

## SY303 Lab 8

## MBED Assembly 1 – Memory Access

Name: McKenzie Eshleman

Partner(s):

Section: 3321

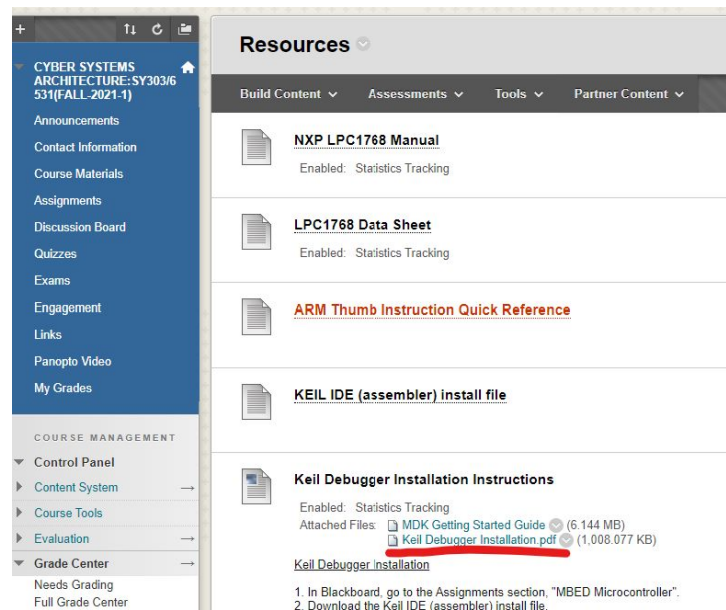
**References:** 1) [Getting started with MDK](#)

2) [LPC1 LPC176x/5x User manual](#) (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)  $\mu$ Vision integrated development environment (IDE) to practice writing to registers and memory of our device.

### 1) Pre-Lab: Creating a new Assembly based Project

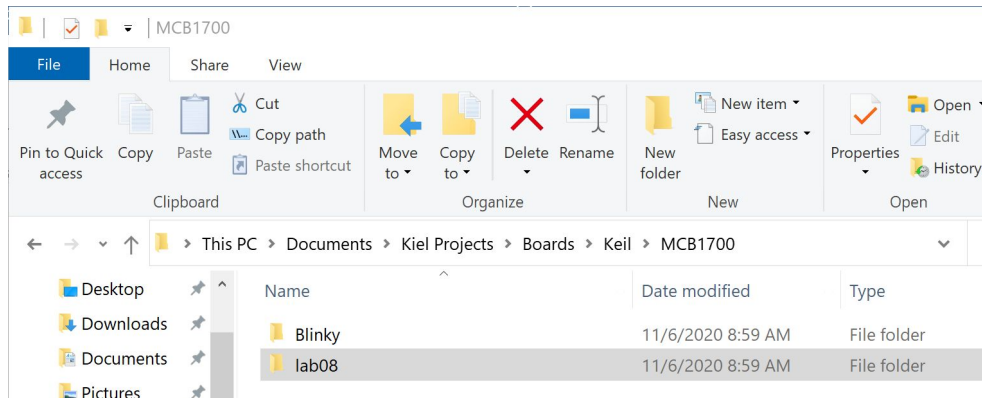
- a. Install Kiel, follow the instructions in Blackboard under /Assignments/MBED Microcontroller/Resources/Keil Debugger Installation Instructions.



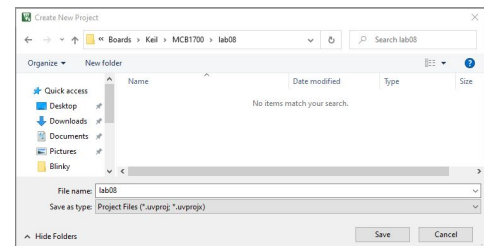
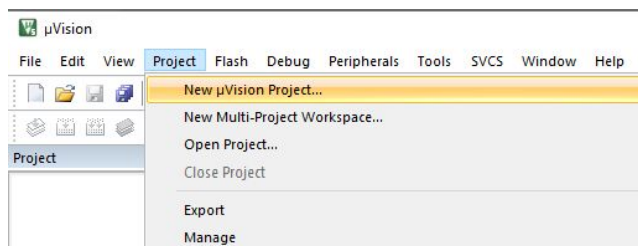
The screenshot shows a Blackboard course interface. On the left is a navigation menu with options like Announcements, Contact Information, Course Materials, Assignments, Discussion Board, Quizzes, Exams, Engagement, Links, Panopto Video, and My Grades. The main content area is titled 'Resources' and lists several files for download:

- NXP LPC1768 Manual** (Enabled: Statistics Tracking)
- LPC1768 Data Sheet** (Enabled: Statistics Tracking)
- ARM Thumb Instruction Quick Reference**
- KEIL IDE (assembler) install file**
- Keil Debugger Installation Instructions** (Enabled: Statistics Tracking)
  - Attached Files: MDK Getting Started Guide (6.144 MB), Keil Debugger Installation.pdf (1,008.077 KB)
  - [Keil Debugger Installation](#)
  - 1. In Blackboard, go to the Assignments section, "MBED Microcontroller".
  - 2. Download the Keil IDE (assembler) install file.

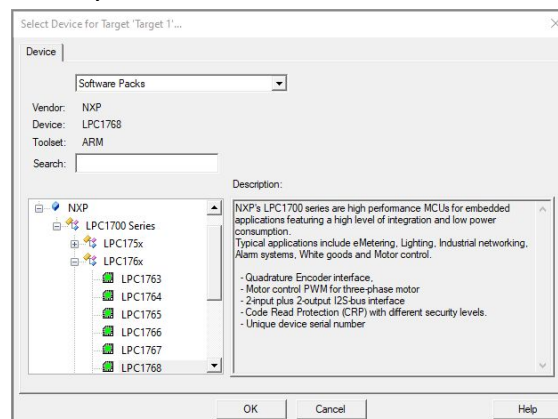
- b. Find the folder where your Blinky project was saved, then make a new folder and name it “lab08”



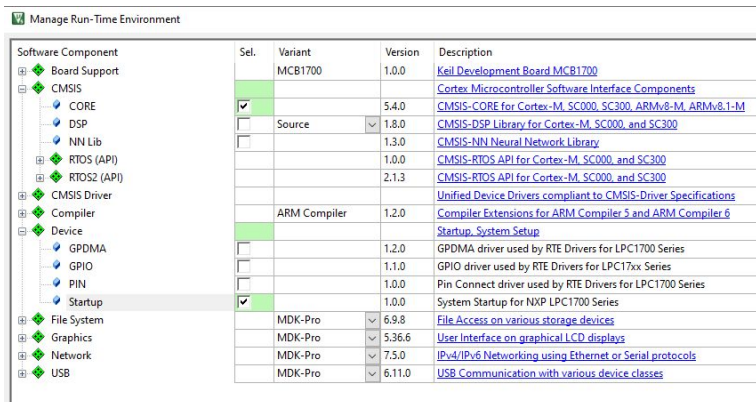
- c. In uVision, close your Blinky project, if it is open, by selecting Project -> Close Project.
- d. Create a new project. Navigate to your lab08 folder and insert “lab08” as the project file name in that folder.



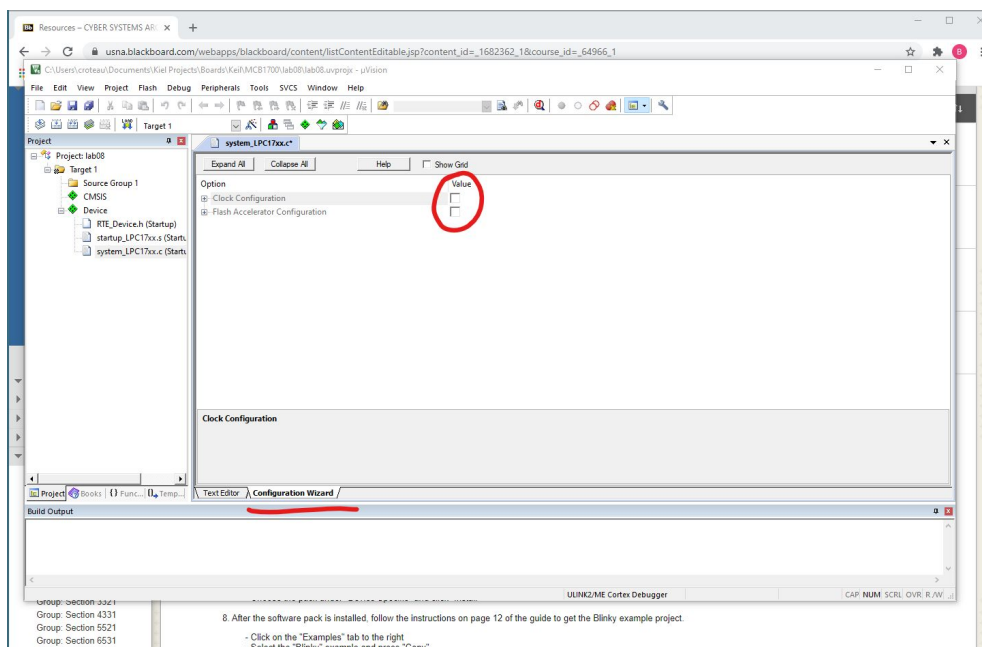
- e. Select your board in the Select Device dialog to set your compilation target.



- f. 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.



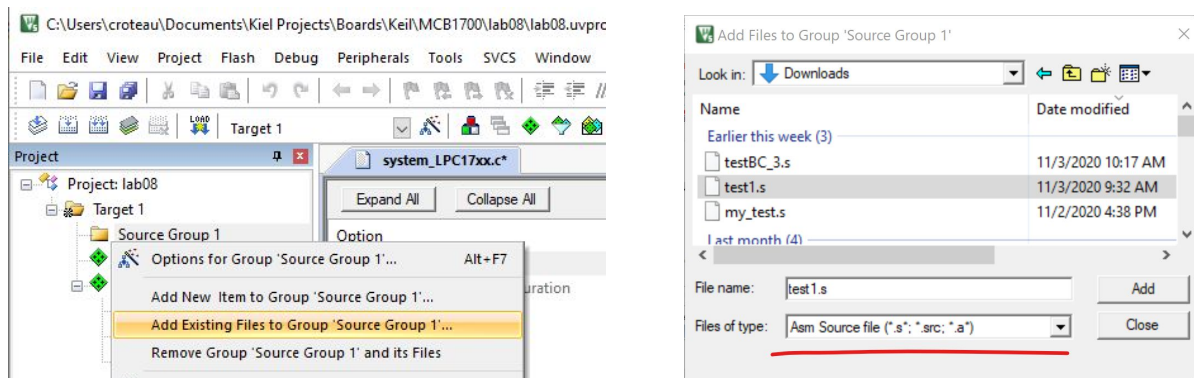
- g. This will create your project, open the + under Target 1 and Device, then double click on the system\_LPC17xx.c file. Select the Configuration Wizard tab at the bottom of the window, then **uncheck** the two boxes under Value. This will keep the simulator from having to worry about the clocks and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



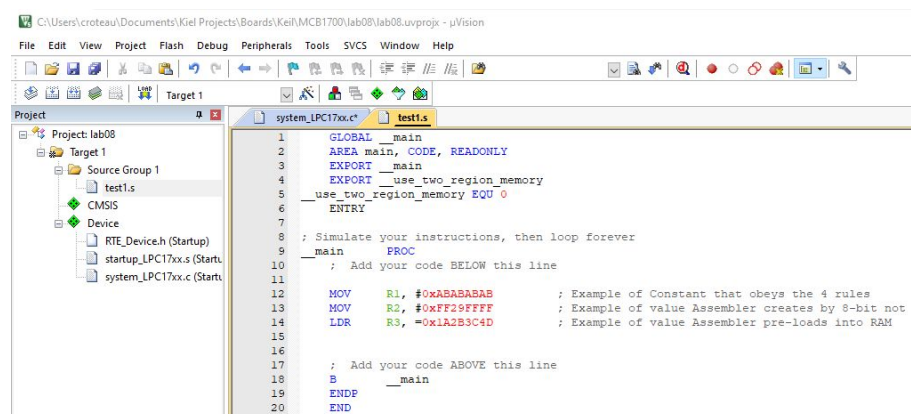
- h. Download the test1.s assembly file from our Blackboard (under MBED/Resources).



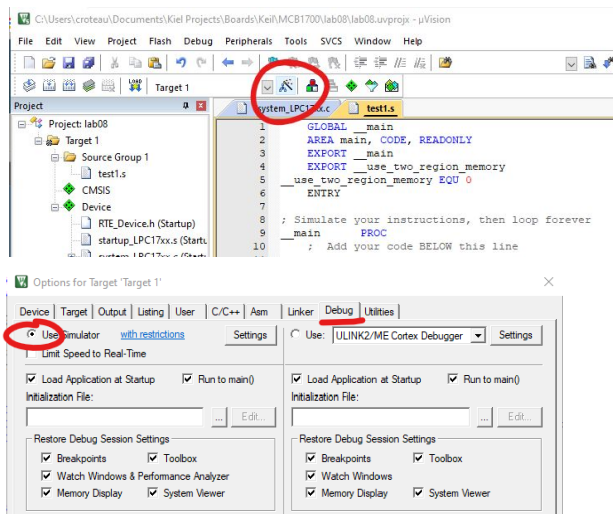
- i. In the Project file tree on the left in the IDE, right-click "Source Group 1" and select "Add Existing Files to Group 'Source Group 1'..." Navigate to where you saved the test1.s file select "Add" then "Close". (Note, you will have to change the file type to see .s)



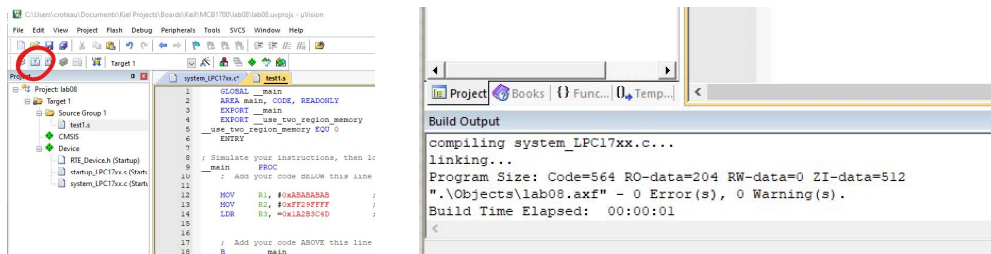
- j. In the file tree open the Source Group 1 and double click the test1.s file to open it in the edit window.



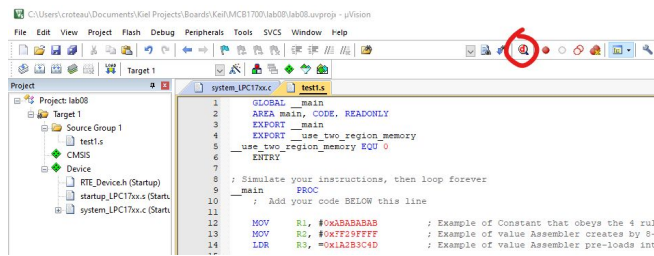
- k. Next, to set up the debugger to use the simulator select the magic wand looking icon labeled “Options for Target...” Open the Debug tab and move the radio button to the left side that indicates the Simulator. Select OK to close that window.



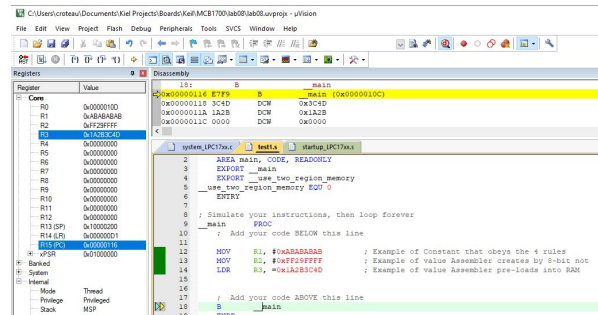
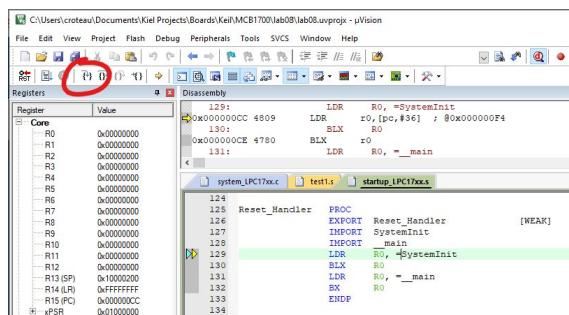
- l. **Build** your project to create the lab08.axf binary file. Check the output in the lower window to make sure there were no errors.



- m. 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.

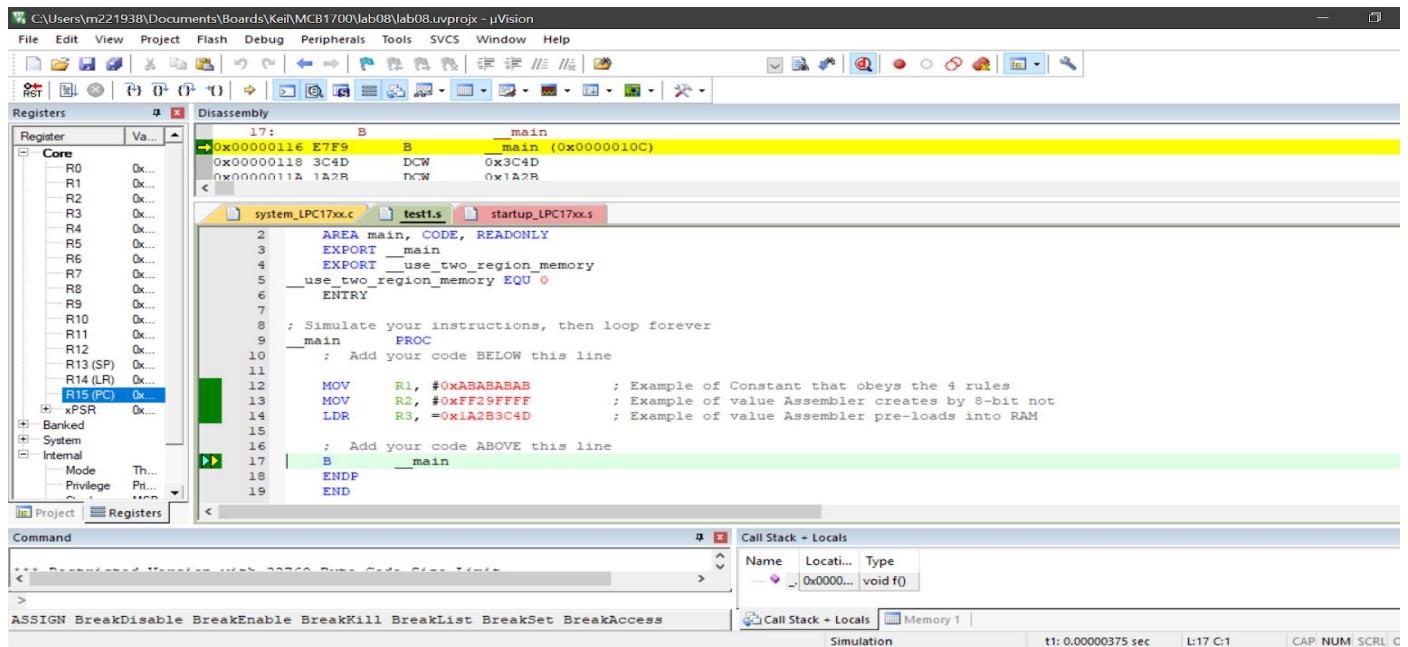


- n. The program will start running from the Reset\_Handler, the code that is run after the board is reset or turned on. Use the Step button to advance the program until you have completed the three assembly instructions in our test1.s main function.



## 2) Analyzing Assembly instructions

- a. 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.

- b. The first instruction was “**MOV R1, #0xABABABAB**”, in your own words, describes how this instruction was able to load a 32-bit value. (Hint: read Section 34.2.3.3.1 Constant) (4 pts) We can produce any constant by shifting an 8-bit value left by any number of bits that is within the 32-bit word. When an operand2 constant is used with the instruction MOV, the carry flag is updated to bit[31] of the constant, if the constant is greater than 255 and can be produced by shifting an 8-bit value. These instructions do not affect the carry flag if Operand2 is any other constant.
- c. Read Section 34.2.5.7 that describes the **MOVT** operation. How does this differ from the regular **MOV** operation? (3 pts) The MOVT operation is Move Top the syntax of the operation is as follows MOVT{cond} Rd, #imm16. The Cond is an optional condition code, Rd is the destination register, and imm16 is the 16-bit immediate constant. The operation writes a 16-bit immediate value(imm16) to the top half word of the destination register. The MOV and the MOVT instruction pair enables you to generate any 32-bit constant.
- d. 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) The MOV and MOVT instruction pair enables the user to generate any 32-bit constant by having the MOV writes the lower half 16-bit constant, while the MOVT writes the 16-bit constant for the upper half. When the two are combined they can generate a 32-bit constant.
- e. 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) The results of the operation is **D60000** or 1101 0110 0000 0000 0000 0000 in binary.

- f. The second instruction was “**MOV R2, #0xFF29FFFF**”, what differed about how this value was entered? (5 pts) This second instruction is different because we are writing the hex value of 0xFF29FFFF (or simply FF29FFFF) to R2, however the flags are not updated they remain the same.
- g. 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)

The Assembler translated the second command to 0x00000110 F46F0256 MVN R2, #0xD60000, this is the hex value we got when we NOT the hex value of 0xFF29FFFF into our windows calculator.

- h. 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)

The section of the manual that describes LDR is section 34.2.4.5. I do not see specifically =0xZZZZZZZZ syntax listed but based off of the section, I presume this is a label which is a PC relative expression. It can also be the #offset. I am not sure if that is a proper syntax if that is supposed to be a hex value since Z is not included in hex values.

- i. 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)

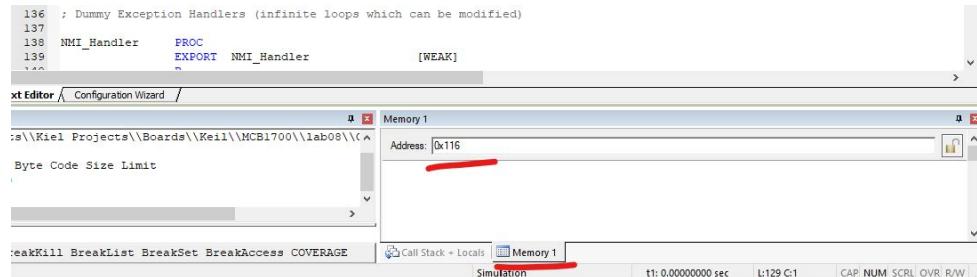
When using LDR a 32-bit value is written when the command is ran. The data width called in the documentation is called an offset, which is from Rn and can be 0 to 255. If offset is omitted, the address is the value of Rn.



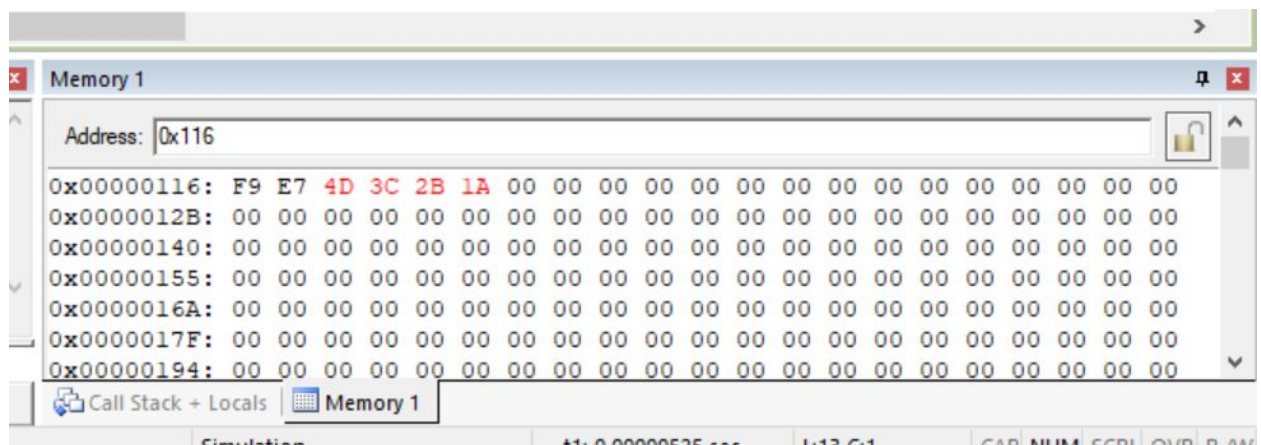
SY303 Lab 8 MBED Assembly 1 – Memory Access

## MBED Assembly 1 – Memory Access

- j. 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)



### 3) 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.

- a. Copy and paste the error code here: (3 pts)

```
..\..\..\..\Desktop\Cyber\SY303\Project 8\test1.s(15): error: A1871E:
Immediate 0x00023300 cannot be represented by 0-255 shifted left by 0-23 or
duplicated in all, odd or even bytes
".\Objects\lab08.axf" - 1 Error(s), 0 Warning(s).
Target not created.
Build Time Elapsed: 00:00:00
```

- b. Describe in your own words what this error code is telling you. (5 pts)

I believe the code is saying that we shifted the values incorrectly, and we duplicated some values that were already programmed. The target is the file we are trying to produce and could not be created because of this error with the shift.

- c. 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) The largest immediate value that can be loaded directly with a MOV command is 65,535.

- d. Is our number above or below this limit? Why? (4 pts) Our number is above this limit because I converted the Hex to a decimal number and got 144,128. Which means we are above the imm16 range therefore the program will not work correctly.

The example code showed one way to get around this limit by loading from memory, let's 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
```

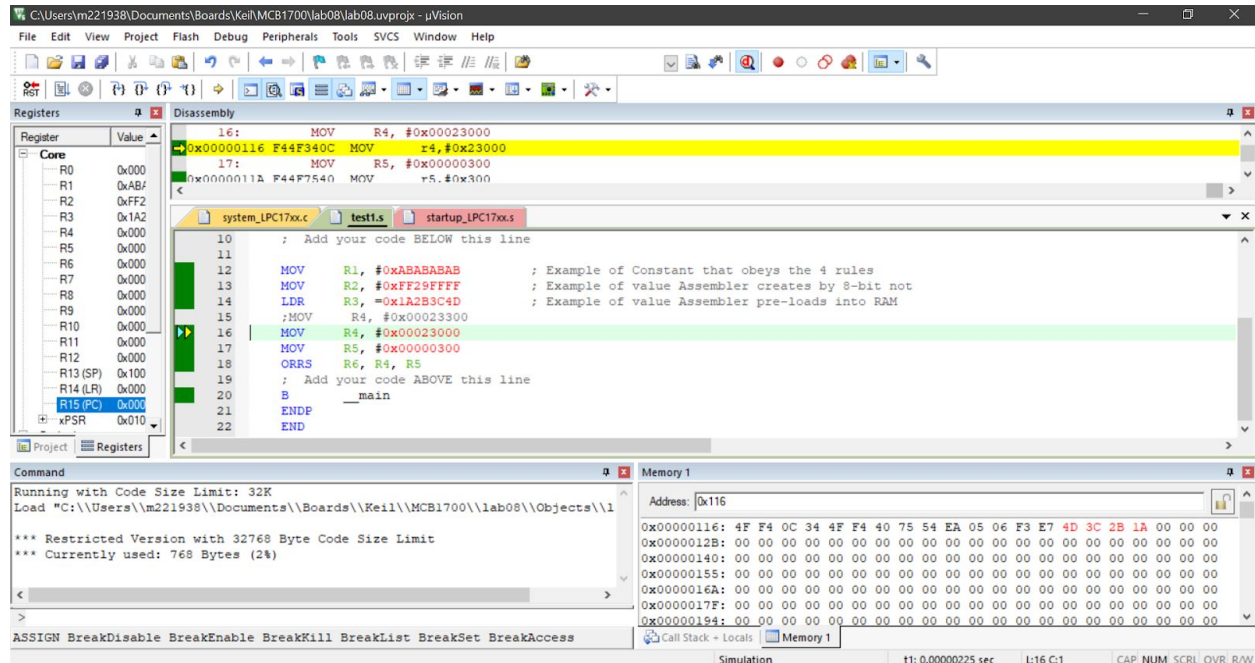
```
MOV R5, #0x00000300
```

```
ORRS R6, R4, R5
```

## SY303 Lab 8

## MBED Assembly 1 – Memory Access

- e. 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)



- f. Describe how these commands were able to get our desired value into a register. (5 pts) These commands are able to get our desired value into a register by going around the limit that is set by MOV, by loading multiple values from memory to get our desired value into the register.

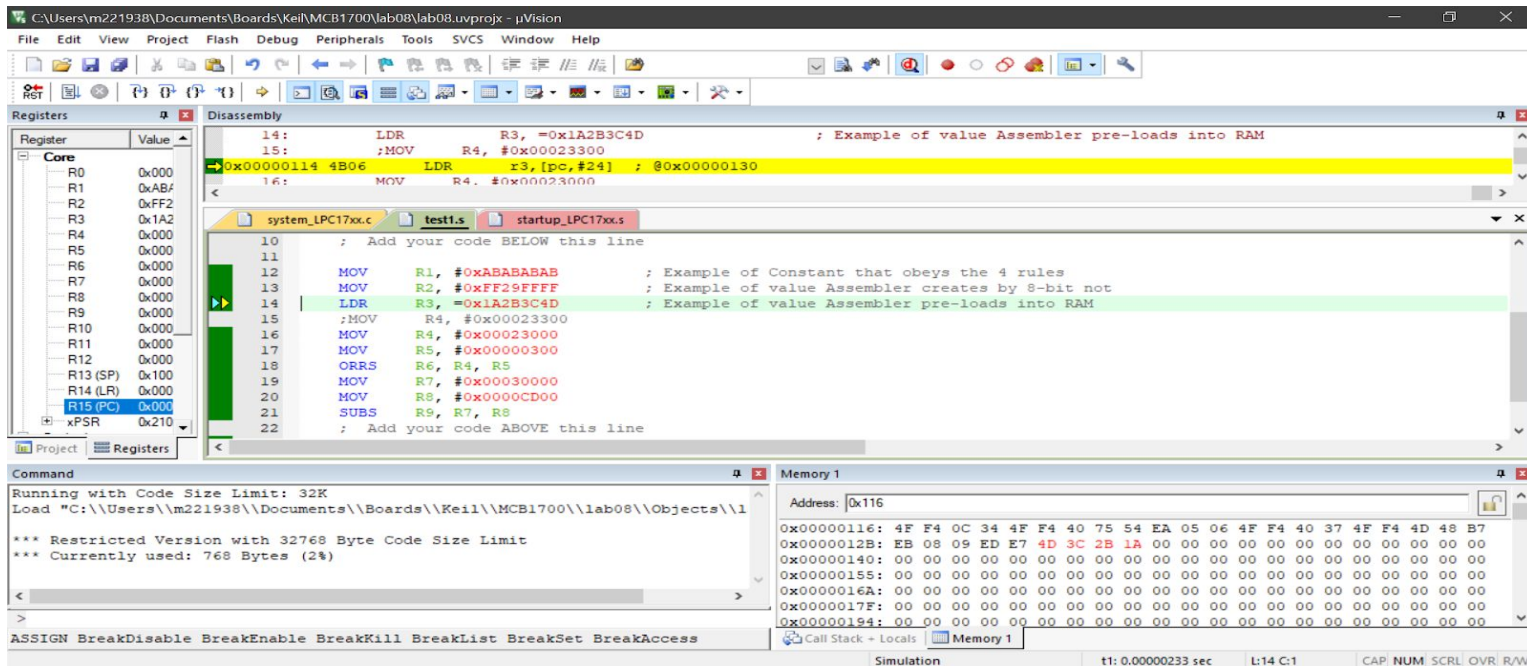
Here is another method to try. Stop the debugger, then add these commands to your program:

```
MOVR7, #0x00030000
```

```
MOVR8, #0x0000CD00
```

```
SUBSR9, R7, R8
```

- g. 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)

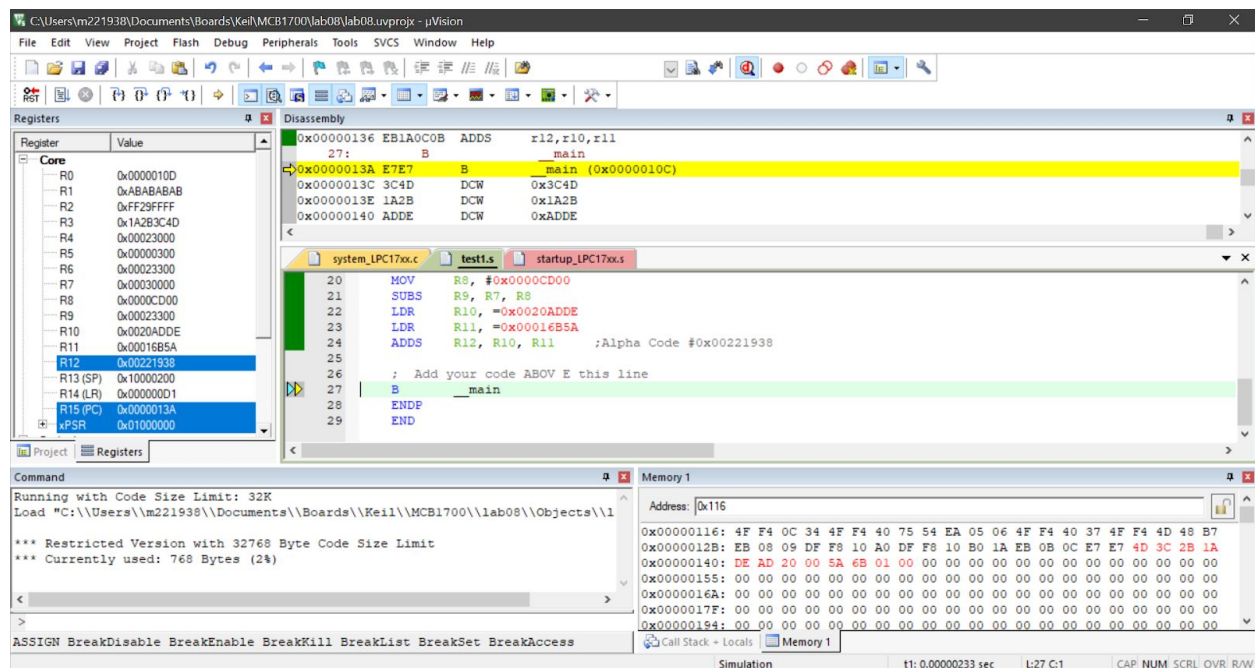


- h. Describe how these commands were able to get our desired value into a register. (5 pts) The commands are able to get our desired value into a register by subtracting Register 7 by Register 8, this will allow us to get the desired value that we want.

#### 4) 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.

- a. Build it and verify it works in the debugger. Paste a screenshot. (7 pts)



b. Additionally, paste in the assembly commands that you used here. (7 pts)

```
LDR R10, =0x0020ADDE
```

```
LDR R11, =0x00016B5A
```

```
ADDs R12, R10, R11 ;Alpha Code #0x00221938
```

- 5) 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) The ARM processor can support other instructions such as the ARM instruction set because of the fact that a single processor is able to support a large number and variety of instructions to offer more functionality. The ARM processor supports ARM instruction and the thumb instruction set. The major difference between these instructions is that the thumb instructions are 16 bit instructions while the ARM instructions are the 32 bit instruction set. The thumb instructions are smaller and faster than the ARM instruction.

<https://developer.arm.com/architectures/instruction-sets>

**6) Does ARM Ltd. make any hardware? Who is trying to buy it for \$40B, and why? (6 pts)**

ARM Ltd. is a British semiconductor and software design company. Nvidia is trying to buy ARM Ltd. for \$40 billion dollars, the reason for them trying to buy ARM is because NVIDIA wants to make their company the world premier computing company for the age of AI. NVIDIA's leadership wants to expand the presence of ARM in the UK by establishing a world-class AI research and education centre.

<https://nvidianews.nvidia.com/news/nvidia-to-acquire-arm-for-40-billion-creating-worlds-premier-computing-company-for-the-age-of-ai>

**7) 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)**

A Complex Instruction Set Computer (CISC), it helps in simplifying the code and makes the code shorter which helps reduce the memory used. An example of a CISC instruction set can be

MOVE A , B

ADD A, C

This example moves the instruction set and adds it. After coding A and B we can save it into A. The same instructions can be implemented in RISC like...

LOAD R2, B

LOAD R3, C

ADD R4, R2, R3

STORE R4, A

This method uses 3 instructions: set load, add, and store. The CISC uses only two lines of code for adding two numbers instead of three with the RISC.

<https://cs.stackexchange.com/questions/269/why-would-anyone-want-cisc/284>

<https://cs.stanford.edu/people/eroberts/courses/soco/projects/risc/riscisc/>