Lab 7: FFT and Spectrum Analysis

EE 352 DSP Laboratory (Spring 2019)

Lab Goals

To understand implementation of Fast Fourier Transform (FFT) on DSP. The following tasks will be carried out here,

- Compute FFT in software for a fixed length sequence.
- Compute FFT using the efficient routines provided by FFT hardware accelerator co-processor **HWAFFT**.
- Compare the performances of software and hardware implementations with the help of profiling.

1 Introduction

The concept of Discrete Fourier Transform (DFT) was introduced to allow us to work with the frequency domain representation of discrete-time signals while staying in discrete (frequency) domain. Equation (1) gives the expression for the DFT calculation of a sequence $x[n]$.

$$X[k] = \sum_{n=0}^{N-1} x[n] W_N^{kn}, \qquad k = 0, 1, \ldots, N-1 \tag{1}$$

where, $x[n]$ is the input sequence of length N, $X[k]$ is the N-point DFT of x[n] and $W_N = e^{-j\frac{2\pi}{N}}$.

In general, to calculate the DFT by this algorithm, the computations (in terms of complex multiplications and additions) required vary with N as $O(N^2)$. So, as N grow higher, the number of computations grows at a higher rate and FFT calculation operation begins to act as a bottleneck for the system. This extremely high number of computations required in the naive implementation of DFT was a major obstacle in its acceptance.

2 Fast Fourier Transform (FFT)

An efficient algorithm to calculate the DFT was proposed by J. W. Cooley and J. W. Tuckey in the year 1965. At first, they themselves did not realize the importance of the algorithm they proposed. But, the algorithm was widely accepted and it was termed as the **Fast Fourier transform**[1]. The algorithm uses the *divide and conquer* technique to calculate DFT of sequences of very large length. It splits a higher order DFT calculation into several lower order ones and then these lower order DFTs into further lower order ones until this cannot be continued. Now, it can be proved that the number of computations required to calculate a higher order DFT directly is very high as compared to splitting the DFT into lower order DFTs, calculating them and combining the results in appropriate manner to get the initial higher order DFT. This brings the order of computations required to calculate an N point DFT from $O(N^2)$ to $O(Nlog_2(N))$ which is a substantial improvement. You are going to study the implementation of FFT on the DSP in this lab session.

A radix-2 decimation-in-time (DIT) FFT is the simplest form of the CooleyTukey algorithm. Radix-2 DIT divides a DFT of size N into two interleaved DFTs (hence the name 'radix-2') of size N/2 with each recursive stage. The full radix-2 Decimation in time FFT decomposition is illustrated in figure 1 using the simplified butterfly structure.

[1]Source: http://www.versci.com/fft/index.html

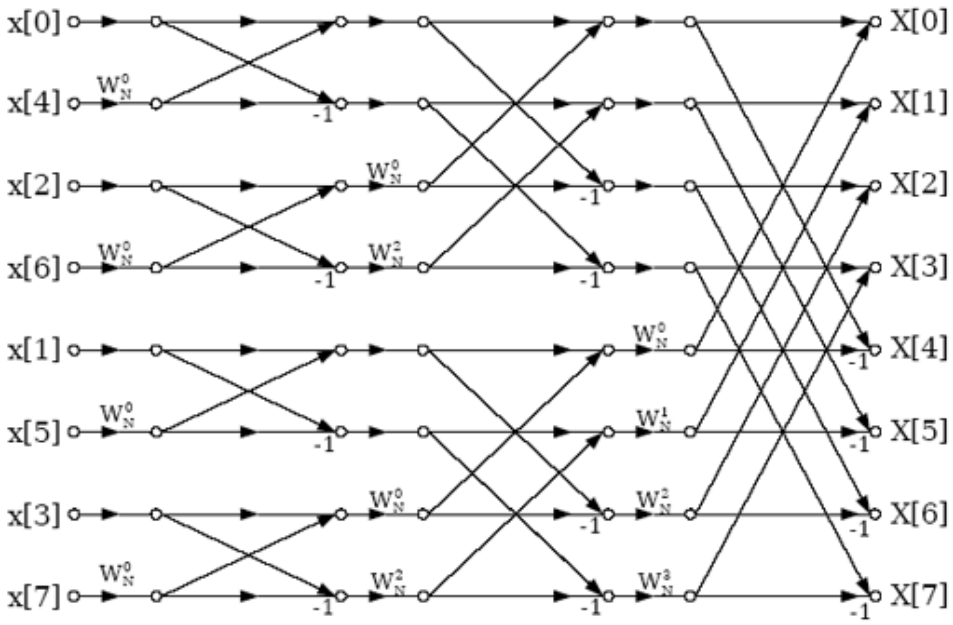

Figure 1: FFT algorithm for a length-8 signal

2.1 Learning checkpoints

All the following tasks need to be shown to your TA to get full credit for this lab session.

1. Writing a C routine to calculate 8-point FFT of a sequence
   (a) You will be using the `main_sw_fft.c` file for this checkpoint.
   (b) The array `x[8]` is a discrete time signal.
   (c) Write a routine to calculate the 8-point FFT of `x` at the indicated place inside the function `sw_fft()`.
   (d) Calculate the FFT of the same array in MATLAB/OCTAVE with the help of appropriate function (Use the command `lookfor` to find the function which calculates FFT.)
   (e) Compare your output with that of the function from MATLAB.

3 Efficient implementation of FFT in hardware

To accelerate the complex process of FFT calculation, it is generally implemented at the processor architecture level. To make the implementation slightly modular, the TMS320C5515 DSP is accompanied by the **HWAFFT** co-processor which is dedicated solely to perform FFT operations. To understand how to utilize it through software, import and open the project `lab7_fft_hw` (provided along with this handout) in CCS. Go through the code written inside the function `do_fft()` in `main.c` file. The following points summarize the working of `do_fft()` function:

- It accepts the data buffer and also the output buffers (separate buffers to return real and imaginary parts of the output) along with the scaling option.
- It first allocates buffers and calls the function `hwafft_br`. Since, the co-processor **HWAFFT** calculates the FFT in *decimation in time (DIT)* format, it expects the inputs already decimated in time. The algorithm to achieve DIT is known as *bit-reversal*. Observe how the indexing demanded by the DIT algorithm can be obtained from normal indexing by just reversing the binary representations of the indices from Table 1. This is achieved by the `hwafft_br()` function.
- Then, the `hwafft_Npts()` calculates the FFT. The `do_fft()` function hides all the book-keeping required to get the `hwafft_Npts` function working. It acts as the wrapper for the same.

- The built-in library (`hwafft.h`) provides functions to compute the FFT and inverse FFT of the given data.
- For more information on this, refer to the sprabb6b.pdf file available in `Ti` website.

Table 1: Relation between normal and bit-reversed indexing

Sr. No.	Normal indexing	Indexing that DIT algorithm demands (bit reversed form)
1	0 (000_b)	0 (000_b)
2	1 (001_b)	4 (100_b)
3	2 (010_b)	2 (010_b)
4	3 (011_b)	6 (110_b)
5	4 (100_b)	1 (001_b)
6	5 (101_b)	5 (101_b)
7	6 (110_b)	3 (011_b)
8	7 (111_b)	7 (111_b)

The definitions of the `do_fft()` and `do_ifft()` (which calculates the inverse FFT) functions are as follow.

- `void do_fft(Int16 *real_data, Int16 *fft_real, Int16 *fft_imag, Uint16 scale)`

  This function takes in real data and stores real part of FFT in `fft_real` and imaginary part of FFT in `fft_imag`.

- `void do_ifft(Int16 *fft_real, Int16 *fft_imag, Int16 *ifft_data, Uint16 scale)`

  This function takes in the complex FFT in `fft_real` and `fft_imag`, calculates its inverse FFT and stores the output real data in `ifft_data`.

- Scale flag : The SCALE version is implemented using only radix-2 stages. This routine prevents overflow by scaling by $\frac{1}{2}$ before each FFT stage.

3.1 Learning checkpoints

All the following tasks need to be shown to your TA to get full credit for this lab session.

1. Calculating the FFT using hardware accelerator and plotting its magnitude.
   - You will be using the `lab6_fft_hw` **project** uploaded on the course-webpage.
   - You will have to calculate the magnitude spectrum of the input signal and display it using the graph window.
   - So, you will have to insert code for magnitude calculation in `/*do frequency domain processing here*/` area in `main( )` function inside the `main.c` file.
   - For magnitude calculation of FFT, you need to square real and imaginary parts of FFT and add. You can use the `_lsmpy` function to multiply two numbers `source1` and `source2` using the syntax, `result = _lsmpy(source1,source2)`, where `source1` and `source2` are of the type `short int`. `result` is of the type `long`.
2. Comparing the performances of hardware and software implementations using profiling.
   - You will be using `lab6_fft_comp` **project** for this checkpoint.
   - Compare the hardware and the software implementations in terms of speed of execution using profiling.
   - Inside `main.c` file, copy your software implementation of FFT in function `sw_fft()`.
   - Call both the functions `sw_fft()` and `do_fft()`, one at a time, 100 times and note the average exclusive counts value.
   - Which is faster? Why?