Name:

Partner(s):

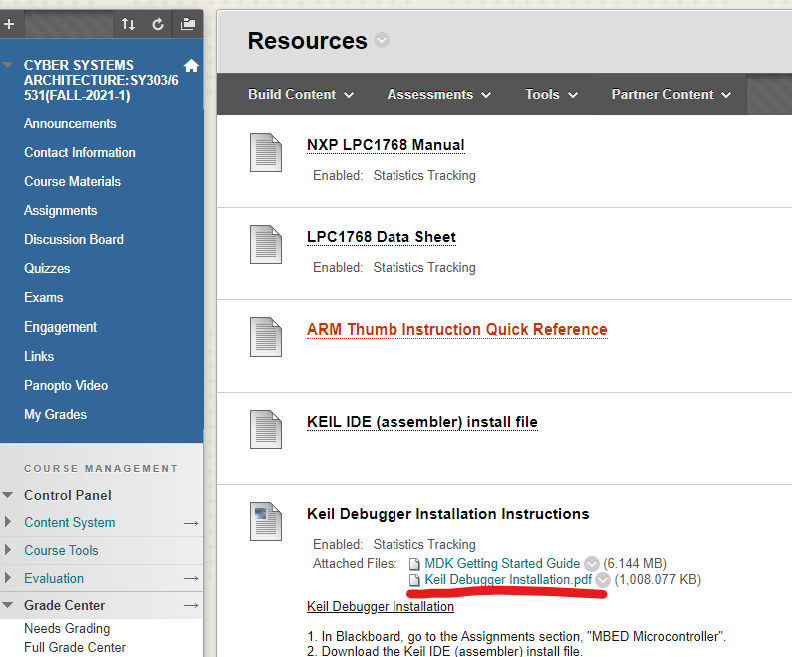
Section:

**References:** 1) [Getting started with MDK](https://www.keil.com/support/man/docs/mdk_gs/gs_MDK5_5_en.pdf)

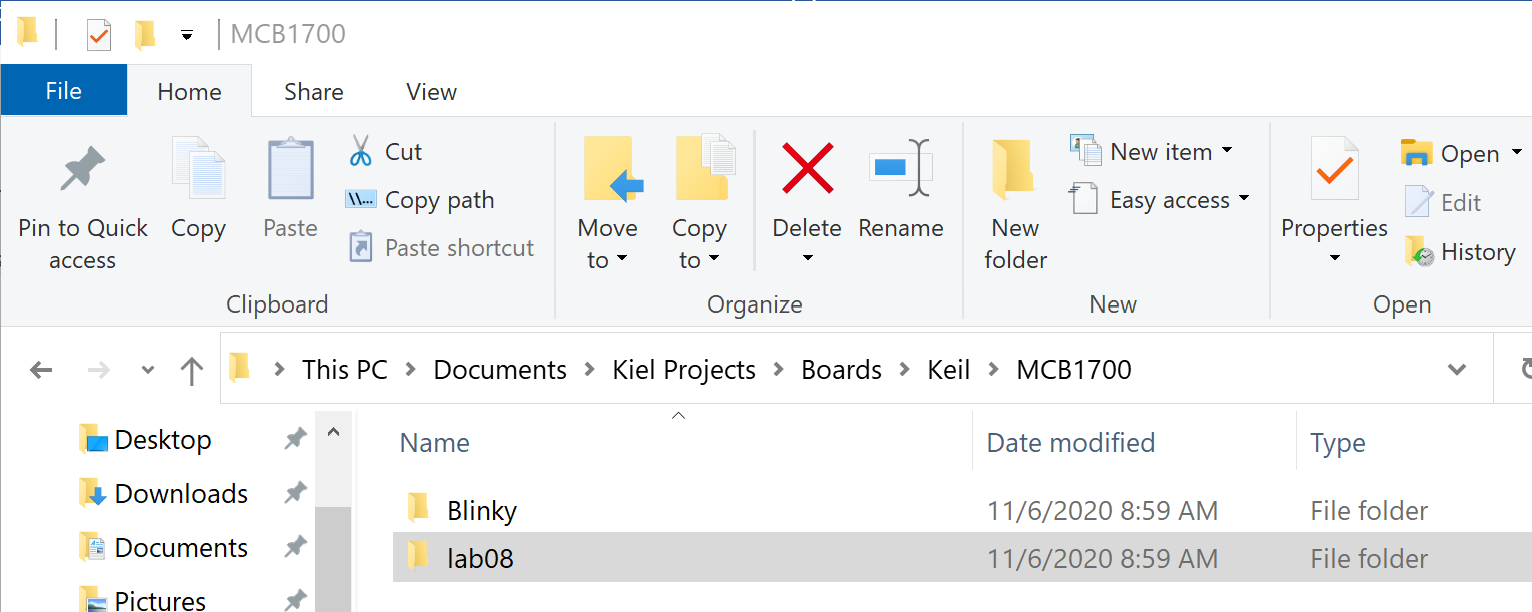
2) [LPC1 LPC176x/5x User manual](https://www.nxp.com/docs/en/user-guide/UM10360.pdf) (Sections 34.2.3, 34.2.4, and 34.2.5)

**Objective:** In this lab, you will use the debug simulator in the Keil Microcontroller Development Kit (MDK) µVision integrated development environment (IDE) to practice writing to registers and memory of our device.

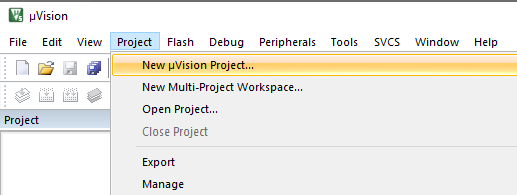
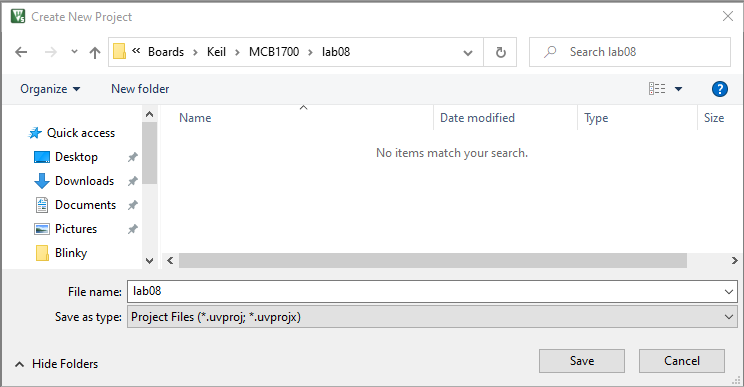
1. **Pre-Lab: Creating a new Assembly based Project** 
   1. Install Kiel, follow the instructions in Blackboard under /Assignments/MBED Microcontroller/Resources/Keil Debugger Installation Instructions.



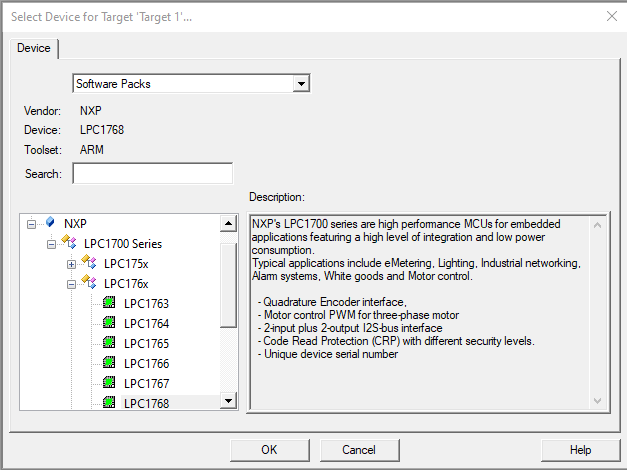
* 1. Find the folder where your Blinky project was saved, then make a new folder and name it “lab08”



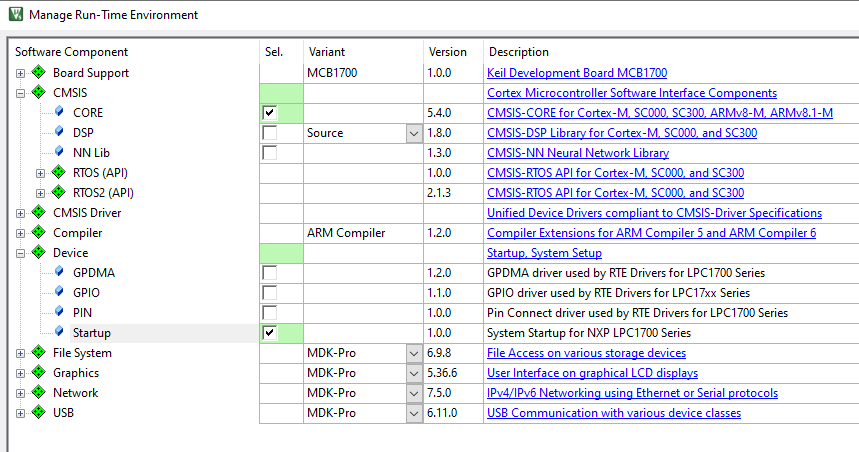
* 1. In uVision, close your Blinky project, if it is open, by selecting Project -> Close Project.
  2. Create a new project. Navigate to your lab08 folder and insert “lab08” as the project file name in that folder.

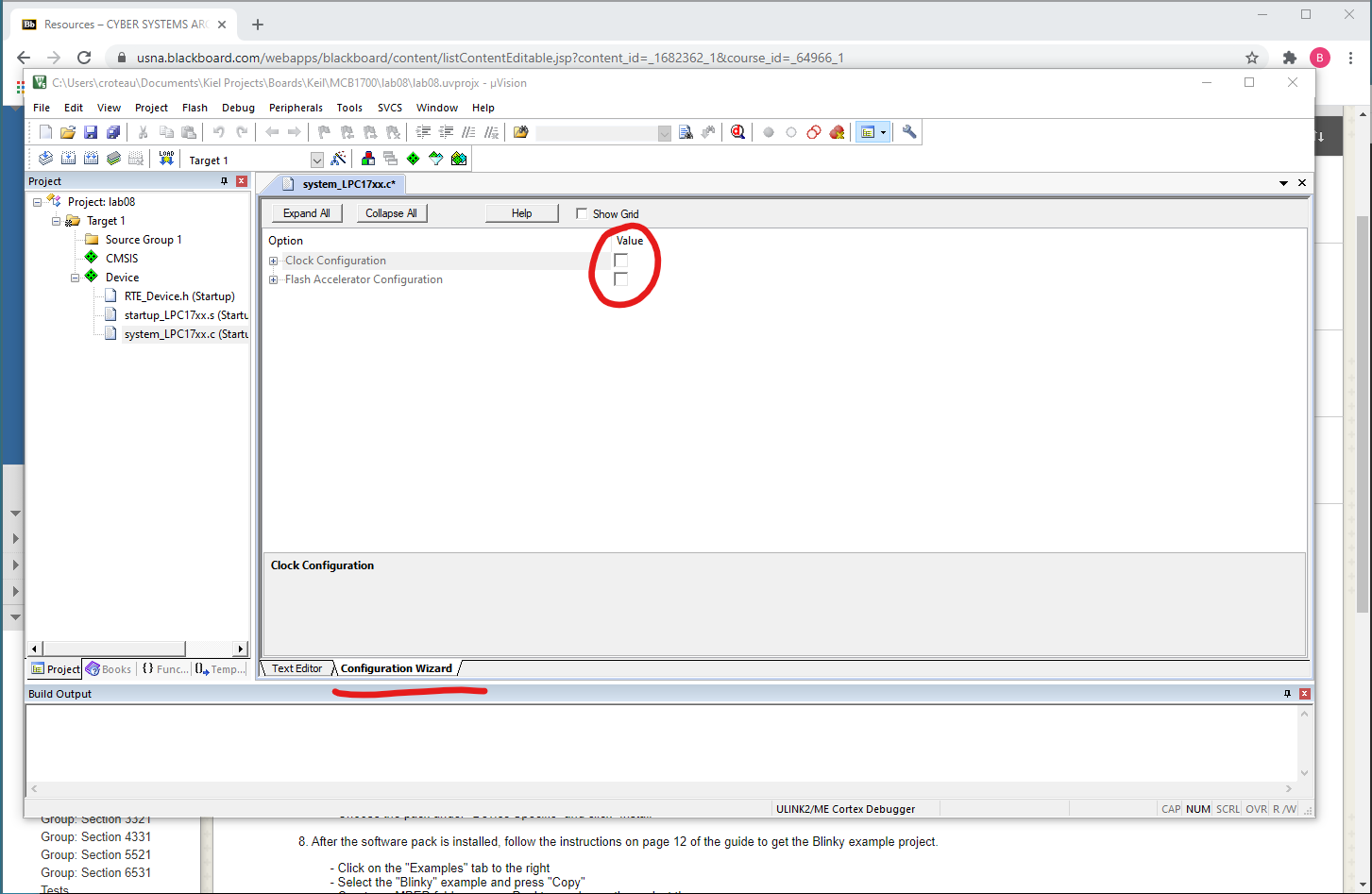
* 1. Select your board in the Select Device dialog to set your compilation target.



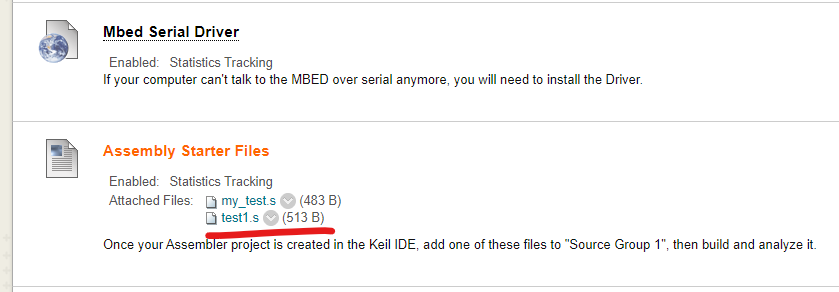
* 1. In the Manage Run-Time Environment window that pops up, open up the CMSIS + icon and check “CORE”. Then also open the Device + and select “Startup”. Then hit OK.



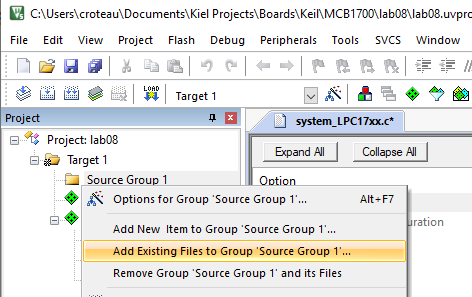
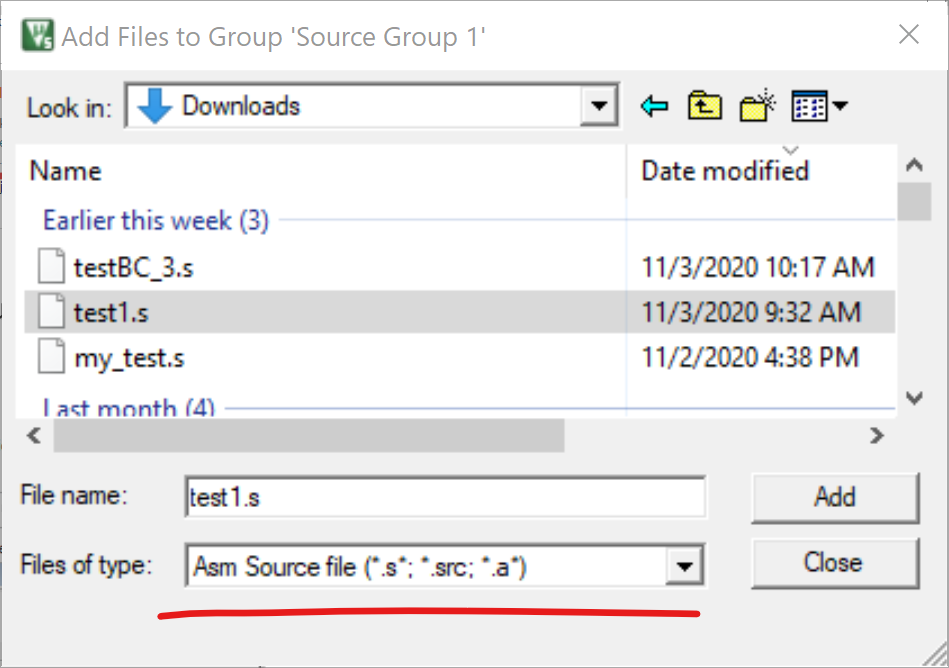
* 1. This will create your project, open the + under Target 1 and Device, then double click on the system\_LPC17xx.**c** file. Select the Configuration Wizard tab at the bottom of the window, then **uncheck** the two boxes under Value. This will keep the simulator from having to worry about the clocks and accelerating the fake flash memory.



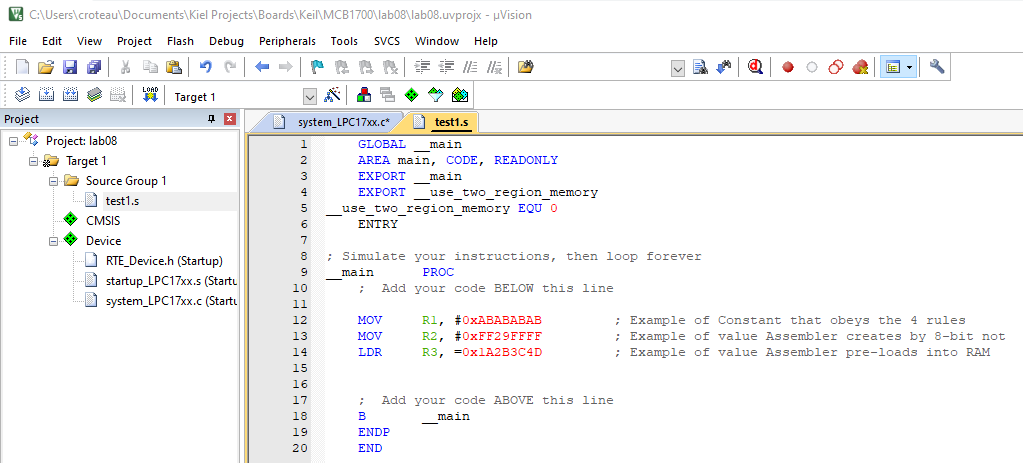
* 1. Download the test1.s assembly file from our Blackboard (under MBED/Resources).

****

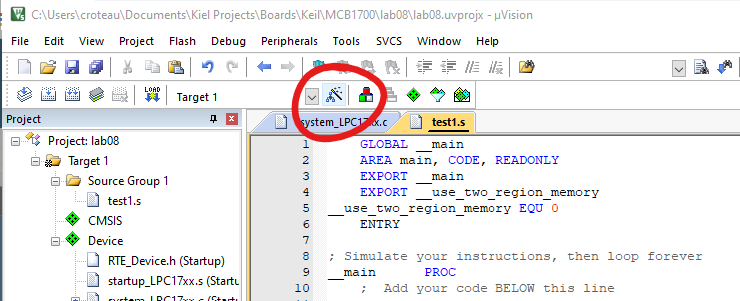
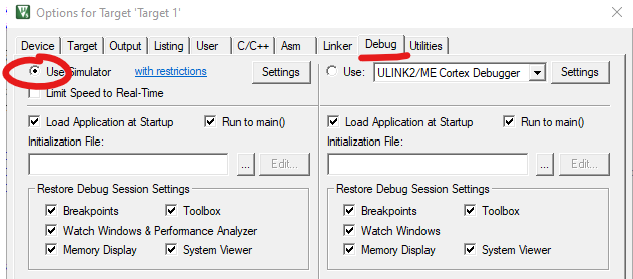
* 1. In the Project file tree on the left in the IDE, right-click “Source Group 1” and select “Add Existing Files to Group ‘Source Group 1’…” Navigate to where you saved the test1.s file select “Add” then “Close”. (Note, you will have to change the file type to see .s)

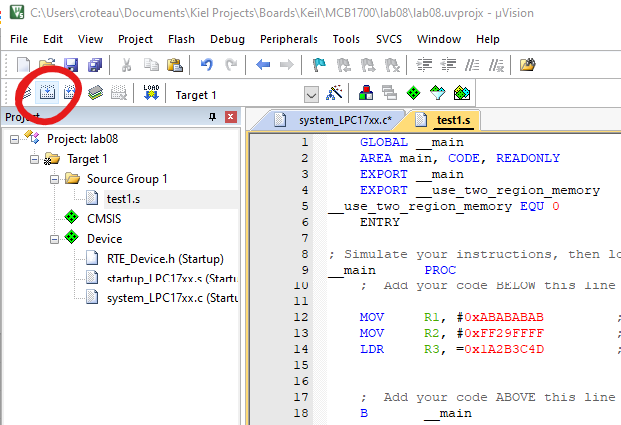
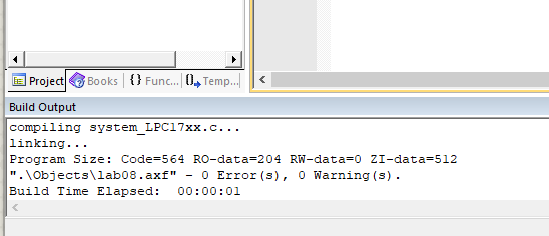
* 1. In the file tree open the Source Group 1 and double click the test1.s file to open it in the edit window.



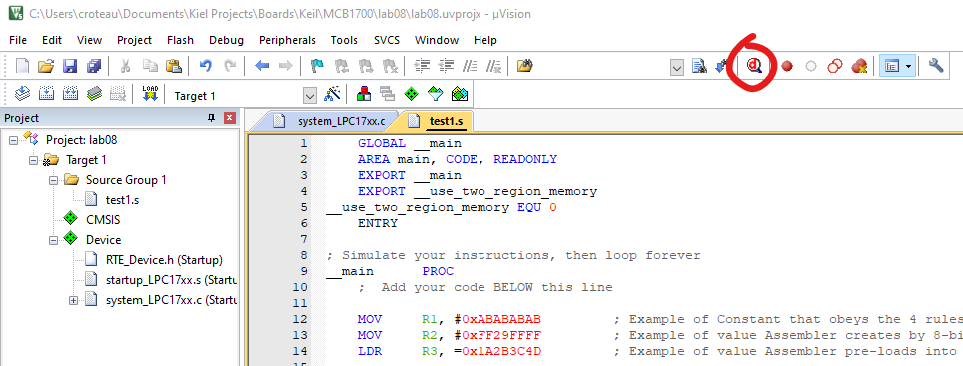
* 1. Next, to set up the debugger to use the simulator select the magic wand looking icon labeled “Options for Target…” Open the Debug tab and move the radio button to the left side that indicates the Simulator. Select OK to close that window.

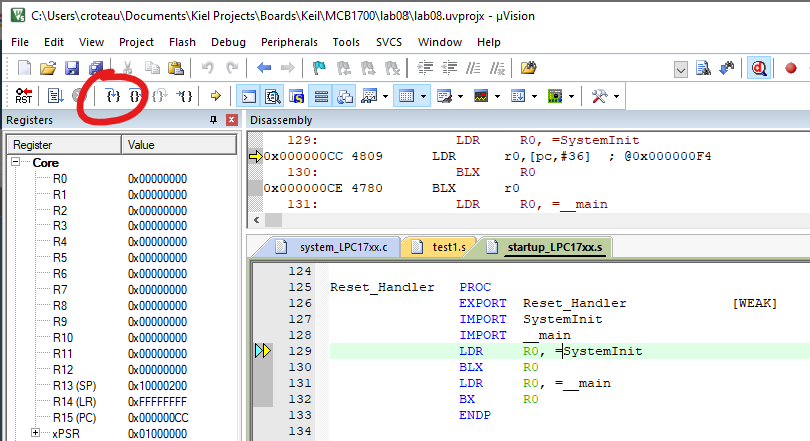
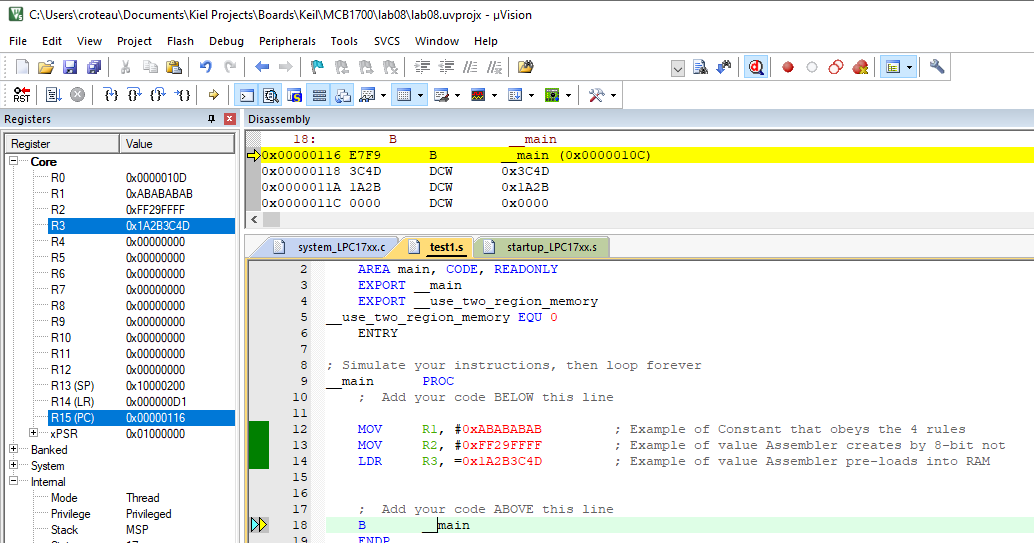
* 1. **Build** your project to create the lab08.axf binary file. Check the output in the lower window to make sure there were no errors.

* 1. Then start the **debugger** by clicking on the magnifying glass icon with a “d”. A warning about be limited to 32K will pop up, this is expected since we are using the free evaluation version of the Keil software.



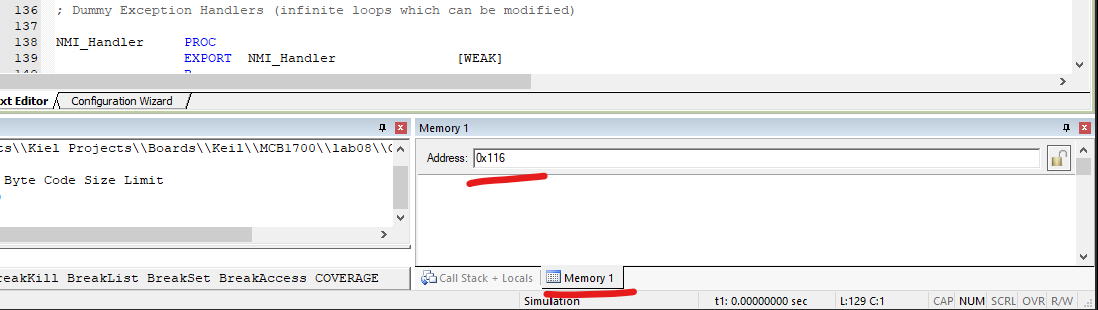
* 1. The program will start running from the Reset\_Handler, the code that is run after the board is reset or turned on. Use the Step button to advance the program until you have completed the three assembly instructions in our test1.s main function.

1. **Analyzing Assembly instructions**
   1. Take a Screen Shot and insert the figure here of your debugger after running the 3 instructions in the test1.s file. (5 pts)

This **MOV** instruction is part of the General-Purpose Data Processing set of assembly commands introduced in Section 34.2.5.6 (that will be covered in further detail next week in Lecture 3). For this lab we will just use the **MOV** command to write an immediate constant value into a general-purpose register. This lab is designed to demonstrate some of the difficulties that exist when trying to write a up-to 32-bit value into a register using a machine code that is only 32 bits in length. We need some number of those bits to contain the operation code, so typically we can only load constants of limited size.

* 1. The first instruction was “**MOV R1, #0xABABABAB**”, in your own words describe how this instruction was able to load a 32-bit value. (Hint: read Section 34.2.3.3.1 Constant) (4 pts)
  2. Read Section 34.2.5.7 that describes the **MOVT** operation. How does this differ from the regular **MOV** operation? (3 pts)
  3. In Section 34.2.5.7.2 the manual states “*The MOV, MOVT instruction pair enables you to generate any 32-bit constant*.” How does the combination of these two achieve this? (3 pts)
  4. Open your Windows calculator, select the programmer mode, Hex input, and set the width to DWORD (32-bits). Type or key in the value **FF29FFFF**. Then hit the Bitwise button and select the NOT operation. What is the result of that operation? (3 pts)
  5. The second instruction was “**MOV R2, #0xFF29FFFF**”, what differed about how this value was entered? (5 pts)
  6. With the debugger running, scroll up or down in the upper Disassembly window until you find a comment for this second command. What actual command did the Assembler translate our written instruction to? (3 pts)
  7. The third instruction was “**LDR R3, =0x1A2B3C4D**”, what section of the manual describes the **LDR** operation? Do you see this =0xZZZZZZZZ syntax listed? (3 pts)
  8. How wide of a value (how many bits) is written when using the LDR command? What is this data width called in the documentation? (3 pts)
  9. With the debugger running select the Memory1 tab in the lower right, then in the Address dialog type in **0x116** and hit Enter.



Do you see the value we loaded from memory in that list? Take a Screen Shot showing that value and paste it here. (3 pts)

1. **Modifying the Assembly Code**

Stop the debugger by deselecting the magnifying glass icon in the toolbar. Type the following assembly command into the test1.s file below the third (LDR) command “**MOV R4, #0x00023300**”. Hit the build icon, it should not work.

* + - 1. Copy and paste the error code here: (3 pts)
      2. Describe in your own words what this error code is telling you. (5 pts)
      3. Look in section 34.2.5.6.1 of the manual, what is the largest immediate value that can be loaded directly with a MOV command? (3 pts)
      4. Is our number above or below this limit? Why? (4 pts)

The example code showed one way to get around this limit by loading from memory, lets look at a few more ways we can with combining multiple MOV commands.

Comment out (or delete) the invalid MOV command (with a “;” at the start of that line) and type these commands:

**MOV R4, #0x00023000**

**MOV R5, #0x00000300**

**ORRS R6, R4, R5**

* + - 1. Build that, restart the debugger then re-step through the program until these operations are completed. Paste in a screen shot showing the desired result in the register. (3 pts)
      2. Describe how these commands were able to get our desired value into a register. (5 pts)

Here is another method to try. Stop the debugger, then add these commands to your program:

**MOV R7, #0x00030000**

**MOV R8, #0x0000CD00**

**SUBS R9, R7, R8**

* + - 1. Build the project, restart the debugger then re-step through the program until these operations are completed. Paste in a screen shot showing the desired result in the register. (3 pts)
      2. Describe how these commands were able to get our desired value into a register. (5 pts)

1. **Your turn**

Figure out a series of assembly commands that loads your alpha code (**in hex**) into a register. You can choose any technique demonstrated or another one you can come up with by reading the manual.

* + - 1. Build it and verify it works in the debugger. Paste a screen shot. (7 pts)
      2. Additionally, paste in the assembly commands that you used here. (7 pts)

1. **Our manual describes a set of assembly instructions known as Thumb. Are there any other instruction sets that run on ARM processors? What makes those different?** (7 pts)
2. **Does ARM Ltd. make any hardware? Who is trying to buy it for $40B, and why?** (6 pts)
3. **A CISC can abstract away the complexity of how a computer must implement an instruction while providing a relatively simple interface to a programmer. A classic example is that a CISC can increment the value of a variable in memory using just one instruction. Describe the steps that a RISC architecture (like ARM) would have to take to accomplish the same task.** (7 pts)