Group Name: 5Alive

CWK 1 Report -

Real-Time Filter Implementation

Aaish Bakhtiar
UNIVERSITY OF WESTMINSTER, MARTIN GILES

Contents

Abstract

This report will explore the design of a digital TDL filter. Initially detailing the background theory and research into digital filtering, going on to describe the design and testing of the digital TDL filter programmed in MPLAB to be run on the PIC microcontroller. It will explain the workings of the TDL filter code and how it is tested. In our case, we will be designing a digital TDL filter with a word length of 12-bits. Using this word length, we will have to design the TDL filter accordingly. With all the background theory to aid us in designing the best possible TDL filter. We will also talk about how another microcontroller being used would affect the filter, with a final conclusion consisting of our evaluation of the filter and what we could've done to make it better.

Chapter 1: Background

1.1- Theory of TDL filter structure

TDL stands for "Tapped Delay Line" and it is an assortment of simple delays with a variety of gains and amounts of delay. It extracts a signal output from anywhere within the delay line, scales the signal and then sums with other taps forming an output signal. Each tap can be non-interpolating or interpolating, for a non- interpolating tap- it extracts the signal at some fixed integer delay comparative to the input. [1, 2]

TDL filter is based on the pointer manipulation using circular addressing, this can be seen in **Table 1** below:

$x(n)$	New input sample or current oldest sample
$x(n-3)$	
$x(n-2)$	
$x(n-1)$	
$x(n-0)$	

Table 1 - Pointer Manipulation using Circular Addressing

$$y(n) = \sum_{k=0}^{m} x(n-k).h(k)$$

Equation 1

Equation 1 : Where x(n) is a sample of the input, y(n) sample of output and h(n) filter coefficient respectively at nth sample instant of a digital system of order m. From the equation above to obtain an output a buffer of m previous delay line value needs to be maintained along with the current sample. Usually, a pointer is set up at the start of the sample array and then is manipulated to access the consecutive values. All values are shifted down when a new sample is added to the delay line. For large values of the delay line (m) will cause additional over- head of shifting the large data. Or the oldest value can be overwritten, by implementing a circular mode for pointer access. There is a finite size for the input data buffer, it must be accessed circularly as the new samples are continuously written into the buffer, the oldest samples need to be overwritten so that the buffer memory Is reused. Once the pointer reaches the last location of the buffer, it will need to wrap back to the start of the buffer. The table above illustrates the circular addressing, the input buffer is made circular for that it must be properly aligned on the internal memory. [1, 2]

1.2- MACs & Other Computations

MAC is an expensive plus important operation as it is used in digital signal processing, and video graphic applications. When performing this operation any improvement in delay can be a positive impact on processor, clock speed and instruction time. In embedded code the MAC unit provides hardware support for a limited set of DSPs (digital signal processing) operations. The unit is integrated into the OEP (operand execution pipeline) and it implements a 3-stage arithmetic pipeline adjusted for 16x16 multiplies. This can be seen in Due to this design 16 and 32- bit operands are supported with a full set of extensions for unsigned and signed integers. Also fixed-point fractional input operands are included. [3]

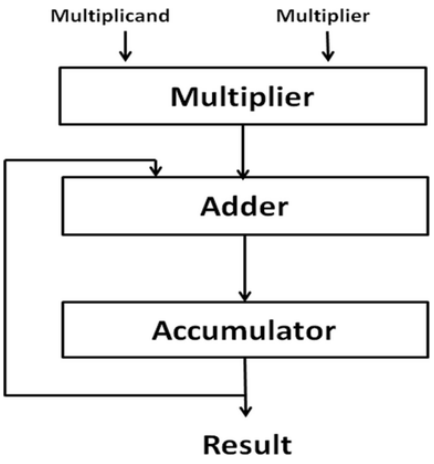

Figure 1 - MAC unit structure

Another operation can be an enhanced multiple and accumulate (eMAC), this unit is more accurate and faster as it's used in more intensive software algorithms- optimizing audio decoding/ encoding for MP3. It contains four 48- bit accumulators, it allows for 16x16 and 32x 32 multiplies with a 40- bit product. Like the MAC the eMAC unit provides functionality in three areas:

- Unsigned and signed integer multiplies
- Provides miscellaneous register operation
- Multiply accumulate operations supporting un/signed integer operands.

Table 2 below shows the differences of the MAC and the eMAC.

MAC	eMAC
3- Stage execution pipeline	4- stage execution pipeline
Optimized for 16- bit operands	Optimized for 32- bit operands
16 x 16multiply array	32 x 32 multiply array
Single 32- bit accumulators	Four 48- bit accumulators
32- bit products	40- bit products

Table 2 - Differences of MAC & eMAC

1.3 - Handling Fractions in Fixed-Point Arithmetic

In computing, a fixed point is a method of representing a fractional number by storing a fixed number of digits of their fractional part. An example of this can be the value 4.35, this number can be stored in a variable as an integer with the value of 4350, it would be stored with a scaling factor of 1/1000 and 4350000 can be represented as 4350 with a scaling factor of 1000. This representation allows standard integer arithmetic units to perform rational number calculations. Most modern computers can support floating point numbers which is how fractional numbers are handled but the use of floating point is not the only way to represent fractional numbers. By reusing all integer arithmetic circuits of a computer, fixed point arithmetic is orders of magnitude faster than floating point arithmetic. The reason we use fixed-point arithmetic is because it is faster than floating point; floating point is too slow, and integers truncate the data. [4]

The binary point is the key to handling fractions in fixed-point arithmetic. The binary point acts as a divider between the integer and fractional part of a number. In the decimal system, the coefficient of the number is represented with e.g 2^0, and the side in which the numbers go from the digit represents either a positive coefficient or a negative coefficient; all digits to the left of the binary point carry a weight of 2^0, 2^1, 2^2... and digits on the right of the binary point carry a weight of 2^{-1}, 2^{-2}, 2^{-3} etc. This is very important in handling fractions fixed-point arithmetic, as the negative coefficients is how fractional binary numbers can be represented.

After familiarising with the binary point in handling fractions, the next important key is shifting between the binary point. Shifting the binary point by 1 bit right or left will either multiply the number or divide it. [5] Shifting an integer to the right by 1 bit position is the equivalent of dividing the number by 2, because the binary point is changed, the coefficient of the numbers has shifted, same way shifting the number to the left by 1 bit position is the equivalent of multiplying the number by 2. [5] By shifting the binary point, it will change the fraction into a real number if done correctly, this will in turn let the computer be able to read the number and be able to represent it.

Fixed point arithmetic is a great way to handle fractions as they are simple and as efficient as integer arithmetic in computers, the hardware built for integer arithmetic can be reused to perform real numbers arithmetic with fixed point arithmetic, without the hardware having to do a lot of processing or work. It is a very powerful way to represent fractional numbers on computers, because of it speed it is used in many different applications. The only downside would be the loss of range and precision, because of floating point number presentation. Some floating-point numbers cannot be represented in an efficient way, when they are represented the range of the integer part is lost. [5]

1.4- Theory of Calculating Timing in Real-Time Programs

Real-time programs are hardware and software systems that are subject to a real time, an event that occurs in real life with the system. It is called real-time since the simulation time in the system follows the speed of a real clock. Real-time operation is the operating mode of a computer system in which a program is always ready to get inputs from the external world and generate results in an appropriate period. [6]

In real-time systems, timing is a very important property of each task. It must be guaranteed that the execution of a task does not take longer than the specified amount of time. This is because each real-time task has a deadline it must meet, otherwise the whole real-time system will fail. Therefore, the theory of calculating timing in real-time programs is important. [7]

Reliable timing is important in many embedded systems & real-time systems, this is especially true when the real-time systems have control over human safety, systems like airplanes, military

equipment and industrial factories. [8] These systems need to be able to meet their timing requirements to ensure that they work safely and on time so that no harm is done to anybody.

Real-time responses are often measured in the order of milliseconds and sometimes microseconds. There are many different methods that that exist to measure execution time, but there is no single definitive technique that is used all over. This is because each technique is a compromise between many attributes, such as resolution, accuracy, granularity & difficulty:

Resolution

- Representation of the imitations of the timing hardware. A stopwatch can measure time with a 0.01 second resolution whereas, computer software like a logic analyser can measure with a resolution of 50 nano seconds.

Accuracy

- The closeness of the value as compared to the actual time if a perfect measurement was obtained. When many measurements are taken and repeated several times, there are usually some differences or errors in the measurements.

Granularity

- This is the part of the code that can be measured. This is a measurement usually specified in a subjective manner. An example can be coarse granularity methods would generally only measure execution time on a per-process, per-procedure, or per-function basis. However, a fine granularity method measures execution time of a loop, small code-segments and or specific functions.

Difficulty

- This measurement defines the level of effort that is needed in order to obtain the measurement.

A table of these measurement methods listed above can be seen below showing the difference in the attributes with each measurement used to measure execute time; seen in **Figure 2** below.

Method	*Typical Resolution*	*Typical Accuracy*	*Granularity*	*Difficulty of Use*
stop-watch	0.01 sec	0.5 sec	program	easy
date	0.02 sec	0.2 sec	program	easy
time	0.02 sec	0.2 sec	program	easy
prof and gprof	10 msec	20 msec	subroutines	moderate
clock()	15-30 msec	15-30 msec	statement	moderate
software analyzers	10 µsec	20 µsec	subroutine	moderate
timer/counter chips	0.5–4 µsec	1-8 µsec	statement	very hard
logic or bus analyzer	50 nsec	half µsec	statement	hard

Figure 2 - Summary of methods to measure execution time

This table lists all the methods and attributes of each method of measurement, this table shows us which method I most viable in specific scenarios of measuring timing in real-time systems. One method may be more useful in certain circumstances and another may prove more useful in

another circumstance of measuring. Knowing these methods and the attributes can aid in measuring timing in real-time systems. [9]

1.5 - Standard Input Test Signals: Impulse, Nyquist & Step Response

We can analyse the response of control systems in both the time domain and the frequency domain. If the output of the system varies with time, it is called the time response of the system. It is knowing how different system variables change their state respect to time. [10] This time responses comes in 2 variations.

- Transient response
- Steady state response

An example of a response of a system in the time domain is shown in **Figure 3** below.

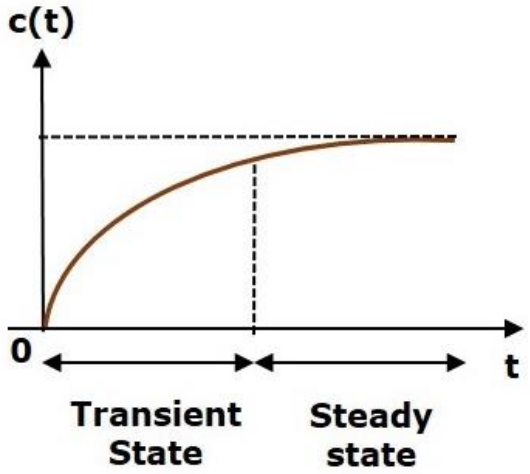

Figure 3 - Response of control system in time domain

Using the variable from the graph in **Figure 3**, we can write an equation for the time response c(t) as:

$c(t)=ctr(t)+css(t)$

where,

- $c_{tr}(t)$ is the transient response
- $c_{ss}(t)$ is the steady state response

The transient response is the state before the steady state, because the output time response takes a bit of time to reach the steady state it will be in the transient stage where it is transitioning into the steady state. When the response time reaches 0, the transient response stage will diminish.

The steady state response is the part of the output time response that stays after the transient response is finished. When the response time reaches 0 and starts to go up, the steady state response stage will start. [11]

Generally, systems are required to perform both states and to control both responses, some sort of feedback is required. This feedback come in the form of standard test signals. Because we cannot know the input ahead of time as it may be random and be situation dependent, we cannot

get an output response. This is why standard test signals are used, these are signals used to judge the performance of a system, with the different signals giving different inputs like a sudden shock, or a sudden change in its properties, these are called standard input test signals. We will be talking about 3 different input test signals: Impulse, Step and Nyquist.

1. Impulse Signal

The impulse input test signal is a signal that sends a sudden shock to the system to see its response. It is a very short input signal, that last for a very short time. The impulse signal is used to check the immediate response of the system after the impulse signal has diminished. [12]

2. Step Signal

The step input test signal is a signal that changes the properties of the actual signal of the system. It gives the system an input that stays there, used to see the transient response of the system. It shows the response of the system when the input signal is completely changed, seeing the stability of the system in response. [12]

3. Nyquist Signal

The Nyquist signal is more of a sampling theorem; a frequency, it's a principle that many engineers use in the digitization of analog signals. It's a frequency that can be best visualised as the frequency that has 2 samples per cycle. The simplest form of this signal is a sine wave in which signal will go above and below a certain frequency creating waves of different inputs. We can use this type of input on the signal to see the response of a continuously changing input or in this case we can use the Nyquist frequency to send a negative impulse signal as the input to see how the system will react. With the Nyquist frequency signal, we can input a negative value, e.g -2, as the input to the system to test the capabilities of the system, to see if it will output anything with a negative input, and if it does what the output means. [13] [14]

By analysing the response from the system after the four test input signals are used, the performance of the system can be judged and used to improve or change anything required to be changed in the system. These signals are key in making sure that the system can provide a response to any input that it may be given.

1.6– The PIC Architecture & Instruction Set: It's Limitations

The PIC architecture is a microcontroller or an MCU, the PIC stans for "Peripheral Interface Controller". A microcontroller is a compact integrated circuit that is designed for a specific operation in a bigger embedded system, [15] it could even be compared to a very small standalone computer with only a specific number of tasks. A PIC microcontroller is a type of microcontroller developed by Microchip. This microcontroller is a very fast and simple microcontroller, that is often used by many engineers when they want to implement an easy program. [16] The PIC architecture usually comes with a bus width of 8-bit,16-bit or 32-bit and it uses Harvard architecture. [17]

Chapter 2: Coding & Design of Filter

The designed filter must be a Tapped Delay Line, nth order filter with a Finite Impulse Response. The input signal x(k) will be a fixed point 8-bit signed integer value and the filter will use 12-bit fractional fixed point saturation arithmetic with two fractional bits to compute the output y(k). The general structure of the nth order FIR TDL filter can be represented as a diagram shown below in **Figure 4**.

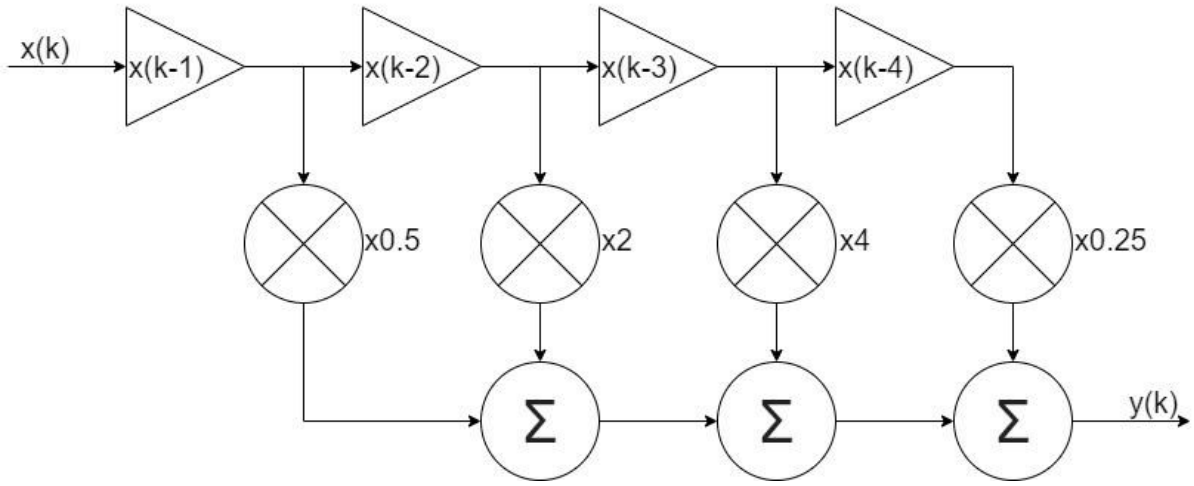

Figure 4 - Filter Structure Diagram

The code of the filter can be reduced into two parts, the real-time code and the set-up code.

2.1 – Explanations of Real-Time Code

The real-time code is referring to the code that is within the main loop of the program. This section of code is meant to execute in real time as the system is being fed with input.

At the beginning of each loop, the input x(k) must be obtained in order to calculate the output. However, before any arithmetic operations can be performed, the input must first be extended from its initial 8-bit signed integer representation into a 14-bit value with two fractional bits. To do this, the input x(k) is represented across two bytes using a sign-extending subroutine that pairs it with an upper byte.

```
sign_extend                ;extend from 8 bits to 12 bits
                    BTFSS     INDF,7   ;bit test if negative lower
                    goto      pos_sign
                    goto      neg_sign
   neg_sign     incf FSR,1 ;increment FSR
                    movlw     0xFF
                    movwf     INDF      ;make upper completely negative and
store in next space
                    return          ;goto      finish
   pos_sign     incf FSR,1 ;increment FSR
                    clrf  INDF
                    return          ;goto      finish
```

Left-Shift

The 14-bit sign extended value is then shifted to the left twice to make room for two bits that will represent the fractional portion of the value.

```
dividebytwo      ;dividebytwo store result in W
```

```
        BSF     STATUS,C
        BTFSS   INDF,7
        BCF     STATUS,C
        RRF     INDF,0
        return
```

The output y(k) can be expressed as the sum of four expressions or taps. These four taps are coded into subroutines that make use of the right and left shifting subroutines to multiply and divide values by two. The results of each tap are summed to compute the output.

2.2 – Set-Up Code

The set-up code is referring to the section of the program that is not computing the output of the filter but setting up the system in order to execute the instructions and they should be.

Chapter 4: Performance Enhancement Using a Different Computer Architecture

4.1 – Processor Architecture Theory & Performance

The processor architecture can come in either 32 or 64-bit, the difference between the two can vary in data path width, integer size and memory address width that the processor is able to work with. A 64-bit processor can process larger “chunks” of data and is able to address more memory than the 32-bit. [18] Within processor architectures can vary in memory bus architecture; RISC & CISC.

1. **R**educed **I**nstruction **S**et **C**omputer – made up of small fast instructions including data and address fields that fit into a single *(binary)* word. RISC tends to be made up of small number of instructions which usually take the same time to execute – 1 instruction cycle. Instructions in RISC can access memory at any point.
2. **C**omplex **I**nstruction **S**et **C**omputer – made up of large instruction which can concurrently do tasks often. The instruction in a CISC have a high-level functionality meaning it can do much of the arithmetic handling such as rounding, sign extension, complex conditioning branching etc.

Both instructions set have their own benefits and drawbacks however it does depend on the code to decide what instruction set is more appropriate. A RISC machine takes less clock cycles to run a command but requires more instructions to perform a task in some scenarios. A CISC machine takes more clock cycles to run a command however doesn’t require a lot of instructions to perform a task. However, from a programmer’s perspective the CISC is easier as it makes a programmer’s life much easier to program in much less code with more clock cycles making the code more efficient.

There are specifically two processor architectures, these are Von Neumann and the Harvard model. The Von Neumann architecture is the most used architecture in modern day computing, programs and data are stored in separated memories which can be accessed equally. We can enhance performance in the von Neumann architecture as something called the von Neumann bottleneck can arise. This is because instructions are done singularly and carried out sequentially. To override this, we can provide this architecture with more cache, RAM, or faster components. [19]

On the contrast, the Harvard model contains separate storage and buses for instructions and data. The advantage of this in comparison to the von Neumann architecture is that CPU can access instructions and read/write data at the same time. [20]

In 1965 Gordon Moore came up with a prediction of the pace of silicone technology, called Moore’s Law. Moore’s Law described the long-term trend of hardware, this meant the number of transistors that is placed onto an IC doubles every two years. [21] However, Dennard Scaling was that as transistors get smaller their power density stays constant, this meant that as a transistor gets smaller the power it uses reduces proportionally to the reduction in transistor size. This in theory enabled voltage scaling which in turn allows for higher clocking rates. Since 2007, the breakdown of Dennard Scaling Rule meant that the reduction of reduced transistor size is so small that the Dennard Scaling rule no longer applies thus greater use of multi-core architectures.

Computer performance refers to performance in terms of processing speed however this isn’t exhaustive meaning it is not just processing speed, as well as power, area and cost is also important. 3 factors which can improve application performance are

1. **CPU** – the more processors you have = higher performance due to multiple processors on the different job's simultaneously
2. **Memory** – faster RAM = the faster the processor speed resulting in memory transfer to other components thus being more efficient
3. **I/O throughout**

4.2 – How Greater Performance could have been Achieved for this Problem – Compare to PIC16fxxx

If we were to use the Blackfin – BF533 core for this project in comparison to the PIC16fxxx core, there would be no need to perform indexing for looping as the Blackfin – BF533 core has its own dedicated loop buffer. This core also allows for any word-length up to 32-bit meaning it can perform calculations in a single cycle whereas the PIC16xxx has no dedicated multiplier. Therefore, unless the PIC16fxxx C is of power 2 you would need to emulate multiply within software.

If we were to use the Blackfin – BF533 core for this project in comparison to the PIC16fxxx core, there would be no need to perform indexing for looping as the Blackfin – BF533 core has its own dedicated loop buffer. This core also allows for any word-length up to 32-bit meaning it can perform calculations in a single cycle whereas the PIC16xxx has no dedicated multiplier. Therefore, unless the PIC16fxxx C is of power 2 you would need to emulate multiply within software. [22]

Appendix

The program is written in assembly language in MPLAB version 3.5 is shown below.

```
        list            p=16f877A    ; list directive to define processor
        #include        <p16f877A.inc>       ; processor specific variable definitions

        __CONFIG _CP_OFF & _WDT_OFF & _BODEN_OFF & _PWRTE_ON &
_RC_OSC & _WRT_OFF & _LVP_ON & _CPD_OFF

; '__CONFIG' directive is used to embed configuration data within .asm file.
; The labels following the directive are located in the respective .inc file.
; See respective data sheet for additional information on configuration words.

;***** VARIABLE DEFINITIONS
w_temp              EQU  0x7D          ; variable used for context saving
status_temp  EQU  0x7E          ; variable used for context saving
pclath_temp  EQU  0x7F          ; variable used for context saving

k               EQU  0x21
xk              EQU  0x22
xk_upper        EQU  0x23
xk_1            EQU  0x24
xk_1_upper      EQU  0x25
xk_2            EQU  0x26
xk_2_upper      EQU  0x27
xk_3            EQU  0x28
xk_3_upper      EQU  0x29
xk_4            EQU  0x2A
xk_4_upper      EQU  0x2B
temp_res        EQU  0x2C
temp_res_upper      EQU  0x2D
tap_res             EQU  0x2E
tap_res_upper       EQU  0x2F
A_lower             EQU  0x30
A_upper             EQU  0x31
B_lower             EQU  0x32
B_upper             EQU  0x33
sum             EQU  0x34
sum_upper       EQU  0x35
y_k             EQU  0x36
y_k_upper       EQU  0x37

;*********************************************************
        ORG     0x000           ; processor reset vector

        nop                     ; nop required for icd
        goto    main            ; go to beginning of program


        ORG     0x004           ; interrupt vector location

        movwf   w_temp          ; save off current W register contents
```

```
        movf   STATUS,w         ; move status register into W register
        movwf  status_temp      ; save off contents of STATUS register
        movf   PCLATH,w     ; move pclath register into w register
        movwf  pclath_temp    ; save off contents of PCLATH register

; isr code can go here or be located as a call subroutine elsewhere

        movf   pclath_temp,w          ; retrieve copy of PCLATH register
        movwf  PCLATH                 ; restore pre-isr PCLATH register contents
        movf    status_temp,w     ; retrieve copy of STATUS register
        movwf  STATUS           ; restore pre-isr STATUS register contents
        swapf   w_temp,f
        swapf   w_temp,w        ; restore pre-isr W register contents
        retfie                  ; return from interrupt

impulse_input
                        addwf       PCL, 1          ;offset pc
                        retlw       d'1'
                        retlw       d'0'
                        retlw       d'0'
                        retlw       d'0'

sign_extend             ;extend from 8 bits to 12 bits, upper stored in FSR+1
                        BTFSS       INDF,7   ;bit test if negative lower
                        goto        pos_sign
                        goto        neg_sign
   neg_sign             incf FSR,1 ;increment FSR
                        movlw       0xFF
                        movwf       INDF      ;make upper completely negative and
store in next space
                        return      ;goto   finish
   pos_sign             incf FSR,1 ;increment FSR
                        clrf  INDF
                        return      ;goto   finish

timestwo                BCF                 STATUS,C    ;multiplies a 8 bit number by
2 and check for overflow
                        RLF                 INDF,0
                        movwf       temp_res                        ;stores result in
Wreg
                        BTFSS       STATUS,C
                        goto        positive
   negative             BTFSS       temp_res,7
                        goto        neg_overflow
                        return      ;goto   finish
   positive             BTFSC       temp_res,7
                        goto        pos_overflow
                        return      ;goto   finish
   neg_overflow         movlw       0x80
                        ;movwf      INDF      ;saturate to -128
                        return      ;goto   finish
   pos_overflow         movlw       0x7F      ;saturate to +127
```

```
                        ;movwf      INDF
                        return      ;goto   finish

add12b                          movf        A_upper,0
                        addwf       B_upper,0
                        movwf       temp_res_upper
                        movf        A_lower,0
                        addwf       B_lower,0
                        movwf       temp_res
                        BTFSS       STATUS,C
                        return      ;goto   no_carry
                        incf temp_res_upper,1
                        goto        addition_saturation
   addition_saturation      ;checking the upper sign bit of each 12b number, sign
bit is 5
                        btfss       A_upper,7
                        goto        positive_sat
           negative_sat
                        btfsc       B_upper,7
                        goto        negative_overflow_test
                        goto        no_overflow_exit
           positive_sat
                        btfss       B_upper,7
                        goto        positive_overflow_test
                        goto        no_overflow_exit
           negative_overflow_test
                        btfsc       temp_res_upper,7
                        goto        no_overflow_exit
           neg_sat      movlw       0x80   ;most negative value
                        movwf       temp_res_upper
                        goto        exit_test
           positive_overflow_test
                        btfss       temp_res_upper,7
                        goto        no_overflow_exit
           pos_sat      movlw       0x7F            ;most positive value
                        movwf       temp_res_upper
                        goto        exit_test
           exit_test    return
           no_overflow_exit
                        return

dividebytwo             BSF                 STATUS,C    ;dividebytwo store result in
W
                        BTFSS       INDF,7
                        BCF                 STATUS,C
                        RRF                 INDF,0
                        return

tap1                    movlw       0x24   ;tap1=(xk_1)/2
                        movwf       FSR
                        call dividebytwo
                        movwf       tap_res
```

```
                    incf FSR
                    call dividebytwo
                    movwf   tap_res_upper
                    return

tap2                movlw   0x26
                    movwf   FSR
                    call timestwo
                    movwf   tap_res
                    incf FSR
                    call timestwo
                    movwf   tap_res_upper
                    return

tap3                movlw   0x28
                    movwf   FSR
                    call timestwo
                    movwf   tap_res
                    incf FSR
                    call timestwo
                    movwf   tap_res_upper

                    movlw   0x2E  ;temp res lower x2
                    movwf   FSR
                    call timestwo
                    movwf   tap_res
                    incf FSR
                    call timestwo
                    movwf   tap_res_upper

                    return

tap4
                    movlw   0x2A
                    movwf   FSR
                    call dividebytwo
                    movwf   tap_res

                    incf FSR
                    call dividebytwo
                    movwf   tap_res_upper

                    movlw   0x2E
                    movwf   FSR
                    call dividebytwo
                    movwf   tap_res

                    incf FSR
                    call dividebytwo
                    movwf   tap_res_upper
                    return
```

```
main
; remaining code goes here

filter_loop             ;INPUT xk taken from function IS SIGN EXTENDED AND
SHIFTED TWICE
                        movf    k,0
                        call impulse_input
                        movwf   xk
                        movlw   0x22
                        movwf   FSR
                        call sign_extend
                        decf    FSR,1       ;FSR back to 0x22 = xk lower
                        call timestwo
                        movwf   INDF
                        call timestwo
                        movwf   INDF

                        incf FSR,1          ;shift upper byte
                        call timestwo
                        movwf   INDF
                        call timestwo
                        movwf   INDF

                        ;Output y_k is calculated as the sum of taps 1-4
                        ;clrf sum
                        call tap1
                        movf    tap_res,0
                        movwf   sum
                        movf    tap_res_upper,0
                        movwf   sum_upper

                        call tap2
                        movf    tap_res,0
                        movwf   A_lower
                        movf    tap_res_upper,0
                        movwf   A_upper
                        movf    sum,0
                        movwf   B_lower
                        movf    sum_upper,0
                        movwf   B_upper
                        call add12b     ;call   addition_saturation
                        movf    temp_res,0
                        movwf   sum
                        movf    temp_res_upper,0
                        movwf   sum_upper

                        call tap3
                        movf    tap_res,0
                        movwf   A_lower
                        movf    tap_res_upper,0
                        movwf   A_upper
```

```
                    movf    sum,0
                    movwf   B_lower
                    movf    sum_upper,0
                    movwf   B_upper
                    call add12b     ;call   addition_saturation
                    movf    temp_res,0
                    movwf   sum
                    movf    temp_res_upper,0
                    movwf   sum_upper

                    call tap4
                    movf    tap_res,0
                    movwf   A_lower
                    movf    tap_res_upper,0
                    movwf   A_upper
                    movf    sum,0
                    movwf   B_lower
                    movf    sum_upper,0
                    movwf   B_upper
                    call add12b     ;call   addition_saturation
                    movf    temp_res,0
                    movwf   y_k
                    movf    temp_res_upper,0
                    movwf   y_k_upper

                    nop

update_states           movf    xk_3,0
                    movwf   xk_4
                    movf    xk_3_upper,0
                    movwf   xk_4_upper

                    movf    xk_2,0
                    movwf   xk_3
                    movf    xk_2_upper,0
                    movwf   xk_3_upper

                    movf    xk_1,0
                    movwf   xk_2
                    movf    xk_1_upper,0
                    movwf   xk_2_upper

                    movf    xk,0
                    movwf   xk_1
                    movf    xk_upper,0
                    movwf   xk_1_upper

                    incf k,1    ;you may wish to stop the loop after k is too big
                    movlw   0x05   ;kmax
                    subwf   k,0

                    btfss   STATUS,Z
```

```
                    goto        filter_loop
                    goto endprog

endprog                     nop

        END                 ; directive 'end of program'
```

Bibliography

[1] "Tapped Delay Line," DSPRelated.com, [Online]. Available: https://www.dsprelated.com/freebooks/pasp/Tapped_Delay_Line_TDL.html . [Accessed 16 November 2021].

[2] M. I. Akram, "Efficient implementation of tap delay line filter using high speed Digital Signal Processor," IEEEXplore, 24 September 2012. [Online]. Available: https://ieeexplore.ieee.org/document/6310632. [Accessed 16 November 2021].

[3] V. G. O. Paul F. Stelling, "Implementing Multiply-Accumulate Operation in Multiplication Time," [Online]. Available: https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=614884. [Accessed 16 November 2021].

[4] "Fixed-Point Math & Other Optimizations," UNOCHARIOTTE, [Online]. Available: https://webpages.uncc.edu/~jmconrad/ECGR6185-2007-01/notes/UNCC-IESLecture23%20-%20Fixed%20Point%20Math.pdf. [Accessed 15 November 2021].

[5] H. So, "Introduction to Fixed Point Number Representation," 02 February 2006. [Online]. Available: https://inst.eecs.berkeley.edu/~cs61c/sp06/handout/fixedpt.html . [Accessed 15 November 2021].

[6] K. M. S. A. Halang, "Real-time systems, implenetation of industrial computerized process automation," 2001.

[7] P. P. Puschner, "Calculating the maximum execution time of real-time programs," 7 April 1989. [Online]. Available: https://www.researchgate.net/publication/226495155_Calculating_the_maximum_execution_time_of_real-time_programs. [Accessed 15 November 2021].

[8] A. Ermedahl, "Execution Time Analysis for Embedded Real-Time Systems," Malardalen University, [Online]. Available: http://www.es.mdh.se/pdf_publications/2840.pdf. [Accessed 15 November 2021].

[9] D. B. Stewart, "Measuring Execution Time & Real-Time Performance," InHand Electronics, inc, September 2006. [Online]. Available: https://www.embedded.com/tutorial-techniques-for-measuring-execution-time-and-real-time-performance-part-1/. [Accessed 15 November 2021].

[10] "Introduction to TIme response Analysis & Standard Test Signals 2.1," CircuitBread, 14 December 2020. [Online]. Available: https://www.circuitbread.com/tutorials/introduction-to-time-response-analysis-and-standard-test-signals-2-1. [Accessed 15 November 2021].

[11] "Control Systems - Time Response Analysis," tutorialspoint, [Online]. Available: https://www.tutorialspoint.com/control_systems/control_systems_time_response_analysis.htm. [Accessed 15 November 2021].

[12] Nasir, “Typical Test Signals in Time Domain Analysis,” Electrical Equipment, 2014. [Online]. Available: https://engineering.electrical-equipment.org/panel-building/typical-test-signals-in-time-domain-analysis.html. [Accessed 15 November 2021].

[13] “Nyquist Theorem,” TechTarget Contributor, September 2005. [Online]. Available: https://whatis.techtarget.com/definition/Nyquist-Theorem. [Accessed 15 November 2021].

[14] “Nyquist frequency, Aliasing, and Color Moire,” imatest, 2021. [Online]. Available: https://www.imatest.com/docs/nyquist-aliasing/. [Accessed 15 November 2021].

[15] B. Lutkevich, “Microcontroller (MCU),” TechTarget - IoT Agenda, November 2019. [Online]. Available: https://internetofthingsagenda.techtarget.com/definition/microcontroller. [Accessed 16 November 2021].

[16] “Difference between 8051 and PIC,” GeeksforGeeks, 07 October 2020. [Online]. Available: https://www.geeksforgeeks.org/difference-between-8051-and-pic/. [Accessed 16 November 2021].

[17] C. P. SOLUTIONS, “What is a PIC Microcontroller: The Harvard Architecture,” cadence, [Online]. Available: https://resources.pcb.cadence.com/blog/2020-what-is-a-pic-microcontroller-the-harvard-architecture. [Accessed 16 November 2021].

[18] J. Faircloth, “Processor Architecture - SQL Server 2000 Overview & Migration Strategies - Servers,” ScienceDirect, 2014. [Online]. Available: https://www.sciencedirect.com/topics/computer-science/processor-architecture. [Accessed 02 November 2021].

[19] “Von Neumann Architecture,” GeeksforGeeks, 04 August 2021. [Online]. Available: https://www.geeksforgeeks.org/computer-organization-von-neumann-architecture/. [Accessed 02 November 2021].

[20] “Harvard Architecture,” GeeksforGeeks, 09 May 2020. [Online]. Available: https://www.geeksforgeeks.org/harvard-architecture/. [Accessed 02 November 2021].

[21] G. E. Moore, Cramming more components onto integrated circuits, USA: IEEE Solid-State Circuits Society Newsletter, 2006.

[22] M. Giles, University of Westminster. [Online].