ECSE444 Microprocessors

Winter 2021

Lab 1: Kalman Filter using Floating-Point Assembly Language Programming and its Evaluation with C and CMSIS-DSP

Objective

This exercise introduces the ARM Cortex and FP coprocessor assembly languages, instruction sets and their addressing modes. The ARM calling convention will need to be respected, such that the assembly code can be used with C programming language. The lab and a prior tutorial will introduce you to the STM32CubeIDE, including the compiler and associated tools. In the second part of the exercise, the code developed here will be used in a larger program written in C and the Cortex Microprocessor Software Interface Standard (CMSIS-DSP) application programming interface (API) that incorporates a large set of routines optimized for different ARM Cortex processors.

Hence, this lab consists of two components, each requiring a week to compete:

Part 1: Assembly language exercise – Kalman filter in one dimension
Part 2: Combining assembly/embedded C and optimizing performance; CMSIS-DSP

Background – ARM Calling Convention

In assembly and C, parameters for a subroutine are passed via stack or internal registers. In ARM processors, the registers R0:R3 are used for passing integer or pointer variables. Up to four parameters are placed in these registers, and the result is placed in R0 and R1. 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. Since we will be also dealing with the floating-point parameters on hardware that performs floating-point arithmetic, be aware of having the option of using either software or hardware floating-point linkage, depending on whether the parameters are passed via general purpose or floating-point registers. The objective here is to use the hardware linkage, hence the floating-point registers will be used for parameter and result passing.

In addition to the class notes, please refer to the document "Procedure Call Standard for the ARM Architecture", especially its sections describing The Base Procedure Call Standard. Other documents that will be of importance include the Cortex M4 programming manual, quick reference cards for ARM ISA and the (vector) floating point instructions, all available within the course online documentation. This particular order of passing parameters is applied by major compilers.

Using the STM32CubeIDE Integrated Development Environment Tool

To prepare for Lab 1, you will need to go through Tutorial 1, where you will learn how to create and define projects, including assembly code projects. The tutorial shows you how to let the tool insert the proper startup code for the given processor, write and compile the code, as well as provide the

basics of the program debugging.

Lab 1: Definition

You will develop the working assembly language code for single-variable Kalman filter that can be used in later exercises. The single-variable version avoids the use of matrix operations required for larger Kalman filters, and makes it amenable to an assembly code implementation, while it still allows experimenting with and appreciating the features of this filter.

Kalman Filter

Kalman filter is a state-based adaptive estimator of a physical process. Its estimation error is provably minimal for linear systems with Gaussian noise. It is the type of an *adaptive filter*, which is generally preferred to the fixed linear filters. The state space adaptation is performed by a sequence of discrete steps, during which the parameters of the filter change depending on the observed physical value, as well as the current state.

Kalman filter performs the adaptation by maintaining the internal state, consisting of the estimated value *x,* the adaptive tuning factor *k* and the estimation error, represented by its covariance *p*. To obtain these values, it requires the knowledge of the noise parameters of the input measurements and the state estimation, represented by their respective covariances *q* and *r*.

The high-level description of the Kalman filter code is given in the working python program in

```
class KalmanFilter(object):
        q = 0.0 # process noise variance, i.e., E(w^2)
        r = 0.0 # measurement noise variance, i.e., E(v^2)
        x = 0.0 # value
        p = 0.0 # estimation error covariance
        k = 0.0 # kalman gain

        def __init__(self, q, r, p=0.0, k=0.0, initial_value=0.0):
                self.q = q
                self.r = r
                self.p = p
                self.x = initial_value

        def update(self, measurement):
                self.p = self.p + self.q
                self.k = self.p / (self.p + self.r)
                self.x = self.x + self.k * (measurement - self.x)
                self.p = (1 - self.k) * self.p

                return self.x
```

Figure 1. While the code is fully functional, and it can be directly run within a larger (Python) program, it is used here as a compact high-level specification. Please note that only the update function is required in the assembly part of Lab 1. In the second part, when you include your assembly code with the C code, the initialization function will be needed. That part will be written in C. Note also that there will be differences in the code caused by different syntax and semantics of C, compared to Python. For instance, you will need to carefully specify the data types and include the function prototypes in the code to be able to correctly link the assembly and C code.

```
class KalmanFilter(object):
        q = 0.0 # process noise variance, i.e., E(w^2)
        r = 0.0 # measurement noise variance, i.e., E(v^2)
        x = 0.0 # value
        p = 0.0 # estimation error covariance
        k = 0.0 # kalman gain

        def __init__(self, q, r, p=0.0, k=0.0, initial_value=0.0):
                self.q = q
                self.r = r
                self.p = p
                self.x = initial_value

        def update(self, measurement):
                self.p = self.p + self.q
                self.k = self.p / (self.p + self.r)
                self.x = self.x + self.k * (measurement - self.x)
                self.p = (1 - self.k) * self.p

                return self.x
```

Figure 1: Python Code Definition of the Kalman Filter Class used in Lab 1

One interpretation of Kalman filter is that of tracking (or estimating the trajectory of) device for the input signal stream. At each time instance *i*, it generates the tracking consisting of the value vector *x*[*i*], that aims to reconstruct the original signal *measurement*[*i*]. An important feature of the Kalman filter is that the estimated value differs from the input by a value that has the statistical properties of the white noise. You will be asked later to inspect those properties using your additional statistical processing.

Part 1: Exercise

Write a subroutine *kalman* in ARM Cortex M4 assembly language that processes one measurement input to update the local state required for the process estimation. You should naturally use the built-in floating-point unit by using the existing floating-point assembler instructions

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

In your case, there is a need for one fixed-point (the address of the struct) detailed below and one floating-point input parameter (value of current measurement). The state of the Kalman filter is the result of your subroutine, but keep in mind that this can be achieved in a way that no output value is produced by the subroutine. This is effectively, the "call by reference" that you should be familiar with from the C programming.

The filter will hold its state as a quintuple (q, r, x, p, k) that are five single-precision floating-point

numbers. In the next lab, it will be convenient to keep these state variables in a single C language `struct`, holding Kalman filter state in each time step, consisting of:

```
float q; //process noise covariance
float r; //measurement noise covariance
float x; //estimated value
float p; //estimation error covariance
float k; // adaptive Kalman filter gain.
```

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

Function Requirements

1. All registers specified by ARM calling convention to be preserved, as well as the stack position should be unaffected by the subroutine call upon returning.
2. The calling convention will obey that of the compiler. Two registers, R0 and S0 contain the arguments, which are respectively the pointer to the state variables structure and the current measurement value. While there is no required result value, please note that the state variables pointed to by the pointer, will be modified.
3. No memory location should be used to store any intermediate data. Not only that it should be faster, but no memory allocation is needed that way.
4. The subroutine should be location independent. It should be able to run properly when it is placed in different memory locations.
5. The subroutine should not use any global variables.
6. The subroutine should be as fast as possible, but robust and correct for all cases of positive and negative numbers, as well as the overflows. Grading of the experiment will depend on how efficiently you solve the problem, with all the corner cases being correct.

Hints:

ARM Assembly Code

1. It helps to solve the problem conceptually at first, taking into account all corner cases, including the arithmetic overflows. Move to assembly code when the algorithm is well defined. (If you wish to implement the code in C, it can only help you here and also for Part 2, where you will be asked to produce several variations to the basic solution.)
2. Follow the examples of codes and instructions elaborated in classes, tutorials and material posted online for getting quickly to the working code. When optimizing for speed, some ARM Cortex instructions, such as those replacing directly bits between two words could help.
3. Document assembly code thoroughly and thoughtfully. It takes only a few hours for you to forget what each register holds and what implicit assumptions were used. It is useful to consider the assembler features that improve the coding style, such as the declaration of register variables, symbolic constants and similar.
4. Please keep in mind that the processor does not activate the floating-point unit on powerup. Following the example given in the tutorial, ensure that the floating-point unit is not bypassed.

Linking

1. When creating a new project, it is simplest to follow the created template. You will notice

that your IDE creates main.c and also the startup code in assembly. You can either
   a. modify the startup code to branch to `kalman` rather than __main for testing assembly code alone, prior to embedding into C main program, or
   b. you can call your assembly code from the main (all calling conventions need to be observed). Please note that you will need to declare as global (exported) the subroutine name in your assembly code, as well as follow up the syntax rules for GCC Assembly language.
2. For linking with C code that has main.c, no special measures should be needed, so option b) above is better and is readily expandable to the Part 2 extension.
3. If the linker complains about some other missing variable to be imported in startup code, you can either declare it as "dummy" in your assembly code, or comment its mention in the startup code.

Test Samples

Be prepared to use your and TA-provided test samples exercising all the relevant cases.

Part 1: Demonstration and Documentation

The demonstration involves showing your source code and demonstrating a working program. Your program will be placed at an arbitrary memory location. 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 the tutorial 1; ask if you do not know!

While the full report will be needed for Part 2, which will include the code developed here, it is by far the most efficient that you have the full documentation on your assembly code subroutine completed upon demonstrating the Part 1, especially that the second part will require lots of documentation and additional code development on its own.

Part 2 - Exercise

You will perform this exercise in several stages and record the results obtained at each step in your lab report.

First, you will augment the main C program to use the subroutine *kalman* repeatedly for an array of inputs.

Your subroutine call will appear in the C source code exactly as if the C subroutine is used. In doing this, you should follow all the usual rules of writing C programs, keeping in mind that the executable program consists of multiple independently compiled modules.

The filter state is a quintuple (q, r, x, p, k) kept in a single C language `structkalman_state`, consisting of:

```
float q; //process noise covariance
```

```
float r; //measurement noise covariance
float x; //estimated value
float p; //estimation error covariance
float k; // adaptive Kalman filter gain.
```

The operation of the program should be correct and as efficient as possible. To illustrate the operation of Kalman filter within your main program, Table 1 lists the state vector entries before and after the execution of first four iterations for the case where the initial value of x=5 and *measurement*(i)=i.

Table 1: Evolving Kalman Filter State Space

measurement(i)	Before 0	0	1	2	3	4
q	0.1	0.1	0.1	0.1	0.1	0.1
r	0.1	0.1	0.1	0.1	0.1	0.1
p	0.1	0.067	0.0625	0.06	0.06	0.06
x	5	1.67	1.25	1.71	2.5	3.43
k	0	0.67	0.625	0.62	0.62	0.62

Processing filtered data

In the second part of the lab, you will implement functions in embedded C to track the input variable, following the Kalman filter tracking algorithm. The code should execute on a fixed-length input value vector, consisting, say, of 100 values that will be provided by TAs.

1. To assess the properties of the Kalman filter tracking of the input stream, you will add four additional operations at the end:

    a. Subtraction of original and data obtained by Kalman 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.

    d. Calculation of the convolution between the two vectors.

You will then profile the code and compare the speed of your assembly code implementation of the Kalman filter with the equivalent C code subroutine.

You will also re-implement the C and assembly code for *kalman* using the CMSIS-DSP library with the aims of:

1. Cross-validating the C code against the CMSIS-DSP implementation – one lab group member can start preparing the CMSIS-DSP implementation from the beginning, so that your assumptions about the code get validated as well.

2. Comparing the performance and appreciating the utility of CMSIS-DSP.

3. Becoming proficient in CMSIS-DSP and capable of using any part of the CMSIS in future labs.

Some observations from these calculations are in place:

- Kalman filter convergence depends on the initial values. The closer the *x* is to the measurement, the smaller the need for correction.

- By deviating from C programming practices, you risk passing the wrong parameters, so please be aware of that

Function Requirements

Design a C function:

```
int Kalmanfilter(float* InputArray, float* OutputArray, kalman_state* kstate, int Length)
```

Here: -InputArray is the array of measurements
-OutputArray is the array of values x obtained by Kalman fiter calculations over the input field (cf. Table 1)
-The kstate struct contains the Kalman filter state that is readily available for your debugging
-Integer Length specifies the length of the data array to process

The output 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)

1. The function *KalmanFilter* should call your 'kalman' subroutine wherever appropriate. Unnecessary operations should be avoided.
2. Use the ARM calling conventions for later inclusion into a C program.
3. Upon returning from subroutines, the registers and stack should be untouched.
4. The subroutine should be location independent.
5. The subroutine should be robust, correct and then fast.

The idea is that the C function will be the interface to the test harness provided by TAs, and with the same interface (often referred to as a *wrapper*) you can call either the assembly, plain C or the CMSIS-DSP code.

Useful Notes

In realizing your code, you are encouraged to use good C coding practices, such as the use of conditional compiling for retargeting to different execution cases given above. Feel free to consult our online tutorial on embedded C, found in the Documentation part of the Content.

The Debugger and Profiler

While developing your code, you will spend substantial time using the debugger. Please follow the instruction from Tutorial 2 to ensure that the compilation for the simulator (and the debugger running on simulated code) compiles properly.

The profiler will be a useful tool when the code is bug free and you want to assess and improve the performance of your code.

Experimental Results to Report

You are asked to perform the following experiments and include the results in your report.

1. Run the program and record the execution for a trace of your choice that nicely illustrates the properties of Kalman filter, such as the convergence towards the input stream and the statistical properties of the deviation to the input,
2. Explicitly calculate in your program the difference to the input stream, standard deviation and average of that difference, as well as the correlation and convolution between the two streams.
3. Using CMSIS-DSP, perform the same functions as in 2.
4. Using code profiling, report the running times for all the procedures with your code and CMSIS-DSP. You should get the results as fast as possible.
5. Rewrite your core Kalman filter code in plain C as well as in C using the CMSIS-DSP. Keep in mind that it is possible to obtain the "scalar" version of CMSIS-DSP when needed.
6. Report the profiling data that compares the three versions of Kalman filter core (assembly, plain C and CMSIS-DSP).
7. Summarize the lessons learned on the use of assembler, C and CMSIS-DSP by selecting the suitable profiling data.
8. Finally, run the code on the actual processor and use the debugger to inspect and modify the program variables while the code is running. Summarize in the report on the rules: which variables can be watched, modified and whether and how that can be done without stopping the processor.

Final Report

Once you have all the parts working, include all the relevant data to your report, which will be due first day in a week following the Lab 2 demonstration, but not later than 4 days after the demo. Please follow the report guidelines included in the lecture slides.