1) From mycourses download this pdf, dxp_Lab2_c1.c, and dxp_Lab2_a1.asm.
2) The purpose of this lab is to become familiar with assembly language instructions, assembler directives and addressing modes. Understanding of the use of instructions, directives and addressing modes is essential to writing effective assembly language programs, allocation of the address space in terms of memory (volatile and non-volatile), and interfacing to peripherals.
3) **Part 1: Lab Exercise, learn what 'bis' and 'bic' do.**
4) The goal of this exercise is to write a small program, which will do the same thing, i.e. have the same effect, as the code between lines 41-47 in your fmlxxxx_Lab2_a1.asm program from last week's lab. First, create a new assembly only project in Code Composer named fmlxxxx_Lab3_a1. Then, add your fmlxxxx_Lab2_a1.asm file from last week to this project. Change its name to fmlxxxx_Lab3_a1. Your goal is to produce the same resultant values in r12, r13, r14 and r15 using the 'bis' and 'bic' instructions instead of 'mov' and 'clr'. In other words, comment out the program statements you recently added to fmlxxxx_Lab2_a1.asm, and replace them with a series of equivalent 'bis' and 'bic' instructions that will result in the same contents in registers r12, r13, r14 and r15. You can determine the function and use of the 'bis' and 'bic' instructions by carefully reading Chapter 3 of the MSP430 Users Guide, posted on mycourses, or the printed lecture handout. The statements you'll have to modify are listed below:

```
mov.w #0x12EF,r12 ;
clr.w r13 ;
clr.w r14 ;
clr.w r15 ;
mov.b r12,r13 ;
mov.w r12,r14 ;
mov.w r12,r15 ;
```

Notice that a "0" in the bit-mask used for bis or bic means the particular bit in that register will be left untouched. That means the register value before the instruction runs is important. Make sure your code does not rely on previous register values. DO NOT assume any initial register value! Include the series of bis and bic statements in your report.

5) **Part 2: How code and data storage is designated via the ASSEMBLER and LINKER programs.**
6) The MSP430 programmer's model centers around 16 general purpose 16-bit Registers (R0-R15), four of which are reserved for specific CPU functions (R0-R3). Given their limited number, these registers are usually used only as temporary storage locations within the microprocessor. Much more extensive information storage capabilities are available in the byte-

addressable memory locations of the RAM (volatile storage) and ROM (non-volatile storage). This memory can be used to establish named variables and constants typically found in higher-level languages like C, and often used in assembly language.

7) In a high-level language such as C, the C **COMPILER** decides, to a large extent, how to designate and allocate memory locations for variables and constants. In assembly language programming, it is up to the programmer to decide how and where variables and constants will be allocated in the memory space accessible to the processor. The final assignment and distribution of code (instructions) and data (variables and constants) is generally carried out by the **LINKER** in C and Assembly**.** So, both a COMPILER (for C) and an **ASSEMBLER** (for assembly) should have a mechanism to provide information regarding addressing and memory utilization to the **LINKER.**
8) In assembly language, most of the information regarding allocation and designation of memory space is carried out by directions to the **ASSEMBLER** (that can be passed on to the **LINKER),** also known as ***assembler directives*** or ***pseudo-operations,*** together with user-defined ***labels*** that symbolically represent specific addresses in memory.
9) Assembly language directives are designated by a leading period. For example: **.text**
is an assembler directive that indicates that the following assembly language code should be stored in a particular section of the address space associated with initialized binary information, in this case the binary encoding of the assembly language instructions.
10) If you wanted to store these instructions in a non-volatile fashion so that the code would be available for execution when power was applied to the microcomputer system, that section of memory would normally be assigned to some ROM device (e.g., FLASH memory). It would be up to the LINKER make sure that the executable binary information to be loaded into the target system contained enough information to make this possible, i.e., it would make sure that the binary information related to the assembly language code is assigned to a series of addresses allocated in the ROM on the target system.
11) Information regarding the ***mapping*** of the assembly language binary information to locations in the physical address space on the target system can be conveyed either via ***assembly language directives*** or via instructions in a ***linker command file*** provided to the LINKER when it is invoked along with the binary object file(s) produced by the ASSEMBLER.
12) (NOTE: Generally speaking, such information can also be provided by command line arguments when the ASSEMBLER or LINKER is invoked.)
13) Information regarding the ASSEMBLER and LINKER programs including their conventions and organization can be found in the MSP430 Assembly Language Tools User's Guide (Texas Instruments Document slau131g), posted on mycourses or accessible from the TI website. In particular, the following lab exercise will refer to information in Chapters 1, 2, 3, 4 and 7 in that document. It is a pdf file that is searchable (Ctrl – F will bring up a search window).

14) Refer to the MSP430 Assembly Language Tools User's Guide, and determine the meaning of the following assembler directives and record your understanding on the sheet attached to this lab report.
15) **.equ**
16) **.cdecls**
17) **.bss**
18) **.sect**
19) **.text**
20) **.end**
21) **.byte**
22) **.word**
23) **.short**
24) **Part 3: Understanding assembler directives and assembly language instructions.**
25) Create a new assembly only project named fmlxxxx_Lab3_a2. Create a new assembly language source file named fmlxxxx_Lab3_ArrayFill_a2.asm and enter the assembly language program listed in the file dxp_Lab3_ArrayFill_a2. ***It will be more time consuming, but much more useful if you type it and NOT copy/paste it.*** Build the project. Write and include in your report a short description of how you think this program works in terms of its operation and manipulation of the variable storage locations. Basically, it fills the locations in a manner that stores an increasing index value in each location as well as the sum of the index values. If there is a problem in understanding a part of this code, remember that it can be stepped through in the debugger and locations in the memory can be observed. **The use of stepping through code and observing what it does is essential to debugging and understanding an assembly language program.** Also, when you are going through the program, identify each assembler directive in it. Indicate in your report which assembler directive appears on which line.
26) In the disassembly pane locate the locations of the first and last instructions of the program. Calculate the length of the assembled machine code in hexadecimal and bytes. You'll need this value in part 5.
27) **Part 4: Understanding assembler directives and assembly language instructions.**
28) For the code in dxp_lab3_ArrayFill_a1.asm, write down the addressing mode used for the source and/or destination in each line of the code. In your report, list the line number and the addressing mode(s) used. Observe how these addressing modes affect the values in the registers and memory locations, before and after each instruction execution.
29) **Part 5: C code that has the same effect as the assembly code in part 3.**
30) The C code in dxp_Lab3_ArraFill_c1.c does the same as the assembly code in dxp_lab3_ArrayFill_a1.asm. Create a new project fmlxxxx_Lab3_c1, and copy the code into the file fmlxxxx_Lab3_ArrayFill_c1.c.
31) First, run the code with breakpoints in lines 23, 38, and 47. After each run to the next breakpoint, look at the contents of the memory locations starting at

address 0x09EA. Identify in which location the indices i and j, the sum, and the elements of the array are stored. Include these in your report.

32) In the disassembly pane locate the locations of the first and last instructions of the program. Calculate the length of the assembled machine code in hexadecimal and bytes. Compare this with the length of your assembled machine code from part 3 or step26. Most probably the C code is ~3 times larger than the assembly code. Why do you think this is the case? Include the two code sizes, and explain why you think the C code is larger, in your report.

33) Make sure you write the report and uploaded along with your project archives on myCourses. As time permits, demo the intermediate steps to the TA.

34) Grading:
    a. Part 1 = 10 points
    b. Part 2 = 5 points
    c. Part 3 = 5 points
    d. Part 4 = 5 points
    e. Part 5 = 5 points.