Lab 3: FIR Filter Implementation using Linear Buffering

EE 352 DSP Laboratory

Lab Goals

The purpose of this lab is to build the framework for real-time filtering of an incoming signal. We will use a linear buffer to implement an FIR filter in C as well as in assembly language and compare the performance of the two implementations. Also, we will analyze the speed of execution of the assembly language program with and without some powerful instructions (like `MAC`). For the comparison purpose, we will be using the profiling feature of the CCS. This lab session requires you to complete the following tasks.

- Implementation of convolution using linear buffering in C
- Compare the performances of the linear buffering implementations in c and assembly using profiling

1 Introduction

1.1 Convolution

The convolution operation which is the fundamental operation in realizing a finite impulse response (FIR) filter, is given by,

$$y[n] \stackrel{def}{=} \sum_{m=0}^{N-1} h[m]x[n-m]$$

Where $h[n]$ is the impulse response (N length FIR Filter) of the filter, $x[n]$ is the input signal and $y[n]$ is the filtered output signal. In non-real-time processing, the input and the impulse response samples are stored in the memory and the entire data is available for processing. But in real-time processing, we are interested in acquisition and processing of samples simultaneously, which means that the processing delay must be bounded even if the processing continues for an unlimited time. So it can be said that the mean processing time per sample should not be greater than the sampling period. In simpler words, the number of input samples entering the DSP in a particular time interval is same as the number of processed samples delivered as output in the same time interval.

1.2 Linear buffering

In this technique, we need a buffer whose length is greater than or equal to the length of impulse response (here we consider buffer length equal to impulse response length). Linear buffering can be explained using Figure 1. In the figure, memory locations are indicated by 1, 2, $\cdots$, 5. We can observe from the figure, that most recent sample will always be stored in memory location 1 (in this example). In order to accommodate this operation and avoid old data being overwritten, we need to first move the data in the memory location to next subsequent location ($4 \Rightarrow 5$, $3 \Rightarrow 4$, $2 \Rightarrow 3$, $1 \Rightarrow 2$) and then write the new data in location 1. Thus, this shifting will always start at the end of the buffer. Every memory location will successively pass its value to its successor. Finally, the new input is received in the first memory location. Shifting in memory is an expensive operation (in terms of CPU cycles) and considering that it has to be performed in addition to the processing, linear buffering becomes costly.

1.3 Learning checkpoint

The following task needs to be shown to your TA to get full credit for this lab session.

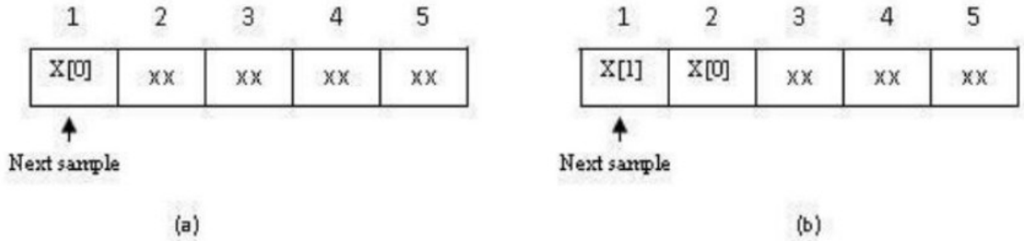


Figure 1: Linear buffer operation (a) new sample is put in position 1, (b) before the next sample comes in elements of the buffer have to be shifted by one step and next sample has to be accommodated in position 1.

You are provided with the skeleton code, in the file `main_lincbuff.c`. This file declares a function `linearbuff()`, that is supposed to perform linear buffering action. Read the following points carefully before starting out to code.

- The function `linearbuff` should neither receive any input argument nor it should return anything.
- It has access to the global pointers `inPtr`, `outPtr` and `coeff`. Observe that, the main function takes care that these pointers acquire addressess of correct variables before the function `linearbuff()` is called.
- You are allowed to use a buffer (essentially a standard array in C!) of the same size as the length of the impulse response, inside the function. Thus, if length of the impulse response is `IR_length`, then this buffer will hold `IR_length` values of input from the recent past. It is this buffer which you will perform the *linear buffering* action on.
- Note that, this buffer needs to hold its values between two different calls of the function. Is this possible? In general, the variables declared in any function are local to that parent function and get deleted as soon as the control reaches outside the function. Take help from your TA to understand this.
- After the function call in `main()` inside the infinite `while` loop, the variable `output` is directly assigned to the transmit data registers which means that `linearbuff` should store correct value in variable `output` while ending.

Your job is to use the mentioned pointers inside `linearbuff()` function to convolve the input with the impulse response to give one sample of the output every time the function is called.

2 Profiling

Profiling is a technique used to determine how long a processor spends in each section of a program. Profiling helps reduce the time it takes to identify and eliminate performance bottlenecks. The profiler analyzes program execution and shows where your program is spending its time. For example, a profile analysis can report how many cycles a particular function takes to execute and how often it is called. The data collected during profiling is displayed in `Profile Viewer Window`.

There are mainly two types of cycle counts,

1. **Inclusive counts**: The inclusive type counts the instruction cycles of the profiling area including the number of cycles taken by any subroutine that may be called by the profiling area.
2. **Exclusive counts**: The exclusive type counts the instruction cycles of the profiling area excluding the number of cycles taken by any subroutine that may be called by the profiling area.

Exclusive counts prove useful in situations when, the code in the profiling area uses many subroutines from pre-compiled libraries which are already optimized in terms of speed, memory usage and/or any other criteria (like power consumption). Their inclusion in profiling becomes redundant as they cannot be optimized further. Thus, exclusive counts, which are a result of skipping all the subroutines, point to the sections in *our own* code which may be sub-optimal in terms of speed.

Based on the above types, different types of cycle count are as follows: Exclusive max, Exclusive min, exclusive average, exclusive total, Inclusive max, Inclusive min, Inclusive average, Inclusive total etc.

2.1 Steps for profiling

Use the code you wrote in checkpoint 1.3 for this section.

2.1.1 Step 1

Start debug session and go to the Menu `Tools` ⇒ `Profile` ⇒ `Setup Profile Data Collection` as shown in figure 2.

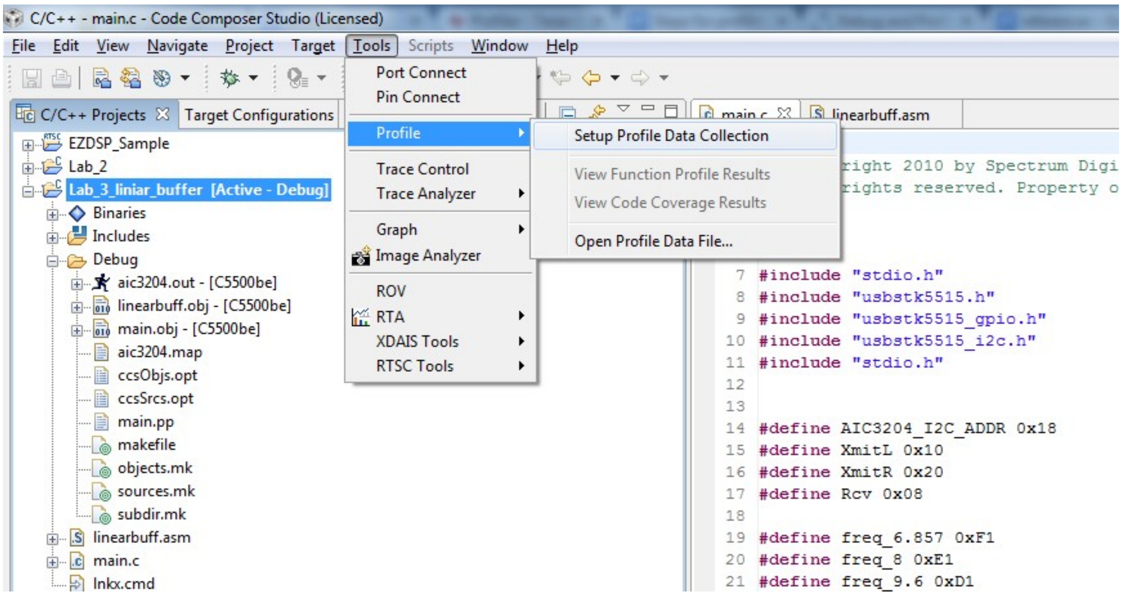


Figure 2: Step 1: Location of `Setup profile data collection` in the CCS Menu

2.1.2 Step 2

Check `Profile all Functions for CPU Cycles` option and save profiling setup as shown in the figure 3. Press `activate` button on bottom right corner before starting debugging program.

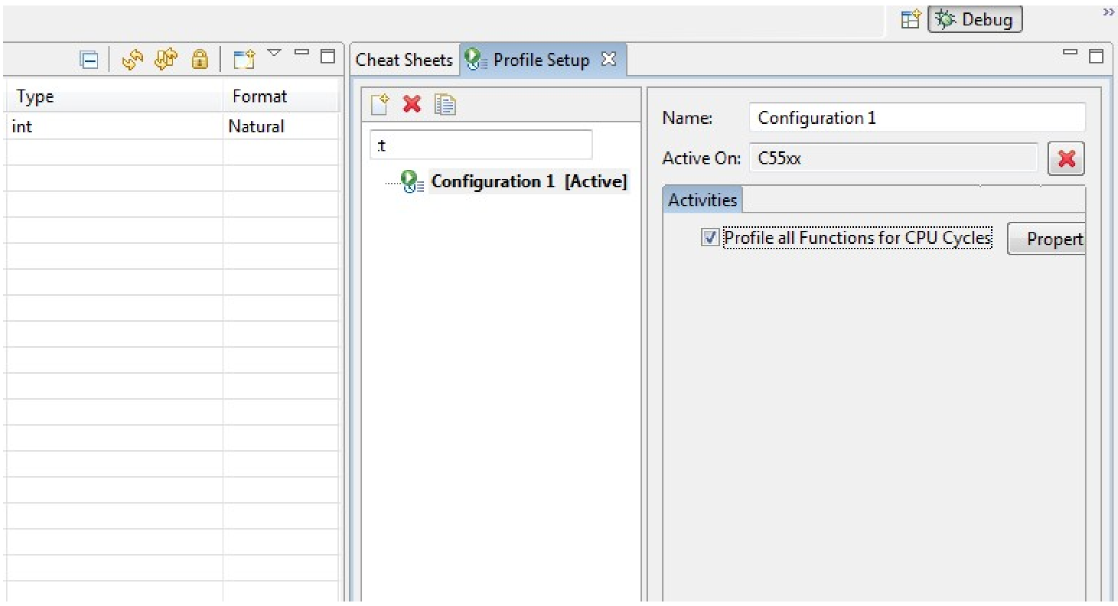


Figure 3: Step 2: Profile setup

2.1.3 Step 3

Run program on C5515 ezDSP kit. Use `while(count<100) { ··· }` instead of `while(1){ ··· }`. Use such a finite loop for all the profiling exercises you perform in this and next lab sessions. **Never use an infinite loop**. For profiling results, go to the Menu `Tools` ⇒ `Profile` ⇒ `View Function Profile Results` as shown in the figure 4.

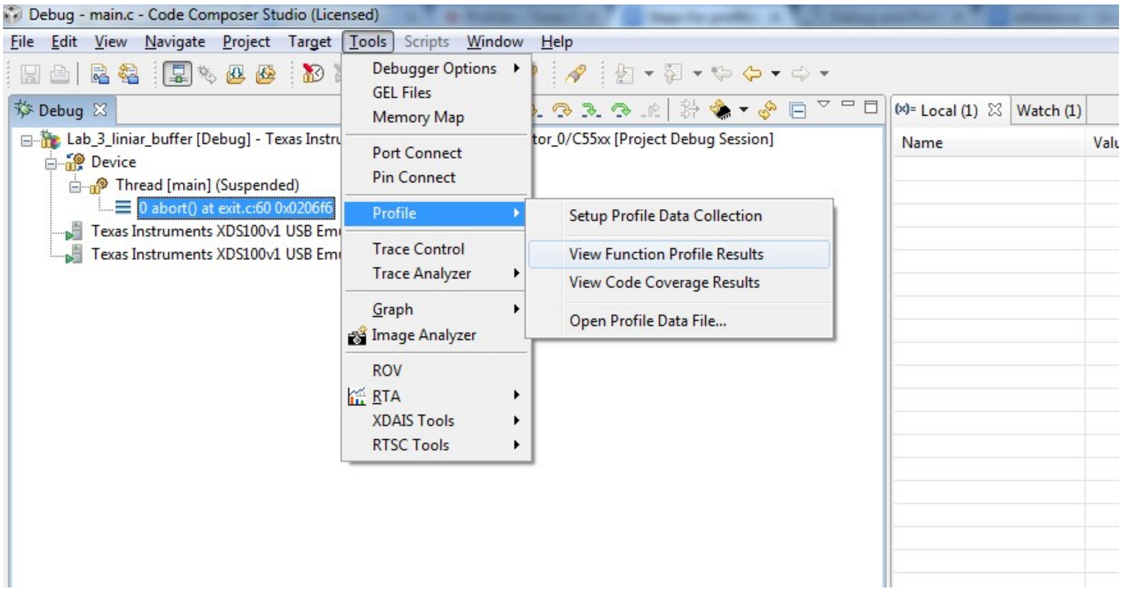
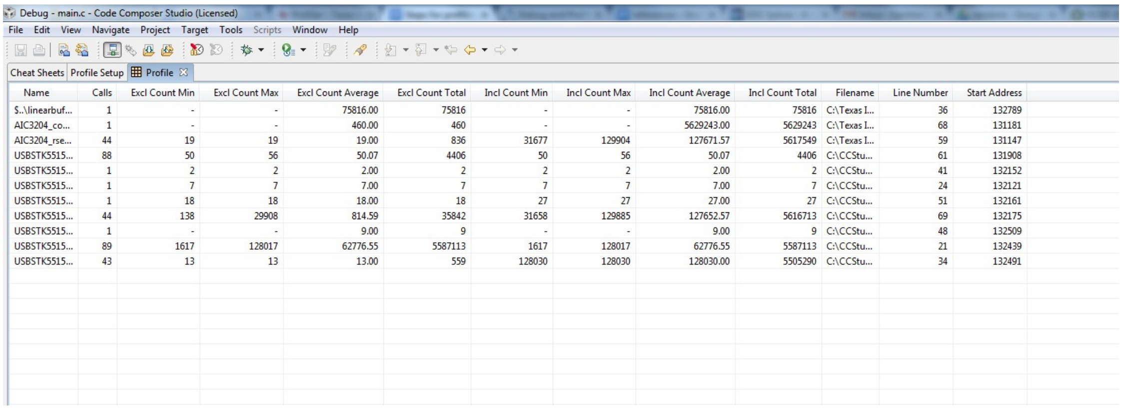


Figure 4: Step 3: Location of `View Function Profile Results` in the CCS Menu



Name	Calls	Excl Count Min	Excl Count Max	Excl Count Average	Excl Count Total	Incl Count Min	Incl Count Max	Incl Count Average	Incl Count Total	Filename	Line Number	Start Address
$..\linearbuf...	1	-	-	75816.00	75816	-	-	75816.00	75816	C:\Texas I...	36	132789
AIC3204_co...	1	-	-	460.00	460	-	-	5629243.00	5629243	C:\Texas I...	68	131181
AIC3204_rse...	44	19	19	19.00	836	31677	129904	127671.57	5617549	C:\Texas I...	59	131147
USBSTK5515...	88	50	56	50.07	4406	50	56	50.07	4406	C:\CCStu...	61	131908
USBSTK5515...	1	2	2	2.00	2	2	2	2.00	2	C:\CCStu...	41	132152
USBSTK5515...	1	7	7	7.00	7	7	7	7.00	7	C:\CCStu...	24	132121
USBSTK5515...	1	18	18	18.00	18	27	27	27.00	27	C:\CCStu...	51	132161
USBSTK5515...	44	138	29908	814.59	35842	31658	129885	127652.57	5616713	C:\CCStu...	69	132175
USBSTK5515...	1	-	-	9.00	9	-	-	9.00	9	C:\CCStu...	48	132509
USBSTK5515...	89	1617	128017	62776.55	5587113	1617	128017	62776.55	5587113	C:\CCStu...	21	132439
USBSTK5515...	43	13	13	13.00	559	128030	128030	128030.00	5505290	C:\CCStu...	34	132491

Figure 5: Profile results

Some important tips

- For more details, go to *http://processors.wiki.ti.com/index.php/Profiler*.
- The `calls` column should show the number of times a particular function is called throughout the project and not the number of times the control enters that function.
- If you are getting some garbage results (which may happen in first few attempts), go through section 2.1 once again carefully. Make sure that you have followed the exact procedure mentioned there. If you are not getting the correct output even after this, do the following. Remove any breakpoints, terminate the debug session, close the entire software and restart it.

2.2 Learning checkpoints

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

1. Use the project you used in checkpoint 1.3. Receive the input from function generator. Use a finite `while` loop and carry out profiling of all the functions. Obtain the **average exclusive counts** of all the functions defined in the project. Identify the function which is consuming bulk of the time for which the program runs. Also, note down the **average exclusive counts** value for the `linearbuff()` function.

2. You are provided with the files `main_checkpoint.c` and `linearbuff.asm`. Here, the linear buffering is carried out in assembly language. The conversation between the `main()` and the `linearbuff()` function takes place through pointers in the same way as that in the checkpoint 1.3. To understand the functioning of various assembly language instructions, make full use of the appendices provided at the end of the manual.

   (a) Perform the following tasks.

      i. Define your own set of inputs in `main_checkpoint.c`.

      ii. Put a breakpoint where the function `linearbuff()` appears and run the code. The execution halts at `linearbuff()`, (To add a break point, double click in the left side margin in the code window).

      iii. Now single step into the code (Use `F11`). You will go into the asm file.

      iv. Continue single stepping and each time, verify whether the operation intended by the instruction is being performed.

      Ex: suppose you are moving address of a memory location into a register, verify whether the action is being performed by: `View` ⇒ `Memory` ⇒ `(enter the address)`. To view registers, `View` ⇒ `Registers` ⇒ `CPU registers`.

   (b) Now, receive the input from function generator. Use a finite `while` loop and carry out profiling of all the functions. Note down the **average exclusive counts** value for the `linearbuff()` function.

3. Observe that, the linear buffering action written in the `linearbuff.asm` is just the translation of C code you implemented in checkpoint 1.3 into assembly language.

   (a) Now, understand the `MAC` instruction from appendix A and rewrite the code in `linearbuff.asm` using the `MAC` instruction. To get an idea about the various registers at your disposal, refer to the `registers.pdf` file uploaded on the course-wikipage.

   (b) Receive the input from function generator. Use a finite `while` loop and carry out profiling of all the functions. Note down the **average exclusive counts** value for the `linearbuff()` function.

4. Compare the **average exclusive counts** values from checkpoints 1, 2b and 3b from section 2.2. What do these values indicate? Explain this to your TA.

5. Read the audio input signal(s) given and process them through the filter you implemented. Listen to the unfiltered version (in PC and through the kit) and filtered version of the input signals. Can you recognise the difference ? What was the filter you used ? Is it a low-pass, high-pass, or band-pass filter ? Plot the frequency response of the filer in Matlab and convince yourself.

Appendices

A Useful instructions to implement linear and circular buffer action

- `MOV` **source, destination**

  Source ⇒ destination

  Where, "source" could be a value, auxiliary register, accumulator or a temporary register. "Destination" could be an auxiliary register, accumulator or a temporary register.

- `ADD` **source, destination**

  destination = destination + source.

  Where, "source" could be a value, auxiliary register, accumulator or a temporary register. "Destination" could be an auxiliary register, accumulator or a temporary register.

- `SUB` **source, destination**

  destination = destination - source.

- `MAC` **mul 1, mul 2,** `ACx`

  This single instruction performs both multiplication and accumulation operation. "x" can take values from 0 to 3.

  `ACx` = `ACx` + (mul_1 * mul_2);

  mul_1 and mul_2 could be an `ARn` register, temporary register or accumulator. (But both cannot be accumulators).

- `BSET ARnLC`

  Sets the bit `ARnLC`.

  The bit `ARnLC` determines whether register `ARn` is used for linear or circular addressing.

  `ARnLC`=0; Linear addressing.

  `ARnLC=1`; Circular addressing.

  By default it is linear addressing. "n" can take values from 0 to 7.

- `BCLR ARnLC`

  Clears the bit `ARnLC`.

- `RPT #count`

  The instruction following the `RPT` instruction is repeated "count+1" no of times.

- `RPTB label`

  Repeat a block of instructions. The number of times the block has to be repeated is stored in the register `BRC0`. Load the value "count-1" in the register `BRC0` to repeat the loop "count" number of times. The instructions after `RPTB` up to label constitute the block. The instruction syntax is as follows

  Load "count−1" in `BRC0`
  `RPTB` label
  ⋯ block of instructions⋯
  Label: last instruction

  The usage of the instruction is shown in the sample asm code.

- `RET`

  The instruction returns the control back to the calling subroutine. The program counter is loaded with the return address of the calling sub-routine. This instruction cannot be repeated.

B Important points regarding assembly language programming

- Give a tab before all the instructions while writing the assembly code.
- In Immediate addressing, numerical value itself is provided in the instruction and the immediate data operand is always specified in the instruction by a `#` followed by the number(ex: `#0011h`). But the same will not be true when referring to labels (label in your assembly code is nothing more than shorthand for the memory address, ex: `firbuff` in your sample codes data section). When we write `#firbuff` we are referring to memory address and not the value stored in the memory address.
- Usage of `dbl` in instruction `MOV dbl(*(#_inPtr)), XAR6`

  `inPtr` is a 32 bit pointer to an `Int16` which has to be moved into a 23 bit register. The work of `dbl` is to convert this 32 bit length address to 23 bit address. It puts bits `inPtr(32:16)` ⇒ `XAR6(22:16) and inPtr (15:0)` ⇒ `XAR6(15:0)`

  Example: In c code, the declaration `Int16 *inPtr` creates a 32 bit pointer `inPtr` to an `Int16` value. Then the statement `MOV dbl(*(#_inPtr)),XAR6` converts the 32 bit value of `inPtr` into 23 bit value. If `inPtr` is having a value 0x000008D8 then `XAR6` will have the value 0008D8 and `AR6` will have the value 08D8. So any variable which is pointed by `inPtr` will be stored in the memory location 08D8. We can directly access the value of variable pointed by `inPtr` by using `*AR6` in this case.
- If a register contains the address of a memory location, then to access the data from that memory location, `*` operator can be used.
- `MOV *AR1+, *AR2+`

  The above instruction will move "the contents pointed" by `AR1` to `AR2` and then increment contents in `AR1`, `AR2`.
- To view the contents of the registers, go to `view` ⇒ `registers` ⇒ `CPU register`.
- To view the contents of the memory, go to `view` ⇒ `memory` ⇒ enter the address or the name of the variable.

C Some assembly language directives

- `.global`: This directive makes the symbols global to the external functions.
- `.set`: This directive assigns the values to symbols. This type of symbols is known as *assembly time constants*. These symbols can then be used by source statements in the same manner as a numeric constant. Ex. `Symbol .set value`
- `.word`: This directive places one or more 16-bit integer values into consecutive words in the current memory section. This allows users to initialize memory with constants.
- `.space(expression)`: The `.space` directive advances the location counter by the number of bytes specified by the value of expression. The assembler fills the space with zeros.
- `.align`: The `.align` directive is accompanied by a number (X). This number (X) must be a power of 2. That is 2, 4, 8, 16, and so on. The directive allows you to enforce alignment of the instruction or data immediately after the directive, on a memory address that is a multiple of the value X. The extra space, between the previous instruction/data and the one after the `.align` directive, is padded with NULL instructions (or equivalent, such as `MOV EAX, EAX`) in the case of code segments, and NULLs in the case of data segments.