Lab 4: FIR Filter Implementation using Circular Buffering

EE 352 DSP Laboratory (Spring 2019)

Lab Goals

The purpose of this lab is to continue building the framework for real-time filtering of an incoming signal. We will use a more efficient technique known as circular buffering to implement an FIR filter and compare its performance with that of the linear buffer implementation. This lab session requires you to complete the following tasks.

- Configuring the DSP to use the inbuilt circular buffer
- Implement convolution using circular buffering technique
- Compare the performance of the linear and circular buffering techniques using profiling
- Designing an FIR filter using Matlab's *fdatool* and implementing it in real time on DSP.

1 Introduction

In lab-3, we implemented an FIR filter using linear buffering technique wherein old input samples in the input buffer had to be shifted by one memory location every time a new incoming input sample arrived. This process can become expensive when the filter/input buffer size is quite large. In this lab, we will be learning an efficient technique i.e circular buffering, to implement an FIR filter.

1.1 Circular 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). Figure 1 illustrates a length five circular buffer. Similar to the linear buffering case, five contiguous memory locations, 0001 to 0005, are assigned for the buffer. Figure 1(a) shows the five contiguous memory locations just before the first input arrives, Figure 1(b) shows the change after the first sample comes in. The idea of circular buffering is that the end of this linear array is connected to its beginning. The memory location 0001 is viewed as being next to 0005, just as 0004 is next to 0005.

We keep track of the array by a pointer (a variable whose value is an address) that indicates where the next sample has to be put. When a new sample is acquired, it replaces the oldest sample in the array, and the pointer is moved one address ahead. For instance, in Figure 1(a) the pointer contains the address 0001, which corresponds to the oldest data, while in Figure 1(b) it contains 0002, similarly, Figure 1(c). In Figure 1(d) we can see how the pointer again points back to the start of the buffer (oldest sample). Circular buffers are efficient because 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.

- A pointer that indicates the start of the circular buffer in memory (in this example, 0001).
- A pointer indicating the end of the array (e.g., 0005), or a variable that holds its length (e.g., 5).
- The step size of the memory addressing must be specified. 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 is frequently not the case. For instance, the addressing may refer to bytes, and each sample may require two or four bytes to hold its value. In these cases, the step size would need to be two or four, respectively.
- A pointer to the oldest sample.

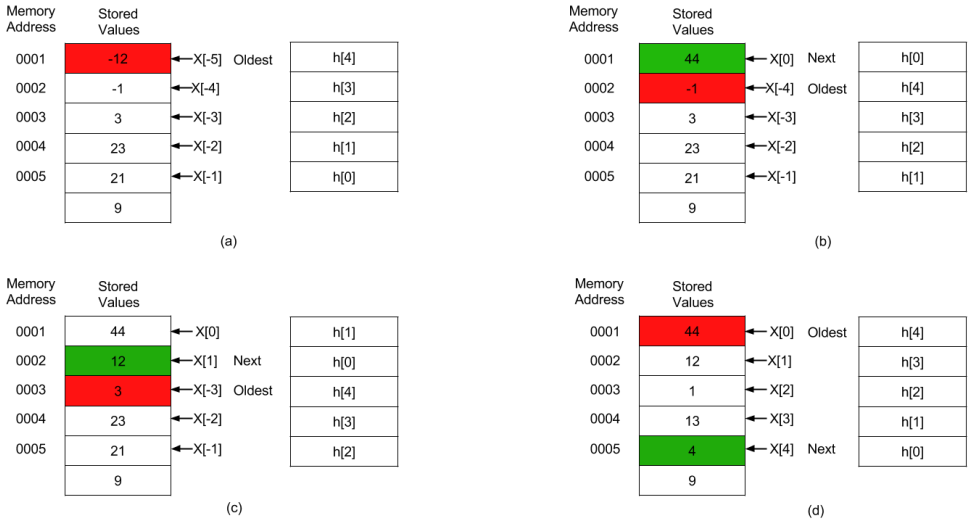

Figure 1: Circular buffer operation (a) Buffer state just before the new sample comes in. Location 0001 points to the oldest sample. (b) Most recent sample x[0] is put in 0001 and oldest data pointer is updated to 0002. (c)Circular buffer state when x[1] comes in, oldest state pointer updated. (d)When x[4] comes in it is put in location 0005 and oldest sample pointer is reset to 0001.

The first three values define the size and configuration of the circular buffer, and will not change during the program operation. The fourth value, the pointer to the oldest sample, must be modified as each new sample is acquired. So there must be program logic that controls how this fourth value is updated based on the first three values. DSPs are equipped with hardware for modulo addressing, which allows for this circular buffer logic to be easily implemented. The C5515 DSP is equipped with the registers `(AR0-7)` to support the circular buffer action.

***Note**: Refer to the* `circular buffer implementation.pdf` *file uploaded on the course wikipage. It shows how to enable the circular buffer feature of the DSP. Here you can see how a register (pointer to a data location) is configured to reset after it reaches the end of buffer.*

1.2 Learning checkpoint

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

You will be using the `circbuff_config.c` and `circbuffconfig.asm` files for this checkpoint. Read the following points carefully before setting out to code.

- The `circbuff_config.c` does nothing but calls the `circbuffconfig()` function repeatedly.
- The function `circbuffconfig()` gets the memory space named `firbuff` to work with. With the help of `Memory` view in CCS, note down the `buff_length` number of values present at the memory locations in the buffer `firbuff`.
- We want the state changes shown in Table 1 to occur. You can neither change the `RPT` instruction already present in the skeleton code in `circbuffconfig.asm` nor the instruction it repeats.
- From table 1, it is obvious that we need a pointer to the start of the buffer `firbuff` which would be configured to
  - (i) add 5 to the memory location it is currently pointing to,
  - (ii) increment (i.e., point to the next memory location),
  - (iii) check if the memory location is a part of the memory buffer, if not, then wrap around to the start of the buffer, else stay where it is,
  - (iv) go to step (i)
- We will be using the `AR1` auxiliary register for this purpose.

Iteration	firbuff	firbuff+1	firbuff+2	firbuff+3	firbuff+4
0	0	0	0	0	0
1	5	0	0	0	0
2	5	5	0	0	0
3	5	5	5	0	0
4	5	5	5	5	0
5	5	5	5	5	5
6	10	5	5	5	5
7	10	10	5	5	5
8	10	10	10	5	5
9	10	10	10	10	5
10	10	10	10	10	10
11	15	10	10	10	10
12	15	15	10	10	10
13	15	15	15	10	10
14	15	15	15	15	10
15	15	15	15	15	15

Table 1: State changes for circular buffer example

- Observe the `ADD` instruction which is configured to repeat 3×`buff_length` times. It takes care of addition (obviously!), but since `AR1` is always increasing, it will eventually cross the boundary of the buffer and add 5 to the memory locations beyond.
- Your job is to configure `AR1` to be circularly rotating over the buffer so that if incremented at the end, it wraps around to the start of the buffer. Refer to our demands from the pointer (`AR1` here) mentioned in point 1.2 while setting out to code.
- Refer to `circular buffer implementation.pdf` file uploaded on the course wikipage and the appendices for information on how to achieve this. Make sure that you do not include any unwanted instruction in the code.
- Get this task verified by the TA/RA by showing him/her, the circular addresssing by the `AR1` register and the incrementation of values in the `firbuff` buffer through single-stepping.

1.3 Learning checkpoint

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

You will be using the `main_test.c` and `asmcircbuff.asm` files for this checkpoint. Read the following points carefully before setting out to code.

- The set-up is very much similar to the FIR convolution using linear buffering you performed in assembly language in the previous lab session. Synthetic inputs and FIR filter coefficients are defined in the arrays `sytheticInput` and `coeffs` respectively in the `main_test.c` file.
- Copy your circular buffer configuration code from checkpoint 1.2 at appropriate place in the `asmcircbuff.asm` file.
- Observe the usage of the variable `oldindex`. Try to understand what it is meant for 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 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.

1.4 Learning checkpoint

All the following tasks need to be shown to your TA to get full credit for this lab session. You will be using the `main_circbuff.c` and `asmcircbuff.asm` files for this checkpoint. Read the following points carefully before setting out to code.

- Now just replace the 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. Compare the frequency response with the one obtained using *freqz* command in MATLAB and comment.
- Compare the circular and linear buffer (lab 3) implementations of the FIR convolution by profiling the functions independently. Which one is faster and why ? Can you justify ?

1.5 Learning checkpoint/exercise

All the following tasks(***at ONCE***) need to be shown to your TA to get full credit for this lab session.

- First, you need to design an FIR filter using MATLAB's `fdatool` having filter specifications as follows:
  `wpass = 0.05 and wstop = 0.5 (normalised to 1)`
- To get started, type `fdatool` in the Matlab's command window. A Filter design and Analysis Tool GUI will appear. Explore the GUI and try to understand various functionalities provided by the tool.
- For this checkpoint, select '**Equiripple**' option in '**FIR Design Method**' menu, '**Minimum order**' option in '**Filter Order**' menu and '**Normalised (0 to 1)**' option in '**Frequency Specifications Units**'. Then, click on '**Design Filter**'.
- Generate a C header file containing the filter coefficients by going to `Targets` $\Rightarrow$ `Generate C header`. Specify the Data type as 'Signed 16-bit integer'. Include the generated header file in your project folder and use the coefficients in the header file for performing the filtering operation.
- Determine the cutoff frequency by providing sine waves of different frequencies from function generator to the kit and observing the output on DSO. Is it matching with the specifications? Comment.

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.