ECE4703 B Term 2010 Laboratory Assignment 5

Project Code and Report Due by 3:00pm 08-Dec-2011

The goals of this laboratory assignment are:

- to develop an understanding of *frame-based* digital signal processing,
- to familiarize you with computationally efficient techniques that reduce computational load not through compiler optimization or architectural optimization but, rather, through clever *algorithm optimization*,
- to allow you to experimentally verify *asymptotic complexity* predictions, and
- to allow you to apply the FFT to a practical signal processing problem.

1 Problem Statement

In this assignment, you will perform frame-based linear convolution of two length-M sequences first using direct linear convolution and then using FFT-based fast linear convolution. Specifically, we wish to compute

$$z_k = \sum_{m=0}^{M-1} x_m y_{k-m}$$

for $k = 0, \ldots, 2M-1$ where x_m and y_m are real-valued samples from the left and right channels of the codec, respectively. Note that x and y are both real-valued sequences, hence z will also be real-valued. All math in this assignment should be performed in single-precision floating point and all of your code will be written in C. To allow testing at larger values of M, the sampling rate for all parts of this assignment should be set to $f_s = 8$kHz.

As discussed in lecture, all frame-based processing should be performed by a function called from `main()`. The ISR should only fill buffers and set flags to indicate a new buffer has been filled. To run in real-time, you need to complete all of your computations before the next incoming buffer is full. You do not need to send any output signals to the AIC23 codec.

2 Part 1: Direct Linear Convolution

In this part of the assignment, you will perform direct linear convolution of the x and y sequences. A suggested flow diagram for the assignment is shown in Figure 1. You will need the following buffers:

- Left channel incoming buffer (M floats)

- Right channel incoming buffer (M floats)
- Left channel active buffer (M floats)
- Right channel active buffer (M floats)
- Output buffer ($2M - 1$ floats)

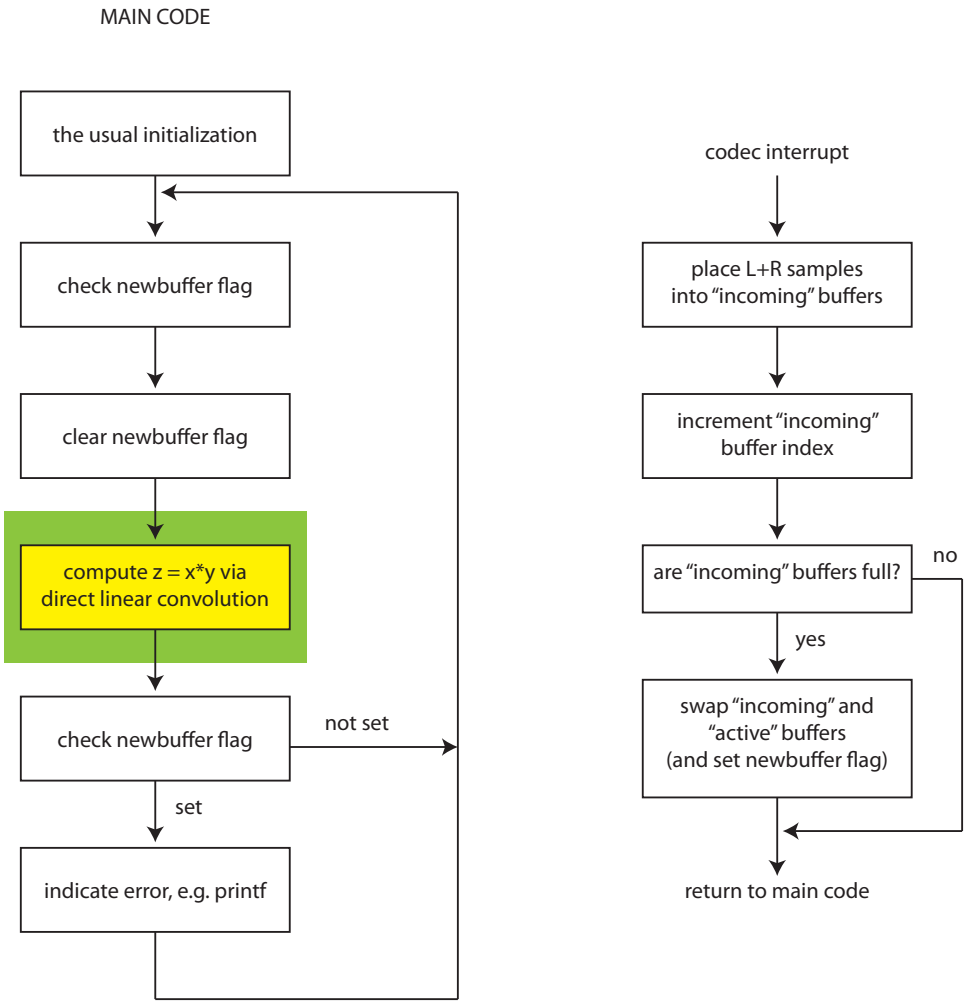

Figure 1: Suggested flow of your direct linear convolution code. The green box represents the direct linear convolution function that you should profile for different values of M until the code no longer runs in real-time.

Experimentally determine how large M can be before your function no longer runs in real-time for both unoptimized code and optimized code, i.e. with and without compiler optimization. In the cycle accurate simulator, profile the CPU cycles of your direct linear convolution function for several different values of M up to the value of M where the function no longer runs in real-time for both unoptimized code and optimized code.

3 Part 2: Fast Convolution

In this part of the assignment, you will perform fast convolution of the x and y sequences using the technique described in lecture and in the Lyons and Welch textbook handouts. A suggested flow diagram for the assignment is shown in Figure 2. You will need at least the following buffers:

- Left channel incoming buffer (N COMPLEX)
- Right channel incoming buffer (N COMPLEX)
- Left channel active buffer (N COMPLEX)
- Right channel active buffer (N COMPLEX)
- Output buffer (N COMPLEX)

Even though the samples from the left and right channels are real-valued, the FFT code you will use requires the input signal to be formatted as COMPLEX, which we define as

```
// complex typedef
typedef struct {
  float re,im;
} COMPLEX;
```

Note that we also declare these buffers as length N and initialize them to zeros to automatically perform the required zero padding for linear convolution via the FFT (the ISR will only fill the first M elements of each buffer). Hence, an explicit zero-padding step is not required. Also, recall that you should select the smallest value of N such that $N \geq 2M - 1$ and N is an integer power of two.

3.1 Steps 1-2

Due to the wide variety of applications for the FFT, TI provides an optimized linear assembly function to implement FFTs on the C6x. You will need three functions to use TI's optimized FFT routines. These functions are `cfftr2_dit`, `digitrev_index`, and `bitrev`. The typical usage is

```
digitrev_index(iw,N/RADIX,RADIX); //produces index for bitrev() W
bitrev(w,iw,N/RADIX);      //bit reverse W
cfftr2_dit(x,w,N) ;      //TI floating-pt complex FFT
digitrev_index(ix, N, RADIX);     //produces index for bitrev() X
bitrev(x,ix,N);      //freq scrambled->bit-reverse X
```

Note that the first two lines only need to be run once in order to get the twiddle factors in the bit-reversed order expected by the TI FFT function. It is probably a good idea to take care of that in the initialization part of your code. You can also try doing the unscrambling of the FFT output *after the product in step 3* since it should be more efficient to do the unscrambling once rather than twice. In other words, you may want to try performing the product on the scrambled FFT outputs and then unscrambling the product, rather than unscrambling both FFTs and then performing the product.

You should read the header comments in the provided FFT files to make sure that you know how to use them correctly. The FFT function is called like this:

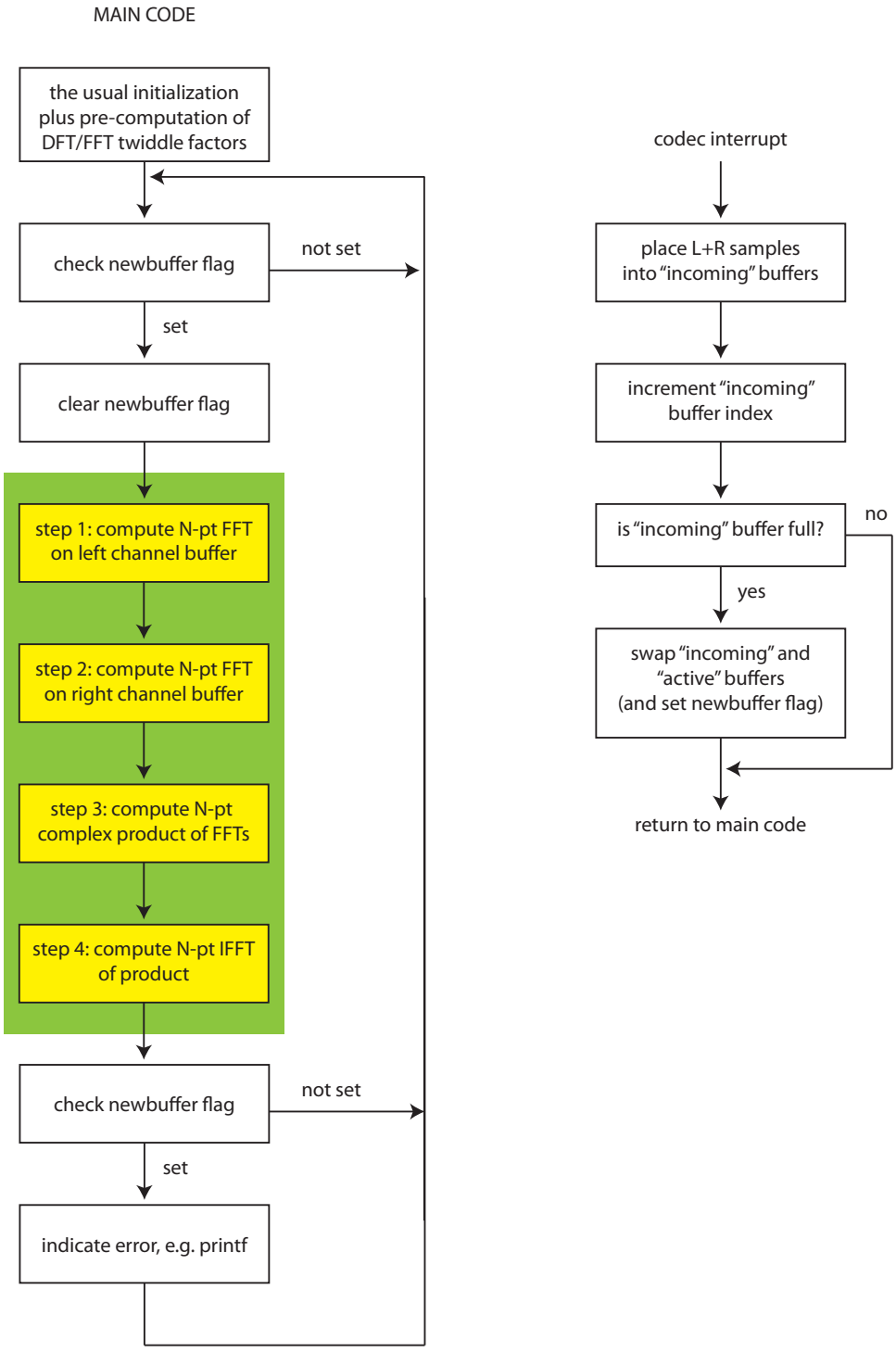

Figure 2: Suggested flow of your fast linear convolution code. The green box represents the fast linear convolution function that you should profile for different values of M until the code no longer runs in real-time.

```
void cfftr2_dit( float *x, const float *w, short N)

// x  Pointer to Array of Dimension 2*N elements holding
//    Input to and Outputs from the function
// w  Pointer to an array holding the complex twiddle factors
// N  Number of complex points in z
```

The input to the FFT is in the array x and the result of the FFT is returned in the array x (the input is overwritten by the FFT function). You should compute the sin/cos portions of the "twiddle factors" during the initialization part of your code (prior to the FFT function call) and store them in W. You will probably want to include the header file `math.h` to allow for pre-computation of the sin and cos terms needed in for the complex "twiddle-factors". All input/output arrays and twiddle factors should be single-precision floating point datatypes.

You should confirm that this part of your code is working correctly before proceeding. One way to do this is just provide some known input to the FFT function and compare the output to Matlab.

3.2 Step 3

This step should be straightforward. Recall that the product of two complex numbers

$$(a + jb)(c + jd) = (ac - bd) + j(bc + ad).$$

3.3 Step 4

Refer to the comments in TI's FFT function to determine how to perform an IFFT using the same function. After the IFFT has been performed, the desired result will be in the first $2M - 1$ real elements of the output buffer. The imaginary elements of the output buffer should all be very close to zero.

3.4 Verification and Testing

Confirm your fast convolution code gives the same results as your direct convolution code. Experimentally determine how large M can be before your function no longer runs in real-time for both unoptimized code and optimized code, i.e. with and without compiler optimization. In the cycle accurate simulator, profile the CPU cycles of your fast linear convolution function for several different values of M up to the value of M where the function no longer runs in real-time for both unoptimized code and optimized code.

4 In Lab

You will work with the same lab partner as in the prior laboratory assignments. Please contact the instructor if your lab partner has dropped the course or if you have concerns about your lab partner's performance on the prior assignment.

5 Code Submission and Specific Items to Discuss in Your Report

Your code will be tested for correct functionality and profiled by the grader for select values of M to determine if the profiling results in your report are accurate. Please be sure to write structured C code and to comment your code liberally to facilitate testing by the grader.

In addition to verifying the correct operation of your code, you discuss and compare your profiling results for direct and fast linear convolution, with and without optimization. Can you explain your profiling results? You should plot your profiling results to demonstrate the trends clearly and discuss what parts of the fast convolution code dominates the overall complexity at large M. Try to fit the asymptotic complexity curves $c_1 M^2$ and $c_2 M \log_2 M$ to the actual profiling results to see how closely the actual computational burden tracks the predicted complexity (you can try to find a least-squares fit for c_1 and c_2 or just try different values until you get the best fit to your eye). Your plots should include distinct line types and clearly labeled legends. You may want to try generating semilog or loglog plots to see if that makes your results easier to interpret.