ENSC 254 – Lab 1

Clarifications on May 21, 2021

1 Introduction

In this lab, we are getting our first experience with the ARM architecture and will use the Keil µVision 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 versions of 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 µVision 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 at this point. We will work with Little Endian hardware later in the course). The 7TDMI-S is an older ARM CPU that is covered extensively in the text book *ARM Assembly Language* by Hohl and Hinds, the 2nd Edition of which can be viewed online for free from the library (the 1st edition is perhaps even more useful if you want to buy a copy). The primary textbook for the course focuses on the ARMv7A instruction set, which is a superset of the ARMv4 instruction set. Note: it's easy to get the ARM 7 processors mixed up with the ARMv7 instruction set, but they refer to different things. Remember that ARM 7 processors use the ARMv4 instruction set.

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 in class this week. 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. This reference manual documents everything in the ARMv4 instruction set, and provides further information regarding extensions added in the ARMv5 instruction set. To get you started, you can find information about the *mvn* and *mov* instructions on pages A4-82 and A4-68, easily navigated to by clicking on the PDF bookmark for sections A4.1.41 and A4.1.35 under the heading "ARM Instructions". On those pages, note that "SBZ" means "Should Be Zero". Also, to learn about

[illegible]

uncondit

<shifter_operand>, I recommend looking at Section A5.1.1 on page A5-3. You can get close to that section if you click on the "Addressing Mode 1…" hyperlink on your current page.   Once you have completed the machine code for the first instruction, you might want to skip ahead to the section on simulation below in this document and start using the simulator so that you can see if you encoded the first instruction correctly.   Make sure to come back and complete all the remaining instructions too.

0 1 2 3

31-28

4

Step	Machine Code	Instruction
		`wl  equ   4       ;@ bytes (Word Length is 4)`
0	E 3E 0 0 0 00	`    mvn   r0, #0      ;@ load 255 or -1`
1	E 3A 0 2 0 01	`    mov   r2, #1      ;@ needed for 2's comp.`
2	E 3A 0 1 0 03	`    mov   r1, #3      ;@ calculate r1 – 3 below`
3	E 1E 0 1 00 1	`    mvn   r1, r1      ;@ take 1's comp`
4	E 08 1 1 00 2	`    add   r1, r1, r2 ;@ add 1 for 2's comp`
5	E 08 0 0 00 1	`    add   r0, r0, r1 ;@ complete the subtract'n`
6	E 5C F 0 003	`    strb  r0,[pc,#3] ;@ PC is R15 on an ARM`
7	E 3A 0 F 0 08	`    mov   pc, #2*wl  ;@ take us back to step 2`

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 an ARM machine-language 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

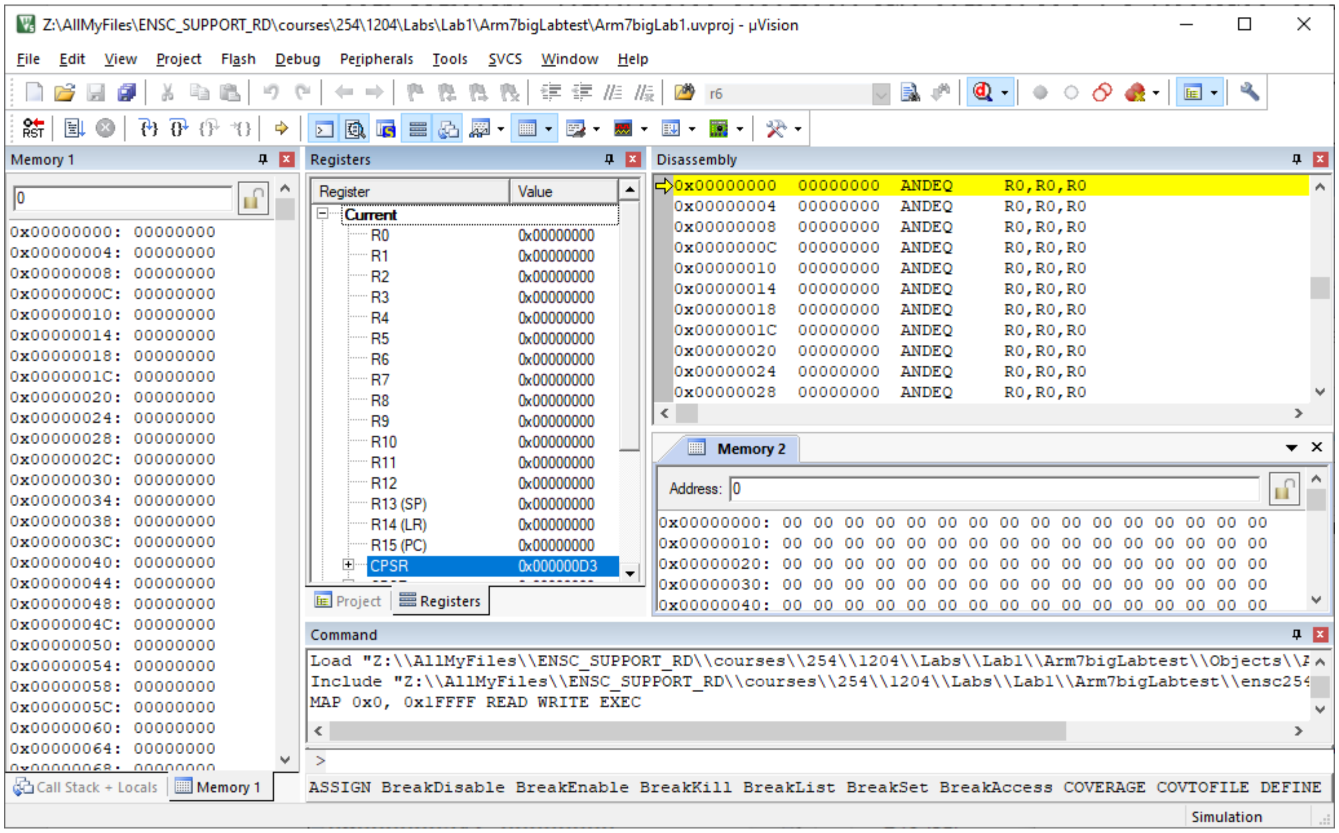
In these early labs for the course, we are using the simulation capabilities in Keil's µVision IDE, a tool for embedded systems offered by ARM as part of their Microcontroller Development Kit (MDK).

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. On Windows computers, open the unzipped directory and double click on the file “Arm7bigLab1.uvproj”. That should launch µVision and open the project template. On Mac computers, you may need to use the “Open Project…” item in the Project menu from within the uVision program.

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.



Z:\AllMyFiles\ENSC_SUPPORT_RD\courses\254\1204\Labs\Lab1\Arm7bigLabtest\Arm7bigLab1.uvproj - µVision
File Edit View Project Flash Debug Peripherals Tools SVCS Window Help
r6
Memory 1
Registers
Disassembly
Memory 2
Address: 0
Command
Call Stack + Locals
Memory 1
Project
Registers
Simulation

```
0x00000000: 00000000
0x00000004: 00000000
0x00000008: 00000000
0x0000000C: 00000000
0x00000010: 00000000
0x00000014: 00000000
0x00000018: 00000000
0x0000001C: 00000000
0x00000020: 00000000
0x00000024: 00000000
0x00000028: 00000000
0x0000002C: 00000000
0x00000030: 00000000
0x00000034: 00000000
0x00000038: 00000000
0x0000003C: 00000000
0x00000040: 00000000
0x00000044: 00000000
0x00000048: 00000000
0x0000004C: 00000000
0x00000050: 00000000
0x00000054: 00000000
0x00000058: 00000000
0x0000005C: 00000000
0x00000060: 00000000
0x00000064: 00000000
```

Register	Value
Current	
R0	0x00000000
R1	0x00000000
R2	0x00000000
R3	0x00000000
R4	0x00000000
R5	0x00000000
R6	0x00000000
R7	0x00000000
R8	0x00000000
R9	0x00000000
R10	0x00000000
R11	0x00000000
R12	0x00000000
R13 (SP)	0x00000000
R14 (LR)	0x00000000
R15 (PC)	0x00000000
CPSR	0x000000D3

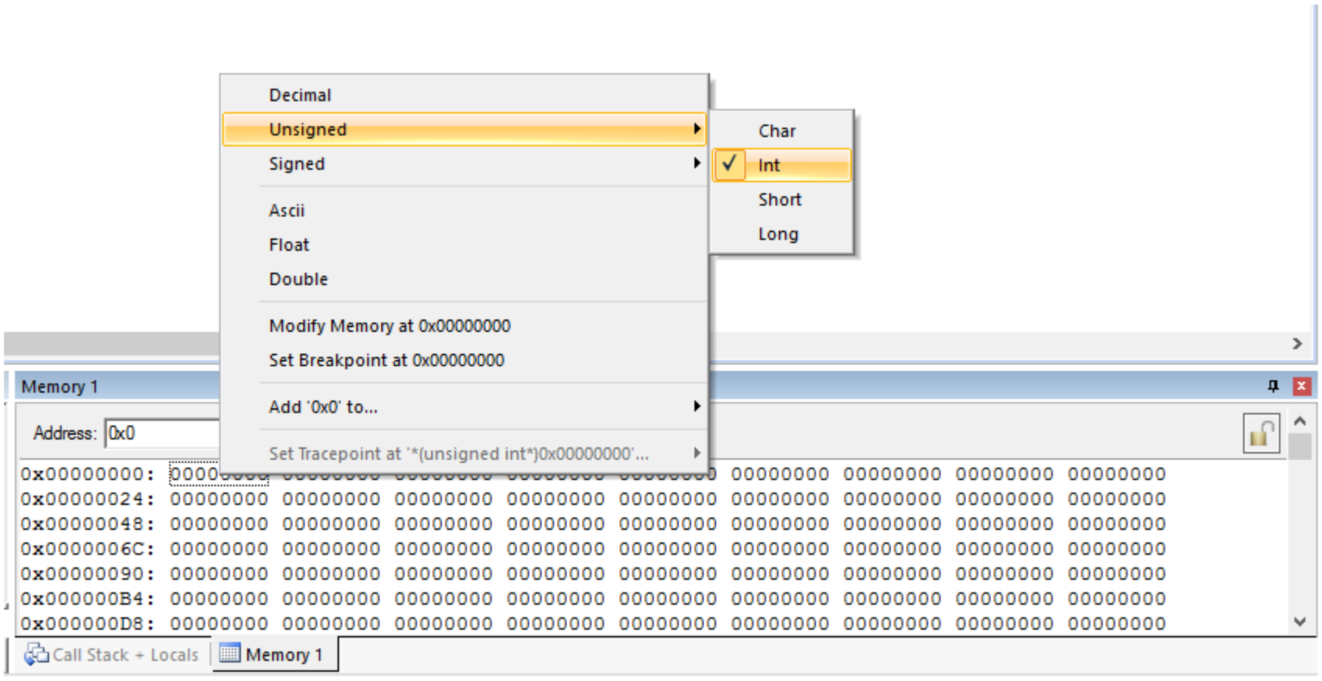
```
0x00000000  00000000  ANDEQ    R0,R0,R0
0x00000004  00000000  ANDEQ    R0,R0,R0
0x00000008  00000000  ANDEQ    R0,R0,R0
0x0000000C  00000000  ANDEQ    R0,R0,R0
0x00000010  00000000  ANDEQ    R0,R0,R0
0x00000014  00000000  ANDEQ    R0,R0,R0
0x00000018  00000000  ANDEQ    R0,R0,R0
0x0000001C  00000000  ANDEQ    R0,R0,R0
0x00000020  00000000  ANDEQ    R0,R0,R0
0x00000024  00000000  ANDEQ    R0,R0,R0
0x00000028  00000000  ANDEQ    R0,R0,R0
```

```
0x00000000: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0x00000010: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0x00000020: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0x00000030: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0x00000040: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
```

```
Load "Z:\\AllMyFiles\\ENSC_SUPPORT_RD\\courses\\254\\1204\\Labs\\Lab1\\Arm7bigLabtest\\Objects\\
Include "Z:\\AllMyFiles\\ENSC_SUPPORT_RD\\courses\\254\\1204\\Labs\\Lab1\\Arm7bigLabtest\\ensc254
MAP 0x0, 0x1FFFF READ WRITE EXEC
>
ASSIGN BreakDisable BreakEnable BreakKill BreakList BreakSet BreakAccess COVERAGE COVTOFILE DEFINE
```

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 real ARM processor, the Program Counter (PC) is R15. The top-right pane shows the addresses, machine-code instructions at those address, and symbolic instructions as well. Note that at this point all the machine code instructions are 0x00000000 and that apparently decodes to an instruction with symbolic form “ANDEQ R0, R0, R0”.

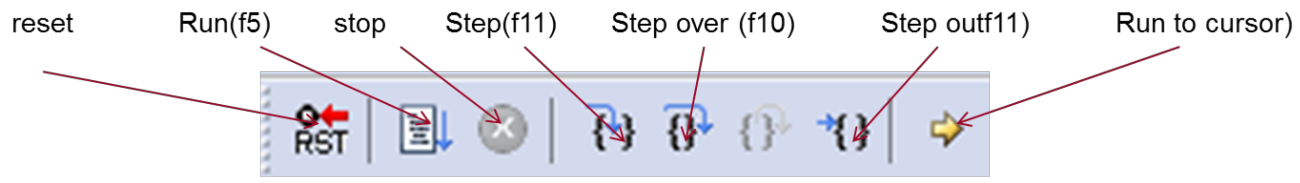
On the left of the window, the Memory 1 pane allows us to see values stored in memory grouped into 32-bit unsigned hexadecimal values. You may need to put your cursor in the address field at the top of the pane and hit enter in order to populate the pane with data. If you make a change and want to return to unsigned 32-bit values, right-click in the pane and select **Unsigned Int**. Ensure **Decimal** remains unselected if you want the values to be displayed in hexadecimal. As well, ensure you are displaying values starting from address `0x00000000`. (Pardon the change in geometry of the pane in the below figure.)



```
0x00000000: 0000                                                 00000000 00000000 00000000 00000000
0x00000024: 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
0x00000048: 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
0x0000006C: 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
0x00000090: 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
0x000000B4: 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
0x000000D8: 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
```

The Memory 2 or Memory 3 pane shows each memory location displayed as an individual 8-bit byte.

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. You won't need those functions in this lab, though.)



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 fictionalities 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 1 pane at the left side of the window, 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 1 pane, the values will be updated in the right-side panes. 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, any changes that you made to the instruction at the Program Counter (the instruction at address 0x0 if you haven't executed any instructions yet) and the following instruction will be ignored. This is because the instructions were not fetched into and decoded properly in the CPU simulator when you entered the instructions in a 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 and decoded 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, located in r1 towards the bottom 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. Use R2 as the base address register. 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:

$R0 \leftarrow M[21] \rightarrow 33$

Step	Machine Code	Instruction
		`wl   equ    4    ;@ bytes (Word Length is 4)`
0	E 3A 0 2 0 01	`     mov    r2, #1`
1	**E 5D 2 0 D20**	***`     ldrb   r0, [r2, #32]`*** −1
2	**E 5D 2 1 01F**	***`     ldrb   r1, [r2, #31]`*** 3
3	E 1E 0 1 00 1	`     mvn    r1, r1      ;@ take 1's comp`
4	E 08 1 1 00 2	`     add    r1, r1, r2 ;@ add 1 for 2's comp`
5	E 08 0 0 00 1	`     add    r0, r0, r1`
6	E 5C F 0 003	`     strb   r0,[pc,#3] ;@ PC is R15 on an ARM` 18
7	E 3A 0 F 0 08	`     mov    pc, #2*wl`

Test the program by inserting testing bytes immediately after the last instruction. Does your program load those testing bytes properly? $R1 \leftarrow M[20] \Rightarrow 32$

Since the input data has been placed immediately after the end of the program, it might make sense to use the Program Counter as the base register used to calculate the addresses from which the data will be loaded. That way, if the entire current program including the data is moved somewhere else in memory (perhaps to make room for more instructions at the top of the program), the program won't have to be changed in order to find the new addresses for the data. Perhaps the data are constants that belong with the program. Change the below program to load R1 and R0 using PC as the base address register.

"3 stage." "+8"

pc execute

R0 ← M[21] → 33 Dec →

R1 ← M[20] → 32 Dec →

Step	Machine Code	Instruction
		`wl   equ    4    ;@ bytes (Word Length is 4)`
0	E 3A 0 2 0 01	`     mov    r2, #1` C
1	E 5D F 0 015	`     ldrb   r0, [pc, #21]` pc=12
2	E 5D F 1 010	`     ldrb   r1, [pc, #16]` pc=16 10
3	E 1E 0 1 00 1	`     mvn    r1, r1      ;@ take 1’s comp`
4	E 08 1 1 00 2	`     add    r1, r1, r2 ;@ add 1 for 2’s comp`
5	E 08 0 0 00 1	`     add    r0, r0, r1`
6	E 5C F 0 003	`     strb   r0,[pc,#3] ;@ PC is R15 on an ARM`
7	E 3A 0 F 0 08	`     mov    pc, #2*wl`

PC = 4

PC = 8

Does your program again load those testing bytes properly? If not, what is being loaded? If necessary, fill some memory locations after the output result with some hexadecimal values such as 01, 02, 03, 04, 05, 06, 07, 08, 09, 0a, 0b, 0c and run the program again. Does this help to figure out where data is being loaded from if the intended data was not loaded? Fix your program if needed and verify that the fix works.

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, 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, #___]