\_writeRegister(64, 0x80); // 64 DAC\_R1 to HPROUT Volume

<- [Routed]

AIC3106\_writeRegister(65, 0x09); // 65 HPROUT Output

<- [Mute=OFF][Power=ON] AIC3106\_writeRegister(72, 0); AIC3106\_writeRegister(82, 0x80); AIC3106\_writeRegister(86, 0x09); AIC3106\_writeRegister(92, 0x80); AIC3106\_writeRegister(93, 0x09); AIC3106\_writeRegister(101, 0x01); AIC3106\_writeRegister(102, 0);

// AIC3106\_writeRegister(43, 0x28); //turn down the L DAC gain

// AIC3106\_writeRegister(44, 0x28); //turn down the R DAC gain

**return** (ERR\_NO\_ERROR);

}

//

// /brief Read data from a register on the AIC3106.

//

// /param uint8\_t in\_reg\_addr: The address of the register to be read from.

//

// /param uint8\_t \* dest\_buffer: Pointer to buffer to store retrieved data.

//

// /return uint32\_t ERR\_NO\_ERROR on sucess

//

//

uint32\_t **AIC3106\_readRegister**(uint8\_t in\_reg\_addr, uint8\_t

\*dest\_buffer)

{

uint32\_t rtn;

// write the register address that we want to read.

rtn = I2C\_write(I2C\_PORT\_AIC3106, I2C\_ADDR\_AIC3106, &in\_reg\_addr, 1, SKIP\_STOP\_BIT\_AFTER\_WRITE);

**if** (rtn != ERR\_NO\_ERROR)

**return** (rtn);

// clock out the register data.

rtn = I2C\_read(I2C\_PORT\_AIC3106, I2C\_ADDR\_AIC3106, dest\_buffer, 1, SKIP\_BUSY\_BIT\_CHECK);

**return** (rtn);

}

//

// /brief Write a register on the AIC3106.

//

// /param uint8\_t in\_reg\_addr: The address of the register to be written to.

//

// /param uint8\_t data: Data to be written to the register

//

// /return uint32\_t ERR\_NO\_ERROR on sucess

//

//

uint32\_t **AIC3106\_writeRegister**(uint8\_t in\_reg\_addr, uint8\_t in\_data)

{

uint32\_t rtn; uint8\_t i2c\_data[2];

i2c\_data[0] = in\_reg\_addr; i2c\_data[1] = in\_data;

// write the register that we want to read.

rtn = I2C\_write(I2C\_PORT\_AIC3106, I2C\_ADDR\_AIC3106, i2c\_data, 2, SET\_STOP\_BIT\_AFTER\_WRITE);

**return** (rtn);

}

//

// Private Function Definitions

//

uint32\_t **I2CGPIO\_init**(uint16\_t in\_addr)

{

uint32\_t rtn = ERR\_INVALID\_PARAMETER; uint8\_t i2c\_data[3];

**if** ((I2C\_ADDR\_GPIO\_EX == in\_addr) || (I2C\_ADDR\_GPIO\_UI == in\_addr))

{

// make sure polarity is not inverted. i2c\_data[0] = CMD\_BYTE\_POLARITY0; i2c\_data[1] = 0;

i2c\_data[2] = 0;

rtn = I2C\_write(I2C\_PORT\_GPIO, in\_addr, i2c\_data, 3, SET\_STOP\_BIT\_AFTER\_WRITE);

**if** (rtn != ERR\_NO\_ERROR)

**return** (rtn);

// set config regs on I/O expander.

**if** (I2C\_ADDR\_GPIO\_EX == in\_addr)

{

}

## else

{

}

i2c\_data[0] = CMD\_BYTE\_CONFIG0; i2c\_data[1] = I2C\_GPIO\_CONFIG0\_EX; i2c\_data[2] = I2C\_GPIO\_CONFIG1\_EX;

i2c\_data[0] = CMD\_BYTE\_CONFIG0; i2c\_data[1] = I2C\_GPIO\_CONFIG0\_UI; i2c\_data[2] = I2C\_GPIO\_CONFIG1\_UI;

rtn = I2C\_write(I2C\_PORT\_GPIO, in\_addr, i2c\_data, 3, SET\_STOP\_BIT\_AFTER\_WRITE);

}

**return** (rtn);

}

//

// \brief get gpio input from one pin of the i2c I/O expander.

//

// \param uint16\_t in\_addr - desired expander i2c address.

//

// \param uint8\_t in\_pin\_num - pin on expander to be read.

//

// \param uint8\_t \*data - gpio data from expander

// 0 -> pin is clear

// 1 -> pin is set

//

// \return uint32\_t

// ERR\_NO\_ERROR - input in bounds...gpio state returned in data.

// ERR\_INVALID\_PARAMETER - input out of bounds.

// else - something happened with i2c comm.

//

uint32\_t **I2CGPIO\_getInput**(uint16\_t in\_addr, uint8\_t in\_pin\_num, uint8\_t \*data)

{

uint32\_t rtn = ERR\_INVALID\_PARAMETER;

// check address and pin number.

**if** (((I2C\_ADDR\_GPIO\_EX == in\_addr) || (I2C\_ADDR\_GPIO\_UI == in\_addr)) && (in\_pin\_num < I2C\_GPIO\_PIN\_MAX) && (data != NULL))

{

uint8\_t i2c\_data; uint8\_t gpio\_bit = 0;

// set command byte to read appropriate input.

**if** (in\_pin\_num < 8)

{

}

## else

{

}

i2c\_data = CMD\_BYTE\_INPUT0; gpio\_bit = 1 << in\_pin\_num;

i2c\_data = CMD\_BYTE\_INPUT1; gpio\_bit = 1 << (in\_pin\_num - 8);

// send i2c command.

rtn = I2C\_write(I2C\_PORT\_GPIO, in\_addr, &i2c\_data, 1, SKIP\_STOP\_BIT\_AFTER\_WRITE);

**if** (rtn != ERR\_NO\_ERROR)

**return** (rtn);

// read the gpio data.

rtn = I2C\_read(I2C\_PORT\_GPIO, in\_addr, &i2c\_data, 1, SKIP\_BUSY\_BIT\_CHECK);

**if** (rtn != ERR\_NO\_ERROR)

**return** (rtn);

// check the input pin value and set var.

**if** (i2c\_data & gpio\_bit)

\*data = 1;

## else

}

\*data = 0;

**return** (rtn);

}

//

// \brief get gpio input from all pins of the i2c I/O expander.

//

// \param uint16\_t in\_addr - desired expander i2c address.

//

// \param uint16\_t \*data - gpio data from expander.

//

// \return uint32\_t

// ERR\_NO\_ERROR - input in bounds...gpio state returned in data.

// ERR\_INVALID\_PARAMETER - input out of bounds.

// else - something happened with i2c comm.

//

uint32\_t **I2CGPIO\_getInputAll**(uint16\_t in\_addr, uint16\_t \*data)

{

uint32\_t rtn = ERR\_INVALID\_PARAMETER;

**if** ((I2C\_ADDR\_GPIO\_EX == in\_addr) || (I2C\_ADDR\_GPIO\_UI == in\_addr)

&&

(data != NULL))

{

uint8\_t i2c\_data;

// send i2c command to read input0. i2c\_data = CMD\_BYTE\_INPUT0; rtn = I2C\_write(I2C\_PORT\_GPIO, in\_addr, &i2c\_data, 1,

SKIP\_STOP\_BIT\_AFTER\_WRITE);

**if** (rtn != ERR\_NO\_ERROR)

**return** (rtn);

// read the gpio data for input0.

rtn = I2C\_read(I2C\_PORT\_GPIO, in\_addr, &i2c\_data, 1, SKIP\_BUSY\_BIT\_CHECK);

**if** (rtn != ERR\_NO\_ERROR)

**return** (rtn);

// copy gpio data into var.

\*data = i2c\_data;

// send i2c command to read input1. i2c\_data = CMD\_BYTE\_INPUT1; rtn = I2C\_write(I2C\_PORT\_GPIO, in\_addr, &i2c\_data, 1,

SKIP\_STOP\_BIT\_AFTER\_WRITE);

**if** (rtn != ERR\_NO\_ERROR)

**return** (rtn);

// read the gpio data for input1.

rtn = I2C\_read(I2C\_PORT\_GPIO, in\_addr, &i2c\_data, 1, SKIP\_BUSY\_BIT\_CHECK);

**if** (rtn != ERR\_NO\_ERROR)

**return** (rtn);

// copy gpio data into var.

\*data += (i2c\_data << 8);

}

**return** (rtn);

}

//

// \brief set gpio output for one pin of the i2c I/O expander.

//

// \param uint16\_t in\_addr - desired expander i2c address.

//

// \param uint8\_t in\_pin\_num - pin on expander to be read.

//

// \param uint16\_t in\_val - 0/1 to set or clear the pin.

//

// \return uint32\_t

// ERR\_NO\_ERROR - pin set successfully.

// ERR\_INVALID\_PARAMETER - invalid pin number.

// else - something happened with i2c comm.

//

uint32\_t **I2CGPIO\_setOutput**(uint16\_t in\_addr, uint8\_t in\_pin\_num, uint16\_t in\_val)

{

uint32\_t rtn = ERR\_INVALID\_PARAMETER;

**if** (((I2C\_ADDR\_GPIO\_EX == in\_addr) || (I2C\_ADDR\_GPIO\_UI == in\_addr)) && (in\_pin\_num < I2C\_GPIO\_PIN\_MAX))

{

uint8\_t i2c\_data[2]; uint8\_t gpio\_bit = 0;

change

{

}

## else

{

}

// set command byte to read appropriate output, so we do not

// any data that we do not want to.

**if** (in\_pin\_num < 8)

i2c\_data[0] = CMD\_BYTE\_OUTPUT0; gpio\_bit = 1 << in\_pin\_num;

i2c\_data[0] = CMD\_BYTE\_OUTPUT1; gpio\_bit = 1 << (in\_pin\_num - 8);

// send i2c command.

rtn = I2C\_write(I2C\_PORT\_GPIO, in\_addr, i2c\_data, 1, SKIP\_STOP\_BIT\_AFTER\_WRITE);

**if** (rtn != ERR\_NO\_ERROR)

**return** (rtn);

// read the gpio data.

rtn = I2C\_read(I2C\_PORT\_GPIO, in\_addr, &i2c\_data[1], 1, SKIP\_BUSY\_BIT\_CHECK);

**if** (rtn != ERR\_NO\_ERROR)

**return** (rtn);

// update the data to set/clr bit for pin num.

**if** (in\_val)

SETBIT(i2c\_data[1], gpio\_bit);

## else

CLRBIT(i2c\_data[1], gpio\_bit);

// write the gpio data back to the I/O expander. rtn = I2C\_write(I2C\_PORT\_GPIO, in\_addr, i2c\_data, 2,

SET\_STOP\_BIT\_AFTER\_WRITE);

}

**return** (rtn);

}

//

// \brief set gpio output for all pins of the i2c I/O expander.

//

// \param uint16\_t in\_addr - desired expander i2c address.

//

// \param uint16\_t in\_val - pattern data to set I/O expander pins.

//

// \return uint32\_t

// ERR\_NO\_ERROR - pins set successfully.

// else - something happened with i2c comm.

//

uint32\_t **I2CGPIO\_setOutputAll**(uint16\_t in\_addr, uint16\_t in\_val)

{

uint32\_t rtn = ERR\_INVALID\_PARAMETER;

**if** ((I2C\_ADDR\_GPIO\_EX == in\_addr) || (I2C\_ADDR\_GPIO\_UI == in\_addr))

{

uint8\_t i2c\_data[3];

// load up the array with the cmd and input data.

i2c\_data[0] = CMD\_BYTE\_OUTPUT0; i2c\_data[1] = (uint8\_t) (in\_val & 0x00FF); i2c\_data[2] = (uint8\_t) (in\_val >> 8);

// write the gpio data to the I/O expander.

rtn = I2C\_write(I2C\_PORT\_GPIO, in\_addr, i2c\_data, 3, SET\_STOP\_BIT\_AFTER\_WRITE);

}

**return** (rtn);

}

uint32\_t **I2C\_init**(i2c\_regs\_t \*i2c, i2c\_clk\_e in\_clock\_rate)

{

// set the pinmux for the given i2c port.

**switch** ((uint32\_t)i2c)

{

**case** I2C0\_REG\_BASE:

EVMC6748\_pinmuxConfig(PINMUX\_I2C0\_REG, PINMUX\_I2C0\_MASK, PINMUX\_I2C0\_VAL);

## break;

**case** I2C1\_REG\_BASE:

EVMC6748\_lpscTransition(PSC1, DOMAIN0, LPSC\_I2C1, PSC\_ENABLE); EVMC6748\_pinmuxConfig(PINMUX\_I2C1\_REG, PINMUX\_I2C1\_MASK,

PINMUX\_I2C1\_VAL);

## break;

**default**:

**return** (ERR\_INIT\_FAIL);

}

// set global clock rate for future use. g\_clock\_rate = in\_clock\_rate;

// put i2c in reset. i2c->ICMDR = 0;

// configure clocks.

// set prescaler for ~8MHz interal i2c clock. i2c->ICPSC = 2;

**switch** (in\_clock\_rate)

{

// set prescaler and clock dividers to precomputed values for

// input clock rate.

**case** *I2C\_CLK\_100K*: i2c->ICCLKL = 35; i2c->ICCLKH = 35;

## break;

**case** *I2C\_CLK\_400K*: i2c->ICCLKL = 5; i2c->ICCLKH = 5;

## break;

}

// release i2c from reset. SETBIT(i2c->ICMDR, IRS);

**return** (ERR\_NO\_ERROR);

}

//

// \brief read data from i2c bus.

//

// \param i2c\_regs\_t \*i2c - pointer to reg struct for the desired i2c port.

//

// \param uint16\_t in\_addr - i2c address to read from.

//

// \param uint8\_t \*dest\_buffer - pointer to memory to copy the data being received.

//

// \param uint16\_t in\_length - number of bytes to receive.

//

// \return uint32\_t

// ERR\_NO\_ERROR - input in bounds, data received.

// ERR\_INVALID\_PARAMETER - null pointers.

//

uint32\_t **I2C\_read**(i2c\_regs\_t \*i2c, uint16\_t in\_addr, uint8\_t

\*dest\_buffer, uint16\_t in\_length, uint8\_t chk\_busy)

{

uint32\_t rtn = ERR\_INVALID\_PARAMETER;

**if** ((i2c != NULL) && (dest\_buffer != NULL))

{

uint32\_t cnt = 0; uint16\_t i;

info.

// wait for bus to be clear...we may want to skip this depending on which

// device we are talking to. see device datasheets for more

USTIMER\_delay(1000);

**if** (chk\_busy)

**while** (CHKBIT(i2c->ICSTR, BB)) {}

// set byte count and slave address. i2c->ICCNT

= in\_length;

i2c->ICSAR = in\_addr;

// configure i2c for master receive mode and release from reset. i2c->ICMDR = STT | MST | ICMDR\_FREE | IRS;

// receive data one byte at a time.

**for** (i = 0; i < in\_length; i++)

{

// do not want to send an ack on last byte.

**if** (i == (in\_length - 1))

{

SETBIT(i2c->ICMDR, NACKMOD);

}

// wait for data to be received. cnt = 0;

## do

{

**if** (cnt++ > I2C\_TIMEOUT)

{

// timed out waiting for data...reinit and return

error.

I2C\_init(i2c, g\_clock\_rate);

**return** (ERR\_TIMEOUT);

}

} **while** (!CHKBIT(i2c->ICSTR, ICRRDY));

dest\_buffer[i] = i2c->ICDRR;

}

// send stop condition. SETBIT(i2c->ICMDR, STP);

rtn = ERR\_NO\_ERROR;

}

**return** (rtn);

}

//

uint32\_t **I2C\_write**(i2c\_regs\_t \*i2c, uint16\_t in\_addr, uint8\_t

\*src\_buffer, uint16\_t in\_length, uint8\_t set\_stop)

{

uint32\_t rtn = ERR\_INVALID\_PARAMETER;

**if** ((i2c != NULL) && (src\_buffer != NULL))

{

uint32\_t cnt = 0; uint16\_t i;

// wait for bus to be clear. USTIMER\_delay(1000);

**while** (CHKBIT(i2c->ICSTR, BB)) {}

// set byte count and slave address. i2c->ICCNT = in\_length; i2c->ICSAR = in\_addr;

reset.

// configure i2c for master transmit mode and release from i2c->ICMDR = STT | MST | ICMDR\_FREE | TRX | IRS;

USTIMER\_delay(10);

// transmit data one byte at a time.

**for** (i = 0; i < in\_length; i++)

{

i2c->ICDXR = src\_buffer[i];

// wait for data to be copied to shift register. cnt = 0;

## do

{

error.

**if** (cnt++ > I2C\_TIMEOUT)

{

// timed out waiting for data...reinit and return

I2C\_init(i2c, g\_clock\_rate);

**return** (ERR\_TIMEOUT);

}

} **while** (!CHKBIT(i2c->ICSTR, ICXRDY));

}

**if** (set\_stop)

SETBIT(i2c->ICMDR, STP);

rtn = ERR\_NO\_ERROR;

}

**return** (rtn);

}

uint32\_t **testAudioLineIn**(**void**)

{

uint32\_t rtn = ERR\_NO\_ERROR;

SETBIT(MCASP->XGBLCTL, XHCLKRST);

**while** (!CHKBIT(MCASP->XGBLCTL, XHCLKRST)) {} SETBIT(MCASP->RGBLCTL, RHCLKRST);

**while** (!CHKBIT(MCASP->RGBLCTL, RHCLKRST)) {}

SETBIT(MCASP->XGBLCTL, XCLKRST);

**while** (!CHKBIT(MCASP->XGBLCTL, XCLKRST)) {} SETBIT(MCASP->RGBLCTL, RCLKRST);

**while** (!CHKBIT(MCASP->RGBLCTL, RCLKRST)) {}

SETBIT(MCASP->XGBLCTL, XSRCLR);

**while** (!CHKBIT(MCASP->XGBLCTL, XSRCLR)) {} SETBIT(MCASP->RGBLCTL, RSRCLR);

**while** (!CHKBIT(MCASP->RGBLCTL, RSRCLR)) {}

/\* Write a 0, so that no underrun occurs after releasing the state machine \*/ MCASP->XBUF3 = 0;

SETBIT(MCASP->XGBLCTL, XSMRST);

**while** (!CHKBIT(MCASP->XGBLCTL, XSMRST)) {} SETBIT(MCASP->RGBLCTL, RSMRST);

**while** (!CHKBIT(MCASP->RGBLCTL, RSMRST)) {}

SETBIT(MCASP->XGBLCTL, XFRST);

**while** (!CHKBIT(MCASP->XGBLCTL, XFRST)) {} SETBIT(MCASP->RGBLCTL, RFRST);

**while** (!CHKBIT(MCASP->RGBLCTL, RFRST)) {}

**while**(!CHKBIT(MCASP->SRCTL3, XRDY)) {} MCASP->XBUF3 = 0;

**return** (rtn);

}

// audio codec interface using MCASP

## int main(void)

{

int32\_t dat; USTIMER\_init(); I2C\_init(I2C0, *I2C\_CLK\_400K*);

MCASP\_init();

AIC3106\_init();

testAudioLineIn();

**while**(1) {

**while** (!CHKBIT(MCASP->SRCTL3, XRDY)) {} dat = MCASP->XBUF4; MCASP->XBUF3 = dat \* 3;

}

}

# EXPERIMENT 4

# LINEAR CONVOLUTION

**Objective:** Write a C function for the linear convolution of two arrays

**Outcome**: Students will be able to perform the linear convolution of two arrays and to implement it on the DSP hardware

**Requirement**: Turbo C, DSP hardware

**Theory:** The linear convolution of two finite duration sequences x[n] and h[n] is given below. It is assumed that x[n] extends from n=0 to L-1 and h[n] extends from n=0 to M-1. Length of y[n] is then L+M-1. The function **conv()** implements the convolution operation given by The sequence x[n] is reflected and shifted. For each n, the range over which both sequences overlap needs to be determined.

## Algorithm:

1. Obtain the input signal and the impulse response as two distinct arrays.
2. Make the length of both the arrays same.
3. For each value of n, the sum of outputs is calculated by taking a different X(k) value in each iteration.
4. This result is stored in an array y(n).
5. Display the value of output,y(n).
6. Stop

## Output:

Linear convolution using convolution sum formula output response y =

**Result:** Linear convolution of two arrays is performed and implemented on DSP hardware

**Linear Convolution Program:**

**#include**<math.h> **#include**<stdio.h>

## void main()

{

**int** \*Xn,\*Hn,\*Output;

**int** \*XnLength,\*HnLength;

**int** i,k,n,l,m;

Xn=(**int** \*)0x80010000; //input x(n) Hn=(**int** \*)0x80011000; //input h(n) XnLength=(**int** \*)0x80012000; //x(n) length HnLength=(**int** \*)0x80012004; //h(n) length Output=(**int** \*)0x80013000; // output address

l=\*XnLength; // copy x(n) from memory address to variable l m=\*HnLength; // copy h(n) from memory address to variable m

**for**(i=0;i<(l+m-1);i++) // memory clear

{

Output[i]=0; // o/p array Xn[l+i]=0; // i/p array Hn[m+i]=0; // i/p array

}

**for**(n=0;n<(l+m-1);n++)

{

**for**(k=0;k<=n;k++)

{

Output[n] =Output[n] + (Xn[k]\*Hn[n-k]); // convolution operation.

}

}

}

# EXPERIMENT 5 IMPLEMENTATION OF N-POINT FFT OF A SIGNAL

**Objective:** To compute the N-Point FFT of a given sequence and to implement it in DSP hardware **Outcome**: Students will be able to perform the Fast fourier transform of a given sequence and to implement it on the DSP hardware

**Requirement**: Turbo C, DSP hardware

**Theory:** A fast Fourier transform (FFT) is an algorithm that computes the discrete Fourier transform (DFT) of a sequence, or its inverse (IDFT). Fourier analysis converts a signal from its original domain (often time or space) to a representation in the frequency domain and vice versa. The DFT is obtained by decomposing a sequence of values into components of different frequencies. This operation is useful in many fields, but computing it directly from the definition is often too slow to be practical. An FFT rapidly computes such transformations by factorizing the DFT matrix into a product of sparse factors. A fast Fourier transform (FFT) is an algorithm that calculates the discrete Fourier transform (DFT) of some sequence – the discrete Fourier transform is a tool to convert specific types of sequences of functions into other types of representations. Another way to explain discrete Fourier transform is that it transforms the structure of the cycle of a waveform into sine components.

**Algorithm:** The FFT has a fairly easy algorithm to implement, and it is shown step by step in the list below. This version of the FFT is the Decimation in Time Method

1. Pad input sequence, of N samples, with Zero’s until the number of samples is the nearest power of two.
2. Bit reverse the input sequence.
3. Compute (N / 2) two sample DFT's from the shuffled inputs
4. Compute (N / 4) four sample DFT's from the two sample DFT's.
5. Compute (N / 2) eight sample DFT's from the four sample DFT's.
6. Until the all the samples combine into one N-sample DFT

## Output:

Enter the length of sequence : Enter the value of x[0] :

Enter the value of x[1] :

Enter the value of x[2] :

Enter the value of x[3] :

Enter the number of the points in FFT

**Result:** The Fast fourier transform of a given sequence is performed and implemented it on the DSP hardware

# EXPERIMENT 6 IMPLEMENTATION OF N-POINT IFFT OF A SIGNAL

**Objective:** Write a C function to compute the N-Point IFFT of a given sequence and to implement it in DSP hardware

**Outcome**: Students will be able to perform the Inverse Fast fourier transform of a given sequence and to implement it on the DSP hardware

**Requirement**: Turbo C, DSP hardware

**Theory:** FFT is a fast algorithm to perform inverse (or backward) Fourier transform (IDFT), which undoes the process of DFT. The dsp. IFFT System object computes the inverse discrete Fourier transform (IDFT) of the input. The object uses one or more of the following fast Fourier transform (FFT) algorithms depending on the complexity of the input and whether the output is in linear or bit-reversed order. An FFT algorithm can be used to compute the inverse DFT by replacing x[n] by X(k), taking the negative powers of WN, and dividing the output by N.

## Algorithm:

1. Take X(k) as the input sequence and x[n] as the output sequence
2. Compute FFT by replacing the twiddle factors WN by 𝑊𝑁 −1
3. Divide the output sequence by N

## Output:

Enter the length of sequence :

Enter the real and imaginary bits of X(0):

Enter the real and imaginary bits of X(0):

Enter the real and imaginary bits of X(0):

Enter the real and imaginary bits of X(0):

Enter the number of points in the IFFT:

The IFFT of the sequence is:

x[0]=

x[1]=

x[2]=

x[3]=

**Result**: The Inverse Fast fourier transform of a given sequence is performed and implemented it on the DSP hardware

## NFFT NIFFT Program:

**#include** "VSK\_6748.h" **#include**<math.h> **#include**<stdio.h> **#define** SPIFLASH\_SPI (SPI1)

**#define** SPIFLASH\_SPI0 (SPI0)

uint32\_t **config\_pll0**(uint32\_t clkmode, uint32\_t pllm, uint32\_t postdiv, uint32\_t plldiv1, uint32\_t plldiv2, uint32\_t plldiv3, uint32\_t plldiv7);

**#define** PI 3.141592653589 //Pi, 12 decimal places

//#define N 8 //Fourier transform points

//#define M 3 //The number of butterfly operations, N = 2^M

**#define** N 64 //Fourier transform points

**#define** M 6 //The number of butterfly operations, N = 2^M

**typedef double** ElemType; //The data type of the original data sequence can be set here

**typedef struct** //Define complex structure

{

ElemType real,imag;

}complex;

complex data[N]; //Define the storage unit, the original data and negative results are used ElemType result[N]; //Store the modulus of the complex number result afterFFT

//Index

**void ChangeSeat**(complex \*DataInput)

{

**int** nextValue,nextM,i,k,j=0; complex temp; nextValue=N/2;

//Indexing operation, that is, changing the natural order into the inverted order, using the Reid algorithm

nextM=N-1;

**for** (i=0;i<nextM;i++)

{

**if** (i<j) //If i<j, then index

{

temp=DataInput[j]; DataInput[j]=DataInput[i];

DataInput[i]=temp;

}

k=nextValue; //Find the next inverse order of j

**while** (k<=j) //If k<=j, the highest bit of j is 1

{

j=j-k; //Change the highest bit to 0

k=k/2; //k/2, compare the next highest bit, and so on, compare one by one, until a bit is 0

}

j=j+k; //Change 0 to 1

}

}

//Complex number multiplication

complex **XX\_complex**(complex a, complex b)

{

complex temp;

temp.real = a.real \* b.real-a.imag\*b.imag; temp.imag = b.imag\*a.real + a.imag\*b.real; **return**

temp;

}

//FFT

## void FFT(void)

{

**int** L=0,B=0,J=0,K=0;

**int** step=0; ElemType P=0,T=0; complex W,Temp\_XX;

//ElemType TempResult[N]; ChangeSeat(data); **for**(L=1; L<=M; L++)

{

B = 1<<(L-1);//B=2^(L-1)

**for**(J=0; J<=B-1; J++)

{

P = (1<<(M-L))\*J;//P=2^(M-L) \*J

step = 1<<L;//2^L

**for**(K=J; K<=N-1; K=K+step)

{

W.real = **cos**(2\*PI\*P/N); W.imag = -**sin**(2\*PI\*P/N);

Temp\_XX = XX\_complex(data[K+B],W); data[K+B].real = data[K].real - Temp\_XX.real; data[K+B].imag = data[K].imag - Temp\_XX.imag;

data[K].real = data[K].real + Temp\_XX.real; data[K].imag = data[K].imag + Temp\_XX.imag;

}

}

}

}

## void IFFT(void)

{

**int** L=0,B=0,J=0,K=0;

**int** step=0; ElemType P=0,T=0; complex W,Temp\_XX;

//ElemType TempResult[N]; ChangeSeat(data); **for**(L=1; L<=M; L++)

{

B = 1<<(L-1);//B=2^(L-1)

**for**(J=0; J<=B-1; J++)

{

P = (1<<(M-L))\*J;//P=2^(M-L) \*J

step = 1<<L;//2^L

**for**(K=J; K<=N-1; K=K+step)

{

W.real = **cos**(2\*PI\*P/N);

W.imag = **sin**(2\*PI\*P/N);//Inverse operation, here is the opposite sign of FFT

Temp\_XX = XX\_complex(data[K+B],W); data[K+B].real = data[K].real - Temp\_XX.real; data[K+B].imag = data[K].imag - Temp\_XX.imag; data[K].real = data[K].real + Temp\_XX.real; data[K].imag = data[K].imag + Temp\_XX.imag;

}

}

}

}

**float** Xn[N]; **float** fft[N]; **float** ifft[N]; **int main**(**void**)

{

**short** val;

spi\_regs\_t \*spi=SPIFLASH\_SPI0;

spi\_regs\_t \*spi1=SPIFLASH\_SPI; USTIMER\_init(); // timer initialization

Spi\_init(); // spi initialization

// config\_pll0(0,45,1,0,1,11,5);//k600 fft\_ifft\_wave();

}

## void fft\_ifft\_wave()

{

**int** i = 0;

**printf**("input\_sample: ");

**for**(i=0; i<N; i++)//Manufacturing input sequence

{

data[i].real = **sin**(2\*PI\*i/N);

// data[i].real = val;//sin(2\*PI\*i/N);

**printf**("%lf ",data[i].real); Xn[i] = data[i].real;

}

**printf**("\n\n"); FFT();//Perform FFT calculation **printf**("\n\n"); **printf**("FFT: ");

**for**(i=0; i<N; i++)

{ fft[i] = **sqrt**(data[i].real\*data[i].real+data[i].imag\*data[i].imag); **printf**("%lf ",**sqrt**(data[i].real\*data[i].real+data[i].imag\*data[i].imag));

}

# EXPERIMENT 7 FIR LOW PASS FILTER

**Objective:** To plot magnitude and phase response of FIR Low Pass Filter using Hamming Window Method

**Outcome**: Students will be able to implement magnitude and phase response of FIR Low Pass Filter using Hamming Window Method

**Requirement**: Google **Colaboratory**- Online cloud-based Jupyter notebook environment

**Theory:** FIR filters are digital filters with finite impulse response. They are also known as non- recursive digital filters as they do not have the feedback. An FIR filter has two important advantages over an IIR design:

* Firstly, there is no feedback loop in the structure of an FIR filter. Due to not having a feedback loop, an FIR filter is inherently stable. Meanwhile, for an IIR filter, we need to check the stability.
* Secondly, an FIR filter can provide a linear-phase response. As a matter of fact, a linear-phase response is the main advantage of an FIR filter over an IIR design otherwise, for the same filtering specifications; an IIR filter will lead to a lower order.

*FIR FILTER DESIGN*: An FIR filter is designed by finding the coefficients and filter order that meet certain specifications, which can be in the time-domain (e.g. a matched filter) and/or the frequency domain (most common). Matched filters perform a cross-correlation between the input signal and a known pulse-shape. The FIR convolution is a cross-correlation between the input signal and a time-reversed copy of the impulse-response. Therefore, the matched-filter's impulse response is "designed" by sampling the known pulse-shape and using those samples in reverse order as the coefficients of the filter.

When a particular frequency response is desired, several different design methods are common:

1. Window design method
2. Frequency Sampling method
3. Weighted least squares design

*WINDOW DESIGN METHOD:* In the window design method, one first designs an ideal IIR filter and then truncates the infinite impulse response by multiplying it with a finite length window function. The result is a finite impulse response filter whose frequency response is modified from that of the IIR filter.

## Algorithm:

1. Define the filter requirement with Sample Period, Sampling Freq, Total Samples, Signal

Freq, Nyquist Frequency, order

1. Choose the Window Type - hamming
2. Approximate the Window Length
3. Find the Appropriate Ideal Filter
4. Apply timeshift and multiply with window
5. Plot the magnitude response and phase response of the filter.

**Output:** The following waveform is obtained:

**Result:** Implemented magnitude and phase response of FIR Low Pass Filter using Hamming Window Method

## FIR LPF Program:

import numpy as np

import matplotlib.pyplot as plt from scipy import signal N=50

w=np.hamming(N)

i= np.arange(-(N-1)/2,(N-1)/2+1) wc=0.1\*np.pi hd=wc/np.pi\*np.sinc(wc/np.pi\*i) h=hd\*w plt.figure(figsize=(15,5)) plt.subplot(121)

plt.stem(h , linefmt = "Green" , markerfmt = 'D') plt.title('Impulse Response')

w, H=signal.freqz(h,1); plt.subplot(122)

plt.plot(w/np.pi , abs(H) , label = "Magnitude Response" , color = 'Magenta' , linewidth = 1.5 ) plt.title('Magnitude Response');

# EXPERIMENT 8

**IMPLEMENTATION OF OVERLAP ADD ALGORITHM USING DSP HARDWARE**

**Objective:** To compute the DFT of a given sequence using overlap add method and to implement it in DSP hardware

**Outcome**: Students will be able to perform the DFT of a given sequence using overlap add method and to implement it on the DSP hardware.

**Requirement:** Turbo C, DSP hardware

**Theory:** An input sequence x(n) of long duration is to be processed with a system having impulse response of finite duration by convolving long sequences.So the input sequence must be divided into blocks and the successive blocks are processed separately one at a time and the results are combined later to yield the desired output sequence.In overlap add method , if the length of the sequence be L s and length of the impulse response is M, Then the sequence is divided into blocks of data size having length L and M-1 zeros are appended to it to make the data size of L+M-1.

## Procedure:

1. Load the out file into VSK-6748 kit
2. Press the resume button to run the program

**Result:** Implemented the DFT of a given sequence using overlap add method on DSP hardware.

**Overlap add Program: #include** <stdio.h>

**#include** <math.h>

**#define** size(x) **sizeof**(x)/**sizeof**(\*x)

**#define** PI 3.141592653589 //Pi, 12 decimal places

**#define** NS 4 //Fourier transform points

**#define** MS 2 //The number of butterfly operations, N = 2^M

//#define N 64 //Fourier transform points

//#define M 6 //The number of butterfly operations, N = 2^M

**int** xsample[] = {3,-1,0,1,3,2,0,1,2,1}; // input sample Xn

**int** hsample[] = {1, 1, 1}; // input impulse response Hn

//yn = {3,2,2,0,4,6,5,,3,3,4,3,1}; // output sample

**typedef double** ElemType; //The data type of the original data sequence can be set here

**typedef struct** //Define complex structure

{

ElemType real,imag;

}complex;

complex data[NS],xndata[NS],hndata[NS]; //Define the storage unit, the original data and

//negative results are used

ElemType result[NS]; //Store the modulus of the complex number result after FFT

// Allocates a 2D array that can be accessed in the form arr[r][c].

// The caller is responsible for calling free() when done.

**void**\*\* **malloc2d**(size\_t rows, size\_t cols, size\_t element\_size)

{ size\_t header = rows \* **sizeof**(**void**\*); size\_t body = rows \* cols \* element\_size; size\_t needed = header + body;

**void**\*\* mem = malloc(needed);

**if** (!mem) {

**return** NULL;

}

size\_t i;

**for** ( i = 0; i < rows; i++) {

**void**\* col\_mem = mem + header + i\*rows\*cols\*element\_size; mem[i] = col\_mem;

}

**return** mem;

}

**void** \* **my\_malloc**(size\_t s)

{

size\_t \* ret = malloc(**sizeof**(size\_t) + s);

\*ret = s;

**return** &ret[1];

}

**void** \* **my\_realloc**(**void** \*ptr,size\_t s)

{

size\_t \* ret = realloc(ptr,**sizeof**(size\_t) + s);

\*ret = s;

**return** &ret[1];

}

**void my\_free**(**void** \* ptr){ free( (size\_t\*)ptr - 1);}

size\_t **allocated\_size**(**void** \* ptr){ **return** ((size\_t\*)ptr)[-1]/**sizeof**(ptr);} **int stagecnt**(**int** X,**int** L){

// Computes quotient

**int** quo = X / L;

// Computes remainder

**int** rem = X % L; **int** temp; **if**(rem == 0) temp = quo;

## else

temp = quo + 1;

**return** temp;}

// Find maximum between two numbers.

**int max**(**int** num1, **int** num2){

**return** (num1 > num2 ) ? num1 : num2;}

// Find minimum between two numbers.

**int min**(**int** num1, **int** num2){

**return** (num1 > num2 ) ? num2 : num1;}

//Index

**void ChangeSeat**(complex \*DataInput)

{

**int** nextValue,nextM,i,k,j=0; complex temp;

nextValue=NS/2; //Indexing operation, that is, changing the natural order

into the inverted order, **using** the Reid algorithm nextM=NS-1;

**for** (i=0;i<nextM;i++)

{

**if** (i<j) //If i<j, then index

{

temp=DataInput[j]; DataInput[j]=DataInput[i]; DataInput[i]=temp;

}

k=nextValue; //Find the next inverse order of j

**while** (k<=j) //If k<=j, the highest bit of j is 1

{

j=j-k; //Change the highest bit to 0

k=k/2; //k/2, compare the next highest bit, and so on, compare one by one, until a bit is 0

}

j=j+k; //Change 0 to 1

}

}

//Complex number multiplication

complex **XX\_complex**(complex a, complex b)

{

complex temp;

temp.real = a.real \* b.real-a.imag\*b.imag; temp.imag = b.imag\*a.real + a.imag\*b.real; **return**

temp;

}

## void FFT\_Xn(void)

{

**int** L=0,B=0,J=0,K=0;

**int** step=0; ElemType P=0,T=0; complex W,Temp\_XX;

//ElemType TempResult[N]; ChangeSeat(xndata); **for**(L=1; L<=MS; L++)

{

B = 1<<(L-1);//B=2^(L-1)

**for**(J=0; J<=B-1; J++)

{

P = (1<<(MS-L))\*J;//P=2^(M-L) \*J

step = 1<<L;//2^L

**for**(K=J; K<=NS-1; K=K+step)

{

W.real = **cos**(2\*PI\*P/NS); W.imag = -**sin**(2\*PI\*P/NS);

Temp\_XX = XX\_complex(xndata[K+B],W); xndata[K+B].real = xndata[K].real - Temp\_XX.real; xndata[K+B].imag = xndata[K].imag - Temp\_XX.imag; xndata[K].real = xndata[K].real + Temp\_XX.real; xndata[K].imag = xndata[K].imag + Temp\_XX.imag;

}

}

}

}

## void FFT\_Hn(void)

{

**int** L=0,B=0,J=0,K=0;

**int** step=0; ElemType P=0,T=0; complex W,Temp\_XX;

//ElemType TempResult[N]; ChangeSeat(hndata); **for**(L=1; L<=MS; L++)

{

B = 1<<(L-1);//B=2^(L-1)

**for**(J=0; J<=B-1; J++)

{

P = (1<<(MS-L))\*J;//P=2^(M-L) \*J

step = 1<<L;//2^L

**for**(K=J; K<=NS-1; K=K+step)

{

W.real = **cos**(2\*PI\*P/NS); W.imag = -**sin**(2\*PI\*P/NS);

Temp\_XX = XX\_complex(hndata[K+B],W); hndata[K+B].real = hndata[K].real - Temp\_XX.real; hndata[K+B].imag = hndata[K].imag - Temp\_XX.imag; hndata[K].real = hndata[K].real + Temp\_XX.real; hndata[K].imag = hndata[K].imag + Temp\_XX.imag;

}

}

}

}

Void **IFFT**(**void**)

{

**int** L=0,B=0,J=0,K=0;

**int** step=0; ElemType P=0,T=0; complex W,Temp\_XX;

//ElemType TempResult[N]; ChangeSeat(data); **for**(L=1; L<=MS; L++)

{

B = 1<<(L-1);//B=2^(L-1)

**for**(J=0; J<=B-1; J++)

{

P = (1<<(MS-L))\*J;//P=2^(M-L) \*J

step = 1<<L;//2^L

**for**(K=J; K<=NS-1; K=K+step)

{

W.real = **cos**(2\*PI\*P/NS);

W.imag = **sin**(2\*PI\*P/NS);//Inverse operation, here is the opposite sign of FFT

Temp\_XX = XX\_complex(data[K+B],W); data[K+B].real = data[K].real - Temp\_XX.real; data[K+B].imag = data[K].imag - Temp\_XX.imag; data[K].real = data[K].real + Temp\_XX.real; data[K].imag = data[K].imag + Temp\_XX.imag;

}

}

}

}

//OVERLAP ADD

**int main**(**int** argc, **char** \*argv[])

{

**int** i = 0,j = 0, k = 0; complex Temp\_XH;

**int** L = 2; // Block length here change **int** X = size(xsample);//length of x(n) **int** M = size(hsample);//length of h(n) **int** N = L + M -1;

**int** \*arr\_xn = my\_malloc(**sizeof**(**int**) \* (X+M-1));

**for**(i=0;i<X+M-1;i++)// M-1 zero padding after the array

{

**if**(i<X)

arr\_xn[i] = xsample[i];

## else

arr\_xn[i] = 0;

}

**int** \*arr\_hn = my\_malloc(**sizeof**(**int**) \* (M+L-1));

**for**(i=0;i< M+L-1;i++)// L-1 zero padding after the array

{

**if**(i < M)

arr\_hn[i] = hsample[i];

## else

arr\_hn[i] = 0;

}

**int** n = stagecnt(X,L);

**printf**("%d",n);

**int**\*\* x\_n = (**int**\*\*) malloc2d(n, N, **sizeof**(**int**)); **float**\*\* y\_n = (**float**\*\*) malloc2d(n, N,

**sizeof**(**float**)); **for**(i=0;i<n;i++) **for**(j=0;j<N;j++)

{

**if**(j<L){

x\_n[i][j] = arr\_xn[k]; k++;}

## else

x\_n[i][j] = 0;

}

**int** h; **printf**("Hn: ");

**for**(i=0; i<NS; i++)//Manufacturing input sequence

{

// data[i].real = 2048.0 \* sin(2\*PI\*i/N); hndata[i].real = arr\_hn[i];//sin(2\*PI\*i/N);

**printf**("%lf ",hndata[i].real);

}

**printf**("\n\n"); FFT\_Hn();//Perform FFT calculation

// printf("\n\n");

// printf("FFT Hn:");

// for(i=0; i<NS; i++){

// printf("%lf

",sqrt(hndata[i].real\*hndata[i].real+hndata[i].imag\*hndata[i].imag));}

/////////////////////////////////////////////////////////

**for**(h=0;h<n;h++)

// start fft inverse fft function

{

**printf**("Xn: ");

**for**(i=0; i<NS; i++)//Manufacturing input sequence

{

// data[i].real = 2048.0 \* sin(2\*PI\*i/N); xndata[i].real = x\_n[h][i];//sin(2\*PI\*i/N);

**printf**("%lf ",xndata[i].real);

}

**printf**("\n\n"); FFT\_Xn();//Perform FFT calculation **for**(i=0; i<NS; i++){ Temp\_XH = XX\_complex(xndata[i],hndata[i]); data[i].real = Temp\_XH.real; data[i].imag = Temp\_XH.imag;

}

IFFT();//Perform FFT calculation

**printf**("\n\n");

**printf**("IFFT: ");

**for**(i=0; i<NS; i++)

{

y\_n[h][i] =data[i].real/NS; **printf**("y\_n:%f",y\_n[h][i]);} **printf**("\n");

}

free(arr\_xn); free(arr\_hn); free(x\_n);

// Dynamically allocate memory using malloc() **float** \*yn = (**float**\*)malloc((X+L) \*

**sizeof**(**float**)); h=0;**int** n\_d = 0;

**int** zp=L;

**for**(i=0; i<(X+L); i++){

**if**(i<L){ yn[i]=y\_n[h][i];} **else**

{

**if**(h<n-1){

yn[i]=y\_n[h][zp] + y\_n[h+1][zp-L];

}

**else** { yn[i]=y\_n[h][zp];} n\_d++; h=n\_d/L;

zp++;

**if**(zp >= N) zp=L;

}

}

**printf**("OVERLAP ADD FFT\_IFFT:");

**for**(i=0; i<(X+L); i++){**printf**("%f ",yn[i]);} free(y\_n); free(yn);

**return** 0;

}

}

}

# MATLAB PROGRAM

close All clear All clc

N=input('Enter the length of x(n) : '); x=rand(1,N); % Random N Numbers h=input('Enter the values of h(n)='); L=length(h);

N1=length(x); M=length(h); lc=conv(x,h);

x=[x zeros(1,mod(-N1,L))]; N2=length(x);

h=[h zeros(1,L-1)];

H=fft(h,L+M-1); S=N2/L;

index=1:L; X=[zeros(M-1)]; for stage=1:S

xm=[x(index) zeros(1,M-1)]; % Selecting sequence to process X1=fft(xm,L+M-1);

Y=X1.\*H;

Y=ifft(Y);

Z=X((length(X)-M+2):length(X))+Y(1:M-1); %Samples Added in every stage X=[X(1:(stage-1)\*L) Z Y(M:M+L-1)];

index=stage\*L+1:(stage+1)\*L;

end i=1:N1+M-1; X=X(i);

figure() subplot(2,1,1) stem(lc);

title('Convolution Using inbuilt function') xlabel('n');

ylabel('y(n)'); subplot(2,1,2) stem(X);

title('Convolution Using Overlap Add Method') xlabel('n');

ylabel('y(n)');

# EXPERIMENT 9

**IMPLEMENTATION OF OVERLAP SAVE ALGORITHM USING DSP HARDWARE**

**Objective:** To compute the DFT of a given sequence using overlap save method and to implement it in DSP hardware

**Outcome:** Students will be able to perform the DFT of a given sequence using overlap save method and to implement it on the DSP hardware

**Requirement:** Turbo C, DSP hardware

**Theory:** An input sequence x(n) of long duration is to be processed with a system having impulse response of finite duration by convolving long sequences. So the input sequence must be divided into blocks and the successive blocks are processed separately one at a time and the results are combined later to yield the desired output sequence. In overlap save method , assume the length of the input sequence be L s and length of the impulse response is M. In this method,the input sequence is divided into blocks of data of size N=L+M-1. Each block consists of last (M-1) data points of previous block followed by L new data points to form a data sequence of length

N=L+M-1. For first block of data,the first M-1 points are set to zero

## Procedure:

1. Load the out file into VSK-6748 kit
2. Press the resume button to run the program

**Result:** Implemented the DFT of a given sequence using overlap save method in DSP hardware

**Overlap save Program: #include** <stdio.h>

**#include** <math.h>

**#define** size(x) **sizeof**(x)/**sizeof**(\*x)

**#define** PI 3.141592653589 //Pi, 12 decimal places

**#define** NS 4 //Fourier transform points

**#define** MS 2 //The number of butterfly operations, N = 2^M

//#define N 64 //Fourier transform points

//#define M 6 //The number of butterfly operations, N = 2^M

**int** xsample[] = {3,-1,0,1,3,2,0,1,2,1}; // input sample Xn

**int** hsample[] = {1, 1, 1}; // input impulse response Hn

//yn = {3,2,2,0,4,6,5,,3,3,4,3,1}; // output sample

**typedef double** ElemType; //The data type of the original data sequence can be set here

**typedef struct** //Define complex structure

{

ElemType real,imag;

}complex;

complex data[NS],xndata[NS],hndata[NS]; //Define the storage unit, the original data **and**

negative results are used

ElemType result[NS]; //Store the modulus of the complex number result after FFT

// Allocates a 2D array that can be accessed in the form arr[r][c].

// The caller is responsible for calling free() when done.

**void**\*\* **malloc2d**(size\_t rows, size\_t cols, size\_t element\_size) { size\_t header = rows \*

**sizeof**(**void**\*);

size\_t body = rows \* cols \* element\_size; size\_t needed = header + body;

**void**\*\* mem = malloc(needed);

**if** (!mem) {

**return** NULL;

}

size\_t i;

**for** ( i = 0; i < rows; i++) {

**void**\* col\_mem = mem + header + i\*rows\*cols\*element\_size; mem[i] = col\_mem;

}

**return** mem;

}

**void** \* **my\_malloc**(size\_t s)

{

size\_t \* ret = malloc(**sizeof**(size\_t) + s);

\*ret = s;

**return** &ret[1];

}

**void** \* **my\_realloc**(**void** \*ptr,size\_t s)

{

size\_t \* ret = realloc(ptr,**sizeof**(size\_t) + s);

\*ret = s;

**return** &ret[1];

}

**void my\_free**(**void** \* ptr){ free( (size\_t\*)ptr - 1);}

size\_t **allocated\_size**(**void** \* ptr){ **return** ((size\_t\*)ptr)[-1]/**sizeof**(ptr);} **int stagecnt**(**int**

X,**int** L){

// Computes quotient

**int** quo = X / L;

// Computes remainder

**int** rem = X % L; **int** temp; **if**(rem == 0) temp = quo;

## else

temp = quo + 1;

**return** temp;}

// Find maximum between two numbers.

**int max**(**int** num1, **int** num2){

**return** (num1 > num2 ) ? num1 : num2;}

// Find minimum between two numbers.

**int min**(**int** num1, **int** num2){

**return** (num1 > num2 ) ? num2 : num1;}

//Index

**void ChangeSeat**(complex \*DataInput)

{

**int** nextValue,nextM,i,k,j=0; complex temp;

nextValue=NS/2; //Indexing operation, that is, changing the natural order into the inverted order, **using** the Reid algorithm nextM=NS-1;

**for** (i=0;i<nextM;i++)

{

**if** (i<j) //If i<j, then index

{

temp=DataInput[j];

DataInput[j]=DataInput[i]; DataInput[i]=temp;

}

k=nextValue; //Find the next inverse order of j

**while** (k<=j) //If k<=j, the highest bit of j is 1

{

j=j-k; //Change the highest bit to 0

k=k/2; //k/2, compare the next highest bit, and so on, compare one by one, until a bit is 0

}

j=j+k; //Change 0 to 1

}

}

//Complex number multiplication

complex **XX\_complex**(complex a, complex b)

{

complex temp;

temp.real = a.real \* b.real-a.imag\*b.imag; temp.imag = b.imag\*a.real + a.imag\*b.real; **return**

temp;

}

## void FFT\_Xn(void)

{

**int** L=0,B=0,J=0,K=0;

**int** step=0; ElemType P=0,T=0; complex W,Temp\_XX;

//ElemType TempResult[N]; ChangeSeat(xndata); **for**(L=1; L<=MS; L++)

{

B = 1<<(L-1);//B=2^(L-1)

**for**(J=0; J<=B-1; J++)

{

P = (1<<(MS-L))\*J;//P=2^(M-L) \*J

step = 1<<L;//2^L

**for**(K=J; K<=NS-1; K=K+step)

{

W.real = **cos**(2\*PI\*P/NS); W.imag = -**sin**(2\*PI\*P/NS);

Temp\_XX = XX\_complex(xndata[K+B],W); xndata[K+B].real = xndata[K].real - Temp\_XX.real; xndata[K+B].imag = xndata[K].imag - Temp\_XX.imag; xndata[K].real = xndata[K].real + Temp\_XX.real; xndata[K].imag = xndata[K].imag + Temp\_XX.imag;

}

}

}

}

## void FFT\_Hn(void)

{

**int** L=0,B=0,J=0,K=0;

**int** step=0; ElemType P=0,T=0; complex W,Temp\_XX;

//ElemType TempResult[N]; ChangeSeat(hndata); **for**(L=1; L<=MS; L++)

{

B = 1<<(L-1);//B=2^(L-1)

**for**(J=0; J<=B-1; J++)

{

P = (1<<(MS-L))\*J;//P=2^(M-L) \*J

step = 1<<L;//2^L

**for**(K=J; K<=NS-1; K=K+step)

{

W.real = **cos**(2\*PI\*P/NS); W.imag = -**sin**(2\*PI\*P/NS);

Temp\_XX = XX\_complex(hndata[K+B],W); hndata[K+B].real = hndata[K].real - Temp\_XX.real; hndata[K+B].imag = hndata[K].imag - Temp\_XX.imag; hndata[K].real = hndata[K].real + Temp\_XX.real; hndata[K].imag = hndata[K].imag + Temp\_XX.imag;

}

}

}

}

## void IFFT(void)

{

**int** L=0,B=0,J=0,K=0;

**int** step=0; ElemType P=0,T=0; complex W,Temp\_XX;

//ElemType TempResult[N]; ChangeSeat(data); **for**(L=1; L<=MS; L++)

{

B = 1<<(L-1);//B=2^(L-1)

**for**(J=0; J<=B-1; J++)

{

P = (1<<(MS-L))\*J;//P=2^(M-L) \*J

step = 1<<L;//2^L

**for**(K=J; K<=NS-1; K=K+step)

{

W.real = **cos**(2\*PI\*P/NS);

W.imag = **sin**(2\*PI\*P/NS);//Inverse operation, here is the opposite sign of FFT

Temp\_XX = XX\_complex(data[K+B],W); data[K+B].real = data[K].real - Temp\_XX.real; data[K+B].imag = data[K].imag - Temp\_XX.imag; data[K].real = data[K].real + Temp\_XX.real; data[K].imag = data[K].imag + Temp\_XX.imag;

}

}

}

}

//OVERLAP ADD

**int main**(**int** argc, **char** \*argv[])

{

**int** i = 0,j = 0, k = 0; complex Temp\_XH;

**int** L = 2; // Block length here change **int** X = size(xsample);//length of x(n) **int** M = size(hsample);//length of h(n) **int** N = L + M -1;

**int** \*arr\_xn = my\_malloc(**sizeof**(**int**) \* (X+M-1+L));

**for**(i= 0;i<X+M-1+L;i++)// M-1 zero padding after the array

{

**if**(i< L) arr\_xn[i] = 0;

**else if** (i >= L && i < (X+L))

arr\_xn[i] = xsample[i-L];

**else** arr\_xn[i] = 0;

}

**printf**("Input sampleX[n]:");

**for**(i=0;i<X;i++)

{**printf**("%d",xsample[i]);}

**int** \*arr\_hn = my\_malloc(**sizeof**(**int**) \* (M+L-1));

**for**(i=0;i< M+L-1;i++)// L-1 zero padding after the array

{

**if**(i < M){

arr\_hn[i] = hsample[i];}

## else

arr\_hn[i] = 0;

}

**printf**("\n\n");

// for(i=0;i< M+L-1;i++){

// printf("%d",arr\_hn[i]);} **printf**("Input sampleH[n]:"); **for**(i=0;i<M;i++)

{**printf**("%d",hsample[i]);}

**int** n = stagecnt(X,L);

**int**\*\* x\_n = (**int**\*\*) malloc2d((n+1), N, **sizeof**(**int**)); **float**\*\* y\_n = (**float**\*\*) malloc2d((n+2), N, **sizeof**(**float**)); **for**(i=0;i<n+1;i++){

**for**(j=0;j<N;j++)

{ x\_n[i][j] = arr\_xn[k]; k++;} k = k -L;}

**printf**("\n\n");

**int** h;

**printf**("Hn: ");

**for**(i=0; i<NS; i++)//Manufacturing input sequence

{

// data[i].real = 2048.0 \* sin(2\*PI\*i/N); hndata[i].real = arr\_hn[i];//sin(2\*PI\*i/N);

**printf**("%lf ",hndata[i].real);

}

**printf**("\n\n"); FFT\_Hn();//Perform FFT calculation

/////////////////////////////////////////////////////////

**for**(h=0;h<n+1;h++)

// start fft inverse fft function

{

**printf**("\n\n");

**printf**("stage:%d",h);

**printf**("\n\n");

**printf**("Xn:");

**for**(i=0; i<NS; i++)//Manufacturing input sequence

{

// data[i].real = 2048.0 \* sin(2\*PI\*i/N); xndata[i].real = x\_n[h][i];//sin(2\*PI\*i/N);

**printf**("%lf ",xndata[i].real);

}

**printf**("\n\n"); FFT\_Xn();//Perform FFT calculation

// printf("\n\n");

// printf("FFT Xn: ");

// for(i=0; i<NS; i++)

// {printf("%lf ",sqrt(xndata[i].real\*xndata[i].real+xndata[i].imag\*xndata[i].imag));}

**for**(i=0; i<NS; i++){

Temp\_XH = XX\_complex(xndata[i],hndata[i]); data[i].real = Temp\_XH.real; data[i].imag = Temp\_XH.imag;

}

IFFT();//Perform FFT calculation

**printf**("\n\n");

// printf("IFFT: ");

**printf**("y\_n: ");

**for**(i=0; i<NS; i++)//Manufacturing input sequence

{

y\_n[h][i] = data[i].real/NS;

**printf**(":%f",y\_n[h][i]);

}

}

free(arr\_xn); free(arr\_hn); free(x\_n);

**int** xc = allocated\_size(arr\_xn);

**float** \*yn = (**float**\*)malloc((xc) \* **sizeof**(**float**)); h=0;**int** n\_d = 0;

**int** zp=L;

**for**(i=0; i<(xc); i++){

**if**(i<L)

yn[i]=0;

## else

{

yn[i]=y\_n[h][zp];

zp++;

**if**(zp >= N) zp=L; h++;

n\_d++; h=n\_d/L;

}

}

**printf**("\n\n");

**printf**("OVERLAP SAVE FFT\_IFFT:");

**for**(i=0; i<(xc); i++){**printf**("%f ",yn[i]);} free(y\_n); free(yn);

**return** 0;

}

# MATLAB PROGRAM

close All close All clear All clc

N=input('Enter the length of x(n) : '); x=rand(1,N); % Random N Numbers h=input('Enter the values of h(n)='); L=length(h);

N1=length(x); M=length(h); lc=conv(x,h);

x=[x zeros(1,mod(-N1,L)) zeros(1,L)]; N2=length(x);

h=[h zeros(1,L-1)];

H=fft(h,L+M-1); S=N2/L;

index=1:L;

xm=x(index); % For first stage Special Case x1=[zeros(1,M-1) xm]; %zeros appeded at Start point X=[];

for stage=1:S X1=fft(x1,L+M-1); Y=X1.\*H;

Y=ifft(Y); index2=M:M+L-1;

Y=Y(index2); %Discarding Samples X=[X Y];

index3=(((stage)\*L)-M+2):((stage+1)\*L); % Selecting Sequence to process if(index3(L+M-1)<=N2)

x1=x(index3); end

end; i=1:N1+M-1; X=X(i);

figure() subplot(2,1,1) stem(lc);

title('Convolution Using inbuilt function') xlabel('n');

ylabel('y(n)'); subplot(2,1,2) stem(X);

title('Convolution Using Overlap Save Method') xlabel('n');

ylabel('y(n)');