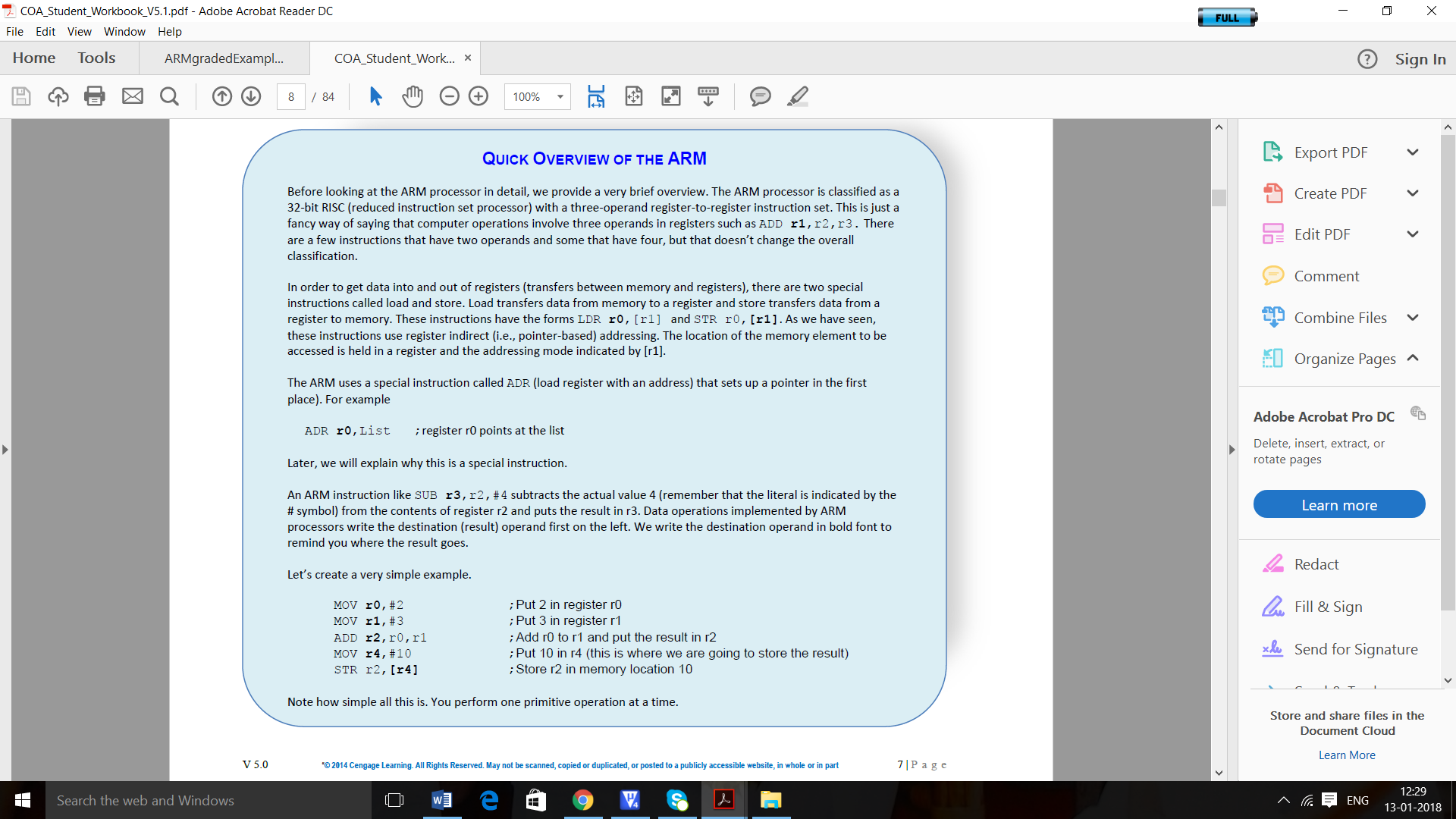
**Lab 7: ARM Assembly Introduction**

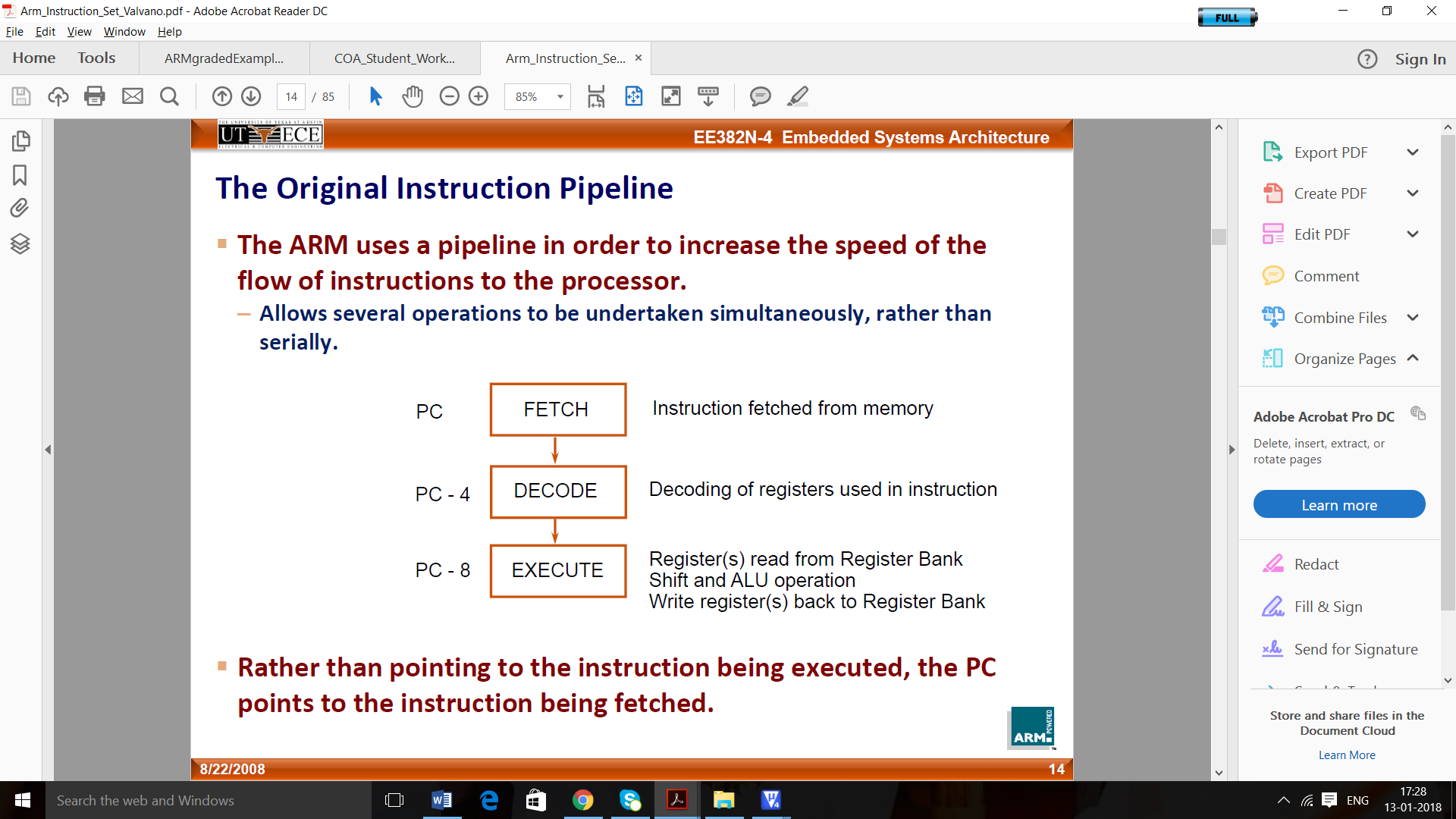
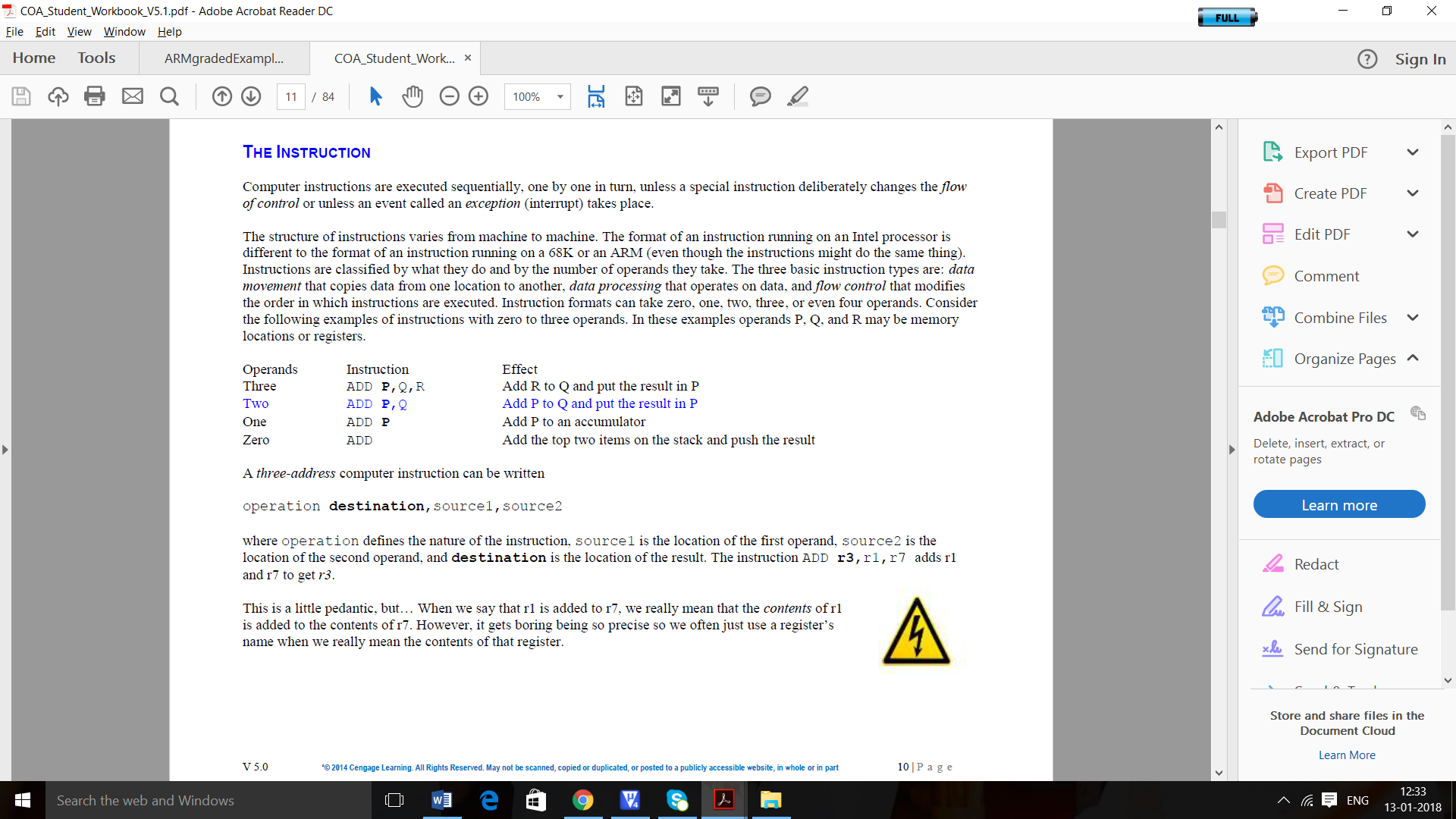
**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

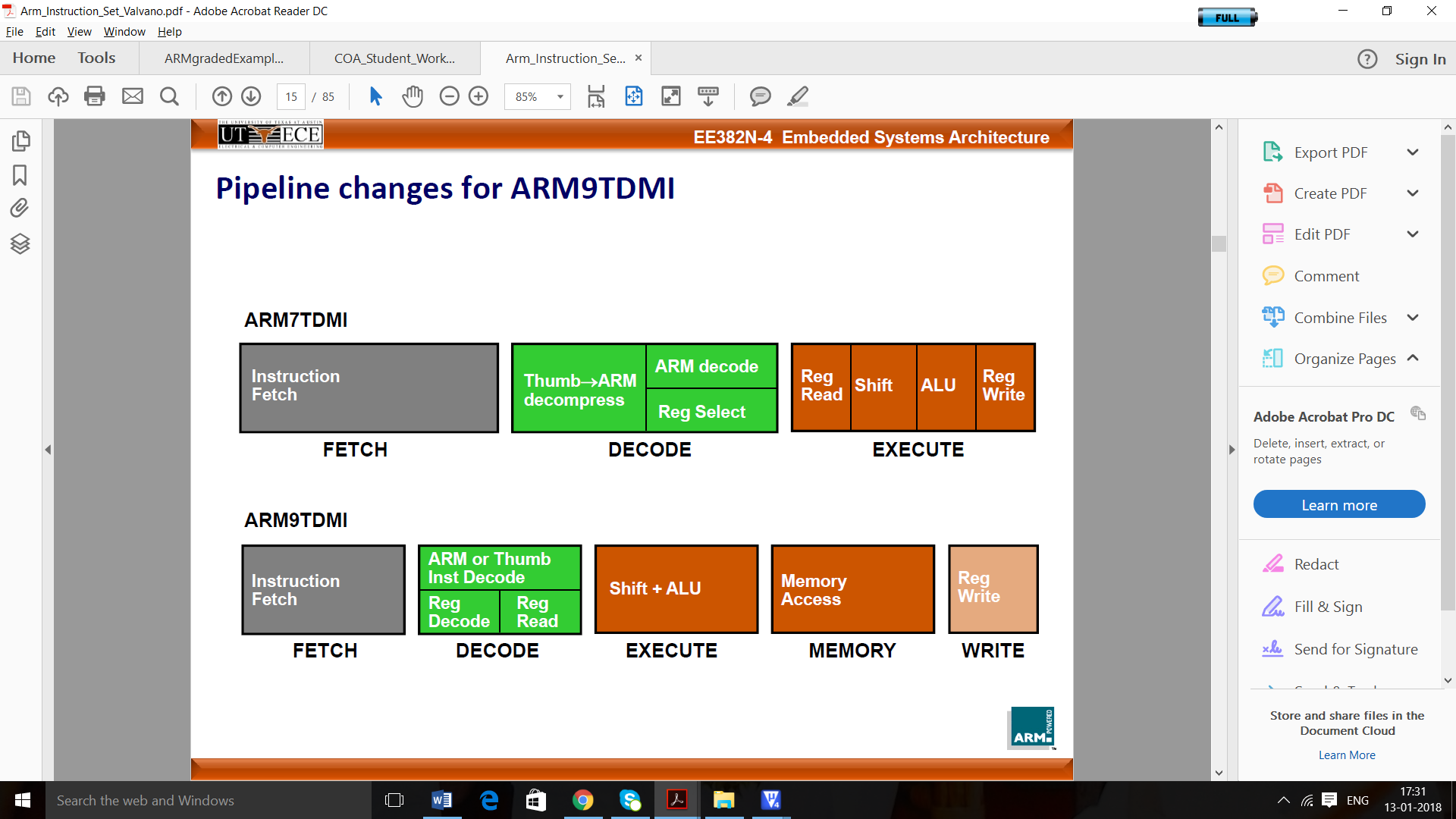
**Name: S Sreya Roll Number:AM.EN.U4CSE17330**

**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

**Part1 – ARM Theory References**







**Part2 – Understanding and writing an ARM Assembly program**

**Familiarization of ARM Simulator**

Keil MicroVision is an integrated development environment (IDE). Keil μVision 4 is the software used for writing program, compile and test the program without real hardware i.e. simulation. Following section describes the steps involved in creating a project, building it and debugging the same.

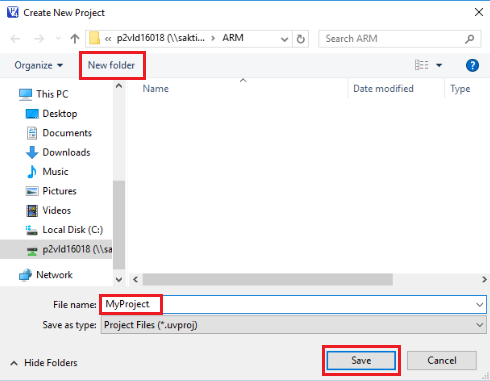
**Creating an Assembly Project in Keil μVision 4**

**Step 1.** Open Keil IDE by clicking  icon on desktop

**Step 2**. Go to **Project** tab  **New** **μVision Project…**



**Step. 3.** Create a **New Folder** and give name to your project. Ensure that you create all the folder under a folder having the name as your roll number. Click Save.

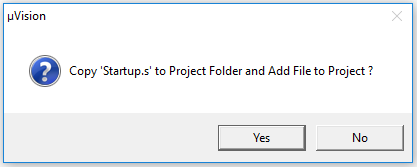


NB: - It is better to always use New Folder for every new Project. Try to give sensible name to the folder and the project. The project name will be stored with .uvproj extension by default.

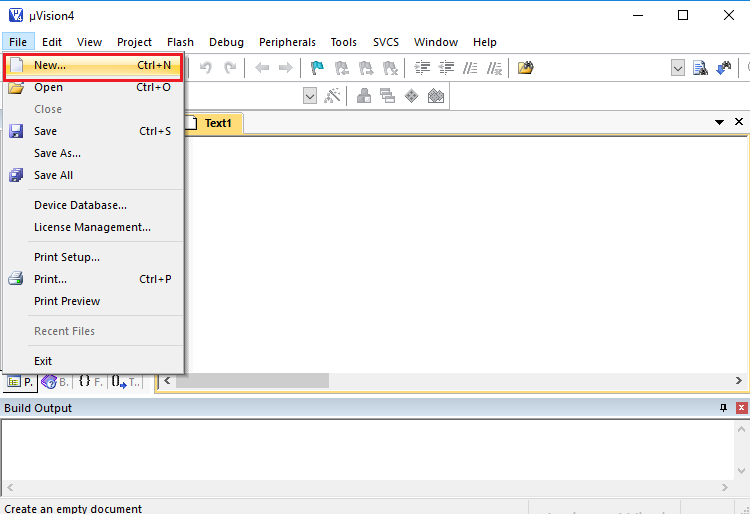
**Step 4.** From the data base of vendors, select the vendor and the device model. We can select ARM7(Little Endian) to work and understand the basics of Assembly language and the working of the Instruction Set



**Step 5.** A message window will be displayed asking whether to add Startup.s file to project. If you are writing program in assembly, click on No. If your program is using embedded, then click on Yes button. Upon clicking Yes, Startup.s file will be added to project and will be present in project directory.



**Step 6.** Make a new text file by clicking New icon (or Ctrl+N or goto Files  New)

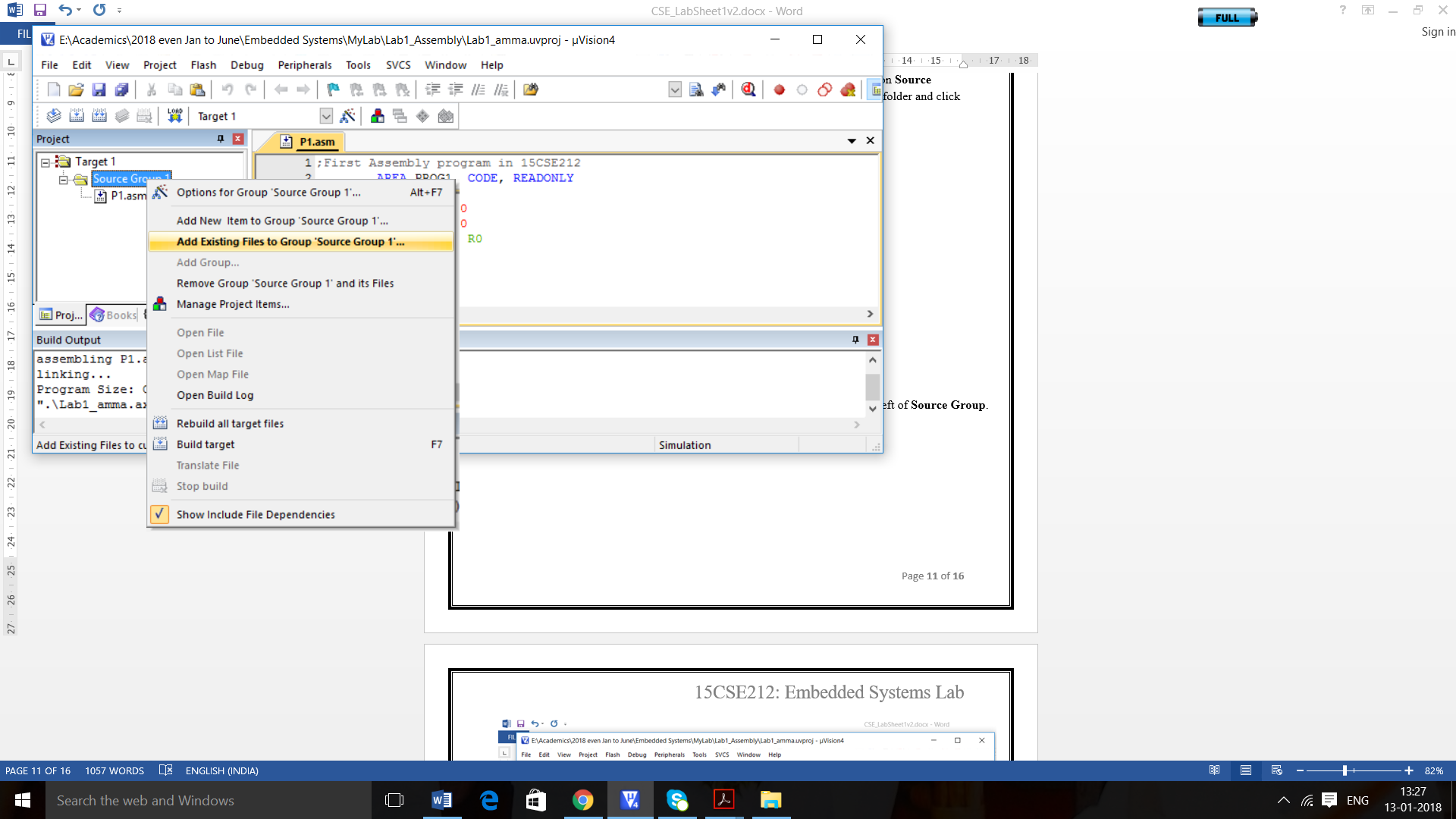
Write your program code in the new white work space created. 

**Note:** - While writing code in assembly, proper indentation must be provided. As you will save the file, the software will detect the .asm language keywords and they become colorful to make them prominent from rest of code.

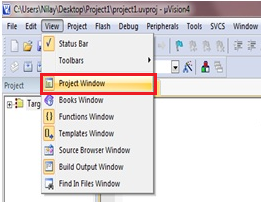
**Step 7.** After writing the codes go to **File**  **Save** (or **Ctrl+S**). Save the file with .c extension if the program is written in C or with .asm if the program is in assembly. The file must be saved in the same project folder.



**Step 8.** Now add the file to your project. In the **Project** window, Right click on **Source Group**  **Add Existing Files to Group Source Group.** Select the file from folder and click on **Add.**

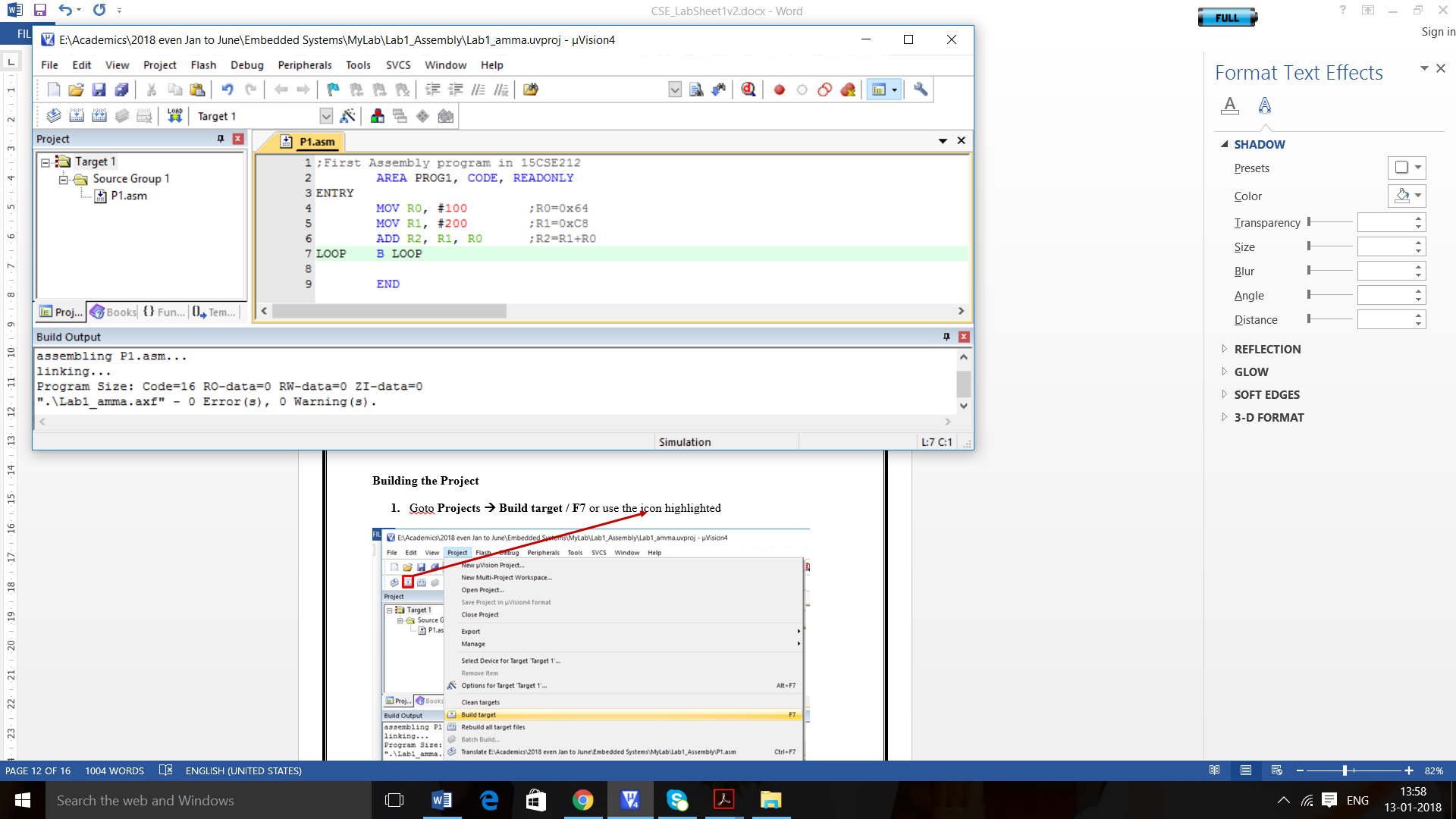


**Note: -** If **Project** **Window** is not there, go to **View**  **Project Window**



**Note: -** You may check the .asm file by clicking on the little plus sign at the left of **Source** **Group**.

**Step 9:** Type your first assembly program in the editor window as shown below,

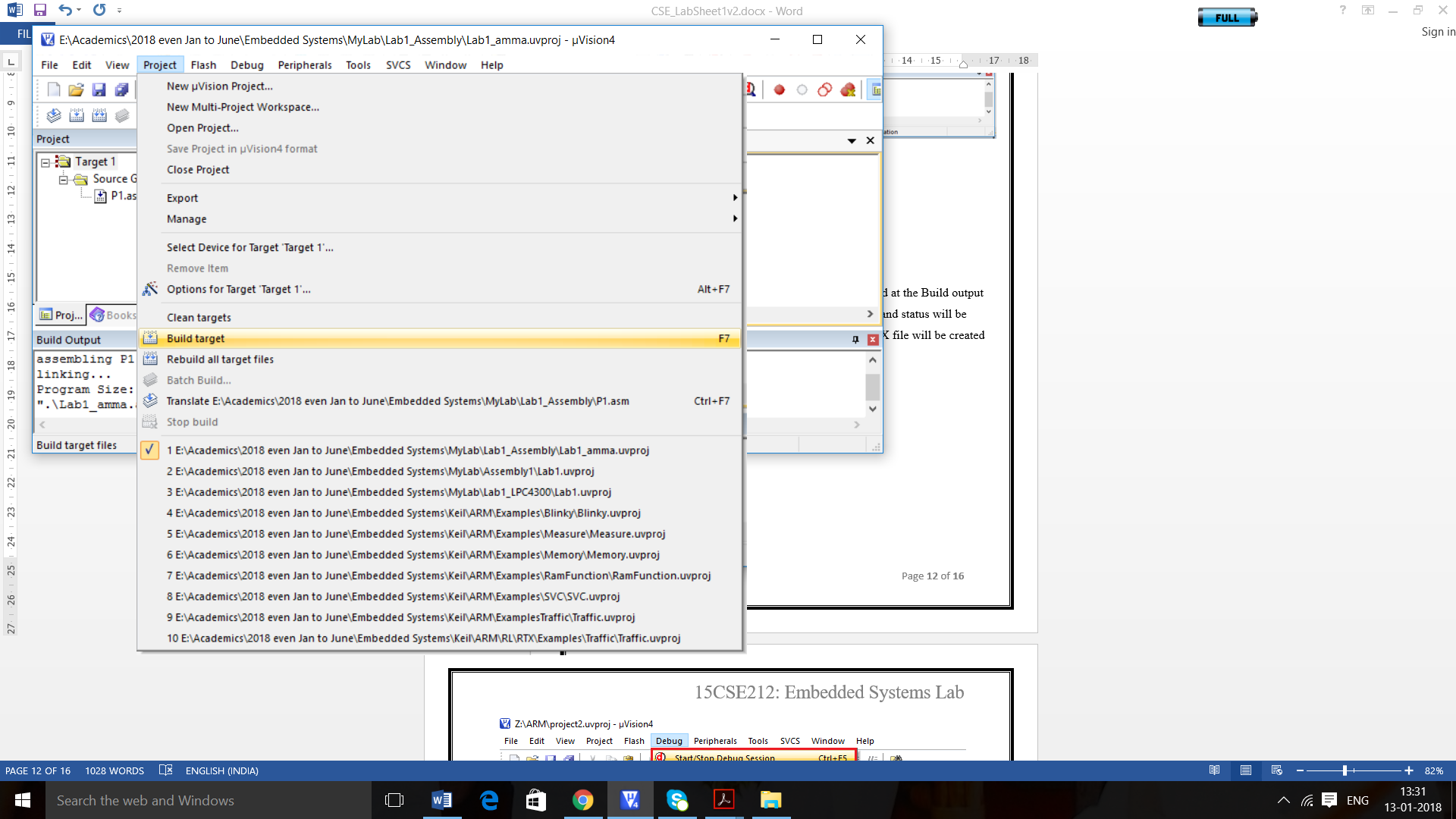


Note:

1. AREA PROG1, CODE, READONLY, ENTRY, END – all these 3 lines are assembler directives and are not part of the assembly language.
2. The purpose of AREA directive is to name the region of memory where the program is located.

**Building the Project**

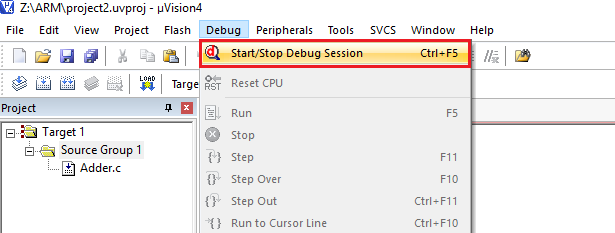
1. Goto **Projects  Build target / F7** or use the icon highlighted

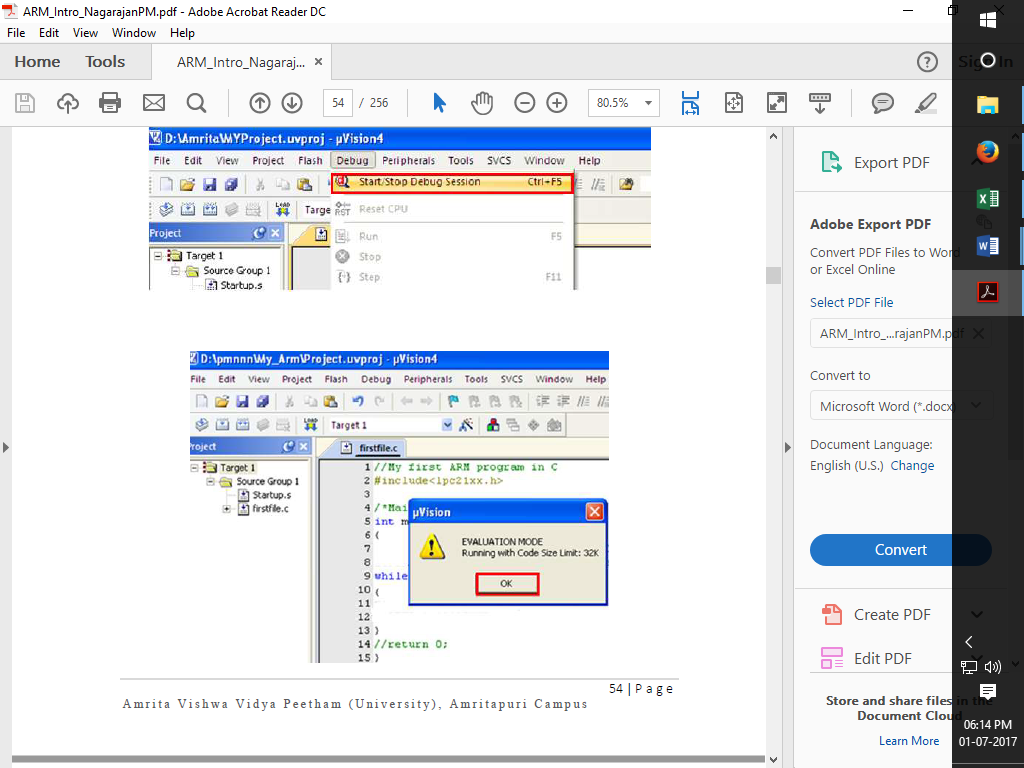


Once you build your project, if there are errors in the code, it will be displayed at the Build output window along with line number. If there are no errors code will be compiled and status will be shown at the Build output window.

**Debugging and Simulation**

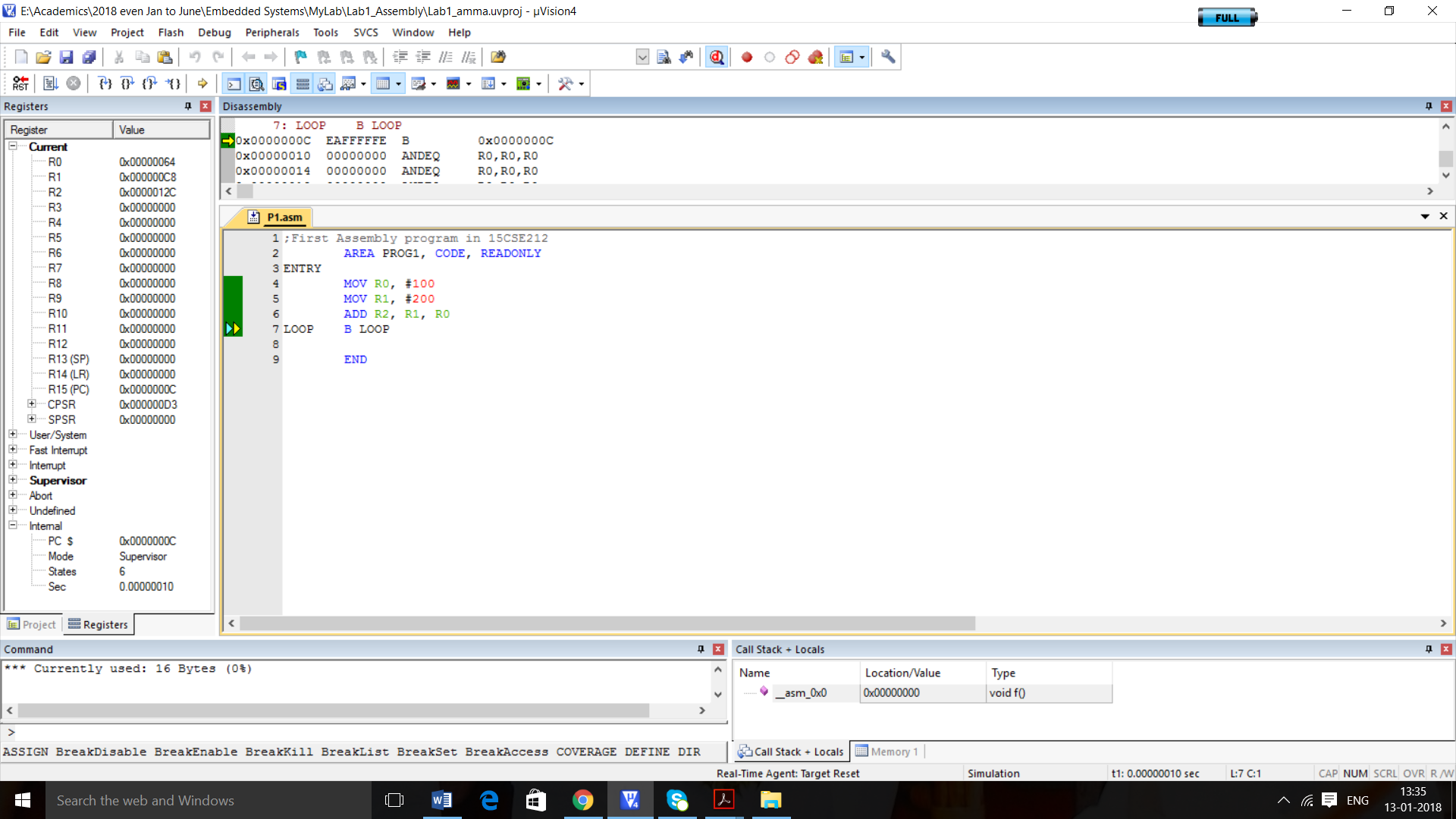
1. **Go to Debug  Start/Stop Debug Session / Ctrl+F5.**





**Debugging windows**:

There are some special windows provided by the Keil IDE, to help you to analyze the code and result through the Registers and Memory.

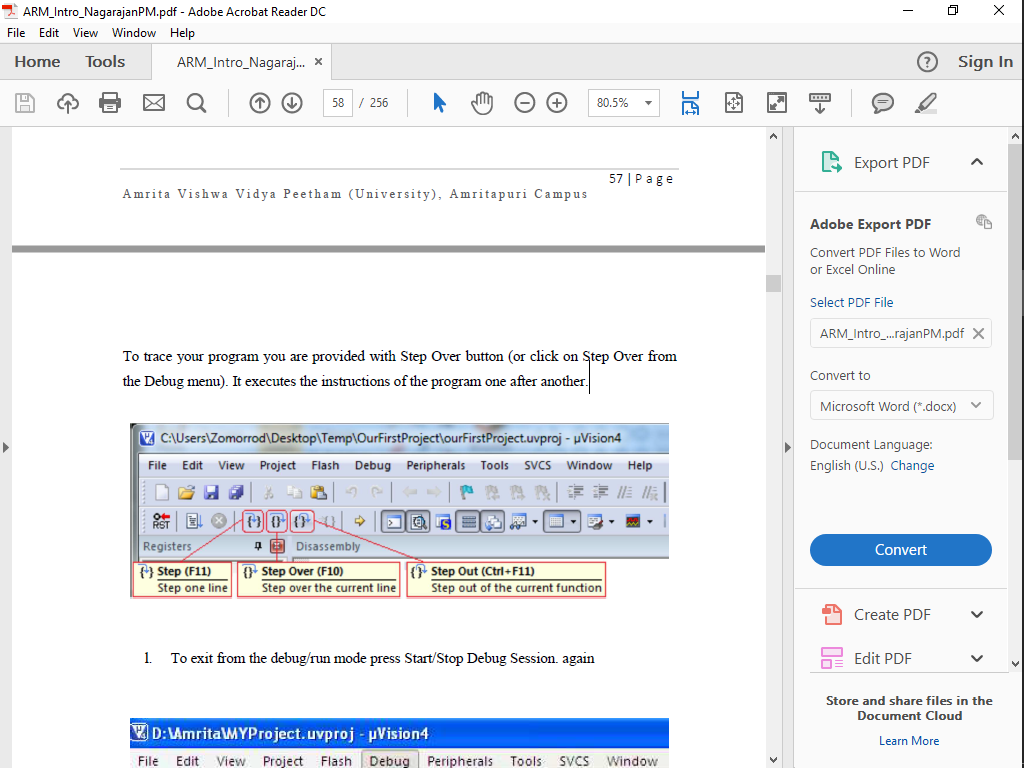


**Register Window**: The left most panel (in the following figure, it is walled in green) is

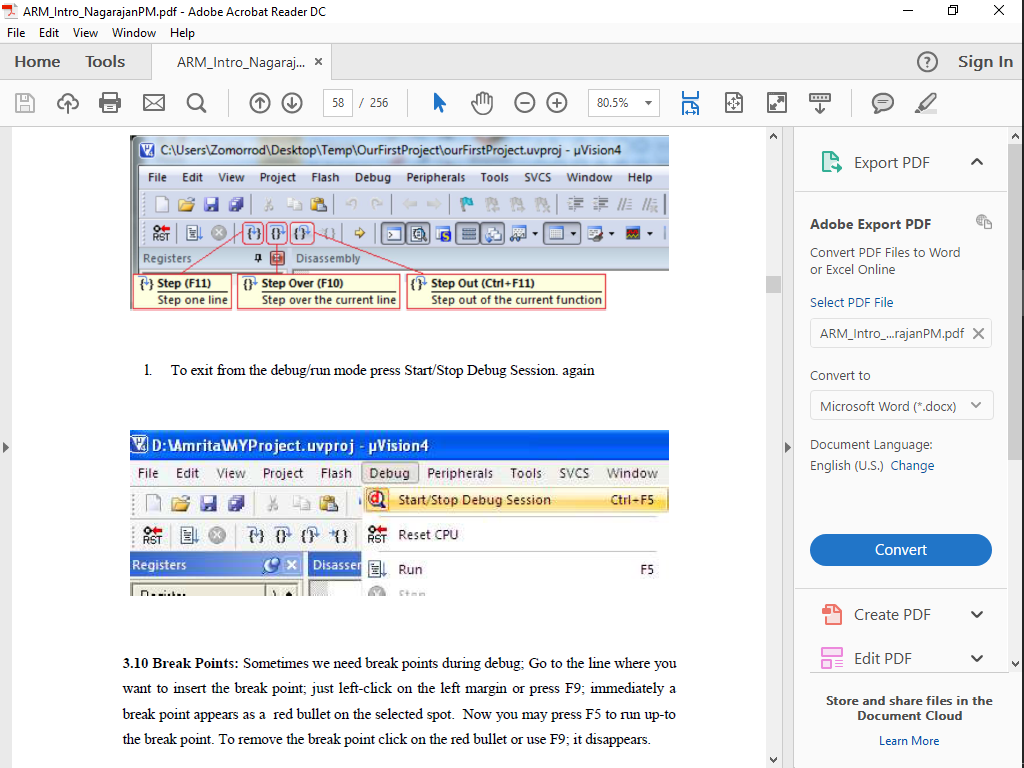
called register window that displays the contents of all the ARM processor registers and allows you to modify their contents. A register can hold any 32-bit value. That value can be a(signed) int, an unsigned int, a pointer (memory address). All C variables map onto the registers and large data structures like arrays will be in a memory. But ARM arithmetic instructions only operate on registers, never directly on memory. Register Window is can be very handy for debugging. Go through the required registers and verify their contents.

**Disassembly window:** In disassembly window, you see the listing, which can contain both C or high-level language program and its corresponding assembly instructions. Current instruction is pointed with yellow arrows; break points are shown in the margin as red bullets.

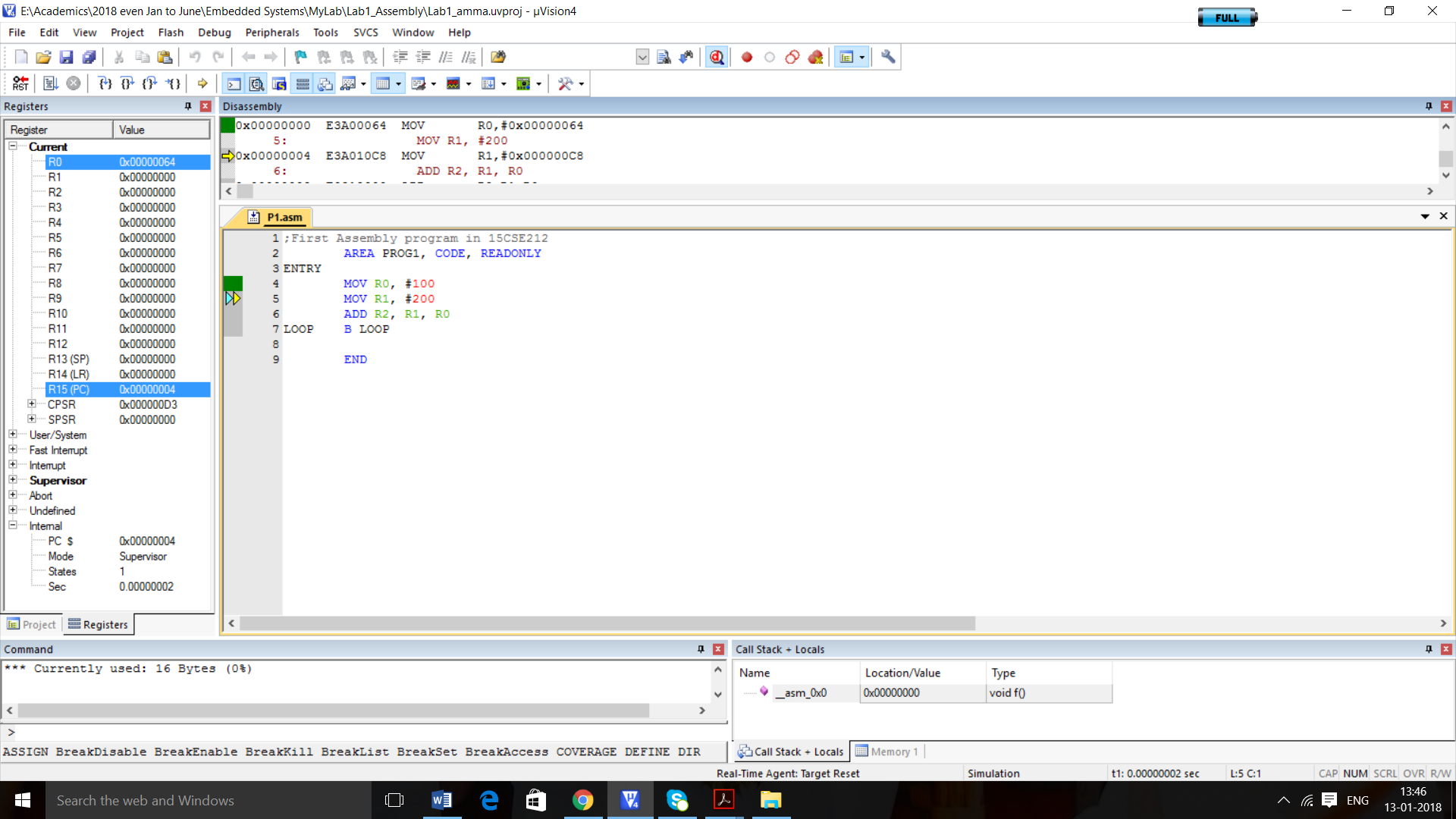
To trace your program you are provided with Step Over button (or click on Step Over from the Debug menu). It executes the instructions of the program one after another.



**After** the debugging is over, stop the debugging session by again selecting **Start/Stop** **Debug Session** from **Debug** menu.



**Questions to Answer:**



1. Based on the screenshot above, What is the current instruction that it is executing/debugging?

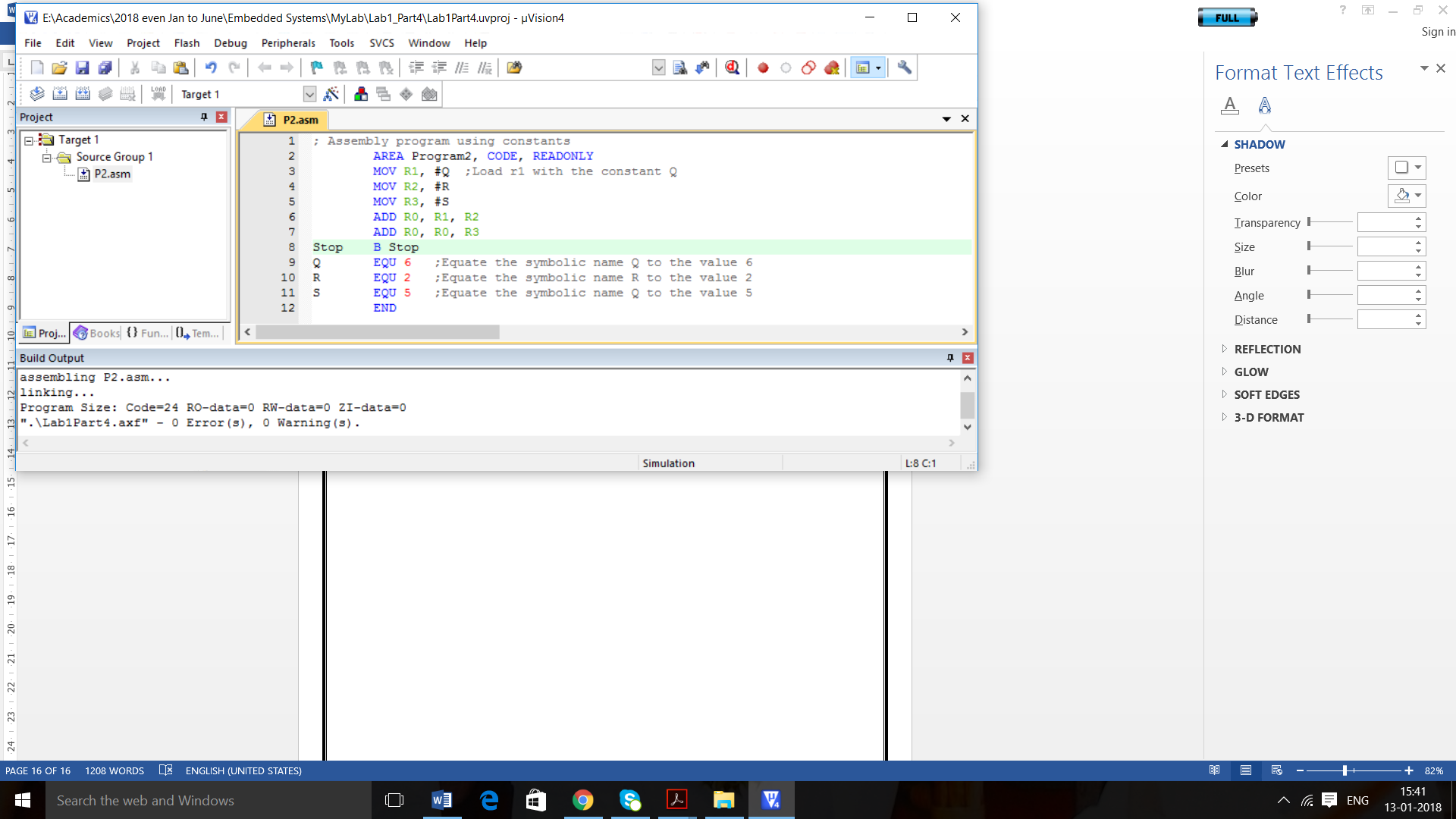
\_MOV R1,#200\_\_\_\_\_\_\_\_\_\_ is the instruction as specified in the P1.asm file and it is located in the address\_\_0x00000004\_\_\_\_\_\_ as shown in the listing in the disassembly window and the current instruction address is specified in the register \_\_R15(PC)\_\_\_\_\_\_\_\_\_\_\_.

1. After the execution of the entire program once, R0 contains the value \_\_\_\_\_64H\_\_\_\_\_\_\_, which is actually the hexadecimal of the decimal \_\_\_\_\_\_\_\_\_100\_\_\_\_\_\_\_. R1 contains the value \_\_\_\_\_\_C8H\_\_\_\_\_\_\_\_\_, which is actually the hexadecimal of the decimal \_\_\_\_\_\_\_\_\_200\_\_\_\_\_\_\_. R2 contains the value \_\_\_\_\_\_\_\_\_\_\_12C\_\_\_\_, which is actually the hexadecimal of the decimal \_\_\_\_300\_\_\_\_\_\_\_\_\_\_\_\_.
2. Write your assembly program to add the contents of registers R1, R2, R3, R4 and R5 and save the result in R0. You can initialize the contents of R1 to R5 as 0x10, 0x2A, 0x3B, 0x4C and 0x5D respectively. Ensure that you create this program file in the folder having the name as your roll number. Give the screenshot

**Part4 – Using Constants**

Problem P=Q+R+S with Q=6, R=2, S=5 and assume R1=Q, R2=R and R3=S. We can use symbolic names Q, R and S using EQU (equate) assembler directive.

The problem is solved using the assembly program shown below



Execute this program which is saved in a folder renamed as your roll number. Give the screenshot of successful execution and answer the following questions.

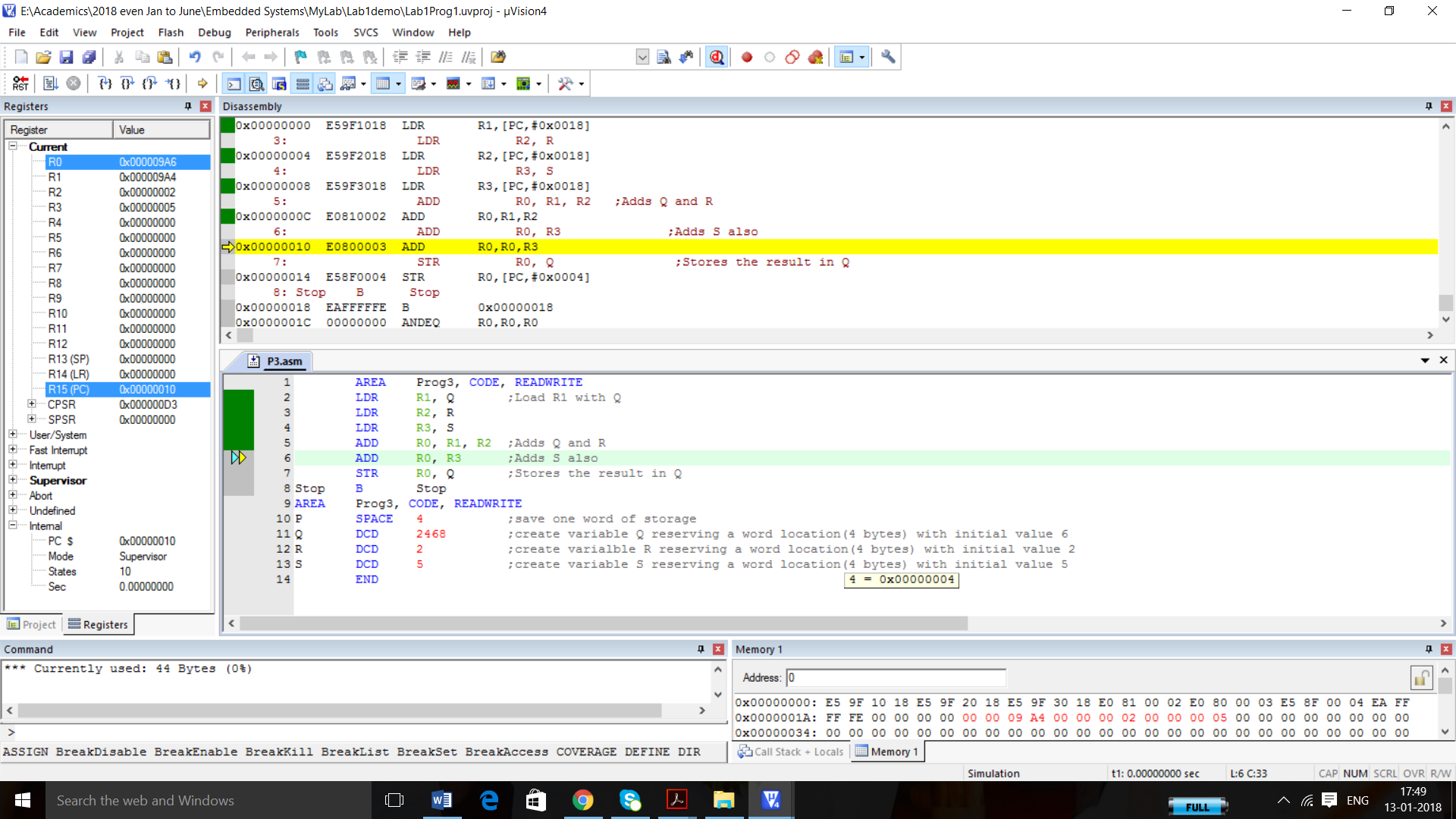
Which addressing mode is used in the first 3 assembly instruction in the program?

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_Indirect Mode\_\_\_\_\_\_\_\_\_\_\_\_\_. Give proper proof with the screenshot captured during the execution in the debug mode.

What is the final value of R0? \_\_\_\_\_D\_\_\_

**Part5 – Using Data in Memory**

Same Problem P=Q+R+S with Q=6, R=2, S=5 and assume R1=Q, R2=R and R3=S. We will use load register LDR R1, Q instruction to load register R1 with the contents of memory location Q. This instruction does not exist and is not part of the ARM’s instruction set. However, the ARM assembler automatically changes it into actual instruction.



Answer the following questions:

1. The code generated by ARM in the disassembly window for the pseudo instruction

LDR R1, Q is \_\_\_\_E59F1018\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

1. The value stored in the address 0x0000001A is \_FF EA 00 00 00 00 A4 09 00 00 02 00 00 00 05 00 00 00 00 00 00\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_
2. The value of PC when it is executing the first instruction is 0x00000004\_\_\_\_\_\_\_\_\_.
3. The addressing mode used in the LDR instruction is \_\_\_\_Indirect\_\_\_\_\_\_\_\_\_\_\_, because \_here we are taking the data from the memory address specified by the label.
4. The address of R is \_\_0x00000024\_\_\_\_\_\_\_\_\_\_\_\_, with the value 2 in memory.
5. The address of S is \_\_0x00000028\_\_\_\_\_\_\_, with the value 5 in memory.
6. In Little Endian (Current Scenario), the first byte address of Q is \_\_0x00000020\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ and it stores the value \_\_A4\_\_\_\_\_\_\_. The second byte address of Q is \_\_0x00000021\_\_\_\_\_ and it stores the value \_09\_\_\_\_\_\_\_\_. The third byte address of Q is \_\_\_0x00000022\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ and it stores the value \_\_\_\_00\_\_\_\_\_. The fourth byte address of Q is \_\_0x00000023\_\_\_\_\_\_\_\_\_\_\_\_\_ and it stores the value \_00\_\_.
7. The address [PC]+0x0018+8 = \_\_\_\_\_0x0020\_\_\_\_\_\_\_\_. If [PC] = 0x00000000 for the first instruction, then this address should refer to the starting address of Q. Note 8 bytes is added in ARM7, because of 3-stage pipeline concept.
8. You can change the Device ARM7 Little Endian to ARM7 Big Endian as shown in the below screenshot

Right click Target1 and select Options as shown below



Now select Device tab in the opening window and select ARM7 big Endian as shown below.

In Big Endian, the first byte address of Q is \_\_\_\_0x00000020\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ and it stores the value \_00\_\_\_\_\_\_\_\_. The second byte address of Q is \_0x00000021\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ and it stores the value \_\_\_00\_\_\_\_\_\_. The third byte address of Q is \_0x00000022\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ and it stores the value \_\_\_\_09\_\_\_\_\_. The fourth byte address of Q is \_\_\_\_\_0x00000023\_\_\_\_\_\_\_\_\_\_\_\_\_ and it stores the value \_A4\_\_\_\_\_\_\_\_.

1. If X=0x12345678 and starting address of X is 0x00000010, How this value will be stored in

Big Endian and Little Endian?

A)

* Little Endian

0x00000010 78

0x00000011 56

0x00000012 34

0x00000013 12

* Big Endian

0x00000010 12

0x00000011 34

0x00000012 56

0x00000013 78