Introduction

Fast Fourier Transform is one of the most commonly used algorithms to calculate the Discrete Fourier Transform of a given signal. Due to its time complexity, specially optimized code for DSP processors have already been implemented as built-in libraries. Code Composer Studio has such a library containing not only FFT routines but also other DSP operations such as convolution, correlation, etc.

In this experiment, you are required to implement Discrete Fourier Transform using your own codes.

Discrete Fourier Transform

The discrete Fourier transform (DFT) is a specific kind of Fourier transform which is used for discrete-time periodic signals and the output of DFT itself is also a discrete signal. DFT transforms a discrete-time signal into another representation that is called the discrete-frequency domain representation, or simply the DFT of the original signal. The DFT operation requires an input signal that is discrete and whose non-zero values have a limited (finite) duration. Such inputs are often created by sampling a continuous signal, like a human speech. And unlike the discrete-time Fourier transform (DTFT), it only evaluates enough frequency components to reconstruct the finite segment that was analyzed. Inverse DFT cannot reproduce the entire time domain, unless the input happens to be periodic (forever). Therefore, it is often said that the DFT is a transform for Fourier analysis of finite-domain discrete-time signals. The sinusoidal basis functions of the decomposition have the same properties.

Since the input signal is a finite sequence of real or complex numbers, the DFT is ideal for processing data stored in computers. In particular, the DFT is widely employed in signal processing and related fields to analyze the frequencies contained in a sampled signal, to solve partial differential equations, and to perform other operations such as convolutions. The DFT can be computed efficiently in practice using a fast Fourier transform (FFT) algorithm.

Since FFT algorithms are commonly employed to compute the DFT, the two terms are often used interchangeably in colloquial settings, although there is a clear distinction: "DFT" refers to a mathematical transformation, regardless of how it is computed, while "FFT" refers to any one of several efficient algorithms for the DFT. This distinction is further blurred, however, by the synonym finite Fourier transform for the DFT, which apparently predates the term "fast Fourier transform" (Cooley et al., 1969) but has the same initialism.

(Reference: wikipedia – DFT)
Definition of DFT

The sequence of N complex numbers \( x_0, \ldots, x_{N-1} \) is transformed into the sequence of N complex numbers \( X_0, \ldots, X_{N-1} \) by the DFT according to the formula:

\[
X_k = \sum_{n=0}^{N-1} x_n e^{-\frac{2\pi i}{N} kn} \quad k = 0, \ldots, N - 1
\]

where \( e^{\frac{2\pi i}{N}} \) is a primitive N'th root of unity.

The transform is sometimes denoted by the symbol \( \mathcal{F} \), as in

\[
X = \mathcal{F} \{ x \} \quad \text{or} \quad \mathcal{F} (x) \quad \text{or} \quad \mathcal{F} x.
\]

The inverse discrete Fourier transform (IDFT) is given by

\[
x_n = \frac{1}{N} \sum_{k=0}^{N-1} X_k e^{\frac{2\pi i}{N} kn} \quad n = 0, \ldots, N - 1.
\]

A simple description of these equations is that the complex numbers \( X_k \) represent the amplitude and phase of the different sinusoidal components of the input "signal" \( x_n \). The DFT computes the \( X_k \) from the \( x_n \), while the IDFT shows how to compute the \( x_n \) as a sum of sinusoidal components \( (1/N) X_k e^{\frac{2\pi i}{N} kn} \) with frequency \( k / N \) cycles per sample. By writing the equations in this form, we are making extensive use of Euler's formula to express sinusoids in terms of complex exponentials, which are much easier to manipulate. In the same way, by writing \( X_k \) in polar form, we immediately obtain the sinusoid amplitude \( A_k \) and phase \( \varphi_k \) from the complex modulus and argument of \( X_k \), respectively:

\[
A_k = |X_k| = \sqrt{\Re(X_k)^2 + \Im(X_k)^2},
\]

\[
\varphi_k = \arg(X_k) = \text{atan}2(\Im(X_k), \Re(X_k)).
\]

Note that the normalization factor multiplying the DFT and IDFT (here 1 and \( 1/N \)) and the signs of the exponents are merely conventions, and differ in some treatments. The only requirements of these conventions are that the DFT and IDFT have opposite-sign exponents and that the product of their normalization factors be \( 1/N \). A normalization of \( 1/\sqrt{N} \) for both the DFT and IDFT makes the transforms unitary, which has some theoretical advantages, but it is often more practical in numerical computation to perform the scaling all at once as above (and a unit scaling can be convenient in other ways).
Preliminary – Understanding and Interpreting the DFT of A Signal

Here is a Matlab code for DFT implementation of an impulse. For the implementation of DFT in Matlab, fft algorithm and the function fft() is used. Execute it and analyze the output signal. What output is expected as the Fourier transform of an impulse. Note that the output of fft function is complex valued. Pay attention to that complex output when you write the c-code for your dft implementation in CCS.

% Here is a simple fft example: fourier transform of an impulse.
signal= zeros(1,256);
signal(128)=32000; % impulse
FSignal = fft(signal,256); % note FSignal is a complex var.
resignal= ifft(FSignal, 256); % % inverse fft % note that resignal is also a complex var.

subplot(2,2,1); plot(signal); title('signal');
subplot(2,2,2); plot(abs(FSignal)); title(abs FSignal');
subplot(2,2,3); plot(real(FSignal)); title('Real part of FSignal');
subplot(2,2,4); plot(imag(FSignal)); title('Imaginal part of FSignal');

% how can we plot resignal?
% type ‘help fft’ for further info!

Shift the position of impulse in the zero-array ‘signal’ and repeat the transformations. What is the difference? Discuss the results of fft and graphics.

Now, plot a Fourier transform of a 1 KHz sine wave (draw your results as graphics). You can easily generate a sine wave of 256 samples in Matlab. (Assume a sampling frequency of 8 KHz). Understanding the frequency phenomena is important for interpreting results of operations.

% Here is a simple fft example: fourier transform of a sine wave.
sine_signal = sin((1:256)*(2*pi*1000/8000));
% generate your 256-samples 1KHz sine wave signal
FSignal = fft(sine_signal,256); % note FSignal is a complex var.
resignal= ifft(FSignal, 256); % % inverse fft % note that resignal is also a complex var.

subplot(2,2,1); plot(sine_signal); title('sine signal');
subplot(2,2,2); plot(abs(FSignal)); title(abs FSignal');
subplot(2,2,3); plot(real(FSignal)); title('Real part of FSignal');
subplot(2,2,4); plot(imag(FSignal)); title('Imaginal part of FSignal');

% Remember that it is very important to read axes correctly.

A discrete signal which is sampled in $f_s = 8000$ Hz, can have harmonics up to $f_s/2 = 4000$ Hz. Hence, in discrete Fourier transform $\pi$ represents 4000 Hz. A 256-point DFT (or FFT) splits frequency axis into 256 segments. This means $[-\pi, \pi]$ is mapped to $[-4000,4000]$ with 256 intervals, and each bin represents $8000/256 = 31.25$ Hz width step of frequency axis. For an N-point FFT; If N is increased, frequency resolution increases but computational cost also increases.
% Remember that it is very important to read axes correctly.
% According to the paragraph above, we will construct a frequency axis for the graph.

F_axis = [(-4000+8000/(256*2)):8000/256:(4000-8000/(256*2))];
% Try to understand how this axis is constructed.
% Now, replot the graphs using this axis values.
subplot(2,2,1); plot(F_axis, abs(FSignal)); title(abs FSignal’);
subplot(2,2,3);
plot(F_axis, real(FSignal)); title(‘Real part of FSignal’);
subplot(2,2,4);
plot(F_axis, imag(FSignal)); title(‘Imaginal part of FSignal’);

Preliminary - Computing DFT manually in Matlab

Now, if you understood how a DFT’ed signal is interpreted, compute your own 256-point DFT using the sum of products formula given in definition of DFT.

\[ X_k = \sum_{n=0}^{N-1} x_n e^{-\frac{2\pi i}{N} k n} \quad k = 0, \ldots, N - 1 \]

In the formula, N is 256, and \( e^{-i \frac{2\pi kn}{N}} = \cos(2\pi kn/N) - i \sin(2\pi kn/N) \).

You should also keep time-domain input signal in complex form even though it has only real part (set imaginal part as zero). Since complex numbers have real and imaginal parts, you will represent each part as separate arrays. Matlab can store complex numbers in one simple array but we will not use that type. To help the conversion-to-c-code operation, we will use separate arrays to keep real and imaginal parts of arrays. You will have four arrays of numbers to keep time-domain and frequency-domain signals. (x_real[256], x_imag[256], FX_real[256], FX_imag[256]). For example, if \( X_k \) is computed, real part of \( X_k \) is kept in FX_real[k], and imaginal part of \( X_k \) is kept in FX_imag[k].

Using the matlab code you wrote, re-compute the DFT of the 1KHz sine-wave signal. Does it match with the result computed by fft version?

LAB WORK - Implementation of DFT on C6713 using Fast Fourier Transform (FFT)

The first C-work will be a non-real-time application that computes the FFT of an impulse. To achieve this goal, disable interrupt initializations and follow the given modifications and write your own c-code.

To implement the FFT of an input array in CSS, define a buffer.

```c
#include <math.h>
#include <stdio.h>

typedef struct {float real, imag;} complex;
/*this structure is defined to make it easy to handle complex numbers*/
#define pi (float)(3.141593)
#define N 256
complex buffer[N]; // the complex buffer to store signal
```
complex W[N]; /* W is the twiddle array for FFT algorithm. 
float Xmag[N]; /* array to store the magnitudes of complex FFT. 

void calc_twiddles(complex *twiddle, short tN); /* calc.twiddle array for FFT-implementation */
void fft(complex *Y, int M, complex *w);
// fft(): Y is complex input for FFT and output is also returned in Y. 
void calc_magn(complex *Y, float *MAG, short tN); /* TO DO: write this function */

void main(void)
{
    init_twiddles(W,N); /* initializing twiddle array */
    // TO DO: initialize buffer array:
    // (reset elements - clear all - and put a real impulse in the middle of array) 
    fft(buffer,N,W);
    calc_magn(buffer, Xmag,N);
    // magnitudes of FFT of buffer is stored in array Xmag
    // After the program executed, check Xmag using graph-display feature of CCS
}

void calc_twiddles(complex *twiddle, short tN)
{
    short n;
    for (n=0 ; n<tN ; n++) /* set up DFT twiddle factors */
    {
        twiddle[n].real = cos(pi*n/tN);
        twiddle[n].imag = -sin(pi*n/tN);
    }
}

void fft(complex *Y, int M, complex *w) /* input sample array, number of points */
{
    complex temp1,temp2; /* temporary storage variables */
    int i,j,k; /* loop counter variables */
    int upper_leg, lower_leg; /* index of upper/lower butterfly leg */
    int leg_diff; /* difference between upper/lower leg */
    int num_stages=0; /* number of FFT stages, or iterations */
    int index, step; /* index and step between twiddle factor */
    i=1; /* log(base 2) of # of points = # of stages */
    do
    {
        num_stages+=1;
        i=i*2;
    } while (i!=M);

    leg_diff=M/2; /* starting difference between upper & lower legs */
    step=2; /* step between values in twiddle.h */
    for (i=0;i<num_stages;i++) /* for M-point FFT */
    {
        index=0;
        for (j=0;j<leg_diff;j++)
        {
            for (upper_leg=j;upper_leg<M;upper_leg+=(2*leg_diff))
            {
                lower_leg=upper_leg+leg_diff;
                temp1.real=(Y[upper_leg]).real + (Y[lower_leg]).real;
                temp1.imag=(Y[upper_leg]).imag + (Y[lower_leg]).imag;
            }
        }
    }
}
LAB WORK 2: Real-time Implementation of FFT on DSK6713

Real-time implementation of FFT needs some improvements to the simple code above. This time, we will combine double buffering scheme with FFT. It is actually some modified version of previous lab part-1. Instead of energy calculation, you will calculate the FFT of the input buffer.

```c
#include <math.h>
#include <stdio.h>
#include "DSK6713_AIC23.h"      // codec support
//#include "lab6.h"
Uint32 fs = DSK6713_AIC23_FREQ_8KHZ;  // set sampling rate
#define DSK6713_AIC23_INPUT_MIC 0x0015
#define DSK6713_AIC23_INPUT_LINE 0x0011
Uint16 inputsource=DSK6713_AIC23_INPUT_MIC; // select input
```
#define SAMPLING_FREQ 8000

typedef struct {float real, imag; } complex;

short in_sample, out_sample, ptr;

#define pi (float)(3.141593)
#define N 256
complex buffer[2][N]; // double complex-buffer
complex W[N]; // twiddles
float Xmag[N]; // magnitudes
float wincoeffs[N]; // window coefficients for suppressing buffer endings

short buf_ID, flag_buffer_full; // flag for task to check buffer switching

void main(void);
interrupt void c_int11();
void task(short buffer_ID);
void fft(complex *Y, int M, complex *w); // Use the function given above.
void calc_twiddles(complex *twiddle, short tN); // Use the function given above.
void calc_magn(complex *Y, float *MAG,short tN); // Use the function you wrote.
void window_cplx(complex *inframe, float *win, complex *outframe, short arrlen);
void init_hamming(float *win, short arrlen);

void main(void)
{
    buf_ID = 0;  flag_buffer_full = 0; //ptr = 0;

    calc_twiddles(W,N);
    init_hamming(wincoeffs,N);

    comm_intr();

    while(1)
    {
        if(flag_buffer_full == 1) // Wait until next
        {
            flag_buffer_full = 0;

            if(DSK6713_DIP_get(0)==0) // switch-0 pressed
            {
                if(DSK6713_DIP_get(1)==0) // switch-1 pressed
                window_cplx(buffer[1 - buf_ID],wincoeffs,buffer[1 - buf_ID],N);

                fft(buffer[1 - buf_ID],N,W);
                calc_magn(buffer[1-buf_ID], Xmag,N);
            }
        }
    }
}

interrupt void c_int11()
{
    in_sample = input_sample();

    buffer[buf_ID][ptr].real = (float)in_sample;
    buffer[buf_ID][ptr].imag = 0;
out_sample = (short)buffer[buf_ID][ptr].real;
output_sample(out_sample);

if (++ptr > N - 1)
{
    ptr = 0; // reset index
    flag_buffer_full = 1;
    buf_ID = 1 - buf_ID;
}

return;
}

void window_cplx(complex *inframe, float *win, complex *outframe, short arrlen)
{
    short k;
    for (k=0; k<arrlen; k++)
        outframe[k].real = inframe[k].real * win[k];
}

void init_hamming(float *win, short arrlen)
{
    short k;
    for (k=0; k<arrlen; k++)
        win[k] = 1 - 0.85185 * cos( 2*pi*(float)k / (float)arrlen);
}

Give different sound inputs such as sine waves generated in matlab and observe the magnitudes for different schemes with or without windowing (use switch-1).

After LAB

Discuss results and your observations in your report.