Lab 2: FIR Filter Implementation using a Linear and Circular Buffer

EE 462 DSP Laboratory (Spring 2023)

Lab Goals

The purpose of this lab is to build the framework for real-time filtering of an incoming signal in an efficient manner. We will first use a linear buffer to implement a 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 using some powerful instructions (like `MAC`). For comparison, we will be using the profiling feature of the CCS. Later we will implement FIR filter using circular buffer and compare the circular buffer implementation with that of linear buffer. This lab session requires you to complete the following tasks.

- Implementation of convolution using linear and circular buffer using assembly level programming
- Compare the performances of the convolution implementations in C and assembly using profiling
- Compare the performances of linear and circular buffer implementations.

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 entire input data and the impulse response samples are stored in the memory, and the processing is carried out on the stored data. But in real-time processing, we are interested in acquisition and processing of samples simultaneously, which means that the processing delay per input sample must be bounded even if the processing continues for an unlimited time. So it can be said that the maximum processing time should not be greater than the sampling period. More generally, 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 Buffer

Here, we will use a buffer, whose length is equal to the length of impulse response, which will store the input data samples required to generate the filtered output sample at a particular time instant.

The linear buffer usage is explained using Figure 1. Here, memory locations are indicated by 0, 1, $\cdots$, 4. We can observe from the figure, that the most recent input sample will always be stored in memory location 0. In order to avoid overwriting the required previous data stored at the same location, we need to first move the data in each memory location to the next subsequent location ($3 \Rightarrow 4$, $2 \Rightarrow 3$, $1 \Rightarrow 2$, $0 \Rightarrow 1$) and then write the new data in location 0. In this process, the data sample in memory location 4 is discarded. But 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, this process becomes costly.

1.3 Circular Buffer

Here we will use a buffer, which we shall call a circular buffer, whose length is equal to filter impulse response length. Figure 2 illustrates a length three circular buffer with three contiguous memory locations, 0001 to 0003.

Figure 2(a) shows the three contiguous memory locations just before first input arrives. Figure 2(b) shows the change after the first sample comes in. The idea of the circular buffer is that the end of this buffer is connected to its beginning i.e, the memory location 0001 is viewed as being next to 0003. We keep a track of the buffer

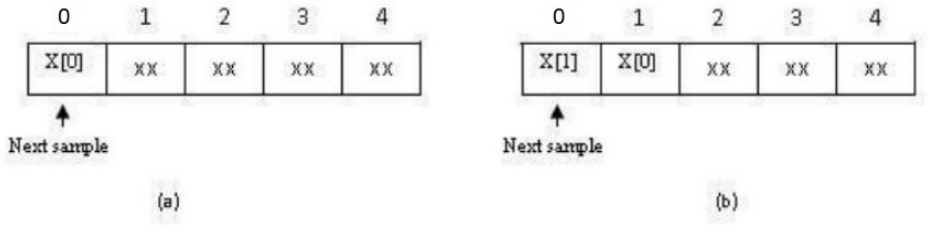

Figure 1: Linear buffer (a) new sample is put in position 1, (b) before the next sample comes, elements of the buffer have to be shifted by one step and the latest sample needs to be accommodated in position 1.

by a pointer (a variable whose value is an address) that points to the oldest input sample stored in the buffer. When the new input sample is received, it replaces the oldest sample in the array, and the pointer is moved one address ahead. For instance, in Figure 2(a) the pointer contains the address 0001, which corresponds to the oldest data, while in Figure 2(b) it contains 0002, and so on. In Figure 2(d) we see that the pointer again points back to the memory location 0001 (which contain the oldest sample). We can see that only one value (i.e the pointer) needs to be changed when a new sample is acquired as opposed to linear buffer where old samples have to be shifted every time a new sample comes in.

Four parameters are needed to manage a circular buffer.

- Length of the buffer (e.g. 3).
- Start address of the circular buffer in memory (in this example, 0001).
- The step size of the memory addressing. In Figure 2 the step size is one, for example: address 0001 contains one sample, address 0002 contains the next sample, and so on. This may not the case every time. For instance, each sample may require two or four bytes to hold its value. In these cases, the step size will be two or four, respectively.
- A pointer to the oldest sample.

The first three values define the size and configuration of the circular buffer, and will not change during the program operation whereas the pointer to the oldest sample, must be modified as each new sample is acquired. The C5515/C5535 DSP is equipped with auxiliary registers `(AR0-7)` to support the circular buffer configuration.

1.4 Learning checkpoint

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

You are provided with a working C code, in the file `main_lincbuff.c` which implements convolution of input data with the impulse response. This file declares a function `linearbuff()`, that is supposed to perform convolution using linear buffer to generate one output sample for each new input sample. This is achieved using two stages: (1) Shifting of the previous input samples in the linear buffer and placing the new input sample at the first position of the buffer, (2) carrying out multiplication and accumulation to generate the output sample. The function `linearbuff` 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.

- Create a new project and use the `main_lincbuff.c` file in it. Run the code on the board and verify the output with pen and paper calculation. You need to use the watch window (You used it in Lab1) to see the output in CCS. Show this to your TA.

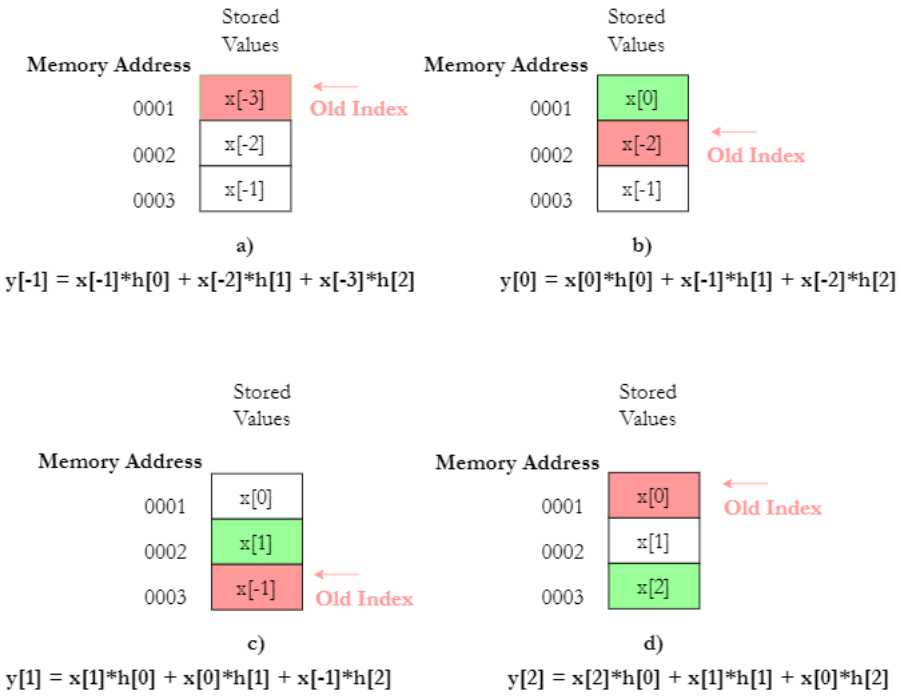

Figure 2: Circular buffer operation (a) Buffer state just before the new sample comes in. Location 0001 points to the oldest sample. (b) Most recent sample x[0] is put in 0001 and Old Index pointer is updated to 0002. (c) Circular buffer when x[1] comes in, oldest state pointer updated. (d) When x[2] comes in, it is put in location 0003 and Old Index pointer is reset to 0001.

- Now comment the code for synthetic input and un-comment the `while(1)` code to take input from the function generator and display the output on the DSO. Change input frequency from the function generator and observe the output waveform. If you are getting saturated output and want to reduce the input signal levels you can use 40 dB attenuation on the function generator. The steps to be followed if you are not getting any output are as follow:
  1. Check if the stereo in and stereo out connections are correct.
  2. Check if the cables are working properly using DSO
  3. Check if the input is available to the board correctly using the graph window
  4. Check that the output levels are atleast above 10000 in the CCS graph window for the output. This can be done using again the graph window. If not then try multiplying the output with a small multiplicative factor to bring the amplitude to around 20000.
  5. If still not getting output call your TA/RA

2 Profiling

Profiling is a technique used to determine how long a processor takes to execute each section of a program. This helps to identify and eliminate performance bottlenecks if present in any section of the program. For example, a profile analysis can report how many cycles a particular function takes to execute and how often it is called. This can be used to understand whether a function is taking unexpectedly long execution time.

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 uses many subroutines from pre-compiled libraries which are already optimized in terms of speed, memory usage and/or any other criteria (like power consump-

tion). 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.

2.1 Steps for profiling

Use the code you wrote in checkpoint 1.4 for this section.

2.2 Step 1

Use such a finite loop for all the profiling exercises you perform in this and next lab sessions. **Never use an infinite loop**. Use `while(count<100)` { ··· } instead of `while(1)`{ ··· }.

2.2.1 Step 2

Start debug session and go to the Menu `Tools` ⇒ `Profile` ⇒ `Setup Profile Data Collection` as shown in Figure 3.

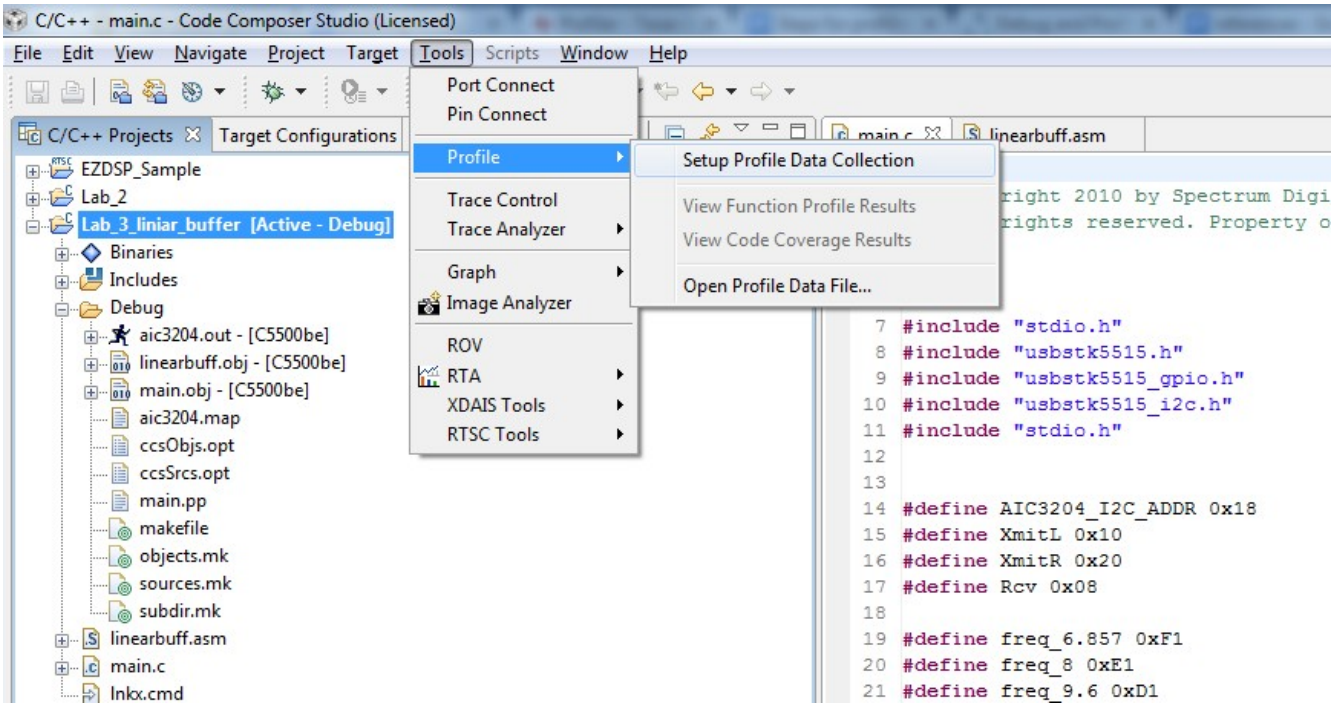

Figure 3: Location of `Setup profile data collection` in the CCS Menu

2.2.2 Step 3

Press `activate` button on bottom right corner before starting debugging program. Check `Profile all Functions for CPU Cycles` option and save profiling setup as shown in Figure 4.

2.2.3 Step 4

Click on the restart button as shown in Figure 5.

2.2.4 Step 5

Run the program on C5515 ezDSP kit. Wait for the profiling to finish, it takes some time.

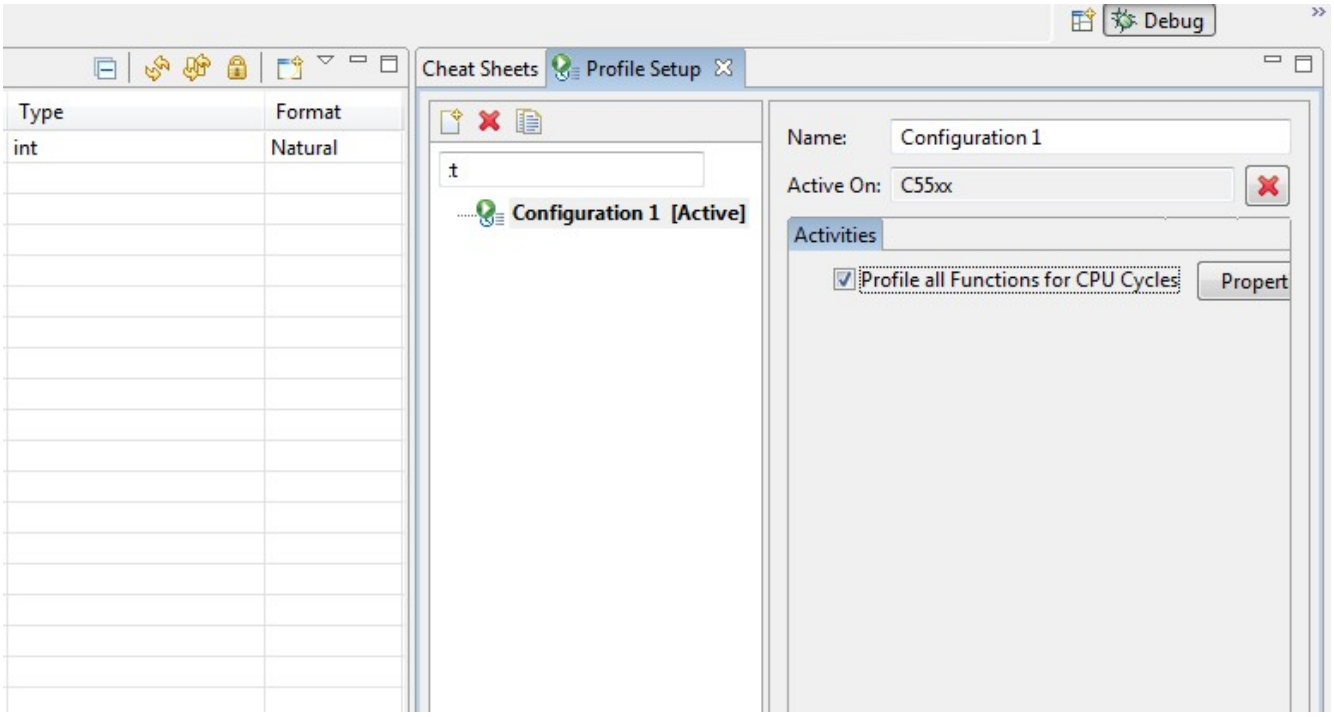

Figure 4: Profile setup

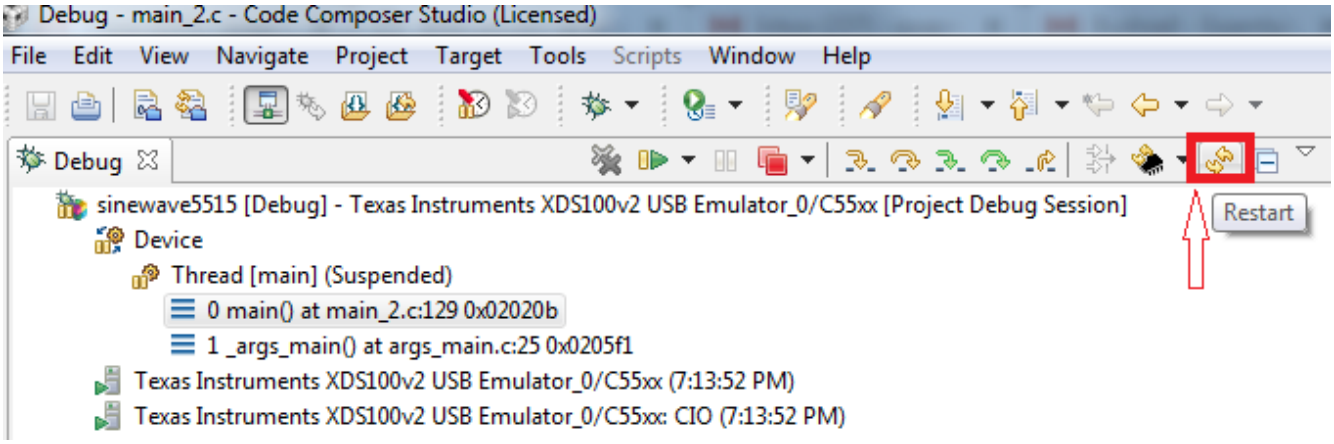

Figure 5: Restart option in CCS debug window

2.2.5 Step 6

To view profiling results, go to the Menu `Tools` ⇒ `Profile` ⇒ `View Function Profile Results` as shown in the figure 6. The results appear in the Profile window as shown in Figure 8. The `calls` column should show the number of times a particular function is called throughout the project.

2.3 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 1.4 (i.e. **main_lincbuff.c**) and use **synthetic input** (for more accurate comparison) to 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`. You need to exclude previous files from project by right clicking on them and selecting **exclude from build** option. Here, the linear convolution is carried out in assembly language. The interaction between the `main()` and the `linearbuff()` function takes place through pointers in the same way as that in the checkpoint 1.4. To understand the functioning of various assembly language instructions, make full use of the Appendices

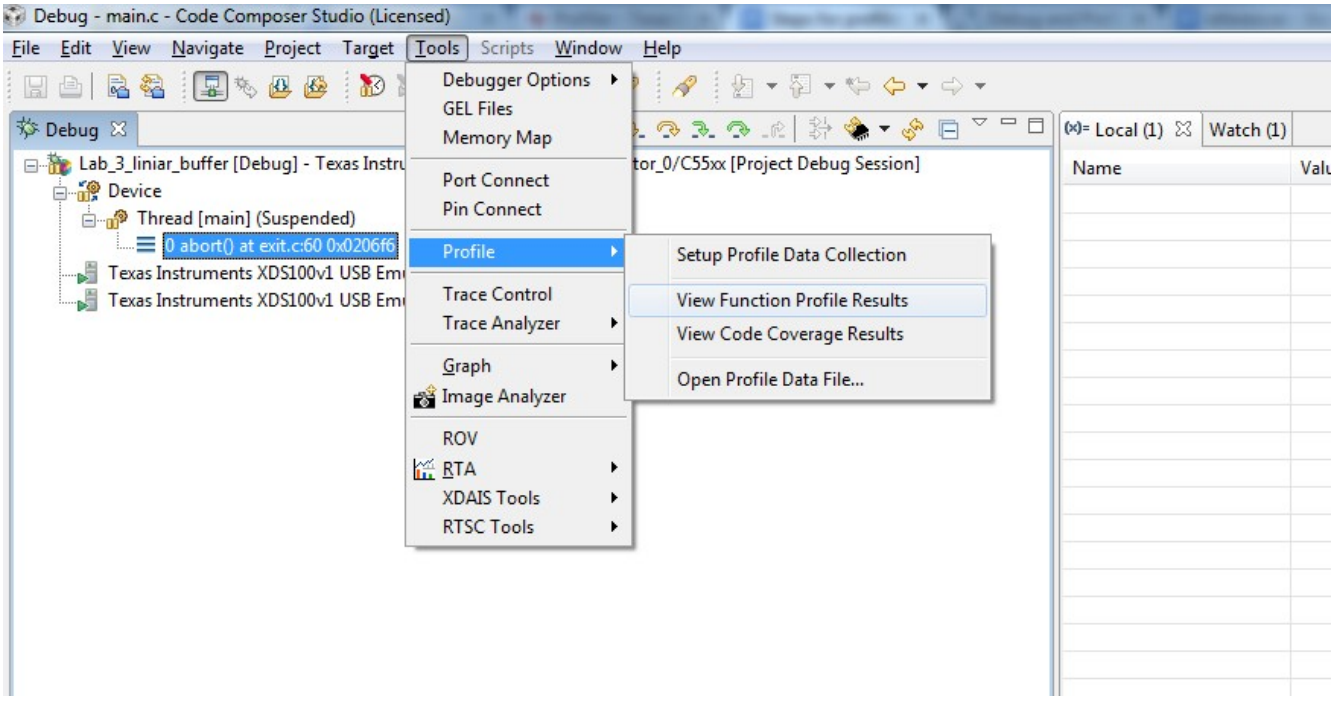

Figure 6: Location of `View Function Profile Results` in the CCS Menu

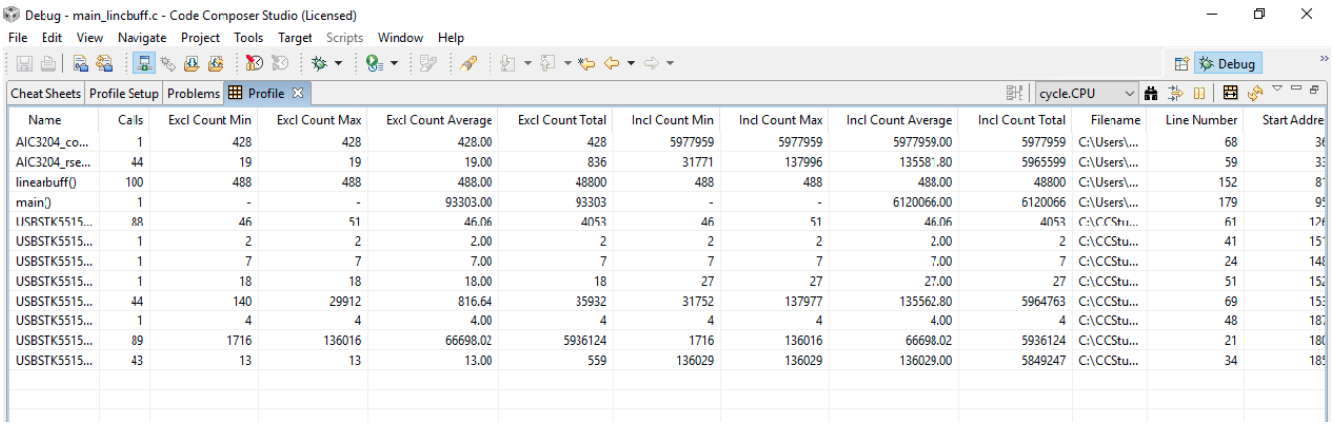

Figure 7: Profile results

provided at the end of the manual.

(a) Go through the following steps to understand code properly.

i. Synthetic input array **x[n] = {1,2,3,4,5,6,7,8,9}** has been defined in `main_checkpoint.c`.

ii. Put a breakpoint where the function `linearbuff()` appears and run the code. The execution halts at `linearbuff()`.

iii. Now single step into the code (Use `F5`). 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, carry out profiling of all the functions. Note down the **average exclusive counts** value for the `linearbuff()` function.

3. **Observe** that, the linear convolution code written in the `linearbuff.asm` is just the translation of C code you observed in checkpoint 1.4, into assembly language.

(a) Now, understand the `MAC` instruction from appendix A and modify the code in `linearbuff.asm` to use the `MAC` instruction. To get an idea about the various registers at your disposal, refer to the `registers.pdf` file given with the material.

(b) Now, carry out profiling of all the functions as you did above . Note down the **average exclusive counts** value for the `linearbuff()` function.

4. Change the FIR filter to have the following impulse response, i.e.,

$$h[n] = \{1, -1, 1, -1, 1, -1, 1, -1, 1, -1, 1, -1\} \tag{1}$$

Set the sampling rate to 12 kHz. Now obtain its magnitude frequency response experimentally by observing the output on DSO for different input sinusoids in the range ($400Hz$ to $4000Hz$) from the function generator. Also comment whether it is a low-pass, high-pass or a band-pass filter. **Note** - If you observe the output as in Figure 8 on DSO, this means that saturation has occurred, i.e., the convolution output value exceeds the maximum allowable range for that particular datatype. To resolve this, reduce the input voltage from function generator. If this also does not resolve the issue, look at the input in the graph window and try to estimate the convolution output looking at the waveform samples. If its more than what could be stored in 16 bit signed, try dividing the input.

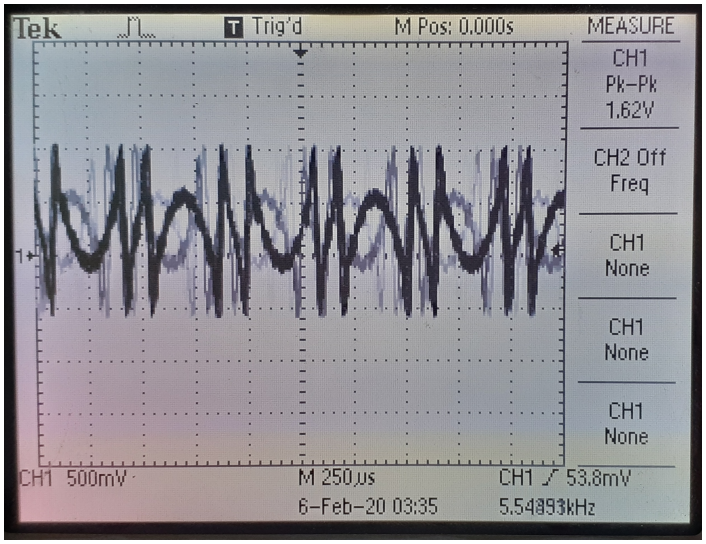

Figure 8: Distorted Output in DSO due to Saturation

2.4 Circular buffer configuration

You will be using the `circbuff_config.c` and `circbuffconfig.asm` files.
To understand different registers used for circular configuration, refer `circular_addressing.pdf`.

The following code example demonstrates initializing and then accessing a circular buffer.

Example code to initialize and access a C55x Circular Buffer:

```
MOV #3, BK03         ;Circular buffer size is 3 words
BSET AR1LC           ;AR1 is configured to be modified circularly
AMOV #010000h, XAR1 ;Circular buffer is in main data page 01
MOV #0A02h, BSA01    ;Circular buffer start address is 010A02h
MOV #0000h, AR1      ;Index (in AR1) is 0000h
MOV *AR1+, AC0       ;AC0 loaded from 010A02h + (AR1) = 010A02h and then AR1 = 0001h
MOV *AR1+, AC0       ;AC0 loaded from 010A02h + (AR1) = 010A03h and then AR1 = 0002h
MOV *AR1+, AC0       ;AC0 loaded from 010A02h + (AR1) = 010A04h and then AR1 = 0000h
MOV *AR1+, AC0       ;AC0 loaded from 010A02h + (AR1) = 010A02h and then AR1 = 0001h
```

This code configures a circular buffer with start address formed by adding contents of XAR1 and BSA01. The index to this circular buffer is AR1. So when we increment the value of AR1 beyond the length specified by

BK03, it gets initialized to 0.
In order to enable the circular buffer feature of the DSP, following steps need to be followed:

- Initialize the appropriate buffer size register (BK03, BK47, or BKC). For example, for a buffer of size 3, load the BK register with 3.
- Each auxiliary register ARn has its own linear/circular configuration bit in ST2_55. Initialize the appropriate configuration bit in ST2_55 (For example, BSET AR1LC).
- Initialize the appropriate extended register (XARn or XCDP) to select a main data page (the 7 most significant bits specify the page). For example, if auxiliary register 3 (AR3) is the circular pointer, load extended auxiliary register 3 (XAR3). If CDP is the circular pointer, load XCDP.
- Initialize the appropriate buffer start address register (BSA01, BSA23, BSA45, BSA67, or BSAC). The main data page, in XARn(22-16) or XCDP(22-16), concatenated with the content of the BSA register defines the 23-bit start address of the buffer.
- Load the selected pointer, ARn or CDP, with a value from 0 to (buffer size - 1). For example, if above example we are using AR1 and the buffer size is 3, load AR1 with a value less than or equal to 3.

2.5 Learning checkpoint

The aim of this checkpoint is to configure a Circular Buffer and check its operation. You will use the `circbuff_config.c` and `circbuffconfig.asm` files for this checkpoint.

- The `circbuff_config.c` calls `circbuffconfig()` function repeatedly. The `circbuffconfig()` definition can be found in `circbuffconfig.asm` file.
- In the asm file, observe that the `ADD` instruction is configured to repeat 3 $\times$ `buff_length` times. This takes care of addition. But as `AR1` is always increasing, it will eventually cross the boundary of the buffer and add 5 to the memory locations beyond.
- But we need updates in the `firbuff` buffer to be as shown in Table 1.

Iteration	`firbuff`	`firbuff+1`	`firbuff+2`	`firbuff+3`	`firbuff+4`
0^{th}	0	0	0	0	0
1^{st}	5	0	0	0	0
2^{nd}	5	5	0	0	0
3^{rd}	5	5	5	0	0
4^{th}	5	5	5	5	0
5^{th}	5	5	5	5	5
6^{th}	10	5	5	5	5
7^{th}	10	10	5	5	5
8^{th}	10	10	10	5	5
9^{th}	10	10	10	10	5
10^{th}	10	10	10	10	10
11^{th}	15	10	10	10	10
12^{th}	15	15	10	10	10
13^{th}	15	15	15	10	10
14^{th}	15	15	15	15	10
15^{th}	15	15	15	15	15

Table 1: State changes for circular buffer example

- Referring the example code discussed in the previous section, configure a circular buffer using `AR1` to circularly rotate over the `firbuff` buffer which will then check if the memory location is a part of the buffer, if not, then wrap around to the start of the buffer.
- Debug the code and obtain start address and data page of the `firbuff` buffer with the help of `Memory` view in CCS Debug mode and make necessary changes in the configuration code in the `circbuffconfig.asm` file and again build it.

- Put a breakpoint before the `ADD` instruction in the asm file. With the help of `Memory` view in CCS, observe the values stored in the `firbuff` buffer every time the code hits the breakpoint.
- Get this task verified by the TA/RA by showing him/her, the circular addressing by the `AR1` register and the increment of values in the `firbuff` buffer through single-stepping.

2.6 Learning checkpoint

In this checkpoint you will implement convolution using the circular buffer. You will be using the `main_test.c`, `main_circbuff.c`, `circbuffconfig.asm` and `asmcircbuff.asm` files for this checkpoint.

- Add the files `main_test.c` and `asmcircbuff.asm` into your project. The set-up is similar to the FIR convolution using linear buffer you performed in assembly language. Synthetic inputs and FIR filter coefficients are defined in the arrays `sytheticInput` and `coeffs` respectively in the `main_test.c` file.
- In the `circbuffconfig.asm` remove the two lines before `RET` i.e the lines adding `5 to *AR1`. Note: You might again need to check the address of the circular buffer using memory view and make the required changes to your assembly code.
- Observe the usage of the variable `oldindex`. Try to understand this from the comments in the program and the appendices.
- Complete the code to convolve the input and the impulse response by using the appropriate registers pointing to them in the assembly language program. **You must use the `MAC` instruction in the convolution implementation**.
- Verify the working of the code for synthetic input with your TA/RA by stepping through the code . Values of the `output` variable or `recent_output` array must match the values obtained by manual calculations.
- Compare Average Exclusive count of the circular and linear buffer implementations of the FIR convolution by profiling the functions independently. Which one is faster and why ? Justify.
- Now just replace the file `main_test.c` with `main_circbuff.c` (exclude the file `main_test.c` by right clicking the file and using the option "exclude from build" and add `main_circbuff.c` to the current project).
- Verify the working of the code with your TA/RA by providing sine waves of different frequencies from function generator to the kit and observing the output on DSO.

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.

- `MPY` **source 1, source 2, destination**

  destination = source 1 * source 2.

- `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.