

# ENSC 254 – Lab 1

May 10, 2019

Copyright © 2019 Craig Scratchley

## 1 Introduction

In this lab, we are getting our first experience with the ARM architecture and will use the Keil  $\mu$ Vision simulation tools. We start by using the ARM Architecture Reference Manual (the so called ARM ARM) to determine the ARM machine code (i.e. numeric code) for a symbolic program and then execute the machine code using a simulator.

## 2 Machine Code for ARM

In class, we discussed CPU operation using a simple model 8-bit CPU. In this lab, we translate this work to a real 32-bit architecture from ARM. We are using a simulation tool, included with the Keil  $\mu$ Vision software, loaded with a 32-bit ARM 7TDMI-S (Big Endian) CPU simulator, which uses the ARMv4 instruction set (I will explain Big Endian later in class, but you will be happy to know that Big Endian makes things simple for us to work with). The 7TDMI-S is an older ARM CPU that is covered extensively in the text book *ARM Assembly Language* by Hohl and Hinds, the 2<sup>nd</sup> Edition of which can be viewed online for free from the library (the 1<sup>st</sup> edition is perhaps even more useful if you want to buy a copy).

For labs in the second half of the course, we will use a physical board based on the newer ARM Cortex-A9 CPU. These later labs will build on the fundamental understanding of the 32-bit ARM architecture you develop using simulators.

### 2.1 Program

Using the reference provided in the Files area on Canvas for the ARM architecture, complete the following machine code to create an ARM version of the program below, similar to what was shown at the end of the last class. This program subtracts 3 by adding the additive inverse (2's complement) of 3. The 2's complement is generated by taking the 1's complement and the adding 1. Before proceeding, I highly recommend downloading the reference manual

ARMv5architectureReferenceManualAbridgedEdited.pdf) and opening it in a dedicated PDF reader that supports Bookmarks. I would recommend against reading the reference manual in a web browser. To get you started, you can find information about the *mov* instruction on page A4-68, easily navigated to by clicking on the PDF bookmark for section A4.1.35 under the heading "ARM Instructions". On that page, note that "SBZ" means "Should Be Zero". Also, to learn about <shifter\_operand>, I recommend looking at Section A5.1.1 on page A5-3. You can get close to there if you click on the "Addressing Mode 1..." hyperlink. Once you have completed the machine code for the first instruction, you might want to skip ahead to the section on simulation and start the simulator so that

you can see if you did the first instruction correctly. Make sure to come back and complete all the remaining instructions too.

Step	Machine Code	Instruction
0	E 3A _ _ 0 _ _	mov r2, #1 ;@ needed for 2's comp.
1	E _ _ _ _ _	mvn r1, #0 ;@ load -1
2	E _ _ _ 0 _ _	mov r0, #3 ;@ calculate r1 - 3 below
3	E _ _ _ 00 _	mvn r0, r0 ;@ take 1's comp
4	E 0 _ _ _ _ _	add r0, r0, r2 ;@ add 1 for 2's comp
5	E 0 _ _ _ _ _	add r1, r1, r0 ;@ complete the subtract'n
6	E 5C _ _ _ _	strb r1, [pc, #3] ;@ PC is R15 on an ARM
7	E _ _ _ 0 _ _	mov pc, #0 ;@ take us back to start

Keep the machine code safe because you will need to enter some of it into a quiz on canvas. It is a hassle to get the simulator discussed below to save the machine code you type in. Notice what a time-consuming process it is to generate machine code, even for such a simple program – this is why programming languages were developed!

## 3 Simulation

Now that we have created a program, it's time to test it. Instead of loading it on a physical CPU, we are instead using a simulator.

A simulator is a software-based model of the real CPU, allowing us to execute programs as if they were running on the physical hardware. In addition to saving us the hassle of loading the program onto a physical board, the simulator gives us easy access to the internals of the CPU, including registers, and the values in memory. As well, it allows us to easily reconfigure the system including changing the version of CPU on which the program is being run.

### 3.1 Simulation Tools

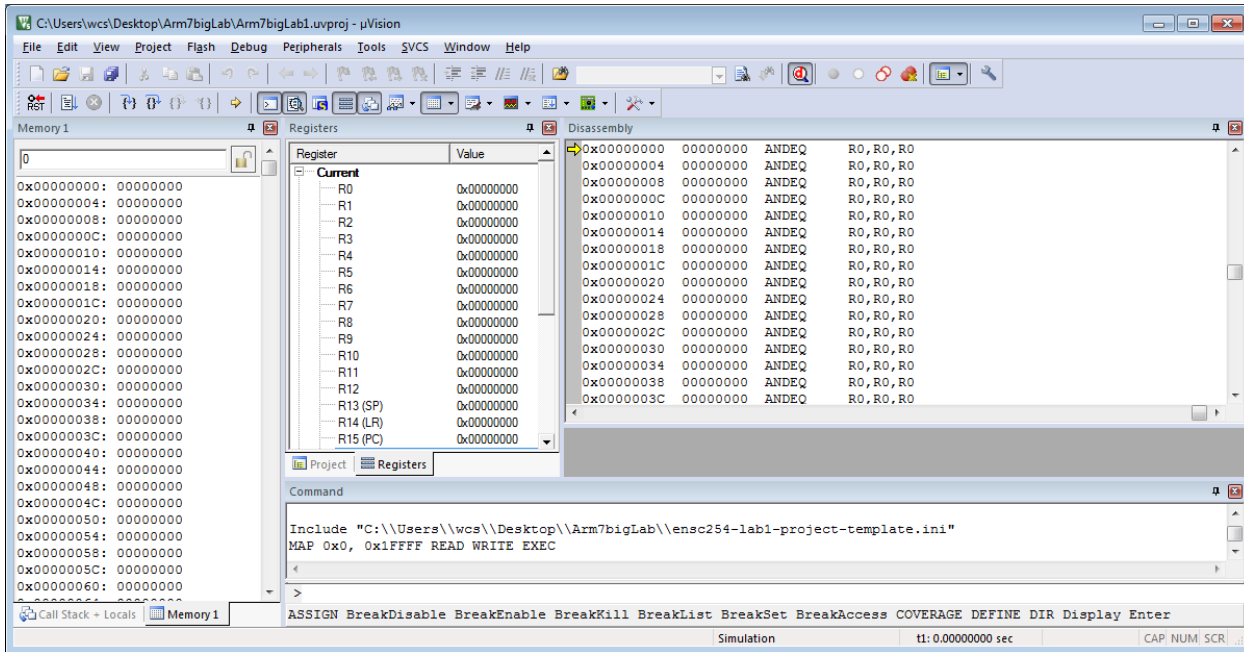
In these early labs for the course, we are using the simulation capabilities in Keil's  $\mu$ Vision IDE, a tool for embedded systems offered by ARM as part of their Microcontroller Development Kit (MDK).

### 3.2 Project Template

We have prepared a project template to allow you to immediately start testing your machine-language program. The template (*ensc254\_lab1\_project\_template.zip*) can be downloaded from Canvas. Unzip this into your desired directory, perhaps on the U: drive on ENSC Lab computers. Open the unzipped directory and double click on the file "Arm7bigLab1.uvproj". That should launch  $\mu$ Vision and open the project template.

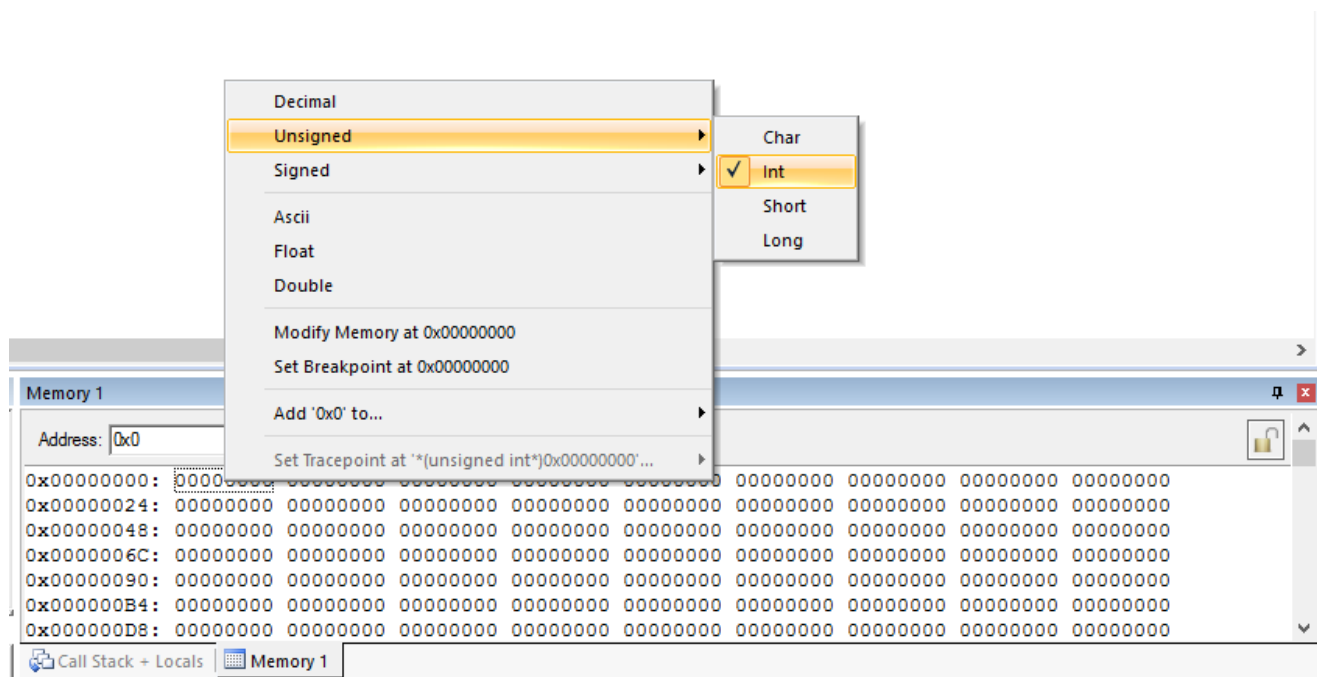
### 3.3 Starting Simulator

The simulation is started from the menu bar, from the appropriate icon (magnified “d”), or from the appropriate key sequence (Ctrl-F5). To use the menu bar, select **Debug** → **Start/Stop Debug Session**. The debug session opens as below. A small window will pop up warning us that the free version of MDK is limited in the amount of RAM that can be used. This is OK for our purposes.

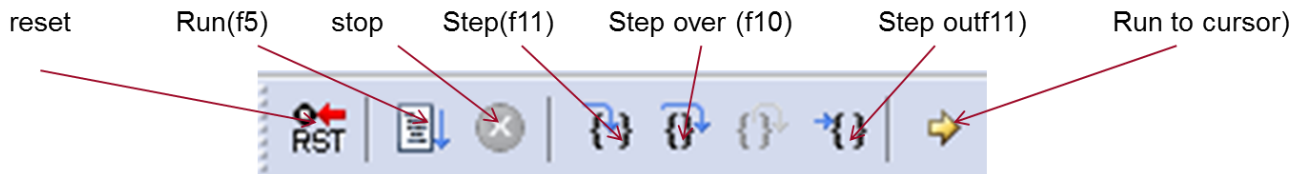


In the middle of the window, the current values of all the registers are displayed (and can be manually modified). As you have seen, on our ARM processor, the Program Counter (PC) is R15. The right pane shows the addresses, machine-code instructions at those address, and symbolic instructions as well.

On the left of the window, the memory display allows us to see the values stored in memory. If the display isn't already configured for 32-bit values, right-click and select **Unsigned Int**. Ensure **Decimal** remains unselected so that the values will be displayed in hexadecimal. As well, ensure you are displaying values starting from address 0x00000000.



To control execution of the ARM program on the simulator, the following buttons and keys can be used. While you can run the program perpetually, it is more illustrative at this point to step through the program. (“Step out” is the icon to the left of the one indicated below, and uses Ctrl+F11 as its key sequence. “Run to cursor line” is also to the left of where has been indicated below, and uses Ctrl+F10 as its key sequence. )



The above buttons, that you will find on the upper left of your window, are extremely important and you will find them in ANY integrated debugging environment. Keep in mind that same functionalities may be used on C sources in the same way

**RESET:** Will take the Program Counter back to the RESET value, thus restarting computation. In most cases, it will also reset the value of internal registers

**STEP:** Will step one line into the disassembled code (depending on your coding style, more than one disassembled lines may correspond to a C or assembly line in your code)

**STEP OVER:** Will step one line into the source code (C or assembly). If the line specifies a function the tool will compute the function in one shot and produce its results

### 3.4 Execution of Machine Code

In the project template we gave you, the program memory is zeroed. `0x00000000` is a valid instruction, interpreted as ARM instruction `andeq r0, r0, r0`. In the memory pane at the side of the screen, replace the zero values with the program you wrote in Section 2. You can enter one 32-bit word at a time, in hexadecimal format.

As you enter values in the memory pane, the values will be updated in the right-side pane. The automatically generated symbolic program with mnemonics will also update, and let you know if your machine code is what you intended.

Reset the CPU with the reset button shown above. If you don't reset the CPU, the first two instructions will not execute, at least not properly. This is because they were not fetched in properly into the CPU simulator when you entered the instructions in the memory pane. We will talk about this later. Step through the program pressing **F11** – you should see the registers updating towards the left side of the screen. Congratulations, your program is running! To restart the program, you can either press the reset (**RST**) button, or directly modify the Program Counter (*PC* or *R15* – they refer to the same thing as already mentioned).

You can also try modifying your program – for example, try using a different register to hold the value 1. Remember that if you modify the instruction in memory at the Program Counter or the next instruction, by default those modifications will not be properly fetched into the simulator for the first

executions of those instructions and you may therefore want to reset the simulated processor. Rerun the program, and you should see any changes reflected.

Go back to the original program (with original registers) shown on page 1. Do you have any theories why the output of the program ended up in memory where it did?

In our 32-bit ARM program, we can load any byte using immediate data. However, 32-bit ARM processors have different instructions to load 16-bit data and 32-bit data and, for those instructions, using immediate data to provide the input data limits us in the values we can use for the input data. Change the program so that it loads the input data from the two memory locations immediately after the last instruction. R1 should be loaded from the byte immediately after the last instruction, and R0 should be loaded from the following byte. Though for this program, which loads only bytes, this change is not needed to resolve a limitation on input values, the change does allow us to put the input bytes next to each other in memory outside of the range of addresses used by program instructions, and such placement may have other advantages. Write the new program here:

Step	Machine Code	Instruction
0	E 3A _ _ 0 _	mov r2, #1
1	<b>E 5</b> _ _ _ _	<b><i>ldrb r1, [pc, #____]</i></b>
2	<b>E 5</b> _ _ _ _	<b><i>ldrb r0, [pc, #____]</i></b>
3	E _ _ _ 00 _	mvn r0, r0
4	E 0 _ _ _ _ _	add r0, r0, r2
5	E 0 _ _ _ _ _	add r1, r1, r0
6	E 5C _ _ _ _	strb r1, [pc, #3] ;@ PC is R15 on an ARM
7	E _ _ _ 0 _	mov pc, #0

Test the program by inserting testing bytes immediately after the last instruction. Does your program load those testing bytes properly?

Make sure you record all your machine code, as I will ask you to enter some bytes from the machine code into a quiz on canvas before next week.

Just like immediate data can limit one when trying to load 16 or 32-bit values, moving immediate data into the PC can limit where the next instruction intended to be executed can be located. One technique to get around this limitation is to place the address of the next instruction intended to be executed in a word of memory close enough to where the mov instruction was and, replacing the mov instruction, to load that stored address into the PC using the same PC-relative addressing we just used to load the subtrahend into R0. We will cover this technique in the next lab but you can try that now if you want, in which case you will want to look into the LDR instruction and make an instruction along the lines of

ldr pc, [pc, #\_\_\_\_]