Lab 5: IIR Filter Implementation and its comparison with FIR

EE 352 : Digital Signal Processing Laboratory (Spring 2019)

Lab Goals

In this lab, we will learn about infinite impulse response (IIR) filter and techniques to implement it on DSP processor. We will also learn about its advantages and drawbacks as compared to finite impulse response (FIR) filter.

- Direct form I implementation of an IIR filter
- FIR vs IIR implementation of a system

1 Introduction

Any discrete-time LTI systems can be described using a difference equation (eq 1) that defines how the output signal is related to the input signal.

$$\sum_{i=-\infty}^{\infty} a_i y[n-i] = \sum_{j=-\infty}^{\infty} b_j x[n-j] \tag{1}$$

where a_i and b_j are constants, characteristic of the particular system, $x[n]$ is the input signal and $y[n]$ is the filtered output signal. We shall consider only *causal* systems, which means both the above summations will operate over the range 0 to ∞ rather than $-\infty$ to ∞.

2 Infinite Impulse Response (IIR) Filters

Consider a special case of eq 1:

- $a_i = 0$ for $M < i < 0$
- $b_j = 0$ for $N < j < 0$
- $a_0 = 1$, and $a_i \neq 0$ for at least one i, $i = 1, \ldots, M$

where M and N are natural numbers. We can rewrite eq 1 as follows:

$$\sum_{i=0}^{M} a_i y[n-i] = \sum_{j=-0}^{N} b_j x[n-j] \tag{2}$$

The transfer function $H(z)$ obtained by applying Z transform on both sides of equation 2 is as follows:

$$H(z) = \frac{\sum_{j=0}^{N} b_j z^{-j}}{\sum_{i=0}^{M} a_i z^{-i}} \tag{3}$$

Time domain impulse response h[n] obtained by applying inverse Z transform to equation 3 is of infinitely long duration. Such a filter is therefore called an Infinite Impulse Response (IIR) filter.

The techniques studied in the previous labs to implement a FIR digital filter are not applicable to IIR filter for obvious reasons. Thus, we need to explore other techniques to implement IIR filters.

As discussed above, an IIR filter can be described using a difference equation, which can be exploited to implement the filter by defining two buffers of lengths $M+1$ and $N+1$, for holding past values of output and input, respectively. Thereby, performing finite multiplication and addition operations, an IIR filter can be easily realized.

The difference equation can be rearranged in different ways thereby obtaining different structures. One such class of structures referred to as direct form structures is introduced in this lab. Other known forms are cascaded sections, parallel sections, lattice structures and state-space structures.

2.1 Direct-form I IIR implementation

Algorithm for implementing an IIR filter using direct-from I structure, defined by (2), on a DSP processor can be written as follows:

1. Allocate memory space to two buffers of sizes $M+1$ and $N+1$ and initialize them with zeros or initial conditions, if known.
2. Start the acquisition of data.
3. Go on calculating the values of the output according to (2) and storing them in the output buffer, one after the other.
4. Simultaneously, keep storing the values of input in the input buffer.
5. Feed the output display unit (the DSP codec combined with the DSO in our case) with every calculated output sample.
6. If any of the buffers runs out of memory space, start overwriting the oldest values with the latest values.
7. Continue steps 3 to 6 till the input data-flow continues.

2.1.1 Learning checkpoint

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

Consider the IIR filter given by equation 4.

$$H_1(z) = \frac{49 + 79z^{-1} + 49z^{-2}}{100 + 50z^{-1} + 40z^{-2}} \tag{4}$$

1. Draw the pole-zero plot of this filter and get it verified by your TA/RA.
2. Using the `freqz` function provided by Matlab, plot the frequency response of the filter and identify the type of the filter (Lowpass / Highpass / Bandpass / Notch) and note down the cutoff frequency.
3. Use `main.c` file provided with this handout for this exercise. You have to write the function `direct_form_1` at the place indicated in the main.c file to realize the direct form I implementation of the above filter (4). Keep in mind that since the first denominator coefficient is not unity, you need to scale down the output by the same. Also, to compress the output to accommodate it in 16 bits, you may have to further scale down the output by 32768.
4. Use the synthetic input given in the main.c file and test your code by first manually calculating the output signal values and then comparing it with the output obtained by running the code in CCS.
5. Verify the frequency response 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.

Note: Please follow steps mentioned in appendix B in case you observe distorted output on the DSO.

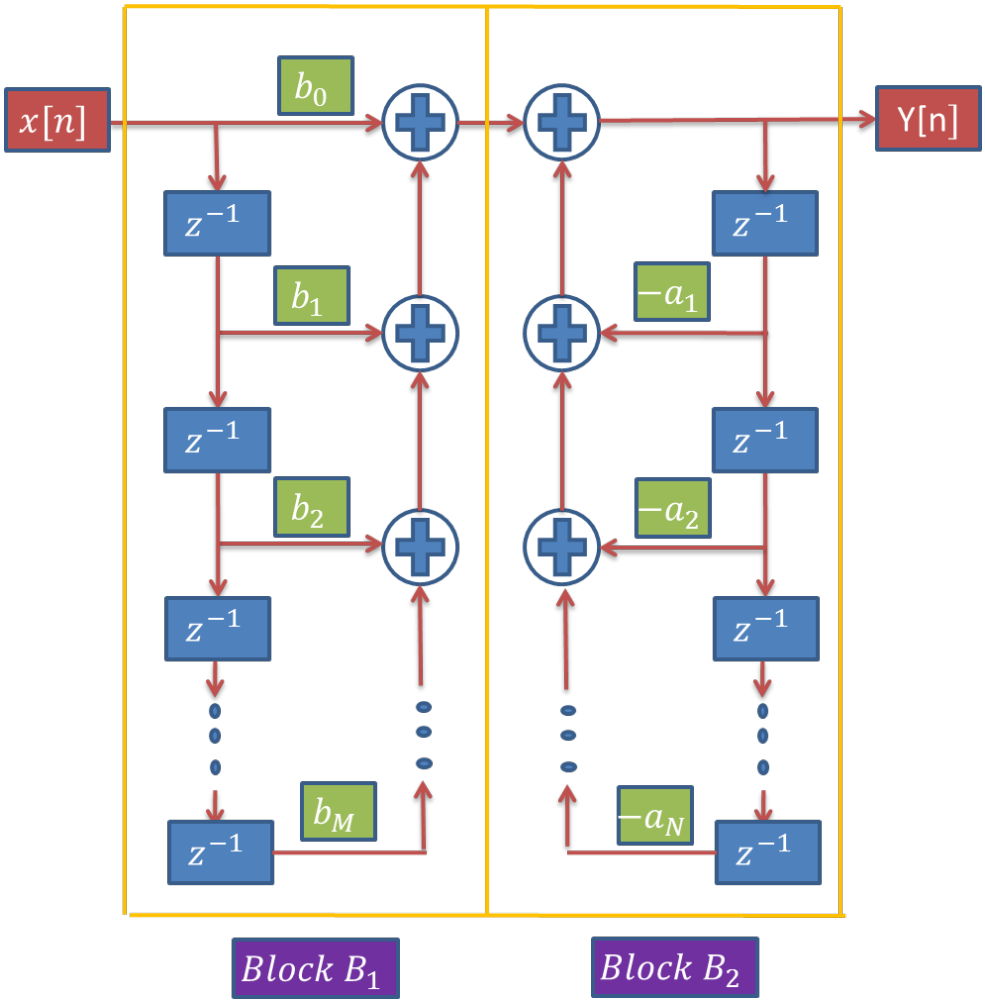

Figure 1: Block diagram for direct-form I structure for the IIR filter given by (2)

2.1.2 Learning checkpoint

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

Now, you will be implementing the IIR filter given by equation 4 in assembly language. You will be using `main_assmbly.c` and `iir_assmbly.asm` for this checkpoint. Please note that you will need two circular buffers (for saving previous output and input samples). Refer `circular_buffering.pdf` to understand how to configure the registers circularly.

1. Use the synthetic input given in the main.c file and test your code by first manually calculating the output signal values and then comparing it with the output obtained by running the code in CCS.
2. Repeat step 5 from checkpoint 2.1.1.
3. Compare the C and assembly implementation of the IIR filter by profiling the functions independently. Which one is faster and why?

3 IIR vs FIR filter implementations

One can observe that the desired frequency response for an LTI system can be obtained through both, FIR and IIR systems. Let us explore the major advantages and disadvantages of both the approaches through working examples in the form of checkpoints.

3.1 Learning checkpoint

Following tasks are to be done on Matlab and shown to your TA to get full credits for this checkpoint.

1. Define the following filter in Matlab with the numerator and denominator coefficients {`10000,-12728,8100`} and {`100, -160, 64`} respectively. In other words, the filter that you are implementing now is,

$$H_2(z) = \frac{1 - 1.2728z^{-1} + 0.81z^{-2}}{1 - 1.6z^{-1} + 0.64z^{-2}}. \tag{5}$$

2. List down the poles and zeros of the system and get them verified from your TA/RA.

3. Using the freqz function, plot the frequency response of the filter in Matlab and identify the type of the filter (Lowpass / Highpass / Bandpass / Notch). Now, let us call the frequencies where the gains fall by 3dB and 20dB w.r.t. the pass-band as `Fpass` and `Fstop`, respectively. Note down `Fpass` and `Fstop` for $H_2(z)$.

Text

4. Give an estimate of the number of computations required to calculate one sample of the output using the filter $H_2(z)$.

5. Finally, is the phase of the filter $H_2(z)$ linearly varying with the frequency `f`? Refer to the plot output by the function `freqz`.

6. Consider the situation where we are constrained to use only 1 digit after decimal point to specify all the coefficients. Now, this is equivalent to introducing round-off error while quantizing the coefficients. Let, $H_3(z)$ be the resulting transfer function. You can use the coefficients `num_coeff = {10000, -12000, 8100}, den_coeff = {100, -160, 60}` in your IIR filter implementation code. Now, repeat steps 2 to 3 of checkpoint 3.1 for $H_3(z)$. Report the observations to your TA/RA. Is the change in the frequency response from that of $H_2(z)$ to that of $H_3(z)$ significant? Also, comment about the stability of $H_3(z)$.

3.2 Learning checkpoint

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

1. Here we will obtain a FIR filter whose frequency response is similar to the IIR and understand effect of quantization. For this checkpoint, you will have to use the `FDAtool` facility provided by Matlab. Type `fdatool` in the Matlab command window to open the same. It allows you to design an FIR or an IIR filter given its desired frequency response. Explore the GUI and try to understand the functionality of the tool. You may also take help from the `help` documentation in Matlab about the tool.

2. Now, considering the frequency response of the filter $H_2(z)$ as the desired one, design an FIR filter using `FDAtool`. Use the `Fpass` and `Fstop` entries as the `Fpass` and `Fstop` values you calculated in step 3 in checkpoint 3.1. `Apass` should be 3dB whereas `Astop` should be 25dB. Click on the `Design Filter` button at the bottom. Click on `File` menu and then on `Export` sub-menu. Click on the `Export` button. You can see the designed FIR filter coefficients in the `Num` variable in the Matlab workspace. Note down the length of the resulting filter. Let's call this filter $H_4(z)$.

3. Estimate the number of computations required to generate 1 output sample using $H_4(z)$.

4. Does $H_4(z)$ have a linear phase?

5. Now, we want to assess the performance of the FIR filter $H_4(z)$ in the scenario where in we are constrained to use only 1 digit after decimal point to specify all the coefficients of $H_4(z)$.

   - First, plot the frequency response of this filter using the function `freqz`.
   - Next, normalize the first coefficient to 1 using the command `Num=Num*(Num(1)^(-1))`.
   - Finally, run the command `Num_single_digit=(round(Num*10))/10`. The vector `Num_single_digit` contains the filter $H_4(z)$ with all coefficients reduced to the precision of 1 digit after the decimal point. Let's call this filter $H_5(z)$.

6. Plot the frequency response of $H_5(z)$. Is it much different from that of $H_4(z)$?. Determine the `Fpass` and `Fstop` of $H_5(z)$ and comment on the stability before and after rounding the coefficients and compare your observations with that of IIR.

Appendices

A Direct form II IIR implementation

In direct form I structure, the total memory space required is of length $M+N+2$ (combining that needed by input and the output buffers). To reduce it, we turn to the direct-form II structure which we shall describe now. **We will not implement it in this lab but interested readers can try it out**.

- If, while implementing direct form I structure, the processing (of multiplication and addition) is performed on the previous inputs first and then on the previous outputs to calculate the current output, then it can be depicted in block-diagram format as shown in Figure 1.

  Note: All block diagrams are drawn assuming that $a_0 = 1$, i.e., all coefficients are normalized by a_0. But, during implementation if it is not, you will have to scale down the output by a_0.

- Let's change the order of operations so that, the two blocks B_1 and B_2 exchange their positions. As a result, we get the block diagram in Figure 2.

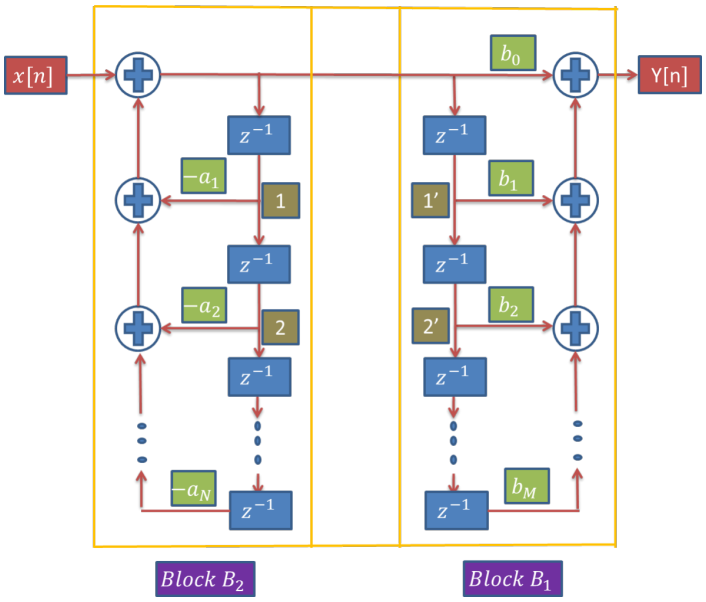

Figure 2: Block diagram obtained by exchanging the positions of blocks B_1 and B_2

- At this point, look at the block diagram in Fig. 2 and think on how you can reduce the total buffer space required to implement the same system. Take help from your TA/RA, if required. Once you have figured it out, read further.

- We observe that, the signals at the points 1 and $1'$, 2 and $2'$ and so on, are identical. So, rather than using separate delay elements (note that each delay element in the structure is equivalent to an extra memory space in the buffer) to generate these pairs of signals, we combine them to get a more efficient structure, in terms of memory usage, as is shown in the block diagram in Figure 3.

- This implementation is called the direct form II structure. Here, only one buffer has to be maintained, for the *intermediate signal*, which we shall denote by $w[n]$.

The algorithm for implementation of an IIR filter, defined by (2), on a DSP by direct-form II can be written as follows:

1. Allocate memory space to just one buffer of size $max\{M+1, N+1\}$ (why?) and initialize it with zeros. Let's call this as *Intermediate signal buffer (ISB)*.
2. Start the acquisition of data.
3. Go on calculating the values of the intermediate signal according to (2) and storing them in the ISB, one after the other, until it saturates.

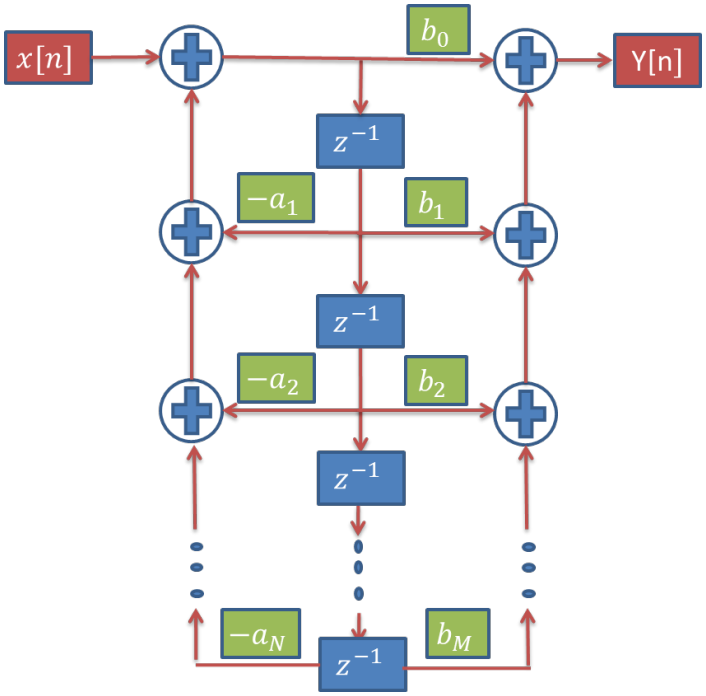

Figure 3: Block diagram for direct-form II structure for the IIR filter given by (2)

4. Feed the output display unit (the DSP codec combined with the DSO in our case) with every output sample, as it is calculated.
5. After the buffer runs out of memory space, start overwriting the oldest values with the latest values.
6. Continue steps 3 to 5 till the input data-flow continues.

B Overflow Avoidance Techniques in Filter Implementations on C5515

DSP programmers are often faced with the problem of dealing with overflows that occur as a result of some computation, especially during IIR/FIR digital filter implementation. In certain cases, an occurrence of an overflow results in distortion. In FIR filters, an overflow in computation affects the result of that particular computation. In contrast, an overflow in computation in IIR filters affects that particular result and all remaining results due to the inherent feedback in their structure.

In C5515/35 DSP kit, left and right channel I/O registers of AIC3204 codec are of 16 bits only. As described above, sometimes the result may require more than 16 bits to store the value. Hence we need to take appropriate measures to avoid this problem as described below:

- **Fixed Scaling of Input Samples:**
  The main idea in this technique is to scale the inputs with a fixed scale factor. Check the range of received input signal on the DSP by storing the input samples in a finite size array and observing the sample values using watch window of CCS. Apply appropriate scale factor before storing the samples in the corresponding input variable.
- **Usage of HI:**
  When the range of the input samples and/or filter coefficients used for filtering are large, please check how many bits are required to store the intermediate results in accumulator (You can do that by adding accumulator (Ex. AC0) in watch window). If the result always requires more than 16 bits, use HI(AC0) (i.e Output = Output / 2^{16}) to move the result in to 16 bit output variable.
- **Scaling Output Samples:** The technique described above generally results in a lower signal-to-noise ratio (SNR), hence we need to scale the output appropriately. To do this, plot the recent_output array using Graph tool in CCS. If you are able to view the expected output waveform on the graph but not on the DSO, check peak-to-peak variations on the Graph window of CCS. If it is found low (ex: peak-to-peak value 100), try setting appropriate gain to the output variable.

C Useful Assembly Instructions

- `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.

D 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.

E 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.