Lab 8: Overlap-Save Method for filtering using FFT

EE 352 DSP Laboratory

Lab Goals

To understand implementation of real-time block processing on DSP. The following tasks will be carried out here,

- Understand the DMA operation in the DSP.
- Implement the real time linear convolution using overlap-save method.

0.1 Learning checkpoint

All the following tasks need to be shown to your TA to get full credit for this lab session. Complete this checkpoint before you start reading the manual.

- Let, $x[n]$ be the sequence formed by taking the entire roll number of one of you two with the letter 'd' (if present) replaced by value 4 and $h[n]$ be the sequence formed by taking last 3 digits of the roll number of the other of you two. For example, if your roll numbers are 45d678912 and 34d567890, then $x[n] = \{4, 5, 4, 6, 7, 8, 9, 1, 2\}$ and $h[n] = \{8, 9, 0\}$.
- You have to convolve $x[n]$ and $h[n]$ using the overlap-save technique. The largest FFT block you can use is of length 8.
- Decide the length of the FFT block you will use.
- Now, do not solve the entire problem. Just, explain your TA the algorithm you would use to do this, writing down any intermediate signals or their expressions (For example, $x_1[n]$, $x_2[n]$,..., $y_1[n]$, $y_2[n]$,...).
- After going to home, in your leisure time, try completing the example. You can let MATLAB take any load of computations.

1 Overlap-Save method

1.1 Overview of Overlap-Save method

In Lab 3 and Lab 4, FIR filtering was implemented in time domain using linear and circular buffering techniques. Here, a similar filtering operation is implemented in frequency domain using *Overlap-Save* method, which is one of the efficient techniques used for frequency domain filtering.

If the length of the filter is A and the input data size is B, then the convolved output will have $C = A + B - 1$ values. So, if we have a real-time data coming continuously and would like to use a frequency domain approach to filtering, then we need to implement product of FFTs. For efficient FFT operation, the following procedure must be followed.

- Let's say we are allowed to use FFT blocks of size N which is a power of 2.
- Let L be the length of the given filter.
- Now, to make full use of the available size of the FFT block, we can set the length of the input blocks to M such that, $N = L + M - 1$ or $M = N - (L - 1)$.
- The critical step is to choose these M values in the input blocks from the incoming data stream. This is performed as follows.

1. If this is the first input block, choose the first $L-1$ values to be zeros. Else, choose them to be the last $L-1$ values from the previous input block.
2. Choose the remaining $N-(L-1)$ values as the latest received values from the input data stream.

- If the i^{th} input block is represented by $X_i[n]; 0 \leq n \leq N-1$ and filter coefficients are represented as $h[n]; 0 \leq n \leq L-1$, N point FFTs of $X_i[n]$ and $h[n]$ are computed (by appending $h[n]$ by $N-L$ zeros) and their product is obtained.
- Now, calculate the i^{th} output block as IFFT(FFT($X_i[n]$) $\times$ FFT($h[n]$)).
- The first $L-1$ values of the resulting output are discarded and remaining output values are concatenated at each stage to form the correct output.

1.2 Example

Suppose our filter length is 12 and the FFT size we are interested in is 128, i.e., $L = 12$ and $N = 128$, then $M = N - L + 1 = 117$. So, every time we collect 117 values of data and process them. Therefore, each time we insert the last 11 values of the previous input data block to the current input data block. Also we append 116 zeros to the filter coefficients. Let the input block be $X[n]$ and impulse response be $H[n]$, now

$X[k] = FFT(X[n])$ and $H[k] = FFT(H[n])$ are computed (128 point FFT).

$Y[n] = IFFT(X[k] \times H[k])$ is computed and the first 11 values are discarded each time and remaining samples are sent as output.

1.3 Learning checkpoint

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

1. Let's say we decide to use the Overlap-Save method to filter an incoming signal with the FFT-blocks of size 1024.
2. Some timing considerations:
   - The DSP is working at a frequency of 100 MHz.
   - The sampling frequency is configured to be 48 kHz (assume that the incoming signal is band-limited to 24 kHz to meet the Nyquist criteria).
   - The FFT hardware accelerator co-processor takes 7315 clock cycles to complete calculating a 1024-point FFT or a 1024-point IFFT.
3. Let's use the following algorithm to filter the incoming signal.
   (1) Acquire 1024 samples.
   (2) Stop accepting the data and perform the frequency domain filtering operation (calculating FFTs, then multiplying them and calculating inverse FFTs) on the data acquired.
   (3) Go to step (1)
4. The pitfall of this algorithm is that the DSP has to finish processing the previous input-block before the 1025^{th} sample arrives. In other words, the processing time for an input-block must be less than inter-sample duration. Let's investigate if this is happening.
5. First, determine the sampling time interval. Confirm it with your TA/RA.(Use the above timing considerations to determine this)
6. Next, find out the time taken by the FFT and the IFFT routines. Confirm it with your TA/RA. (Use the above timing considerations to determine this)
7. Can this algorithm work? Shall we drop any samples generated by the codec while performing the filtering operation?

8. What can you conclude from the number of samples being dropped, if any? Is sequential operation fast enough or is there a need for parallelization? Should the data acquisition operation be assigned to someone else so that the processor can continue processing previously acquired data.

2 Key difference between time domain and frequency domain filtering in real-time (Why do we need DMA?)

In previous lab experiments, the processing of data has been done on a sample-by-sample basis. For example, convolution had to be performed only in those cases where, each time a new sample comes in the multiplication and accumulation. But in signal processing, we encounter applications like block filtering, which can refer to either time domain convolution based filtering or frequency domain based overlap-save processing, where processing is done on a block of data. Naturally this requires buffering the data. Though in the former case also, we required a buffer, the processing happened on a sample by sample basis. But in block-filtering, entire buffer is filled up first and then the processing is done on it. This brings up another issue, what happens to the data that is continuously streaming in when the buffer is being processed? This could lead to loss of data if the processing of the buffer cannot be done before the arrival of next data sample. To avoid this data loss, we use two buffers. One buffer (buffer_2) gets filled in while the data in the other buffer (buffer_1) is being processed. Once the processing of buffer_1 is complete we switch to buffer_2 whose data gets processed and this swapping process continues indefinitely. We now briefly describe, with the help of an example, Ping-Pong buffering using DMA, an approach to efficiently implement the above processing.

2.1 Example

Among various codes uploaded on the EE-352 wiki page, that in `DMA_checkup` folder is an example of how the DMA is implemented. Import the projects `DMA_checkup` and `usbstk5515bsl` into CCS and then perform the experiment.

This example uses *Direct Memory Access* (DMA) feature for efficient transfer of data between memory and the codec. DMA is a special hardware feature that allows transfer of data between memories or between memory and peripheral devices without the intervention of the CPU. So the CPU has to just initiate the DMA controller and can then continue with its job of processing. We have used a template project provided on Texas Instruments webpage. C5515 has 4 DMA units. In this example we are using DMA0. Each DMA has 4 channels out of which channels 0 and 1 are input channels and 2 and 3 are output channels. You can quickly go through files `dma_registers.h`, `dma_setup.h`, `dma_setup.c` to see how different DMA channels have been configured. For more information on DMA, go through references provided on EE352 wiki page.

3 Explanation of DMA Example

The example takes in data through the codec and stores it in `DMA_InpX` buffers (first half part of this buffer is Ping buffer and latter half part of this buffer is Pong buffer) using DMA. in general, the data in these buffers will be processed alternately and the result will be placed in output (`DMA_OutX`) buffers. Here, we are just setting output equal to the input. We now see how this data transfer is efficiently implemented in the example.

Apart from the basic DMA operation, for optimal data transfer, the example uses some special features like.

- Ping-pong buffering
- Automatic channel sorting by the DMA controller
- Automatic re-initialization

`DMA_ISR(void)` is an Interrupt Service Routine defined in `dma_setup.c` file which sets Ping-Pong flags.

3.1 Program Flow

- In `main( )` function, we initialize c5515 board using `InitSystem()`. Then we configure AIC3204 codec. We set I2S bus in slave mode using `I2S0_setup()` and finally we configure DMA0 and its all 4 channels.
- The `DMA_ISR( )` interrupt service routine is called when a buffer has been filled. It modifies a state variable (flags) named `PingPongFlagInX` and `PingPongFlagOutX` to 1 or 0 that indicates whether the buffer is a PING or PONG buffer. Using values of these flags, we transfer data in `while(1)` loop (`main( )` function).

3.2 Important steps in implementation

- Using `usbstk5515bsl.lib` in the project.

  Make sure that you keep `memory model` in run time options `small`. Also don't forget to select `--ptrdiff_size` in `run time options` to be equal to 16. In `File search path`, add `usbstk5515bsl.lib` provided to you on EE-352 wiki page (and not from usual location).

- Real time Auto refresh mode

  The graph window in the CCS can be set in the auto refresh mode using `Silicon Real Time Mode`. Following steps indicate the real time refresh mode settings.

  1. Load the program.
  2. Go to `Target` ⇒ `Advanced` and select `Enable Silicon Real Time Mode` as shown in figure 1.

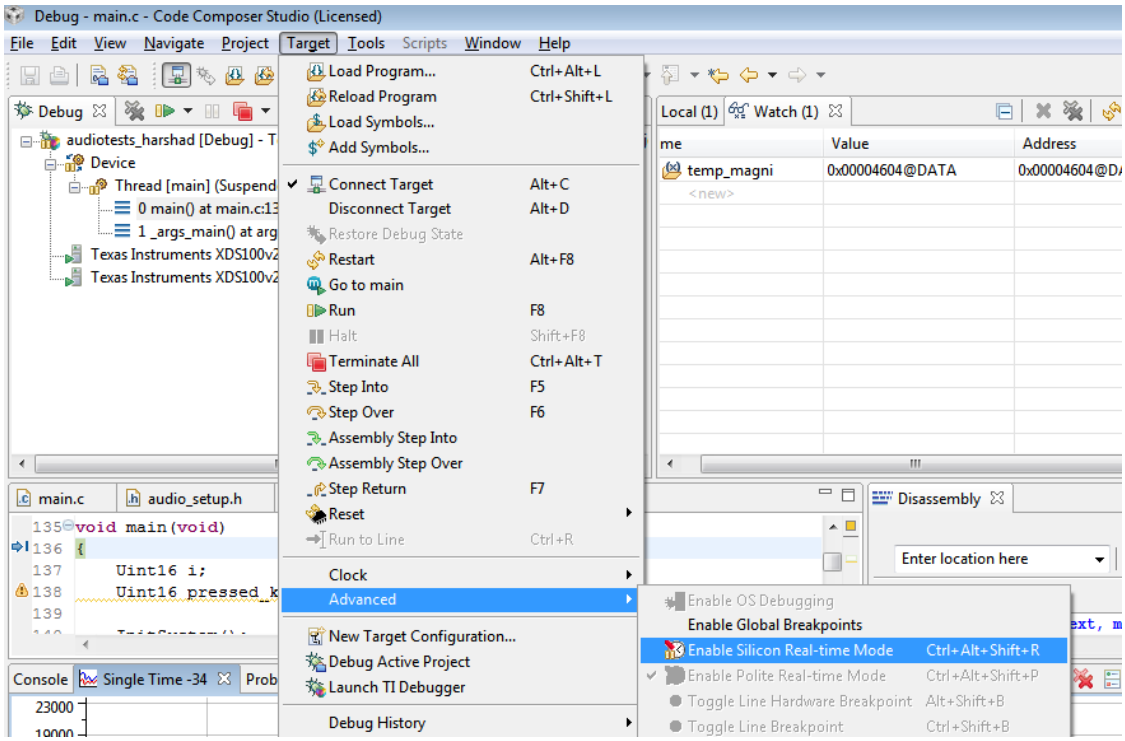

Figure 1: Selecting the Silicon-real time mode.

  3. Go to `tools` ⇒ `Graphs` and select and Enable Continuous refresh in graph window.

With these settings, you should be able to view the continuously refreshing graph in the graph window as shown in figure 2.

Note: For each part of the assignment, enable the real time refresh mode.

Go through the entire code, it is a self-explanatory code where all the details have been furnished.

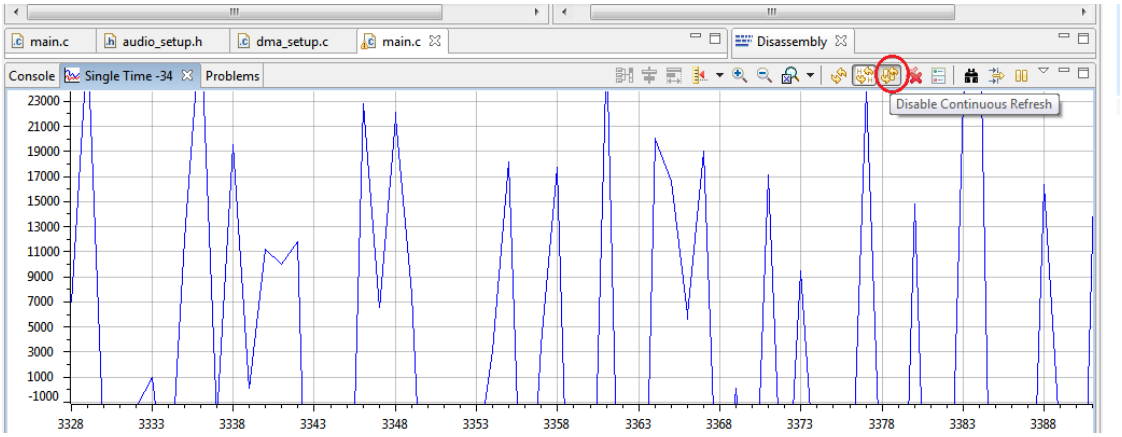

Figure 2: The continuously refreshing graph

3.3 Learning checkpoints

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

1. To verify whether your DMA setup is working correctly, perform the following steps.
   - You will be using the `DMA_checkup` project for this checkpoint.
   - As explained earlier, inside the infinite `while` loop, the program is just reading the values from the DMA buffers depending upon which of the two buffers (PING and PONG) is filled into the array `BufferL`.
   - Later, these values are output, as they are, into DMA output buffers.
   - Try to run this project, **but with the additional settings mentioned in section 3.2.**
   - Input a sine wave of frequency 3kHz to 5kHz from the function generator and observe the output on the DSO. If it is showing a sinusoid with the same frequency, then you are all set to move ahead. Skip the remaining steps in this checkpoint.
   - Graph the array `BufferL` into which the values are being read. Find out the size and the data-type of the array yourself.
   - If the plot is showing a sinusoid, check its amplitude. If it is very low (say, of the order of a few hundreds) then in the statements writing the values to the DMA output buffers, apply a gain to the `BufferL` values such that they occupy the whole range (or at least considerable part of the whole range) of the data-type.
   - If even this doesn't work, approach one of the TAs/RAs to help you with the debugging.
2. To implement the overlap-save method.
   - You will be using the `lab8_filter` project.
   - It is well commented to help you in writing the whole code.
   - You may declare any extra buffers (arrays, essentially), if needed.
   - You have to use a 256 length FFT (as mentioned in the file `audio_setup.h`).
   - Decide the value of the constant `PINGPONGSIZE` declared in the same file.
   - To verify the working of your code, inside the file `lowpass.h`, first define the `COEFFS` array to be an impulse, i.e., set first coefficient to say, 3000, and all others to 0.
   - Now, if the external sine-wave, when input to the kit, is regenerated on the DSO by your code, then it is correct. Move to the next checkpoint. Otherwise, debug your code by single-stepping.
3. Find the cut-off frequency of a filter.
   - You will be using the same project as earlier.

- Now, use the `COEFFS` array as that given in the skeleton code.
- How will you determine the frequency response of the filter? (Hint: You have the signal generator at your disposal).
- Determine the cut-off frequency of the filter in discrete-time domain.