# Digital filtering using XA AN700

*Author: Santanu Roy, MCO Applications Group, Sunnyvale, California*

## SUMMARY

This report describes a method of implementation of FIR filters using Philips XA microcontroller. Appended with this application note is a generic routine that could be used to implement a N-point FIR filter.

## INTRODUCTION

The term “digital filter” refers to the computational process or algorithm by which a digital signal or sequence of numbers (acting as input) is transformed into a second sequence of numbers termed the output digital signal. Digital filters involve signals in the digital domain (discrete-time signals) and are used extensively in applications such as digital image processing, pattern recognition, and spectral analysis.

Digital Signal Processing (DSP) is concerned with the representation of signals (and information they contain) by sequences of numbers and with the transformation or processing of such signal representations by numeric computational procedures. In order to be considered a DSP microcontroller, a part must be able to quickly multiply two values, and add the result to an accumulator register. this is a minimum requirement. “Quickly” implies MAC (Multiply and Accumulate). Typically, the multiply and accumulate path operates on 16-bit values with a 32-bit result. Figure 1 shows a typical Digital Signal Processing hardware used in digital filtering.

Although XA currently does not have a hardware MAC unit, it is quite suitable for some DSP applications, due to its relatively high computational power, and high I/O throughput. This application note is intended to demonstrate such DSP power of the XA through implementation of FIR and IIR digital filters. It is to be noted, though, that this application note is not intended as a learning tool for DSP. It is assumed that the reader is familiar with DSP and filtering basics.

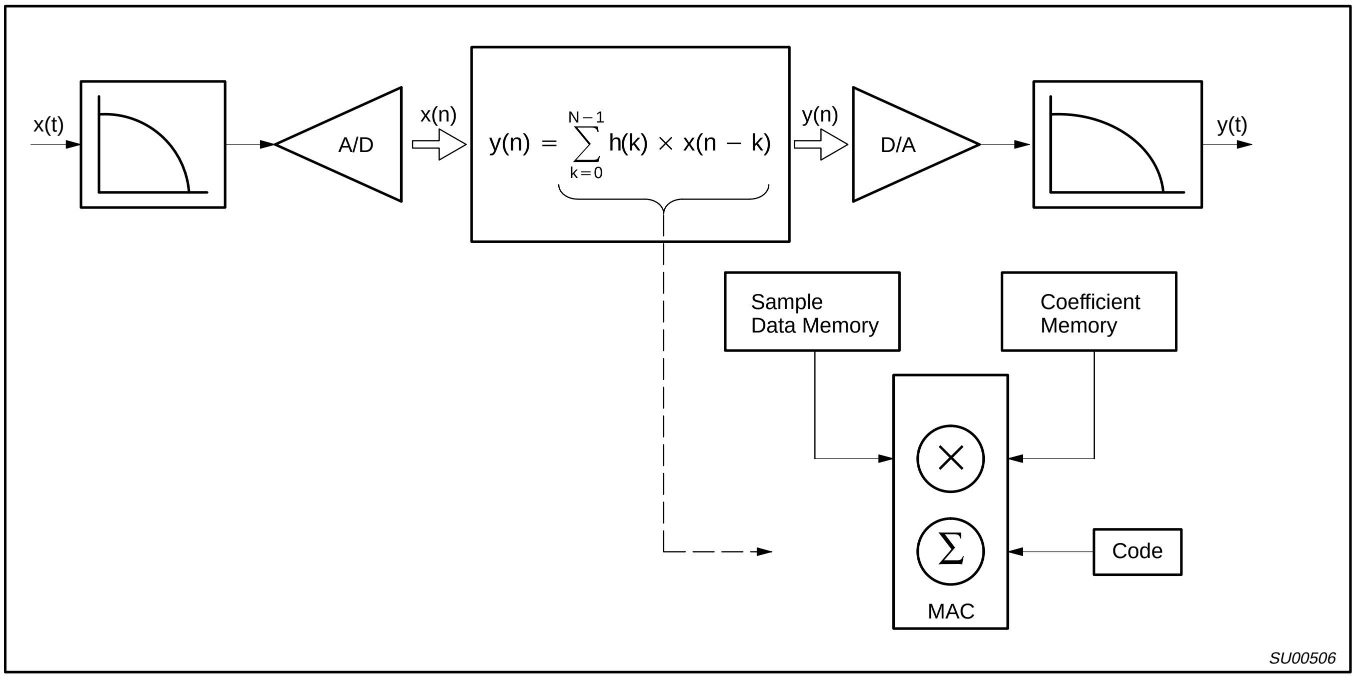


**Figure 1. Typical DSP Hardware**

## Filter Algorithms

For a large variety of applications, digital filters are usually based on the following relationship between the filter input sequence x(n) and filter output sequence y(n);

$$y(n) = \sum_{k=0}^{N} a_k \times y(n-k) \quad \sum_{k=0}^{M} b_k \times x(n-k) \qquad (1)$$

where $a_k$ and $b_k$ represent constant coefficients and N and M represent the number of input samples.

Equation (1) is referred to as a linear constant coefficient difference equation. Two classes of filters can be represented by such equations:

1. Finite Impulse Response (FIR) filters, and
2. Infinite Impulse Response (IIR) filters.

This applications note describes the implementation of the FIR class of digital filters on the XA.

## FIR Filters

FIR filters are preferred in lower order solutions, and since they do not employ feedback (output values used in the calculation of newer output values), they exhibit naturally bounded response. They are simpler to implement, and require one RAM location and one coefficient for each order.

For FIR filters, all of the $a_k$ in equation (1) is zero. Therefore (1) reduces to:

$$y(n) = \sum_{k=0}^{M} b_k \times x(n-k) \qquad (2)$$

As a result, the output of an FIR filter is simply a finite length weighted sum of the present and previous inputs to the filter. If the unit-sample response of the filter is denoted as h(n), then from (2), it is seen that h(n) = b(n). Therefore, (2) is sometimes written as:

$$y(n) = \sum_{k=0}^{N-1} h(k) \times x(n-k) \qquad (3)$$

where N = length of the filter = M+1.

## Digital Filter Implementation

As described above, a digital filter (FIR or IIR) could then be implemented by multiplying a vector of sampled signals with another vector of constants (coefficients) and adding the results to a register. The vectors involved in the filter process are derived from transformation of an S domain transfer function into the sampled Z domain.

## The Multiply-Accumulate (MAC) Function

The MAC speed applies both to finite impulse response (FIR) and finite impulse response (IIR) filters. The complexity of the filter response dictates the number MAC operations required per sample period.

A multiply-accumulate step performs the following:

- Read a 16-bit sample data (pointed to by a register)
- Increment the sample data pointer by 2
- Read a 16-bit coefficient (pointed to by another register)
- Increment the coefficient register pointer by 2
- Sign Multiply (16-bit) data and coefficient to yield a 32-bit result
- Add the result to the contents of a 32-bit register pair for accumulate.

This accumulator should be initialized to zero before calculating each output. It is assumed that the algorithm cannot overflow the accumulator, either by reducing significant bits of samples and/or constants or the number of accumulations.

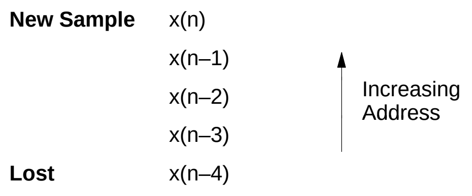
All these above MAC operations take place in addition to a buffer management routine that maintains an updated database for the filters samples, and system coefficients.

## Buffer Management

In order to effectively perform the task of buffer management, the processor should be able to quickly “shift” data (or pointers) in a data array which contains a series of input samples. New data is going in, and oldest sample is disposed off.

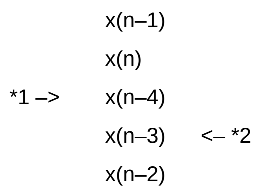
There are few ways to maintain and manage this database. They are as follows:

1. Linear Buffer



Linear buffer management requires the data to physically move down towards the oldest sample, then the newest sample is written into the top (FIFO style).

2. Circular Buffer



*1 At the beginning of filter pass, input pointer points to the oldest (n–4) sample, new sample is stored there.
*2 At the end of the filter pass, pointer now points to the next oldest sample (n–3).

Circular buffer management requires a test to make sure a buffer pointer increment does not move the pointer beyond the “tail” (end) of the buffer. If so, the pointer must be reset to the “head” (beginning) of the buffer.

Selecting one approach versus another depends mainly on the overhead involved with this task over the plain Multiply-Accumulate and loop control operations, and may vary based on the processor architecture, storage access time and other factors.

## MAC Implementation on the XA

An efficient loop for memory mapped vectors is presented below. The loop entry is at an even address, to reduce the fetch overhead after branch to beginning of the loop. Arrays are accessed using the indirect-autoincrement addressing mode.

```
.incld fir.h
  MAC_LOOP:
       mov.w   R3, [R1+]    ; read sample vector entry
       mov.w   R4, [R2+]    ; read coefficient vector entry
       mul.W   R3, R4       ; multiply
       add     R5, R3       ; accumulate into
       addc    R6, R4       ; a 32 bit register pair(R5:R6)
                            ; this serves as the RRP
       djnz    R0, MAC_LOOP ;
                            ;decrement loop counter and
                            ;branch to MAC_LOOP
```

The loop contains 13 bytes and takes 32 clocks (including branch penalty) per iteration (1.6µS at 20MHz and 1.07µS at 30.0MHz).

The following section analyzes the digital filter performance, including initialization, I/O, MAC operations and sample vector buffer management.

## An N-Point FIR Filter Implementation on XA

The FIR filter maintains a list of a fixed number N of recent samples. At each iteration, a new sample is taken, replacing the oldest sample on the list. This list represents a sampled vector. It is then multiplied by an N constant's vector to yield the current output.

As mentioned earlier, there are 2 register pointers fetching data samples and coefficient and feeding it to the ALU for 16-bit signed multiply with the 32-bit result being added to the MAC result register pair (RRP). In addition, a buffer management routine updates the sample data buffer each sample period.

The following sample codes show the mechanism for running filters on successive samples. It reflects the simplest data structures and list management, to simulate an output of a high level compiler.

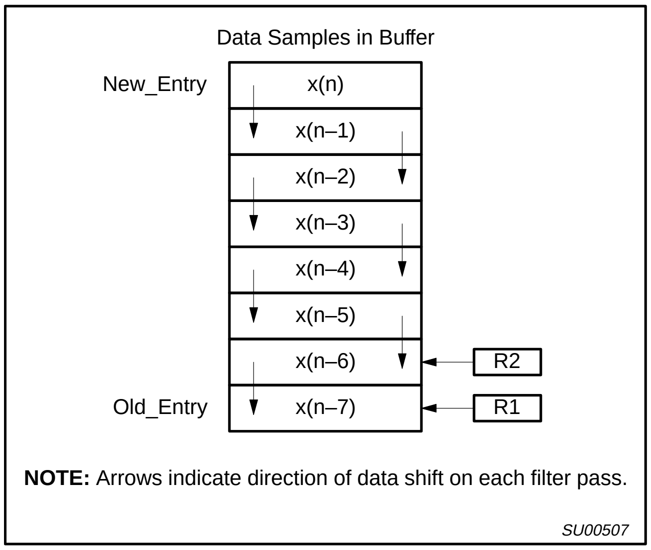


**Figure 2. Buffer Management for FIR Filter**

## FIR Algorithm in XZ

```
;Preliminary initialization for first filter pass:
.incldfir.h
  Start_FIR:
    mov  R0, #N–1           ; N = number of entries in the list
                            ; = loop counter
    mov  R1, #Old_Entry     ; compiler uses 2 pointers
    mov  R2, #Old_Entry+2   ;

Shft_Smpl:
    mov.w  [R1+], [R2+]
    djnz   R0, Shft_Smpl
                            ; – SAMPLE FROM A/D PORT
    mov    R0, A2D          ; input from port
    and    R0, #mask        ; mask upper bits (for N–bit A/D)
    mov    New_Entry, R0    ; add to list

Mac_init:                   ; MULTIPLY ACCUMULATE
    mov    R0, #N           ; N = number of entries in the list
                            ; = loop counter
    mov    R1, #Old_Entry   ; pointer to sample vector
    mov    R2, #Coef_Entry  ; pointer to coefficient vector
    xor    R5, R5           ; zero to accumulator
    xor    R6, R6           ; zero to accumulator

MAC_LOOP:
    mov.w  R3, [R1+]        ; read sample from list to reg
    mov.w  R4, [R2+]        ; read constant from list to reg
    mul.w  R3, R4           ; multiply
    add    R5, R3           ; accumulate in RRP
    addc   R6, R4           ; complete 32 bit add
    djnz   R0, MAC_LOOP

ACC_corr:                   ; – NORMALIZE RESULT BY SHIFTING
    asl    R5, #norm**      ; correction for non–significant LSBs
                            ; for eight 10 bit samples and 16 bit
                            ; constants, #norm=3
                            ; i.e. take only most significant 29 bits of the result
                            ; [16+10 + 3 (for 8 iterations)]
                            ; – OUTPUT TO D2A PORT
    mov    DAC, R6          ; send to DAC

        A total of 62 bytes and 370 clocks for this FIR algorithm.
```

** For N=8, 10 bit A/D, 16 bit filter coefficients; 8, 12, 16 bits clock very similar performance.

Total time for an 8-point filter at 20 MHz is 19.0 microseconds and 12.7 microseconds at 30 MHz. This would translate to a maximum sampling rate of 52 KHz at 20 MHz and 78 KHz at 30 MHz clock.
If this filter algorithm is interrupt driven, then additional 20 clocks would be required for latency, which would then translate to 50 KHz maximum sampling rate at 20 MHz and 75 KHz at 30 MHz. This puts the XA in the bandwidth of Audio Signal Processing (44.1 KHz) applications.

**NOTES:**

1. The above FIR algorithms are assembled with “asmxa rev 1.4” , the first XA absolute assembler for verification. It is to be noted in this context, that this assembler is a beta-site tool and still under evaluation. The syntax used in the assembler might be subjected to change. The functionality of the code is not checked at this stage using any simulator or ICE.
2. It is possible in the above MAC operation to extend the length of the accumulator to accommodate more iterations and higher precision (greater than 10-bit A/D) sample values with some additional overhead, e.g., using ’ADDC Rn, R6H”, etc., after the 32-bit accumulate, where Rn is a byte-size register to increase the length of the accumulator to accommodate more accumulations and higher precision (greater than 10-bit A/D) sample values.

## Author's Note

All addresses and constants assumed 16 bit for generality. Performance is calculated for a work-aligned branch targets which is mandated in the XA architecture for performance reasons. Misalignment will result in addition of NOPs by the assembler causing penalty in both code density and execution times. It is also to be mentioned that this is not the fastest executable code for the XA. A good programmer can combine the two loops into one, and data can be kept in registers. For low order filter implementation, code can be written in-line, and can utilize direct addressing mode for samples array.

This code was written in a way that reflects minimum expected optimization form a compiler (local loop optimization only), and it shows the expected speed for code written in a high level language, without rewriting routines in assembly language. also, this is not the ultimate performance for the XA architecture. The register banks can be used to store coefficients and samples, resulting in slightly faster execution time.

## Author's Acknowledgement

The author recognizes the following Philips Semiconductors XA team members for their review and inputs on this article:

Greg Goodhue, Ori Mizrahi-Shalom, and Ata Khan.

## References

*XA User Guide* — Philips Semiconductors
*Digital Signal Processing* — Rosenbaum