ECSE426 Microprocessor Systems

Winter 2017

Lab 1: Simple FIR Filter

Objective

The objective of this experiment is to familiarize you with assembly language programming concepts, ARM's Cortex instruction set (ISA) and assembly addressing modes. The ARM calling convention will need to be respected, such that the assembly code can later be used with the C programming language. In the follow-up experiments, the code developed here will be used in larger programs written in C. We will introduce the Cortex Microprocessor Software Interface Standard (CMSIS) application programming interface that incorporates a large set of routines optimized for different ARM Cortex processors. The tutorial associated with this lab will introduce you to the new version of Keil IDE (5.x) and the ARM compiler, simulator and associated tools.

Preparation for the Lab

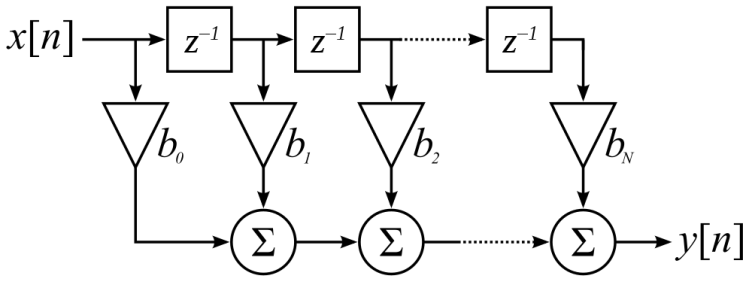
To prepare for Lab 1, you will need to attend Tutorial 1, where you will learn how to create and define projects, specifically purely assembly code projects. The tutorial will show you how to let the tool insert the proper startup code for a given processor, write and compile the code, as well as provide the basics of program debugging. Elaborate details about debugging in Keil are provided in the document entitled ***"Debugging with Keil"*** which will be uploaded on my courses.

Other documents that will be of importance include the Cortex M4 programming manual, and Quick reference cards for ARM ISA, all available within the course online documentation. **More useful references are found in the tutorial slides.** But since the reference material is huge (hundreds of pages over the course of the semester), make sure to attend the tutorials so that the TA will guide you where to look.

Background on FIR filter

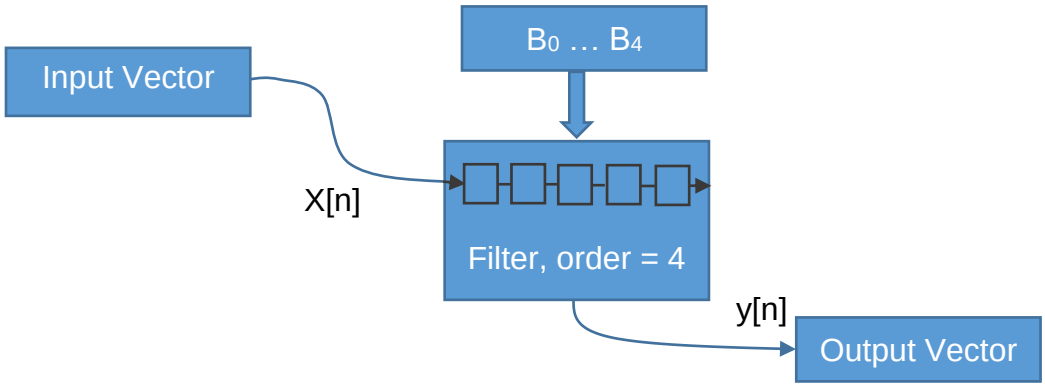
The design of FIR filters using windowing is a simple and quick technique. There are many pages on the web that describe the process, but many fall short on providing real implementation details. All the required information to put together your own algorithm for creating low pass, high pass, band pass and band stop filters based on a number of different windows are accessible online. The impulse response of an Nth-order discrete-time FIR filter lasts exactly N + 1 samples (from first nonzero element through last nonzero element) before it then settles to zero.

For a causal discrete-time FIR filter of order N, each value of the output sequence is a weighted sum of the most recent input values:



$$y[n] = b_0 x[n] + b_1 x[n-1] + \cdots + b_N x[n-N]$$
$$= \sum_{i=0}^{N} b_i \cdot x[n-i],$$

Where:
x[n] is the input signal,
y[n] is the output signal,
N is the filter order,
b_i is a coefficient of the filter.



The Experiment

Part I – Assembly and C implementations

You are required to write an assembly subroutine "`FIR_asm`" in ARM Cortex M4 assembly language that filters the one-dimensional input data and return the filtered data as one-dimensional as well. You should naturally use the built-in floating-point unit by using the existing floating-point assembler instructions.

Your assembly filter will take four parameters:

1. **A pointer** to the input data array
2. **A pointer** to the filtered data array (output)
3. The arrays' length
4. **A pointer** to the filter coefficients (struct)

Your subroutine should follow the ARM compiler parameter passing convention. Recall that up to four integers and 4 floating-point parameters can be passed by integer (R0 – R3) and floating-point registers (S0 – S3), respectively. For instance, R0 and R1 might contain the values of the first two integers and S0 will contain the value of a floating-point parameter. If the datatype is more complex (e.g, struct or a matrix), then a pointer to it is passed instead in R0 or R1. For the function return value, the register R0/R1 or S0/S1 are used for integer and floating-point results of the subroutine, respectively.

The filter will hold its coefficients ($b_0 + b_1 + \ldots$) that are five single-precision floating-point numbers. It is convenient to keep these state variables in a single C language `struct`, holding filter coefficients in each time step of operation. The filtering loop should only **get one value at each time and generate one output**. **At each time, a new input must be added to the filter and the oldest one deleted.** This technique is called *Moving Average*.

You should ensure that the operation of the filter is correct for all operations of inputs and state variables, including when there are arithmetic conditions such as overflow.

ARM CALLING CONVENTION

In assembly, parameters for a subroutine are passed on the memory stack and via internal registers. In ARM processors, the scratch registers are R_0:R_3 for integer variables and S_0:S_3 for floating point variables. Up to four parameters are placed in these registers, and the result is placed in R0 - R1 (S0 – S1). If any parameter requires more than 32 bits, then multiple registers are used. If there are no free scratch registers, or the parameter requires more registers than remain, then the parameter is pushed onto the stack. In addition to the class notes, please refer to the document ***"Procedure Call Standard for the ARM Architecture"***, especially its sections **5.3-5.5**. This particular order of passing parameters is eventually a convention applied by specific compilers. Please be aware that the several different procedures calling and ordering conventions exist beyond the one used here, but this procedure call convention is standardized by ARM.

Initial values

For the purposes of this lab, we will initialize the values $b_0 = 0.1$, $b_1 = 0.15$, $b_2 = 0.5$, $b_3 = 0.15$, $b_4 = 0.1$. The initial value of $\boldsymbol{x}$ is the first element of your input vector.

Performance Evaluation against C

You are required to write the same subroutine described in the previous part in the C language. Then you are to **call both the C and assembly subroutines from main**. You should compare the performance between the two implementations and present analysis on execution time differences. (Use the time measurement features in Keil). The function must get a value N to define the order of the filter. But for demonstration, N = 4 (means 5 coefficients). This is the only difference between

Assembly and C implementations. Moreover, create a FIR filter using CMSIS-DSP library by passing the same coefficients to the appropriate function and compare the performance with your Assembly and C functions. Regardless to say, **the output of all three implementations should match** when functions have the same coefficients. Similar to the assembly part, the prototype of your C-function should look as follows:

```
int FIR_C(float* InputArray, float* OutputArray, FIR_coeff*
coeff, int Length, int Order)
```

Here:

- ***InputArray*** is the array of measurements
- ***OutputArray*** is the array of values obtained by filter calculations over the input field
- ***The coeff struct*** contains the coefficients of filter
- ***Integer Length*** specifies the length of the input data array to process
- ***Order*** specifies the order of the filter (N).

The function return value encodes whether the function executed properly (by returning 0) or the result is not a correct arithmetic value (e.g., it has run into numerical conditions leading to NaN)

The deliverables of this part are:

1. An assembly based 1D-FIR filter subroutine "`FIR_asm`"
2. A C based FIR filter functions "`FIR_C`"
3. A CMSIS-DSP based FIR filter functions "`FIR_CMSIS`"
4. A **C-based test workbench** to call all functions and test for correctness and speed

Part II– CMSIS-DSP

To assess the properties of the FIR filter tracking of the input stream, you will add three additional operations at the end:

A. Subtraction of original data stream and data obtained by filter tracking.
B. Calculation of the standard deviation and the average of the difference obtained in A.
C. Calculation of the correlation between the original and tracked vectors.

For each of the above operations, you will implement the required function twice. The first in your own C implementation and then another time using the ***CMSIS-DSP*** library functions. You are required to profile the code and measure the execution speed of your own subroutines (in C) against the library subroutines.

Function Requirements for all parts

1. Your code should not use registers/variables unnecessarily whenever you can overwrite registers/variables you do not use anymore.
2. Your code should have minimum code density; that is; it should consume minimum memory footprint.

3. You may use the stack for passing parameters if needed (Assembly part)
4. Your code should run as fast as possible. You should optimize beyond your initial crude implementation.
5. Your code should make use of modular design and function reuse whenever possible
6. Codes will be compared against each other for the above criteria during demo time. Those who achieve best results will ***get the highest demo grades.***
7. The subroutine should be robust and correct for all cases. Grading of the experiment will depend on how efficiently you solve the problem, with all the corner cases (if any) being correct.
8. All registers specified by ARM calling convention are to be preserved, as well as the stack position. It should be unaffected by the subroutine call upon returning.
9. The calling convention will obey that of the compiler.
10. The subroutine should not use any global variables besides any that we have specified (if any).

Reference Material and Required Reading

- Doc_01 - Cortex-M4 Devices - Generic User Guide
- Doc_02 - Cortex-M4 Programming Manual
- Doc_04 - The Definitive Guide to ARM Cortex-M3 and Cortex-M4 Processors, 3rd Edition (Optional)
- Doc_07 - Procedure Call Standard for ARM Architecture
- Doc_08 - ARM® and Thumb®-2 Instruction Set
- Doc_09 - Vector Floating Point Instruction Set
- Doc_16 - Debugging with Keil
- Doc_17 - Introduction to Keil 5.xx
- Doc_24 - Doxygen Manual 1.8.2 (Optional)
- Doc_25 - ProGit 2nd Edition (Optional)

Demonstration and Documentation

On demonstration day, you will be provided by a header file ***test.h*** containing the test array ***testArray[]*** of a known length**.** Your code should generate the correct output for that specific array.

The demonstration involves showing your source code and demonstrating a working program. You should be able to call your subroutine several times consecutively. You should be able to explain what every line in your code does – there will be questions in the demo dealing with all the aspects of your code and its performance. The questions might regard the skeleton code to initiate and start the assembly code that we gave you in Tutorial 1; ask if you do not know!

It is by far the most efficient that you have the full documentation on your assembly code subroutine completed upon demonstrating the Lab 1, especially that future labs will require lots of documentation and additional code development on its own and following Doxygen documentation standard.

1. **Demos will be held on Friday, February 3th, 2017. This lab has a weight of 8 marks. There is NO report associated with this lab. The assembly code should be submitted for review.**

2. Demo slots will be announced and students will reserve their own preferred slot on Friday. Students should show up and be ready for demo 10 minutes prior to their reserved slot. **T.A.s will not wait for you** if you are late and will move on to the next ready group. You will demo on Monday with a huge penalty so make sure you are ready on time.

3. Early demos are allowed (**whenever possible, no guarantees**) upon appointment. **Contact the T.A. whose schedule falls on your preferred demo day to check if a demo is possible**. There is a cap on the max number of early demos per day.