ECSE426 Microprocessor Systems

Fall 2012

Lab 1: Moving Average Filter using Assembly Language Programming

Preparation - Objective

This exercise introduces the ARM Cortex assembly language, instruction sets and their addressing modes. The ARM calling convention will need to be respected, such that the assembly code can later be used with C programming language. The lab and a tutorial prior to it will also introduce you to the Keil compiler and simulator and associated tool. In the follow-up exercises, the code developed here will be used in larger programs written in C and will be using as well the Cortex Microprocessor Software Interface Standard (CMSIS) application programming interface that incorporates a large set of routines optimized for different ARM Cortex processors.

Hence, this lab is the first in the sequence consisting of two components:

Lab 1: Assembly language exercise – moving filter

Lab 2: Combining assembly/embedded C and optimizing performance; CMSIS-DSP

Background

Calling convention

In assembly, parameters for a subroutine are passed on the memory stack and via internal registers. In the ARM processors, the scratch registers are R0:R3 for integer 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 5.3-5.5.

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. Further, for comprehensive examples, and the cases requiring more arguments, and with larger datatypes, you can perform a search for “calling convention” in Keil tool help.

This particular order of passing parameters is eventually a convention applied by specific compilers. Please be aware that the several different procedure calling and ordering conventions exist beyond the one used here, but this procedure call convention is standardized by ARM.

Using the Keil 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, specifically purely 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 a moving average filter that can be used in later exercises. The moving average filter outputs an average of the last *d* samples, which makes it amenable to an assembly code implementation, while it would still allow you to experiment with and use it in later experiments.

Moving average Filter

Moving average filter is a simple linear filter that performs the function of a lowpass filter, i.e., smoothing. The moving average filter can in general performs a weighted averaging, often done in a way that puts more weight to the more recent samples.

At each time instance *i*, it generates the output consisting of the value vector $x[i]$, that aims to smoothen the original signal $measurement[i]$ and its d-1 predecessors. You will be asked later to inspect the properties of the filter using your additional statistical processing as well as by employing the advanced features of MDK-ARM.

In this lab instance, you will need to carefully specify the data types and include the function prototypes in Lab 2 to be able to later correctly link the assembly and C code.

Lab 1 - Exercise

Write a subroutine *moving_average* in ARM Cortex M4 assembly language that at every instance processes one measurement input to update the average of the last *d* measurements. 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 16 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 floating-point output and one floating-point input parameter (value of current measurement). The depth of the averaging *d* is not meant to be a variable to your

subroutine, but you should strive to keep it as a symbolic variable that can be changed from outside, for instance your main code in C, which will be written in lab 2.

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. You should check for all the floating-point arithmetic flags. Please keep in mind that as the floating-point unit (FPU) is a co-processor, the arithmetic flags are different than those used in the main Cortex processor. Therefore, you will need to employ special assembly instructions that move the floating-point condition codes to the general-purpose registers.

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. The registers, R0 contains the arguments, which are in this case both the input and the output: the input is the current measurement value, and the output is the filter output. You should provide flexibility in the depth *d* of the
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 except the filter depth *d*.
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 Lab 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 better ARM Cortex instructions could exist, such as those replacing directly bits between two words.
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, include the floating-point activation code at a proper place in the code.

Linking

1. When creating a new project, it is best to include the startup code, for which the tool will ask you whether to include it. Then, modify that code to branch to `moving_average`

rather than __main for testing assembly code alone, prior to embedding into C main program. Please note that you will need to declare as exported the subroutine name in your assembly code.
2. 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 it's mention in the startup code.
3. For linking with C code that has main, none of the above two measures are needed.

Test Samples

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

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 Lab 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 Lab 1, especially that Lab 2 will require lots of documentation and additional code development on its own.