EENG260 (Microcontrollers) Lab 1:

Store and sum a series of integers, on the Tiva C evaluation board using ARM Assembly instructions

Tasks for this lab

- Create a project for this lab in Code Composer Studio (CCS).
- Import the partial main.asm file linked in the Reference Materials below into your project.
- Edit the main.asm file to make an assembly program that does the following:
    - Starting with the memory location 0x20000400 (a location in RAM outside of the space set aside for the stack), store the values 0 through 15 in one byte each, in ascending order.
    - Once all the values have been stored, go back through them, again in ascending order, and add all the values together. The results should be left in register R3 (the expected sum is 0x78).

Reference Materials

- ARM Assembly spnu118u.pdf
- CortexM_InstructionSet.pdf
- Partial main.asm file

The main.asm file

Start by creating a project for this lab, if you don't already have one set up from your work on Lab 0.

Next, you want to import the partial main.asm file listed in the Reference Materials above. Just right-click on your project, then select Import > "Import…". In the import wizard, select File System under the General category, then click Next. Find the main.asm file where you downloaded it, hit Finish, and it will show up under your project.

With the main.asm file imported, double-click on it in the Project Explorer panel to bring it up in a tab in the editing area to the right. Currently, there are only two assembly commands listed, but there are several more lines than that, which aren't all comments. Those are *assembler directives*, instructions intended for the CCS assembler rather than for execution as instructions on the microcontroller. In order, from the top of the file:

- .thumb tells CCS that the file from this point forward is an ARM Assembly (Thumb) program.
- ".global main" tells CCS that the label main is going to be used outside of just this one file (without which, if we tried to build the project, you would get errors preventing the build).
- The space I've marked with the comment "Your constants go here" is where you should use the following directives to store constants that will make your code easier to read:
    - .field tells the assembler that you want to store a value in a location in ROM referenced by the attached label, to be retrieved later by your program. You have to specify not only the value to store, but the size in bits of the data you want to store. For example,

you could (and likely will!) use the line “arrayLocation: .field 0x20000400,32” so you can use arrayLocation to load that value into a register using LDR.
    - .equ is very much like #define in C, where whatever label is attached to it is used at assembly time to swap-in the value defined for that label elsewhere in the program. For example, if you had a line “countSize: .equ 0x10”, then you could use #countSize as a constant for a MOV instruction (like, say, loading a register with a value to count down from), or for an offset on an LDR or STR instruction.
- .asmfunc tells the assembler that what follows is an assembly function. Not strictly necessary, but it never hurts to give the debugger some guidance. Also, note that the main: label will end up applied to the first item of assembly code that follows (which is what we want here).
- “Your code goes here” should be self-evident.
- The two lines that follow are the only actual assembly instructions included. The first line establishes the loop: label, pointing at a MOV instruction that does nothing useful (placing the contents of R0 in R0). The second line then branches to that loop label. This is just included to keep program execution within the confines of this file, to simplify debugging.
- .endasmfunc tells CCS that the .asmfunc we called out at the beginning of the file has come to an end.
- .end tells CCS that our .thumb instructions are at an end.

These are the only assembler directives you should need to use for the foreseeable future, but details on these and many more are available in the “ARM Assembly” document. As a side note, all of these assembler directives are specific to CCS. The textbook for this course was written assuming you would be using ARM’s Keil assembler, which uses a different format for assembler directives. If you need to translate from Keil to CCS, see Appendix 3 of the textbook.

Planning your program

We have a project set up, including the main.asm file that will hold our constants and code. We have a list of tasks our code needs to perform. Now, you need to work out what combination of assembly instructions will let you reach that goal. You need to store and retrieve values from memory, so that will involve LDR and STR instructions (and, since you’re working with byte-sized pieces of data, probably LDRB and STRB specifically). You will need to use ADD and SUB to generate the numbers you need to store, and the memory addresses you need to store those numbers at – and, since you’re likely to use loops with a B instruction of one sort or another to do that (possibly BNE or BEQ), don’t forget that you’ll likely need to use ADDS or SUBS if you want Program Status Register flags to update.

Work out what how you want your program to behave, whether that’s using pseudocode, a flowchart, or any other method you decide. When finished, make the changes you have planned out to the main.asm file, then build and debug your code.

Build and Debug

Once your code is ready, you then have to build your project, which compiles/assembles the various files in your project, then links/loads them to form a bootable batch of data for your microcontroller’s ROM. To start the process, click the hammer icon near the top of the screen. If there are any problems,

errors will show up in the console panel at the bottom of the window, which you will need to decipher and fix before you can continue (just like compiling your C programs in previous courses).

Once you get a complete build without any errors, you are ready to debug. First, connect your Tiva C board to your PC using the included USB cable (and the port to the right of the power switch), then turn on your Tiva C board. Assuming no driver issues at the PC, you can just hit the debug icon (looks like a bug, a couple of icons left of the build icon), and CCS will download your program to the board's ROM, then leave you at a breakpoint in a file named boot_cortex_m.c, paused. This file is just part of the CCS boot routine, and, for our purposes, can be safely ignored. If you're feeling lucky, you can just hit the Resume icon (looks like the "play" icon on many devices), and the program will continue at full speed until you hit the Suspend (pause) or Terminate (stop) icons. Alternately, if you're interested in what's happening at a particular point in your code, you can switch to that file, click on the line of interest, then right-click and choose "run to line" to advance that far in your program. Additional controls are there for stepping into/over code and setting breakpoints. While you are paused, you can view the contents of any register in the Registers tab in the upper-right pane, allowing you to verify that each command you wrote is doing what you intended it to do, as needed. You may also find the Memory Browser (available under the View menu) handy, if you need to verify that the values you intended to store actually got stored where you expected.

To turn in

Make a .zip file with the following contents and upload it to Canvas:

- A brief lab report, in Word or PDF format. Did your program work? What issues did you run into? What did you learn? Keep it under a page, if you can.
- A copy of your modified main.asm file. You can export your file to the file system by right-clicking on it in the Project Explorer pane and selecting Export.