REAL-TIME DSP LABORATORY6: The Fast Fourier Transform (FFT) and Block Convolution FIR filtering on the C6713 DSK

Contents

1 Introduction 1
2 The Discrete Fourier Transform (DFT) 1
3 A Fast Fourier Transform (FFT) Algorithm 2
3.1 An 8-point Decimation-in-Time FFT algorithm . . . 3
4 Implementing An FFT Algorithm on the C6713 DSK 7
4.1 The Bit-Reversing Algorithm* . . . 8
4.2 Implementing the Factored Butterfly* . . . 9
5 FIR Filtering using an FFT Algorithm 15
5.1 Real-Time Block Processing on the C6713 DSK* . . . 15
5.2 Linear Convolution using the DFT . . . 17
5.3 Overlap and Add Block Convolution FIR Filtering* . . . 17
5.4 Implementing the Overlap and Add Algorithm using an FFT* . . . 18
6 End Notes 20
6.1 Advanced Lab Topics . . . 20

1 Introduction

The discrete Fourier Transform (DFT) is the only Fourier transform that can be computed exactly on a digital computer, if by exact we ignore finite precession effects. The DFT, when implemented directly, requires many complex multiplications and additions. When an N-point DFT is highly composite (i.e. N is a power of 2), the number of complex multiplications and additions may be reduced significantly by efficiently factoring the DFT. These factored DFT algorithms are collectively known as fast Fourier Transform (FFT) algorithms. The FFT algorithm was introduced to signal processing in 1965 in a paper published by J. W. Cooley and J. W. Tukey [3]. However, the earliest known discovery of this factorization is credited to C.F. Gauss in 1805 [5], and in 1965 there actually existed some efficient codes that were nearly equivalent to the FFT.

In this lab, you will study

- the DFT,
- the in-place decimation-in-time FFT on the DSK using a C language program, and
- FIR filter implementation using the FFT for fast convolution.

2 The Discrete Fourier Transform (DFT)

The discrete Fourier transform, or DFT, is a one-to-one mapping between an N-point complex discrete-time sequence $\{x[n], n = 0, 1, \ldots N-1\}$ and an N-point complex Fourier series sequence

$\{X[m], m = 0, 1, \dots N-1\}$. The resulting Fourier transform pair is denoted $\{x[n], nt_o\}_N \longleftrightarrow \{X[m], \frac{m\omega_o}{N}\}_N$, where

$$\begin{aligned} X[m] &= \sum_{n=0}^{N-1} x[n] W_N^{-mn} \qquad \text{(analysis)} & (1) \\ x[n] &= \frac{1}{N} \sum_{m=0}^{N-1} X[m] W_N^{mn} \qquad \text{(synthesis)}, & (2) \end{aligned}$$

where $W_N = e^{j\frac{2\pi}{N}}$ and W_N^{mn} are the N roots of unity on the unit circle in the complex plane. Here, the signal samples are spaced every nt_o seconds in times, and the FS coefficients, $X[m]$, are spaced every $\frac{m\omega_o}{N}$ radians per second in frequency. When no confusion will result, we will use the simpler notation $\{x[n]\}_N \longleftrightarrow \{X[m]\}$.

From Fourier analysis, we know that if a signal is discrete in one domain, then its Fourier transform will be periodic. Since the DFT is discrete in both frequency and time, it is periodic in both time and frequency. Consequently, both $x[n]$ and $X[m]$ are N-periodic (i.e. $x[n] = x[n+rN]$ and $X[m] = X[m+rN]$ for all integers r.)

3 A Fast Fourier Transform (FFT) Algorithm

The direct computation of an N-point DFT requires N^2 complex multiplications and $N^2 - N$ complex additions [6]. In terms of real computing, this translates to $4N^2$ real multiplications and $2N^2 + 2(N^2 - N) = 4N^2 - 2N$ real additions[1] [6]. These arithmetic operations may be reduced significantly by using the following two properties about the roots of unity W_N^k (where k is an integer) [7]:

$$\begin{aligned} \text{Symmetry property:} \quad W_N^{k+N/2} &= -W_N^k & (3) \\ \text{Periodicity property:} \quad W_N^{k+N} &= W_N^k. & (4) \end{aligned}$$

When N is a power of two (i.e. $N = 2^\nu$, where $\nu = \log_2 N$ is an integer), a radix-2 FFT algorithm may be used to efficiently compute a DFT. There are two ways of implementing a radix-2 FFT, namely decimation-in-time and decimation-in-frequency. Since these two algorithms are transposes of each other, only the decimation-in-time algorithm will be derived.

To begin this factorization, the DFT sum in eqn (1) is split into a sum of two $N/2$ length sums, one on the even indices of $x[n]$ and one on the odd indices. This process is illustrated below [6, pg. 635–636]:

[1] The complex multiplication of two numbers $z_1 = x_1 + jy_1$ and $z_2 = x_2 + jy_2$ is $z_1 z_2 = (x_1 x_2 - y_1 y_2) + j(x_1 y_2 + x_2 y_1)$, which requires four real multiplications and two real additions. The addition of the two complex numbers z_1 and z_2 is $z_1 + z_2 = (x_1 + x_2) + j(y_1 + y_2)$, which requires two real additions. Note that complex arithmetic requires "two channel" variables that hold the real and imaginary parts of the complex number.

$$\begin{aligned}
X[m] &= \sum_{n=0}^{\frac{N}{2}-1} x[2n]W_N^{-m(2n)} + \sum_{n=0}^{\frac{N}{2}-1} x[2n+1]W_N^{-m(2n+1)} \\
&= \sum_{n=0}^{\frac{N}{2}-1} x[2n](W_N^2)^{-mn} + W_N^{-m}\sum_{n=0}^{\frac{N}{2}-1} x[2n+1](W_N^2)^{-mn} \quad (5)\\
&= \sum_{n=0}^{\frac{N}{2}-1} x[2n]W_{N/2}^{-mn} + W_N^{-m}\sum_{n=0}^{\frac{N}{2}-1} x[2n+1]W_{N/2}^{-mn} \quad (6)\\
&= G[m] + W_N^{-m}H[m], m = 0, 1, \ldots, N-1. \quad (7)
\end{aligned}$$

Equation (6) uses the fact that $W_N^2 = e^{(j\frac{2\pi}{N})2} = e^{(j\frac{2\pi}{N/2})} = W_{N/2}$ to show how an N-point DFT may be split into the sum of two $N/2$-point DFTs, namely $G[m]$ and $H[m]$ in eqn (7). Now, these $N/2$ DFTs are periodic in m with period $N/2$ (i.e. $G[m] = G[m + r\frac{N}{2}]$ and $H[m] = H[m + r\frac{N}{2}]$ for all integers r). This process of grouping the even and odd indexed terms of $x[n]$ into two $N/2$ point DFTs is known as time decimation. As a result of this factorization, the number of complex multiplications has been reduced from N^2 to $(\frac{N}{2})^2 + (\frac{N}{2})^2 + N = \frac{N^2}{2} + N$, where the extra N multiplications are for multiplying the coefficients $H[m]$ by $W_N^{-m}, m = 0, 1, \ldots, N-1$. This factorization has reduced the number of complex multiplications roughly in half. This process may be repeated $\log_2 N$ times until there are $\frac{N}{2}$ 2-point FFT factored butterflies per stage and $\log_2 N$ stages. The final factorization uses the symmetry property to reduce the number of multiplications required in half. The discussion in the following paragraph clarifies this point.

3.1 An 8-point Decimation-in-Time FFT algorithm

To begin factoring an 8-point DFT sum, we group the even and odd terms of the sequence $\{x[n]\}_8$ and perform two 4-point DFTs as follows [6]:

$$\begin{aligned}
X[m] &= \sum_{n=0}^{8} x[n]W_8^{-mn} \quad (8)\\
&= \sum_{n=0}^{3} x[2n]W_4^{-mn} + W_8^{-m}\sum_{n=0}^{3} x[2n+1]W_4^{-mn} \quad (9)\\
&= G[m] + W_8^{-m}H[m], k = 0, 1, \ldots, 7. \quad (10)
\end{aligned}$$

Here, $G[m] = \sum_{n=0}^{3} x[2n]W_4^{-mn}$ is the result of a 4-point DFT on the even indices of $\{x[n]\}_8$, and $H[m] = \sum_{n=0}^{3} x[2n+1]W_4^{-mn}$ is the result of a 4-point DFT on the odd indices of $\{x[n]\}_8$. This is called decimation-in-time. We may draw the signal flow graph for this once-factored DFT as shown in Figure 1.

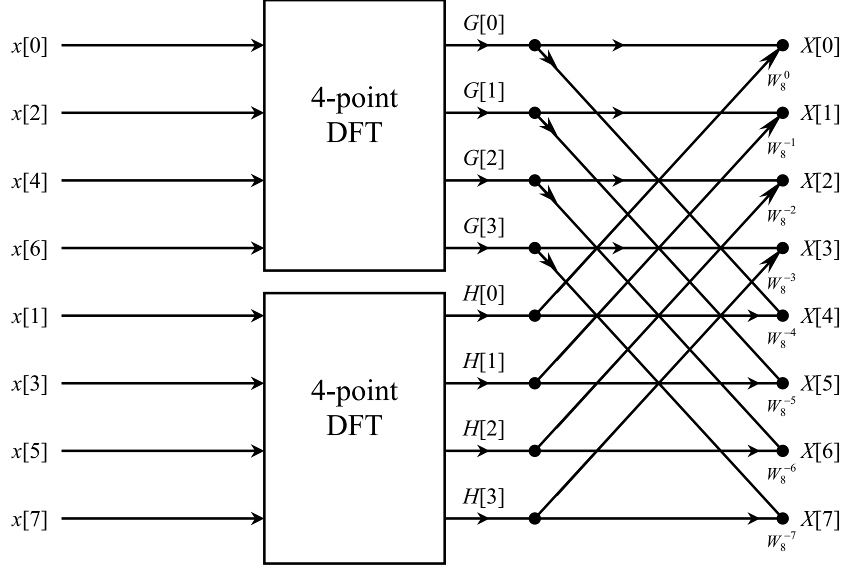

Figure 1: Once factored 8-point DFT. Adapted from [6].

This process may be repeated one more time to yield

$$\begin{aligned} X[m] &= \left(\sum_{n=0}^{1} x[4n]W_2^{-mn} + W_4^{-m}\sum_{n=0}^{1} x[4n+2]W_2^{-mn}\right) \\ &+ W_8^{-m}\left(\sum_{n=0}^{1} x[4n+1]W_2^{-mn} + W_4^{-m}\sum_{n=0}^{1} x[4n+3]W_2^{-mn}\right), \end{aligned} \tag{11}$$

where the four summations are each 2-point sub-DFTs on indices separated by four. Each sum inside the parenthesis is a 4-point sub-DFTs that amounts to a recombination of two 2-point sub-DFTs (one on the even indices and one on the odd indices). The sum of the two 4-point sub-DFTs is an 8-point sub-DFT that amounts to a recombination of the two 4-point sub-DFTs. Using the facts that $W_2^k = W_8^{4k}$ and $W_4^k = W_8^{2k}$ along with the fact that W_8^{4k} is periodic in k with period 2 and W_8^{2k} is periodic in k with period 4 yields the final radix-2 FFT factorization of an 8-point DFT[2]. This is shown in Figure 2.

!

[2]To further clarify this, note that in eqn (11) when $n = 0$ in each of the 2-point sums (i.e. sums involving the time samples $x[0], x[1], x[2]$, and $x[3]$), $W_2^{-mn} = W_2^0 = 1$, so these time samples are not scaled in the first stage. When $n = 1$ in each of the 2-point sums (i.e. sums involving the time samples $x[4], x[5], x[6]$, and $x[7]$), $W_2^{-mn} = W_2^{-m}$, which is equal to W_2^0 when m is even and W_2^{-1} when m is odd. Bearing in mind that $W_2^0 = W_8^0$ and $W_2^{-1} = W_8^{-4}$, use these facts to justify the first stage of Figure 2.

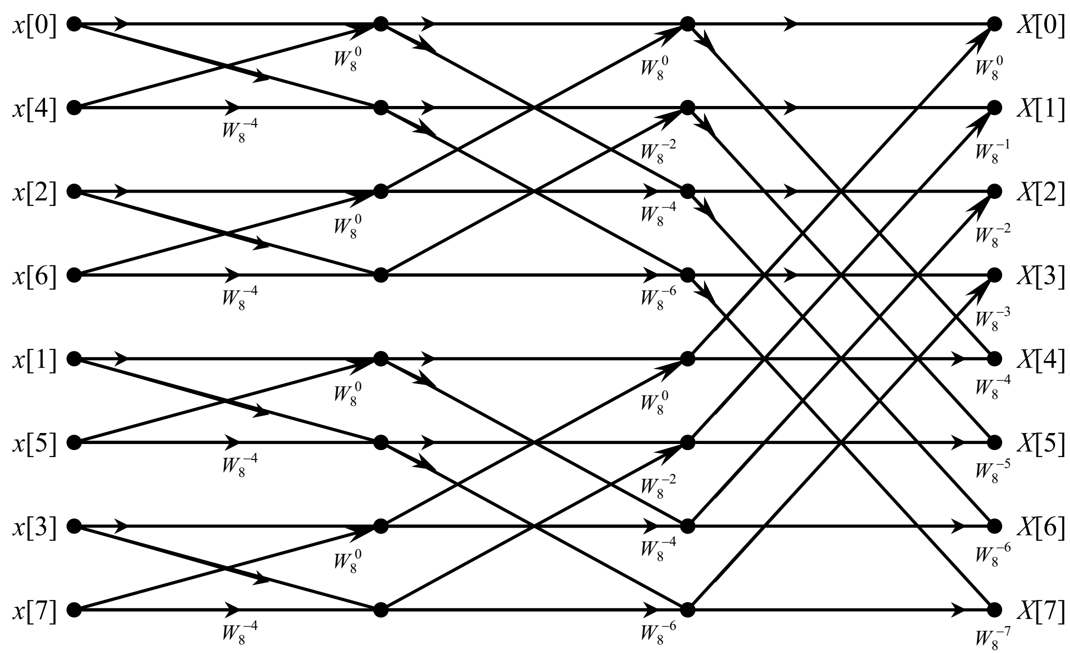

Figure 2: Final factorization of an 8-point DFT. Adapted from [6].

By inspecting Figure 2, there are $8 \log_2 8 = 24$ complex multiplications (W_8^{-m} factors). The direct calculation of an 8-point DFT would require $8^2 = 64$ complex multiplications. This FFT factorization has reduced the number of complex multiplications by 40, which means that this algorithm will run at rate $64/24 = 2.67$ times faster than a direct 8-point DFT computation.

When we implement the FFT algorithm shown in Figure 2, we will over-write the set of N registers from the previous stage. This is known as an *in-place* algorithm. Implementing an in-place algorithm will use the minimum number of memory locations, but it will not save the N input samples. Before we implement an in-place algorithm, we utilize a symmetry condition, which will further cut the number of complex multiplications in half. Notice that at each stage, a set of two nodes from one stage is used to compute the same two nodes in the next stage. Each of these mappings is of the form shown in Figure 3.

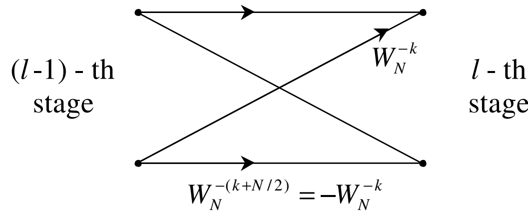

Figure 3: The basic butterfly structure in a radix-2 decimation-in-time FFT algorithm. Adapted from [6].

The structure in Figure 3 is referred to as a butterfly. For a radix-2 decimation-in-time FFT algorithm, the butterfly structures have the same pattern. In particular, the value at the top node in the l-th stage is the value at the top node from the $(l-1)$-th stage plus the value of the bottom node of the $(l-1)$-th stage scaled by the complex value W_N^{-k}. The value at the bottom node in the l-th stage is the value at the top node from the $(l-1)$-th stage plus the bottom node scaled by $W_N^{-(k+N/2)}$. From the symmetry condition in eqn (3), we know that $W_N^{-(k+N/2)} = -W_N^{-k}$. Now, the value at the bottom node in the l-th stage may also be calculated by taking the value at the top node from the $(l-1)$-th stage and subtracting the value of the bottom node of the $(l-1)$-th stage scaled by the complex value W_N^{-k}. This observation leads to the in-place butterfly structure shown in Figure 4.

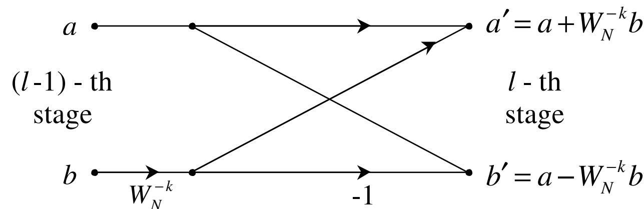

Figure 4: The in-place butterfly structure in a radix-2 decimation-in-time FFT algorithm. Adapted from [6].

When this butterfly structure is implemented in an in-place algorithm, the value in the bottom node of the $(l-1)$-th stage is first copied to a temporary variable which is scaled by a twiddle factor. This happens before either node has been over-written. Then, the bottom node in

the l-th stage of the butterfly is computed and the value over-writes the bottom node. After the bottom node in the l-th stage is calculated in-place, the value in the temporary variable is used to calculate the the value of the top node in the l-th stage, which over-writes the top node. This extra memory requirement is unavoidable in an in-place implementation. However, in assembly code, this temporary variable would be stored in core register and not saved to a memory location. The final in-place decimation-in-time signal flow graph for an 8-point FFT algorithm is shown in Figure 5. Notice that only 12 complex multiplications are required in this 8-point FFT algorithm. Also, only the first $N/2$ roots of unity know are required for this implementation. These roots of unity W_N^k for $k = 0, 1, \ldots, (N/2) - 1$ are known as the *twiddle factors*. In this FFT algorithm, the complex conjugates of the twiddle factors are needed. A point to be clarified later. In general, an N-point in-place decimation-in-time FFT algorithm will require $\frac{N}{2} \log_2(N)$ complex multiplications and $N \log_2(N)$ complex additions.

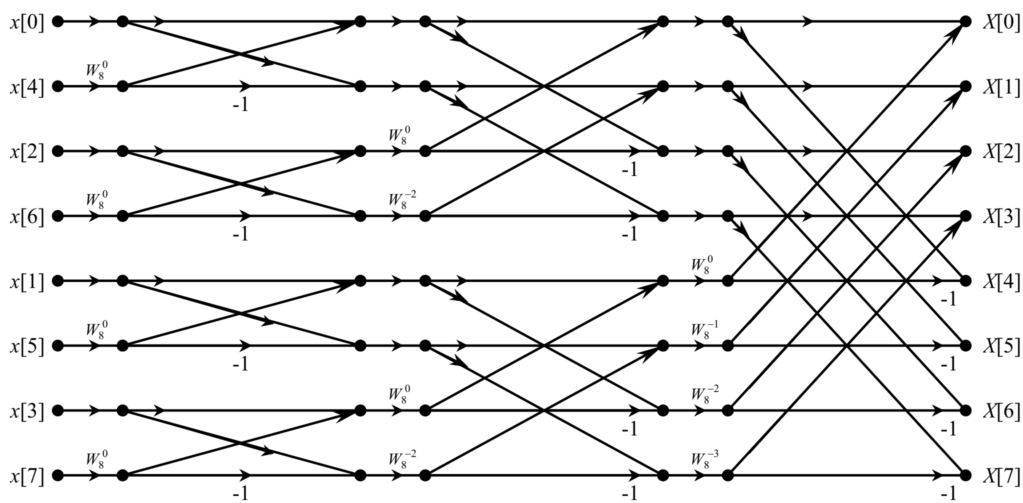

Figure 5: The in-place decimation-in-time signal flow graph for an 8-point FFT algorithm. Adapted from [6].

The trick in the decimation-in-time algorithm is to organize the N input samples in *bit-reversed* order. By this, we mean that we represent the original indices as $(\log_2 N)$-bit binary numbers in chronological order, and then reverse the bits to create the bit-reversed order. In the case $N = 8$, the 3-bit bit-reversed order observed in Figure 5 is illustrated in Table 1.

This re-arrangement can be seen in the arrangement of the inputs in the FFT flow graph in Figure 2.

4 Implementing An FFT Algorithm on the C6713 DSK

In this lab, we will create C code to implement an N-point FFT algorithm, where N is a power of 2. To begin, let's create the C code function `FFT_func(COMPLEX *X, COMPLEX *W)` that will take the N input samples, namely $\{x[n]\}_N$, stored in the complex array `X[]`, and use the $N/2$

n (dec)	n (bin)	bit-reversed (bin)	bit-reversed (dec)
0	000	000	0
1	001	100	4
2	010	010	2
3	011	110	6
4	100	001	1
5	101	101	5
6	110	011	3
7	111	111	7

Table 1: Bit-reversing for an 8-point FFT algorithm.

twiddle factors in `W[]` to compute the N Fourier series coefficients, namely $X[k]$, in-place. This algorithm will over-write the same N registers at each stage of the algorithm. This process will be broken up into two sections:

- Arrange the N inputs in bit-reverse order.
- Compute the $N/2$ butterflies at each of the $\log_2 N$ stages.

4.1 The Bit-Reversing Algorithm*

By comparing the binary values in Table 1, one can see that the bit-reversed order may be calculated by adding a binary one to the most significant bit and having the carryover bits propagate to the right. To begin this process, start with binary zero and add a binary one to the most significant bit (i.e. add $N/2$ in binary to zero). To calculate the next bit reversed index, test the most significant bit to see if it is a binary zero or binary one. If the bit is binary zero, add a binary one to it and move on to the next index. If this bit is a binary one, set it to binary zero and try to add a binary one to the next least significant bit (carryover to the right). If this bit is a binary zero, add a binary one to it and move to the next index; otherwise, clear it and move to the next bit and repeat the process. This continues until a binary zero is found or the last bit-reversed index has been calculated. Due to the way this is implemented, the last index will be calculated before an overflow to the right occurs.

To visualize this, we will work through a bit reversed sequence for $M = 3$ ($N = 8$ as in Table 1). First, we observe that `000`(bin) maps to `000`(bin) and that `111`(bin) maps to `111`(bin), so these indices are not calculated. The first bit-reversed index is $N/2$, which in our case is equal to 4. For intuition into the bit-reversing algorithm, we will represent the decimal number 4 as the binary number `100`(bin).

To calculate the next bit-reversed index, we try to add `100`(bin) to our previous bit-reversed index, `100`(bin). Since the most significant bit in the previous bit-reversed index is a binary one, we clear it by subtracting `100`(bin) from the number to get `100`(bin) - `100`(bin) = `000`(bin) to clear the most significant bit. Then, we try to add `010`(bin) to `000`(bin), which is the result of the previous subtraction (clearing the most significant bit). Since the next least significant bit is a binary zero, our new bit-reversed index becomes `000`(bin) + `010`(bin) = `010`(bin).

To calculate the next bit-reversed index, we again try to add `100`(bin) to the previous bit-reversed index, `010`(bin). Since the most significant bit in `010`(bin) is a binary zero, our new bit-reversed index becomes `010`(bin) + `100`(bin) = `110`(bin).

To calculate the next bit-reversed index, we again try to add `100`(bin) to our previous bit-reversed index, `110`(bin). Since the most significant bit is a binary one, we clear it by subtracting `100`(bin) to get `110`(bin) - `100`(bin) = `010`(bin). Then, we try to add `010`(bin) to `010`(bin). Since the next least significant bit in `010`(bin) is a binary one, we clear it by subtracting `010`(bin) to get `010`(bin) - `010`(bin) = `000`(bin). Then, we try to add `001`(bin) to `000`(bin). Since the next least significant is a binary zero, our new bit-reversed index becomes `000`(bin) + `001`(bin) = `001`(bin).

This process continues until all N bit-reversed indices have been calculated. In practice, we take advantage of the fact that the first and last indices map to themselves, so we only calculate the $N - 2$ indices in between.

Examine the code for the bit-reversing algorithm and associated data swaps listed in Figure 6. Note that this is not a complete program but just a fragment of code. Observe that the bit-reversal mapping is a one-to-one mapping, so we can swap the value at the current index with the value at the bit-reversed index. This can be done in-place using a temporary storage variable, which is illustrated in lines 34 and 35 in Figure 6. Also, the swapping of values between the original and bit reversed indices only needs to occur once. To account for this, we test the current index with the bit-reversed index and only swap if the current index is less than the bit-reversed index. Then, when the current index is greater than the bit-reversed index, we know that the values have already been swapped. This is coded in line 32 of Figure 6, where `i` is the current index and `j` is the bit-reversed index of `i`.

Assignment

1. Work through the description of the 8-point bit-reversing algorithm given on the previous page using decimal numbers instead of binary numbers. Then, briefly explain how the C code in lines 17-27 of Figure 6 implement the bit-reversing algorithm. NB: If `k=01100`$_{\text{bin}}$ (which has decimal equivalent $k = 12_{\text{dec}}$), then the C command `k>>1` returns the binary number `00110`$_{\text{bin}}$ (which has decimal equivalent 6_{dec}) meaning it shifts all of the bits one to the right (and discards the least significant bit). This is the equivalent of dividing the decimal number $k = 12_{\text{dec}}$ by 2 (i.e. $12/2 = 6$).

4.2 Implementing the Factored Butterfly*

Once the input samples have been put into bit reversed order, we can implement the in-place decimation-in-time signal flow graph shown in Figure 5. This signal flow graph is implemented by grouping the butterfly structures at each stage according to their leading twiddle factors. First, all of the butterfly structures that have W_N^0 as their leading factor are calculated. Then, the butterfly structures with W_N^{-r} as their leading factor are calculated. Then, the butterfly structures with W_N^{-2r} as their leading factor are calculated, and so on until the $N/2$ butterfly structures with the W_N^{-qr} as their leading factor have been calculated at a given stage. Here, $qr < N/2$, and q and r are non-negative integers. In the first stage, $r = N/2$ and q may take on the value $\{q = 0\}$, which means that the $N/2$ butterfly structures with W_N^0 as their leading

```
1.)   // Bit reversing algorithm coded in C
2.)   // The input array X (of type COMPLEX)
3.)   // N, M, and COMPLEX defined in FFT_header.h
4.)
5.)   COMPLEX temp;          // temporary storage complex variable swaps
6.)   short i;               // current sample index
7.)   short j;               // bit reversed index
8.)   short k;               // used to propagate carryovers
9.)
10.)  short N2 = N/2;        // N2 = N >> 1
11.)
12.)  // Bit-reversing algorithm. Since 0 -> 0 and N-1 -> N-1
13.)  //  under bit-reversal,these two reversals are skipped.
14.)
15.)  j = 0;
16.)
17.)  for(i=1; i<(N-1); i++)
18.)  {
19.)    k = N2;               // k is 1 in msb, 0 elsewhere
20.)
21.)    while(k<=j)           // Propagate carry to the right if bit is 1
22.)    {
23.)      j = j - k;          // Bit tested is 1, so clear it.
24.)      k = k>>1;           // Carryover binary 1 to right one bit.
25.)    }
26.)
27.)    j = j+k;              // current bit tested is 0, add 1 to that bit
28.)
29.)    //  Swap samples if current index is less than bit reversed index.
30.)
31.)    if(i<j)
32.)    {
33.)      temp.real = X[j].real;      // Hold value at bit reversed location.
34.)      temp.imag = X[j].imag;
35.)      X[j].real = X[i].real;      // Insert value in current location
36.)      X[j].imag = X[i].imag;      // at bit reversed location.
37.)      X[i].real = temp1.real;     // Insert value in bit reversed
38.)      X[i].imag = temp1.imag;     // location at current location
39.)    }
40.)  }                               // end for loop
```

Figure 6: A C coded bit reversing algorithm. Adapted from [8].

factor are calculated. In the second stage, $r = N/4$ and q may take on the values $\{q = 0, 1\}$, which means that the $N/4$ butterfly structures with W_N^0 as their leading factor are calculated and then, the $N/4$ butterfly structures with $W_N^{-N/4}$ as their leading factor are calculated. In the l-th stage, $r = N/(2^l)$ and q may take on the values $\{q = 0, 1, \ldots, 2^{l-1} - 1\}$ and the $N/2$ butterfly structures are calculated accordingly. This continues until all of the $N/2$ butterfly structures in each of the $M = \log_2(N)$ stages have been calculated.

The C coded implementation of the decimation-in-time algorithm is shown in Figure 7 (not a complete program – notice the absence of `main()`. Note that the array `X[]` is assumed to be in bit reversed order (i.e. the input array has been processed by the C code in Figure 6.). Pay particular attention to lines 26 and 27 in Figure 7. The twiddle factors, namely $W[i], i = 0, 1, \ldots, (N/2)-1$, are assumed to be the first $N/2$ roots of unity defined by $W[i] = e^{j2\pi i/N}$, but we need the complex conjugate of these, namely $\overline{W[i]} = e^{-j2\pi i/N}$. This means the values stored the structure `temp` are

$$\begin{aligned}\mathrm{Re}\{temp\} &= \mathrm{Re}\{X[i_lower]\}\mathrm{Re}\{\overline{W[i]}\} - \mathrm{Im}\{X[i_lower]\}\mathrm{Im}\{\overline{W[i]}\} \\ &= \mathrm{Re}\{X[i_lower]\}\mathrm{Re}\{W[i]\} + \mathrm{Im}\{X[i_lower]\}\mathrm{Im}\{W[i]\}\end{aligned}$$

$$\begin{aligned}\mathrm{Im}\{temp\} &= \mathrm{Im}\{X[i_lower]\}\mathrm{Re}\{\overline{W[i]}\} + \mathrm{Re}\{X[i_lower]\}\mathrm{Im}\{\overline{W[i]}\} \\ &= \mathrm{Im}\{X[i_lower]\}\mathrm{Re}\{W[i]\} - \mathrm{Re}\{X[i_lower]\}\mathrm{Im}\{W[i]\},\end{aligned}$$

where i_lower is the the index of the lower butterfly node for a given stage and twiddle factor. The variable `temp` is used to store the value of the lower node scaled by the conjugate of the twiddle factor. This is required so the in-place algorithm can calculate and overwrite the values in the lower node at current stage in lines 29 and 30 and still have the value of the scaled lower node at the current stage to calculate and overwrite the top node in lines 32 and 33. This is a programming concern that comes up often when two or more registers are used for accumulating or in-place processing. This problem is easily solved by using temporary registers.

For sake of clarity, note that the variables `step` and `stage` code the variables r and l from the previous discussion, respectively. Also, at each stage, q takes on the values {`0`, `1`, . . ., `numBF-1`}.

For the final C coded decimation-in-time FFT algorithm, the code in Figure 7 is integrated with the code from Figure 6, along with the header file `FFT_header.h`, to create the C callable function `FFT_func.c` that we will use to implement our FFT algorithm. In the case that $N = 8$, the header file `FFT_header.h` is listed in Figure 8. This header files defines the structure `COMPLEX` (see Lab 2) and the order of the FFT to be implemented. This header file may be created using the homebrew MATLAB function `FFT_header_gen.m`, which is available on the class webpage. Download this file and use it to re-create the file listed in Figure 8.

The C program that we will use to test our FFT function is shown in Figure 9, which also includes `FFT_header.h`. Together, the C code in Figures 6, 7, 8, and 9 have been grouped together into the project `FFT_test.pjt` that is available on the class webpage. You are expected to create a header file like the one listed in Figure 8 using the MATLAB function provided. Download this project file and accompanying files, `FFT_test.c` and `FFT_func.c`, and create the header file in MATLAB. Open the project in CCS, and implement it on the DSK to verify that the FFT algorithm is working correctly.

NB: the program FFT_test.c does not terminate in an infinite loop as do programs which use the codec – it runs to completion and then stops at an exit location. In order to re-run it, you can either reload the program or click on **Debug** → **Restart** followed by **Debug** → **Run**

```
1.)   // N, M, COMPLEX defined in FFT_header.h, N2=N/2 (pre-defined)
2.)   // COMPLEX W[N2]                  // twiddle factors, passed to FFT_func.c
3.)   COMPLEX temp;                     // temporary storage complex variable
4.)   short i,j,k;                      // loop indices
5.)   short i_lower;                    // Index of lower point in butterfly
6.)   short step;
7.)   short stage;                      // FFT stage
8.)   short DFTpts;                     // # of points in sub DFT and offset to next DFT
9.)   short numBF;                      // # of butterflies in one DFT, offset to lower node
10.)
11.)  // Assume X[] is in reverse-bit order and do the M=log2(N) stages of butterflies
12.)  step = N2;                        // step = N/2, N/4, N/8, ... 1
13.)  for(stage=1; stage <= M; stage++)
14.)  {
15.)    DFTpts = 1 << stage;            // DFTpts = 2^stage = points in sub DFT
16.)    numBF = DFTpts/2;               // number of butterflies in sub-DFT
17.)    k = 0;                          // initial twiddle factor index
18.)
19.)    // Do butterflies for current stage
20.)    for(j=0; j<numBF; j++)          // do the numBF butterflies per sub DFT
21.)    {
22.)      // Compute butterflies that use same twiddle factor, W[k]
23.)      for(i=j; i<N; i += DFTpts)
24.)      {
25.)        i_lower = i + numBF;        // index of lower point in butterfly
26.)        temp.real = X[i_lower].real*W[k].real + X[i_lower].imag*W[k].imag;
27.)        temp.imag = X[i_lower].imag*W[k].real - X[i_lower].real*W[k].imag;
28.)
29.)        X[i_lower].real = X[i].real - temp.real;
30.)        X[i_lower].imag = X[i].imag - temp.imag;
31.)
32.)        X[i].real  = X[i].real + temp.real;
33.)        X[i].imag  = X[i].imag + temp.imag;
34.)      }
35.)      k += step;                    // increment twiddle index
36.)    }
37.)    step = step/2;                  // calculate step for next stage
38.)  }
```

Figure 7: A C coded decimation-in-time algorithm. Adapted from [8].

Assignment

2. Using the DSK and the project `FFT_test.pjt`, located on the class webpage, calculate the FS coefficients for $\{x[n] = \cos(\frac{3\pi}{8}n), n = 0, 1, \ldots, 15\}$. What are the values of `N`, `M`, and `N2`? Calculate the 16 discrete Fourier series coefficients by hand using eqn (1) to verify that the C coded FFT function is working properly. If the sample rate of the system were $f_o = 8$kHz, then what frequencies would the non-zero FS coefficients correspond to? If the sequence $x[n]$ were aliased every 16 samples (i.e. $x[n] = x[n+16r]$ for every integer r) and sent to the on-board codec, what would the output be?

3. Create a function that implements an in-place inverse FFT. Label this function `IFFT_func.c`. This function should have two complex arrays passed to it, namely `X[]` and `W[]`, where `X[]` will contain the array of elements that the IFFT algorithm will operate on in-place, and `W[]` will contain the $N/2$ twiddle factors. These twiddle factors should be the same as those used in `FFT_func.c`. Explain what changes need to be made to convert an FFT algorithm to an IFFT algorithm. (HINT: Compare equations (1) and (2).) Use the output of the FFT evaluated in question 3 as your input to your IFFT function. What is the output? (HINT: If your IFFT algorithm is working correctly, you will find that the cascade of the FFT and the IFFT is the identity operator. Thus the output of cascade is the input.)

```
// FFT_header.h
// This file must be included in FFT_func.c
// and by the program that calls FFT_func.c

#define N 8                              // N-point FFT
#define M 3                                // M=log2(N)
#define N2 4                               // N/2 (number of twiddle factors)
#define PI 3.14159265358979                // fixed-point approx. to pi

typedef struct {float real,imag;} COMPLEX;         // structure COMPLEX
```

Figure 8: A listing of `FFT_header.h` for $N = 8$

```
// FFT_test.c
// Used to test FFT_func.c
// N-point FFT, where N is defined in FFT_header.h
/*1 */ #include <math.h>
/*2 */ #include <stdio.h>
/*3 */ #include "FFT_header.h"              // defines COMPLEX structure
/*4 */                                      // and FFT order
/*5 */ void FFT_func(COMPLEX *X, COMPLEX *W); // FFT function prototype
/*6 */
/*7 */ COMPLEX X[N];                        // Declare input array
/*8 */ COMPLEX W[N2];                       // Used to hold the N/2 twiddle factors
/*9 */
/*10*/ int main()
/*11*/ {
/*12*/   short i;                           // loop index
/*13*/
/*14*/   // Calculate twiddle factors
/*15*/   for(i=0; i<N2; i++)
/*16*/   {
/*17*/     W[i].real = cos(2.0*PI*i/N);
/*18*/     W[i].imag = sin(2.0*PI*i/N);
/*19*/   }
/*20*/
/*21*/   // Initialize input array
/*22*/   for(i=0; i<N; i++)
/*23*/   {
/*24*/     X[i].real = cos((float) 2.0*PI*3*i/N);
/*25*/     X[i].imag = 0.0;
/*26*/   }
/*27*/
/*28*/   FFT_func(X,W);                     // perform in-place FFT
/*29*/
/*30*/   // Display results on screen
/*31*/   for(i=0; i<N; i++)
/*32*/         printf("X[%d] = \t%10.5f + j %3.5f\n",i,(X[i]).real, (X[i]).imag);
/*33*/   return 0;
/*34*/ }
```

Figure 9: A C coded program that tests our FFT algorithm. Adapted from [8].

5 FIR Filtering using an FFT Algorithm

5.1 Real-Time Block Processing on the C6713 DSK*

Now that we have working FFT and IFFT algorithms, we need to incorporate them into a real-time system. In this section, we will process blocks or vectors of data sequentially in time instead of calculating scalar outputs sequentially in time. To begin, let's build a template for processing blocks of data on the C6713 DSK. We are going to implement the equivalent of the straight wire program from Lab 2, but we will use it to test our FFT and IFFT algorithms.

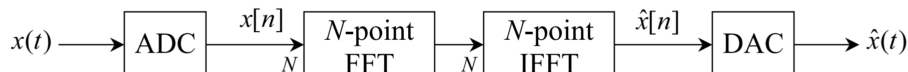

Figure 10: Straight Wire using Coded FFT and IFFT Algorithms.

Consider the system in Figure 10. The sequence $\widehat{x}[n] \approx x[n]$, with discrepancies resulting from finite precision processing. However, these effects will be minimal due to the floating-point algorithm used. The trade-off is that the floating-point algorithm will take more clock cycles to compute than a fixed-point algorithm would, but the finite precision effects will not need to be managed. A C code implementation of Figure 10 is given in Figure 11. Note that the functions `FFT_func.c`, `IFFT_func.c`, and appropriate header files must be included with the project.

The idea in `IO_stream.c` is that samples are read from the on-board codec, converted to floating-point numbers, and stored in the floating-point array `io_buffer` (input - output buffer). Once this buffer is full, the samples are copied to a complex array structure labelled `process`, and previously processed data is copied to the array `io_buffer`. While this is taking place, the imaginary part of the `process` array is set to zero. This is done since the samples being read in are assumed to be purely real. This may seem redundant since the output in the `process` array is assumed to be real, but due to finite precision effects, the imaginary part of the output may not be exactly equal to zero. Next, the FFT algorithm is applied to the data in the `process` array in the `main()` function, while an interrupt is used to output previously processed samples and read in new samples. Now, as a sample is outputted from a memory location in the array `io_buffer`, a new sample is read into the same memory and a global counter (`ctr` in Figure 11) is used to point to the next memory location in `io_buffer`. This continues until `io_buffer` is full, at which point the procedure is repeated.

The DSP algorithm waits for the `io_buffer` to fill in line 28. Once the buffer is full, the variable `flag` is set to true in line 16 (located in the interrupt) and the algorithm, upon return from the interrupt, proceeds to line 30, where it resets the `flag` variable to false. Then, in lines 31 through 37, the data in `io_buffer` is transferred to the real channel of the `process` array, the imaginary part of the process array is zeroed, and previously processed data is moved to the `io_buffer`. These data transfers will take place between the time the last sample of the previous block was read in from the codec and the time the first sample of the next block is sent to the codec. Therefore, the DSP chip will have roughly Nt_o seconds to process the rest of the algorithm, where N is the length of the block being processed and t_o is the sample rate of the system.

```
/*1 */ // io_stream.c - include header files, function prototypes, and W[N2]
/*2 */ COMPLEX process[N];
/*3 */ float tmp;
/*4 */ float io_buffer[N];
/*5 */ short ctr=0;
/*6 */ short flag=0;
/*7 */
/*8 */ interrupt void c_int11()   // interrupt service routine
/*9 */ {
/*10*/  output_sample((short)io_buffer[ctr]);
/*11*/  io_buffer[ctr++]=(float)input_sample();
/*12*/
/*13*/  if(ctr >= N)
/*14*/  {
/*15*/    ctr=0;
/*16*/    flag = 1;
/*17*/  }
/*18*/  return;                    //return from interrupt
/*19*/ }
/*20*/
/*21*/ void main()
/*22*/ {
/*23*/   short i;                  // local counter
/*24*/
/*25*/   // Calculate twiddle factors for (I)FFT
/*26*/
/*27*/   comm_intr();
/*28*/   while(1)
/*29*/   {
/*30*/     while(flag==0);
/*31*/
/*32*/     // Once the buffer is full, implement DSP algorithm here
/*33*/     flag = 0;
/*34*/
/*35*/     for(i=0; i<N; i++)
/*36*/         {
/*37*/           tmp = process[i].real;
/*38*/           process[i].real = io_buffer[i];
/*39*/           process[i].imag = 0.0;
/*40*/    io_buffer[i] = tmp;
/*41*/     }
/*42*/
/*43*/     FFT_func(process, W);   // perform in-place FFT
/*44*/     IFFT_func(process, W); // perform in-place FFT
/*45*/   }
/*46*/ }
```

Figure 11: Stripped Down version of `IO_stream.c`.

Assignment

4. Download the project `IO_stream.pjt`, `IO_stream.c`, and `FFT_header.h` from the class webpage. Incorporate your FFT and IFFT functions into this project and build and run the project on the DSK. Verify that the program is implementing a straight wire. Notice that this FFT header file is designed for a 128-point FFT. This means that we are processing data in blocks of 128 samples. Now, once the buffer has filled and the data has been swapped between the `io_buffer` array and the `process` array, the algorithm has about $Nt_0 = 128 * .125$ms (about 16ms) to execute the rest of the algorithm. In reality, this algorithm will finish after only a sample or two has been read in from the codec (about .25ms), so the DSK will be idle (ignoring the input and output of data via the codec) for about 15.75ms for every block of data that is processed. This extra time will allow us to use the FFT (and IFFT) to implement block convolution, which we will discuss next. This project will serve as a template for doing block processing in real-time.

5.2 Linear Convolution using the DFT

A property that holds for all Fourier transforms is that multiplication in one domain transforms to convolution in the other. Since the DFT is periodic in both domains, the convolution will be cyclic in both domains. In this section, we wish to do linear convolution of two finite length sequences in the time domain by first calculating their respective DFTs, multiplying the DFTs term-by-term[3], and then taking the inverse DFT of the product to get the convolved time sequence. However, if we apply this process to the finite length sequences only, we will get a result that is the circular convolution and not the linear convolution of the two sequences. To get linear convolution from the DFT, we note that the convolution in time of two finite length sequences, say $x_1[n]$ of length L and $x_2[n]$ of length Q, results in $y[n] = (x_1 * x_2)[n]$, which will be of length $L + Q - 1$. Using this fact, we can zero-pad the time sequences, so they are both of length $L + Q - 1$. To do this, we define $\widetilde{x}_1[n] = x_1[n]$ for $n = 0, 1, \ldots, L-1$ and 0 for $n = L, L+1, \ldots, L+Q-1$, and $\widetilde{x}_2[n] = x_2[n]$ for $n = 0, 1, \ldots, Q-1$ and 0 for $n = L, L+1, \ldots, L+Q-1$. Then, we can take the DFTs of $\widetilde{x}_1$ and $\widetilde{x}_2$, multiply the DFTs term-by-term, and inverse DFT the product to get the desired linear convolution $(x_1 * x_2)[n]$.

5.3 Overlap and Add Block Convolution FIR Filtering*

In the previous programming example, `IO_stream.c`, the output block size was the same as the input block size, so there was no overlap between the blocks. However, when block convolution is implemented, blocks of L samples will be read in from the codec and output blocks of $N+Q-1$ samples will be produced, which will overlap by $Q-1$ samples. This leaves two options for dealing with the overlap. Either, read in blocks of L samples, overlap the input blocks (i.e. make the last $Q-1$ samples of a given input block the first $Q-1$ samples of the next input block), convolve the input blocks with the filter response using an FFT method, and save the last L samples, or read in blocks of L samples, convolve the input blocks with the filter response using an FFT method, and add the last $Q-1$ samples (term-by-term) from the processed data

[3] It is assumed apriori that the two sequences are of the same length.

of a given block to the first $Q-1$ samples of processed data in the next block. These two algorithms are known as the *Overlap and Save* and *Overlap and Add* methods, respectively [6]. In this section, we will develop the overlap and add algorithm, which you will ultimately code in C. To visualize this algorithm, consider the sequence $x[n]$ in the top panel of Figure 12.

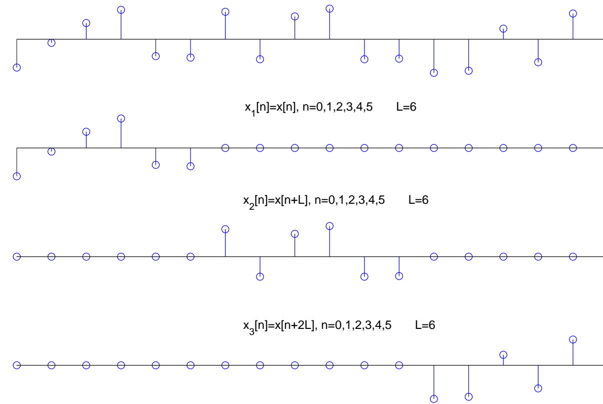

Figure 12: Splitting the input stream into blocks of length $L = 6$.

This input stream, $x[n]$, is split into blocks of length $L = 6$ (shown in the bottom three panels of Figure 12). Let the FIR filter $h[n]$ be the 1kHz notch filter from lab 4 with duration 3 samples (i.e. $Q = 3$). Now, if each of the sequences $x_1[n]$, $x_2[n]$, and $x_3[n]$ is convolved with $h[n]$, then the results will be of length $N = 8$, which are shown in the top three panels of Figure 13. The output is then formed by summing these sequences vertically (term-by-term) to generate the output $y[n]$.

> ***Assignment***
>
> 5. Consider the output $y[n]$ in Figure 13 when $n = 6, 7, 8$, and 9. Show that the summation of the overlap from the blocks $(h * x_1)[n]$ and $(h * x_2)[n]$ gives the desired result $y[n] = h[0]x[n] + h[1]x[n-1] + h[2]x[n-2]$. (HINT: Use the linear shift-invariance of convolution.)

5.4 Implementing the Overlap and Add Algorithm using an FFT*

Located on the class webpage is a homebrew MATLAB function, `Overlap_Add_header_gen.m` that will create the header files you will need to implement an overlap and add algorithm. This function will create the header file `FFT_header.h` and `coeffs.h`, where `FFT_header.h` is the standard FFT header file that we have been using. The new file, `coeffs.h`, must be included in the C source code containing the `main()` function. This file will contain the filter coefficients, stored in the floating-point array `h[]`, define the number of inputs to read in, `L`, and define the number of samples to overlap, `P`. In the previous discussion, L was the number of samples to read in, Q was the filter duration, and $Q-1$ was the number of samples to overlap. Now,

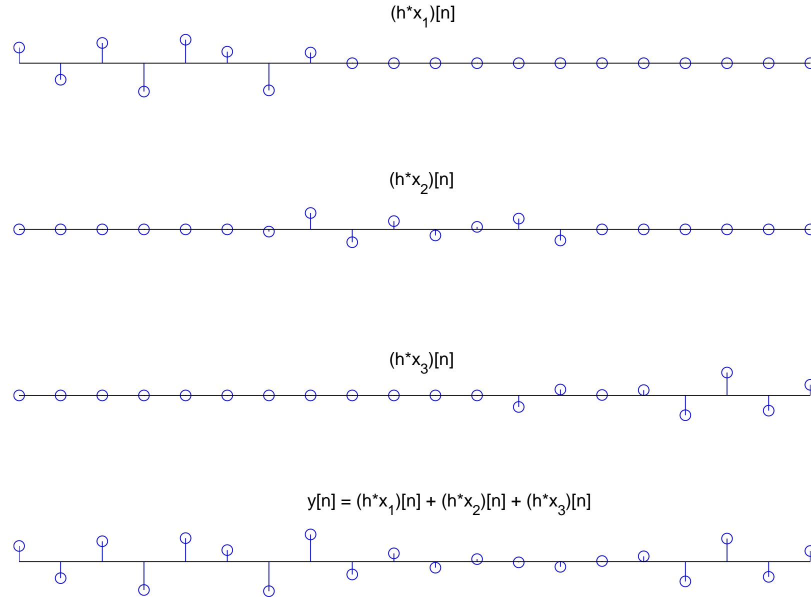

Figure 13: Convolved input blocks and final output $y[n]$.

L remains the same, the number of samples to overlap is $P = Q - 1$, and the FFT order is $N = L + P$ [4]. Note that the function `Overlap_Add_header_gen.m` can generate this information from the filter coefficients `h` and the FFT order `N`, which are the two variables that must be passed to the function. Download this function from the class webpage and use `help` and `type` to see how this works.

For the overlap and add algorithm, you will need the following global variables:

- a complex structure of length N to hold the DFT of the filter response,
- a complex structure of length of length N to use as a process buffer,
- a floating-point array (IO buffer) of length L to interface the codec,
- a floating-point array of length L to hold the samples to overlap from the previous block of processed data; this array will also be used to store the final processed block of data that will be copied to the IO buffer,
- temporary variable(s) for the complex multiplication of the filter FS coefficients with the DFT of the zero-padded input sequence, and
- other miscellaneous variables needed to implement any real-time block processing algorithm.

Once you have your variables in order, you must initialize some of the variables. Before the codec is initialized, initialize the following:

- the twiddle factors, and
- the discrete FS coefficients of the zero-padded filter impulse response.

After these initializations are made, initialize the codec, and code the overlap and add algorithm as follows (in the order given):

[4] Note that P is the filter order as it was defined in Lab 4, except the letter to denote this has been changed from N to P. This is done to avoid a conflict of variables with the FFT order N.

- copy the IO buffer into the first L memory locations in the process array and copy the previously processed data into the IO buffer,
- copy the last P samples from the process array into the overlap array and zero the last $L - P$ samples of the overlap array,
- zero the imaginary part of the process array and perform an in-place FFT on the array,
- multiply the process array with the DFT of the zero-padded filter coefficients,
- IFFT the process array,
- add the first L samples of the process array to the overlap array, which will serve as the output for the next block of data.

Assignment

6. Create a project that implements the overlap and add algorithm given above. Include a copy of the C source code that you used to achieve this. Briefly explain how your program works. What is the filter latency? Numerically, compare the computations required to implement this algorithm with the direct convolution of Lab 4.

6 End Notes

In this lab, we have explored the FFT algorithm and one of its applications to real-time systems. Other applications include spectrum analysis and orthogonal frequency division multiplexing (OFDM) for modulation and demodulation, which we will explore in more depth in later labs.

6.1 Advanced Lab Topics

- Implement a decimation in frequency FFT algorithms. Refer to [6].
- Implement a radix-4 (or higher order radix) FFT algorithm.
- Implement the DFT using recursive linear filter methods like the Goertzel algorithm and the Chirp Z-transform algorithm. Refer to [6].
- Implement an *overlap and save* block convolution FIR filter.

References

[1] Rulph Chassaing. *DSP Applications Using C and the TMS320C6x DSK.* Wiley, New York, 2002.

[2] Colorado State University, Fort Collins, CO. *Signals and Systems Laboratory 11: The FFT and its Applications*, 2001.

[3] J. W. Cooley and J. W. Tukey. An algorithm for the machine calculation of complex fourier series. *Mathematics of Computation*, 19:297–301, 1965.

[4] Simon Haykin and Barry Van Veen. *Signals and Systems.* Wiley, New York, 1999.

[5] M.T. Heideman, D.H. Johnson, and C.S. Burrus. Gauss and the history of the fast fourier transform. *IEEE Transactions on Acoustics, Speech, and Signal Processing*, 1:14–21, October 1984.

[6] Alan V. Oppenheim and Ronald W. Schafer. *Discrete-Time Signal Processing*. Prentice Hall, Uper Saddle River, NJ, 1989.

[7] John G. Proakis and Dimitris G. Manolakis. *Digital Signal Processing: Principles, Algorithms, and Applications*. Prentice Hall, Uper Saddle River, NJ, 1996.

[8] Steven A. Tretter. *Communication Design Using DSP Algorithms: With Laboratory Experiments for the TMS320C6701 and TMS320C6711*. Kluwer Academic/Plenum Publishers, New York, 2003.