Module 3

Lab 3: ARM Cortex M Architecture

Lab 3: ARM Cortex M Architecture

3.0 Objectives

The purpose of this lab is to introduce the architecture of the Cortex M.

1. You will learn about registers, RAM, and flash ROM.
2. You will write an assembly function with input and output parameters, which includes conditional and arithmetic operations.
3. You will learn debugging techniques like single stepping, breakpoints, and watch windows.
4. You will use an automated test approach called black-box functional testing to verify your algorithm is operating properly

Good to Know: We will be programming the robot challenge in C. However, the compiler converts the C code into assembly code. It is this low-level code that actually runs on the MSP432, which is a Cortex-M microcontroller. In this lab, you will experience some of the details of how the microcontroller executes software. Knowing these low-level details will make you a better high-level software developer.

3.1 Getting Started

3.1.1 Software Starter Projects

Look at these three projects:
SimpleProject_asm (a simple project that implements a random number generator),
LinearInterpolation_asm (an implementation of sine), and
Lab03_Assembly (starter project for this lab)

3.1.2 Student Resources (in datasheets directory-Links)

spmu159a.pdf , Cortex-M3/M4F Instruction Set

3.1.3 Reading Materials

Volume 1 Section 1.7, Chapter 3, and Section 5.3
Embedded Systems: Introduction to the MSP432 Microcontroller",
or
Volume 2 Sections 1.1, 2.1, and 2.5
Embedded Systems: Real-Time Interfacing to the MSP432 Microcontroller",

3.1.4 Components needed for this lab

Quantity	Description	Manufacturer	Mfg P/N
1	MSP-EXP432P401R LaunchPad	TI	MSP-EXP432P401R

3.1.5 Lab equipment needed (none)

3.2 System Design Requirements

Throughout the course you will acquire knowledge that will allow you to solve many robot challenges. The goal of this lab is to better understand how the computer performs tasks. We expect most students will complete the robot challenge programming in C. However, in this lab you will write a simple function in assembly.

Note: In the robot challenge you will use a distance measuring sensor unit composed of an integrated position sensitive detector and an IR sensor. This is also called a Proximity sensor which will be placed on the robot to measure distance.

In lab of module 4 you will develop a C function that converts raw ADC samples into a distance for the GP2Y0A21YK0F proximity sensor.

Let **n** be a 14-bit sample from the ADC (0 to 16383), and **D** be the distance in mm. The basic form of this nonlinear transfer relation is

$$\mathbf{D} = 1195172/(\mathbf{n} - 1058)$$

where 1195172 and -1058 are calibration coefficients to be empirically determined in the ADC lab (Module 15).

The maximum measurement distance for the sensor is 800 mm, so if the ADC value is less than 2552, your function should return 800. The C prototype for your function is

int32_t Convert(int32_t n);

However, since you are writing the function in assembly, you must adhere to a programming standard, called **ARM Architecture Procedure Call Standard** (AAPCS). There are many components of this standard, but the ones relevant to this lab include:

- If there is one input parameter, it is passed in R0
- If there are two input parameters, they are passed in R0, R1
- If there are three input parameters, they are passed in R0-R2
- If there are four input parameters, they are passed in R0-R3
- If there is an output parameter, it is returned in R0
- The function can modify R0-R3, R12 freely
- If a function wishes to use R4-R11 then it must save and restore them using the stack.
- If a function calls another, then it must save and restore LR
- Functions must balance the stack

Adhering this standard will allow you to develop assembly code that can be called from C, and allow your C code to be called from assembly. In particular, the compiler will adhere to this standard when creating object code.

3.3 Experiment set-up

This lab uses the LaunchPad without any input/output hardware.

3.4 System Development Plan

3.4.1 Functions and debugging

In this lab section you will build and debug the **SimpleProject_asm** example. Using the debugger, observe the input and output parameters of the function while you single step through the main program.

Answer the following:

i) How are data passed into **Seed**?
ii) How are results passed back from **Rand**?
iii) What happens to the LR register when a function is called?
iv) How does a function return?
v) How does the software access global RAM?
vi) What is the difference between storing data in a register and storing it in global RAM?
vii) Where is the machine code stored?
viii) What do **.data** and **.text** mean?
ix) Where are the constants 1664525 and 1013904223 stored?
x) You can observe the variables M and n by placing their addresses into a **Memory Browser** window.
xi) Using the step-over command, execute the Rand function multiple times and observe the values in M and n. In particular, look at bit 0 of M; what pattern do you see in bit 0?

Next you will build and debug the **LinearInterpolation_asm** project. If you are unfamiliar with “linear interpolation”, do an internet search on the topic to better understand the math used in this project.

Using the debugger, place a breakpoint inside the **Sin** function, and use the debugger observe the values of the **registers** during one execution of the **Sin** function. From a programming theory standpoint, these registers are considered **local variables** for the function.

Answer the following:

i) Can you prove the three subtract instructions will never overflow, when calculating (Ix-x1), (y2-y1), and (x2-x1)?
ii) Can you prove the multiply instruction will never overflow, when calculating (y2-y1)*(Ix-x1)?
iii) Can you prove it will never divide by zero?
iv) Why do we use SDIV instead of UDIV for this function?

Observe how this main program tests the **Sin** function. We call the form of testing in main.asm **Black Box** functional testing, because the testing just sets inputs and observes outputs. In other words, we look at the outside of the software, and not probe any of the internal details of the function. Black box testing looks at the overall functionality of what software does without know of how it works.

3.4.2 Distance Conversion

Write an assembly function that converts raw 14-bit ADC data to distance in mm. Use **.field** statements to encapsulate the calibration parameters.

```
IRSlope  .field 1195172,32
IROffset .field -1058,32
IRMax    .field 2552,32
```

Lab 3: ARM Cortex M Architecture

You can use the **main** program delivered as part of the **Lab_Assembly** project to test your **Convert** function. Similar to **LinearInterpolation_asm**, this testing approach is **Black Box functional testing**. This test program contains 16 test cases (inputs and expected outputs). The expected results are plotted as Figure 1.

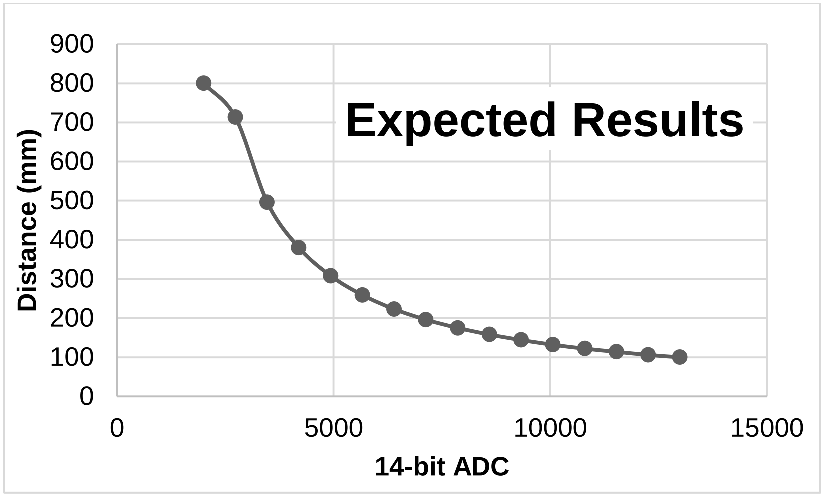

Figure 1. Expected results for the GP2Y0A21YK0F conversion function.

*Run **main** and compare your results with expected values. It is ok if your results differ by ±1 (which could be due to rounding).*

3.4.3. Observing Compiler-Generated Assembly Code

Revisit one of the C examples you ran as part of Lab 1. Within the debugger, open a Disassembly window. Single step the C code and observe the actual instructions

3.5 Troubleshooting

Convert doesn't work:

- Using **main**, find an input value that does not work, write a simple main program that calls your function with just that input, and single step your program comparing your internal calculations with expected values. Observing internal values is called **white-box** testing.
- If you are still having bugs, consult with your instructor and/or fellow students. You may be interpreting the problem in a different way as the testing procedure.

3.6 Things to think about

In this section, we list thought questions to consider after completing this lab. These questions are meant to test your understanding of the concepts in this lab. The goal of this module is for you to know enough assembly language to be able to interpret the machine-executable code generated by the compiler.

- What information do we store in ROM? Why?
- What information do we store in RAM? Why?
- What information do we store in R0-R12 registers? Why?
- How are R4-R11 different from R0-R3, R12?
- How is the LR used?
- How is the SP used?
- How is the PC used?
- How do functions work? Input parameters? Return parameter?
- Can you prove the (**n** – 1058) subtraction never overflows?
- Can you prove the division never attempts a divide by zero?
- Using integer division, what is the result of 1/**n** for any values of **n** greater than 1? This error (loss of information) is called **dropout**.
- The input is a 14-bit number (0 to 16383), but the output is only a 10-bit number (0 to 800). This reduction of four bits is a mild form of **dropout**. How could you have reformulated the problem to have less dropout?
- Notice that **SimpleProject_asm** project uses just one source file, while **LinearInterpolation_asm** **Lab_Assembly** projects use two source files. How are these two files used? What is the advantage of separating the implementation software from the testing software?
- List the debugging techniques used in this lab.

Lab 3: ARM Cortex M Architecture

3.7 Additional challenges

In this section, we list additional activities you could do to further explore the concepts of this module. You could extend the system or propose something completely different. For example,

- Consider exhaustive testing that tries every possible 14-bit input from 0 to 16383. How would you generate the test cases? How would you change the main program? What are the advantages of exhaustive testing?
- The robot can have multiple proximity sensors. Redesign the **Convert** function to handle three sensors, where each sensor has a unique set of three calibration coefficients (IRSlope IROffset IRMax).
- Use the debugger to estimate the time it takes to execute your **Convert** function.
- The Cortex M supports floating point arithmetic. Implement a floating point version of the function and develop a means to test it. Compare the accuracy and execution times for the two versions.

3.8 Which modules are next?

We will use the next few labs to create components we will need to control the robot. The input/output are an important component of an embedded system. The following modules will build on this module:

Module 4) Introduce C and develop some functions needed for the robot.
Module 5) Begin construction of the robot, including battery and voltage regulation
Module 6) Learn how to input and output on the pins of the microcontroller
Module 7) Study finite state machines as a method to control the robot
Module 8) Interface actual switches and LEDs to the microcontroller. This will allow for more inputs and outputs increasing the complexity of the system.

3.9 Things you should have learned

In this section, we review the important concepts you should have learned in this module:

- Understand how the processor uses registers during execution
- Discover the differences between RAM and ROM and how the software uses each.
- Perform arithmetic calculations in assembly with addition, subtraction, multiplication, and division
- Understand how constants are stored on the microcontroller
- Make decisions with conditional branch assembly statements
- Use the debugger to single step and visualize variables
- Perform functional testing