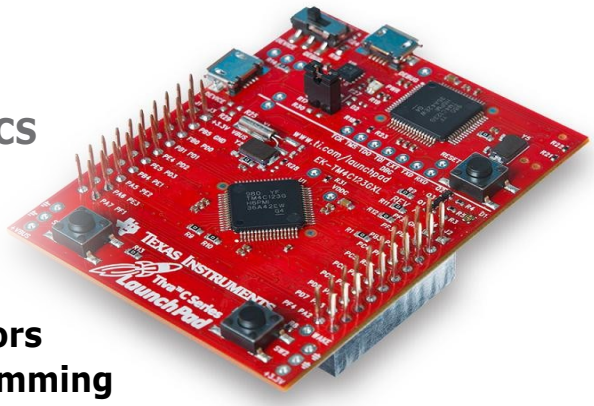
ELECTRICAL AND ELECTRONICS
ENGINEERING
DEPARTMENT



Introduction to Microprocessors
Laboratory with Assembly Programming

Introduction to Microprocessors Laboratory with Assembly Programming

Objectives

The ARM Cortex-M4 processor executes a set of 16 and 32 bit instructions, called the Thumb-2 instruction set, yielding a good tradeoff between code density (16 bit instructions) and performance (32 bit instructions). In EE447, assembly language is the programming language of choice. This laboratory work will introduce the basics of the Thumb-2 Instruction Set and the software development environment for developing and debugging programs using our Arm Cortex M4 processor and Tiva Launchpad development board. At the end of this work, you will be able to:

- Understand the fundamentals of assembly programming
- Recognize and use several Thumb-2 instructions
- Understand Keil μVision (a tool for embedded software developers who write software in assembly language or C for microcontrollers).
- Run a program and examine changes to memory and registers.
- Use breakpoints and stepping to debug a program.
- Understand how to use several assembler directives.
- Write simple assembly programs to perform multi-byte addition and subtraction.
- Use the status flags with conditional jump instructions.
- Input from the keyboard and output to the terminal screen using monitor utilities

A complete understanding of this laboratory work is necessary since you will perform all of these procedures in your future laboratory works and demonstration sessions.

1 Background Information

1.1 Assembly Programming Fundamentals

An assembly language, like many other computer languages, consists of a series of instruction statements that tell the CPU what operations to perform. An assembly language has three types of statements: Assembler Directives, Instructions, and Comments. Assembler directives are interpreted by the assembler to define program constants and reserve space for program variables, as simple examples. Assembly instructions are divided into two sets of fields: Operation (or Mnemonic) and Operand fields. The Operation field consists of the mnemonic names for the machine instructions. The Operand field contains the operand(s) and assembler directive arguments (if necessary). The Comment field is an optional field that allows programmers to document their code. Comment fields are ignored by the assembler and begin with a semicolon ';'. Finally, there is another optional field called the Label field. A label allows a programmer to use symbols for memory locations, which make assembly programs easier to read and write. As it turns assembly code into object code, the assembler "replaces" each label with its respective memory location. Labels will be used extensively when you begin programming, but for now, let's put this idea on hold. When you begin writing assembly programs, you will follow a format specified for the class; however, each instruction will use the following structure: LABEL, OPERATION, OPERAND, COMMENTS.

NOTE: The Operation field contains the assembly mnemonic for the corresponding machine instruction and cannot start in the first column of the program (only labels can exist in the first column). A common practice is to separate each field with a tab (or multiple tabs). The operand field contains the arguments required for the machine instruction or assembler directive, with comments following at the end of the instruction.

Let's look at the example below and clarify some common nomenclature used in this course. First, notice that the label field is not used in this example. Second, the Operation field contains the mnemonic for the machine instruction: MOVe. The first operand, r2, is a register name, in this case the name of the register to be loaded. The second operand, #10, is the decimal number 10. The '#' symbol refers to a type of addressing mode, which is discussed below, but in layman's terms, this instruction is simply telling the CPU to put the value of 10 (base 10) into register r2. The comments field shows this by using some common nomenclature that will be used in this course. The second instruction instructs the CPU to add the value 11 (base 10) to the contents of register r2 and store the result in register r4. The brackets around the register name in its comment field indicate the contents of register r2. Note that these comments, describing the behavior of the instructions, are NOT appropriate when you are writing your programs, but they will be used extensively for instructional purposes here and in class.

LABEL	***OPERATION***	***OPERAND***	***COMMENTS***
	`mov`	`r2, #10`	`; move 10d into reg #2`
	`add`	`r4, r2,#11`	`; [r2] + 11 → r4`

What do you think the contents of registers r2 and r4, i.e. [r2] and [r4], will be after these two instructions are executed?

Now that we have some idea of what instructions look like, let's look at the tool we will be using to work with programs, which are just an ordered collection of instructions, directives, and comments.

1.2 Board startup

CAUTION: Your development board is sensitive to static electricity. Be sure to touch something grounded before touching the board.

The Texas Instruments EK-TM4C123GXL is a single board computer development system for the TI TM4C123GH6PM microcontroller (one of the TM4C123 family). Support software for this development board is provided for Windows XP/Vista/Windows7/Windows10 operating systems. The EK-

TM4C123GXL board is a small development board designed for low cost, and has several common I/O ports. However, you won't have to worry about using any I/O ports or a breadboard for several more lab sessions.

The EK-TM4C123GXL development board uses option select jumpers to configure the board operation. The jumpers should already be set for default board operation.

A "**Jumper**" is a plastic encased wire shunt that connects two terminals electrically. **Make sure that the VDD jumper is placed on your board.**

REMARK: Ensure that the USB cable is plugged in DEBUG port and the SELECT switch is in DEBUG mode.

1.3 Create a Project Folder

You will need to create a project folder in an appropriate place in your computer file system. On your own computer a good folder path could be something like . . . \Documents\EE447\Projects . Each of your EE447 lab projects should go in its own folder within this folder.

Using Keil μVision with development board:

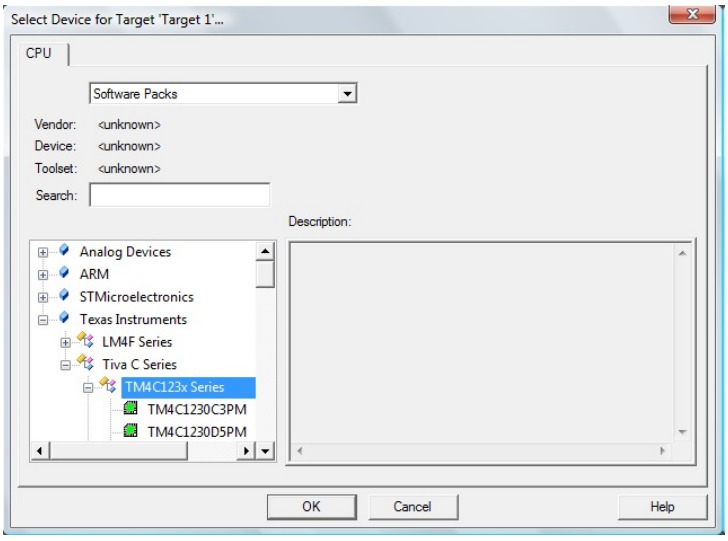
Run the μVision application, making sure you see the 2 sub-windows: the Project window on the left, the Build Output window along the bottom. The Build Output and Project windows can be made visible or invisible using the View menu at the top of the program.

All files associated with each of your projects, debug, and build settings are organized and saved using Project files. Each project will contain all of the assembly files created for an individual lab or project.

To start a new project, go to the Project menu > New μVision Project

Create a project folder, then navigate to the Project Folder you just created and select a name for your project.

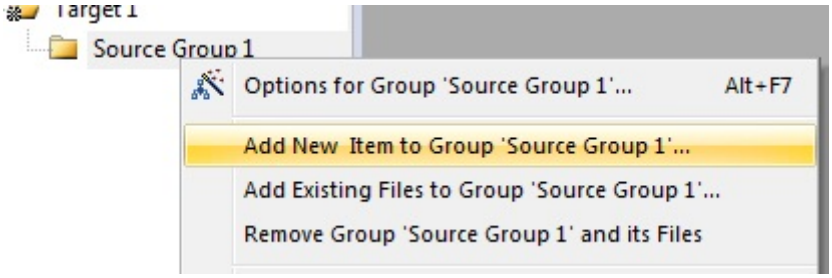
Next, select the device you are using. In our case, the TI TM4C123GXL development board uses the TM4C123GH6PM microcontroller. Expand the tree inside of the Target window to find the correct processor. Texas Instruments > Tiva C Series > TM4C123x Series. Scroll down until you find the TM4C123GH6PM and select it, then click OK.



The next window to pop up is the Manage Run-Time Environment window. However, for an Assembly project, nothing needs to be configured here. Simply select OK.

You will notice that the Project window now has a tree structure. Expanding the Target1 folder shows the Source Group 1 folder. The Source Group 1 folder is where all of your assembly files will be added.

There are two ways to add assembly files to your project. Right-click on the Source Group folder and either select Add New Item to Group –or- Add Existing Files to Group.



The Add **Existing** option is for when you already have an existing file that you would like to use in your project.

The Add **New** Item option is used when starting a file from scratch. Selecting this option walks you through creating a new file and opens the new file as a new tab in the Editor.

Any new or old program file can be opened in the Editor (like any word processor) using either the File menu or double clicking it in the Project window. Saving the files can be done by selecting File > Save, or the save button in the menu.

REMARK: If you want a file to be used in your project it MUST be added to the Source Group folder. Otherwise, μVision does NOT consider it during build time.

1.4 Running an application using μVision

Right-click on the Source Group folder and select Add New Item to Group, name your file as PracticeLab. Open the assembly file and type (or copy / paste) the program in ***PracticeLab.s*** in the Editor Window. Save the file by following File > Save on the menu bar. Download ***Startup.s*** file from ODTUCLASS and add it to the project as well. **You will add this file to all your Assembly projects.** Now build (or assemble) this program by following Project > Build on the menu bar. This will generate PracticeLab.lst in the Listings directory. Connect the board to the PC and download the program to the board.

```
;*************************************************************
; Program PracticeLab.s
; Clear memory locations 0x2000.0400 - 0x2000.041F,
; then load these locations with consecutive numbers starting
; with 00.
; 'CONST' is the number of locations operated on.
; 'FIRST' is the address of first memory address.
;*************************************************************

;*************************************************************
; EQU Directives
; These directives do not allocate memory
;*************************************************************
;SYMBOL          DIRECTIVE       VALUE                   COMMENT
FIRST           EQU             0x20000400
CONST           EQU             0x20
;*************************************************************
; Program section
;*************************************************************
;LABEL           DIRECTIVE       VALUE                   COMMENT
                AREA            main, READONLY, CODE
                THUMB
                EXPORT          __main
__main          LDR             R1,=FIRST               ; Initialize registers
```

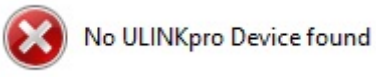
```
                MOV             R0,#0x00                ;
                LDR             R2,=CONST               ;
loop1           STRB            R0,[R1]                 ; Clear memory
                ADD             R1,R1,#1                ; Increment address
                SUBS            R2,R2,#1                ; Decrement contant
                BNE             loop1
                LDR             R1,=FIRST               ; Reset address
                LDR             R2,=CONST               ; Reset constant
loop2           STRB            R0,[R1]                 ; Store value in memory
                ADD             R0,R0,#1                ; Increment value
                ADD             R1,R1,#1                ; Increment address
                SUBS            R2,R2,#1                ; Decrement contant
                BNE             loop2
done            WFI                                     ; Infinite loop to
                B                       done            ; end program
;*************************************************************
; End of the program section
;*************************************************************
;LABEL          DIRECTIVE       VALUE                   COMMENT
                END
```

Once an assembly program has been written, it will need to be built (assembled). This can be done using Project > Build target. When the program is built, other files with the same name, but with other file extensions are created. The .lst file is a common listing file, which provides physical address information with operation and operand information. It also gives the error information above the line having the error. Make sure to edit the main program and not the .lst file.

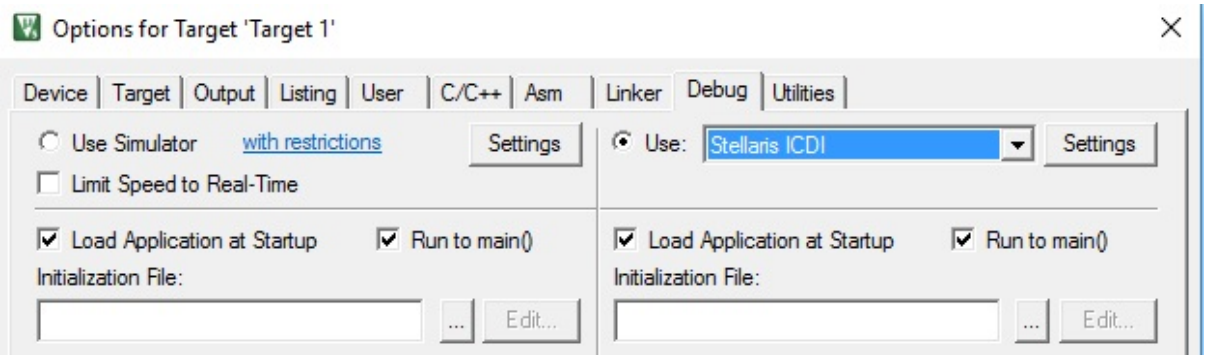
Upon a successful build, the Build Output window will display something similar to the following:

```
Build target 'Target 1'
assembling Lab1.s...
linking...
Program Size: Code=40 RO-data=0 RW-data=0 ZI-data=1024
".\Objects\Lab1.axf" - 0 Error(s), 0 Warning(s).
```

Having built your program, the next step in the process is to download it to your board. From the toolbar, choose Download . The Build Output window shows messages about the download progress. If the pop-up window shown below appears and/or Build Output window shows an error message, then the driver may not be set up properly.



In order to use the correct driver for TM4C123GH6PM microcontroller select Project > Options for target > Debug and choose Stellaris ICDI from drop down menu like shown below.

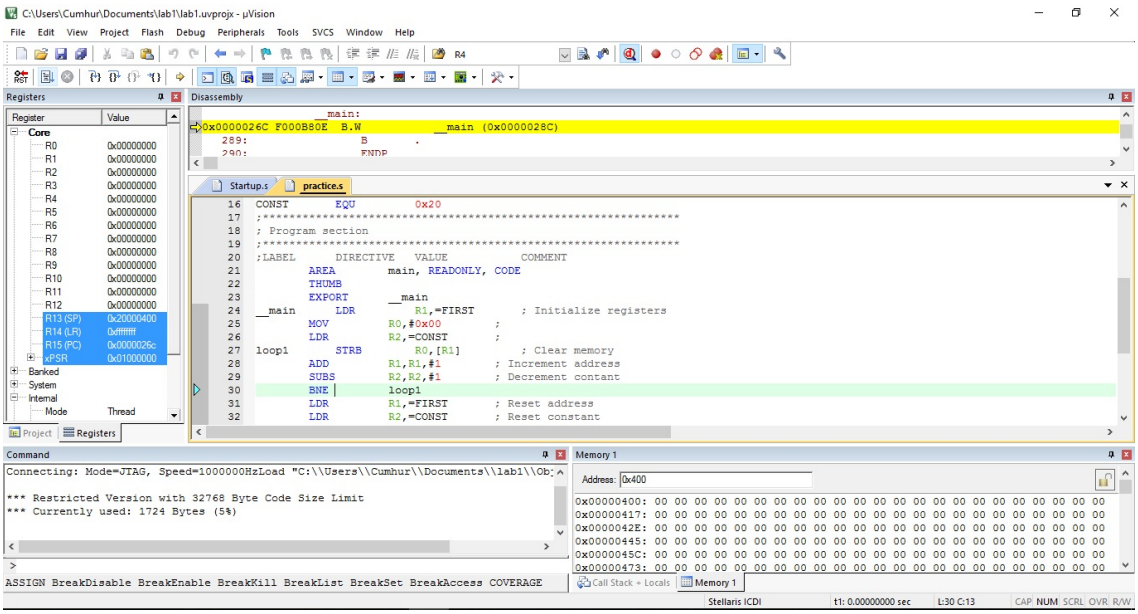


1.5 Debugging:

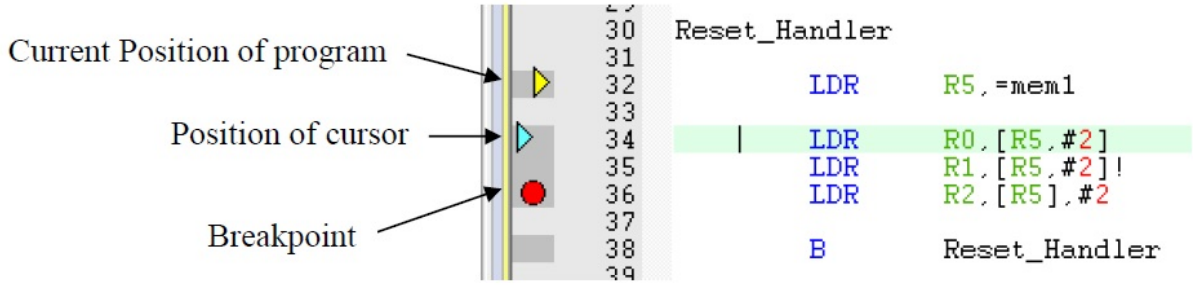
After a successful build/download, you can now enter debug mode using Debug -> Start/Stop Debug Session or the Debug icon in the top menu. When entering the debug session, the look of μVision will change to display windows helpful to debugging.

1. On the left is the Register window that shows the current values of each of the registers.
2. In the middle is the Editor window that contains the files to be debugged.
3. At the top is the Disassembly window which shows the .lst file that was created during the build.
4. On the bottom left is the Command window which shows any processes currently running by μVision . It also shows any errors that may come up along the way.
5. The bottom right contains two windows. The Call Stack and Memory windows. The Call Stack window shows current status of the stack and variables of the project. The Memory window is used to view and (in some cases) modify the values at memory addresses.

Other options to Run, Stop and Step through your program also become available.



When first entering the Debug Session, the program will be pointed to the very first operation of your program. The current position of the program will be indicated by a yellow arrow. Note that the current line of the program has NOT yet been executed. The cyan arrow is simply the line your cursor is on.



Right after entering the Debug Session, your program does NOT automatically run. It waits at the first line of code waiting for input from you. You have the option to Run or Step through your code.

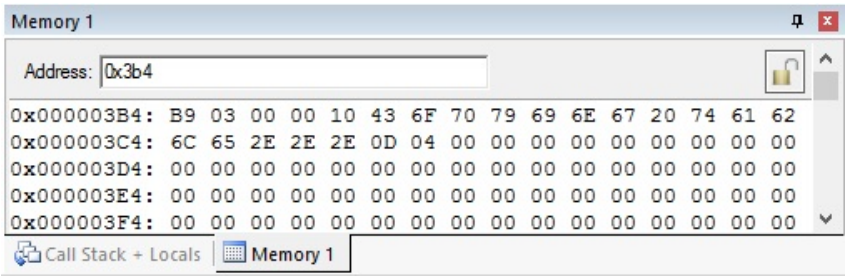
To run the program, select Debug > Run, or the Run icon in the debug toolbar (key F5). Selecting Run will let the program keep executing operations in real time until it either comes to a line with a breakpoint or the end of the program.

To Step through your program line by line select Debug > Step or the Step icon in the debug toolbar (key F11). Selecting Step will execute one line only.

Breakpoints are markers you set at places you would like the program to "pause". When a line with a breakpoint has been reached, the program stops on that line. All values in memory and registers are current allowing you to check if they are the values you expect. Note that breakpoints do not end the program. They simply "pause" it. You can continue the program by clicking Run or by Stepping. You can add breakpoints two different ways. One way is to simply click in the dark gray area left of the line you wish to stop on. Another way is to place your cursor on the line you would like the program to stop on, and then select Debug > Insert/Remove Breakpoint (key F9). If a breakpoint already exists, the same actions will remove the breakpoint.

1.6 Memory and Registers

μVision allows you to view the contents of any memory address using the Memory Window. At the top of the Memory Window you can type in the address you wish to view. For example, typing 0x000003B4 will display the values in memory starting at address 0x000003B4. Additionally, μVision understands 0x000003B4, 0x03B4, and 0x3B4 are the same addresses. Note that μVision assumes leading zeros.



It can be seen that the values are grouped in twos. This is because the Memory Window follows the physical memory addressing design of the TM4C. Each 32-bit memory address is a pointer to an 8-bit space of memory. Each group of two hex numbers are 8-bits (1 byte), therefore each group of two hex numbers are a different address. For instance in the example above the memory and values are as follows:

Address	Value
0x000003B4	0xB9
0x000003B5	0x03
0x000003B6	0x00
0x000003B7	0x00

To modify any of the values, simply double click the value and type in a new one (Yes, you can modify

the memory of your board just like that). Note that some memory locations are read only, and cannot be modified.

The Register window functions in a similar manner, but shows the current values of the registers located on the CPU of the TM4C. Each register is 32-bits wide, so when storing register values in memory they will take up 4 address spaces in memory. To modify a register value, double click on the register value and enter a new value.

Note: *Modifying memory and register values are done only to assist in debugging your program. They should not be modified as a regular process, or as input to a program. Memory and Register windows are most helpful as a way to view the values of specific memory locations and registers.*

Address Start	Address End	Memory application
0x0000.0000	0x0003.FFFF	On-Chip Flash *(Where you will program)*
0x0004.0000	0x1FFF.FFFF	Reserved
0x2000.0000	0x2000.7FFF	Internal SRAM
0x2000.8000	0x220F.FFFF	Reserved
0x2210.0000	0x3FFF.FFFF	Reserved

Figure 1: Memory Map for TM4C123GXL Development Board

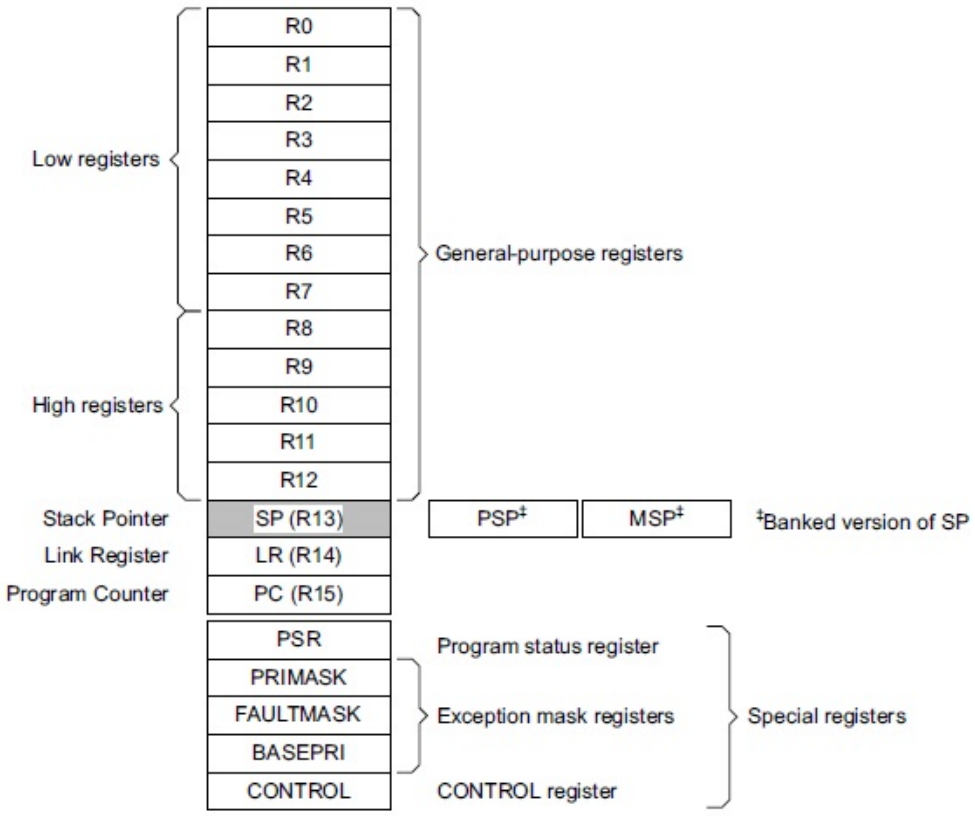


Figure 2: The Cortex-M4F Register Model

When the program is downloaded to the board, it is stored starting at address 0x0000.0000. Enter Debug mode and step through the program. In the MemoryWindow, examine memory locations 0x0000.0000 – 0x0000.003F. These memory locations should contain the object code of the program. You can compare the Disassembly Window to the values in memory. Notice in the Disassembly Window the starting point of the program (__main) is pointed to address 0x0000.028C. If you look at the value at that address it is 0x490A. Look at address 0x0000.028C, 0x0000.028D in the Memory Window and you should see 0A 49.

Now look at memory locations 0x2000.0400 – 0x2000.041F. This program modifies these memory locations by first loading 0x00 into all of them, then storing consecutive numbers starting with 0x00. Step through the program by pressing F11. Watch as the program clears the memory locations and then populates it with consecutive numbers. Also pay attention to the values in the registers and how they relate to the instructions being executed.

1.7 Debugging The Program

Here we will introduce a few more debugging tools. When you start writing larger programs it won't always be practical to single-step the entire program, but you can stop at particular places by using breakpoints. After you have stepped through the program and understand how the program works, reset the program by selecting Debug > Reset CPU or the Reset icon in the debug menu. This places the Program Counter back at the beginning of the program. Place a breakpoint at loop2 and Run the program (F5). The program should have run through the first loop, clearing the memory, and stopped at your breakpoint. Notice that memory locations 0x2000.0400 – 0x2000.041F are zeros again. Pressing F5 again allows the program to run through 1 cycle of loop2 because it hits the breakpoint again. Remember you can still Step after breakpoints as well.

1.8 Assembler Directives:

There are a few assembler directives we need to discuss before you continue writing programs. Refer to the sample program ***Program_Directives.s*** for reference while you are going through this section. You can download this program from ODTUCLASS. This program makes use of some of the most common directives you will be using throughout the semester and is designed to copy a table from one location to another location in the internal RAM (0x2000.0400 – 0x2000.041F). There are several other directives that can be found under the μVision help menu. Some of these we will use and discuss later in the semester, so for now, just get to know the EQU, DCB, SPACE, FILL and END directives discussed below.

EQU

The EQU directive is used to assign a value to a symbol definition. For example, notice in the Data Section of the program below that there are two symbol definitions: OFFSET and FIRST. The OFFSET symbol is assigned a value of 0x10 and the symbol FIRST is assigned a value of 0x2000.0400. At build time, the assembler will "replace" any symbol definition with its corresponding value. For example, the corresponding instruction on line 58, STRB R0,[R1,# OFFSET] is the same instruction as STR R0,[R1,#0x10]. Using symbol definitions for subroutine names and for values used in multiple places not only makes your code easier to follow, but also allows you to make a single change to a value versus having to make multiple changes everywhere the value is used in the code.

DCB

The DCB directive stands for Define Constant Byte, and is used for defining the value of a byte (or bytes) at a given memory location. Let's look at the same example program. In the data section, you notice the instruction CTR1 DCB 0x10. "CTR1" (our abbreviated name standing for "counter 1") is the label, and 0x10 is the byte value. Is this different than the EQU directive? Yes! The EQU directive assigns a value to a label and is only used by the assembler at build time; however, in this case, the CTR1 label does not equal a value of 0x10; but instead refers to the address that the value 0x10 is stored in. If you were to build *Program_Directives.s*, download this project to your TM4C123 and look at memory location 0x3AC, it would contain the value 0x10. Using the label CTR1 in your program code refers to the address 0x3AC where the value 0x10 is stored. Now look at the following line in the program.

You notice the directive MSG DCB “Copying Table...”. This directive will store the ASCII byte values of ”Copying Table...” into memory when the program is downloaded before program execution begins. The first character (quotation mark “) is interpreted as the delimiter and is not included as part of the stored ASCII bytes (this also applies to the last quotation mark as well). At what addresses are these ASCII bytes located? The address where the first ASCII byte value (‘C’) is stored is 0x3AD. The next instruction will continue at the next byte in memory, which means the message ”Copying Table...” is located at addresses 0x3AD – 0x3BC. If you were to load this program to your TM4C123 board and examine memory 0x3AD – 0x3BC, you would see the ASCII byte values of the message ”Copying Table....” It is also important to note that instead of using multiple DCB directives for a sequence of byte definitions (that don't need individual labels) you can simply use a comma or space and define multiple bytes using the DCB directive once. For instance, if you needed to define constant byte values 0x01, 0x02, and 0x03 in sequential memory locations, you could simply type DCB 0x01,0x02,0x03.

SPACE/FILL

SPACE is used to reserve a number of bytes of memory, filling them all with zeros. FILL is used to reserve a number of bytes of memory, filling them all with a desired value.

```
;LABEL          DIRECTIVE   VALUE                    COMMENT
               AREA        myData,DATA,READWRITE
block_empty    SPACE       255                      ;Allocate 255 bytes of
                                                    zeroed memory space
block_fill     FILL        255,0xDD                 ;Allocate 255 bytes and
                                                    set each byte to 0xDD
```

END

Finally, the END directive indicates the logical end of a source program. Any statement following the END directive will be ignored by the assembler. The END directive is not necessary for successful compilation of your code; however it is good practice to use the END directive and show where your source code ends.

As mentioned before, there are many other directives available for the assembler. The five mentioned above will be the ones you use regularly for programming and other directives for different purposes will be introduced later in the semester. For now, make sure you fully understand the directives above and how to use them.

```
;*****************************************************************
; Program_Directives.s
; Copies the table from one location
; to another memory location.
; Directives and Addressing modes are
; explained with this program.
;*****************************************************************
;*****************************************************************
; EQU Directives
; These directives do not allocate memory
;*****************************************************************
;LABEL      DIRECTIVE   VALUE       COMMENT
OFFSET      EQU         0x10
FIRST       EQU         0x20000400
;*****************************************************************
; Directives - This Data Section is part of the code
; It is in the read only section  so values cannot be changed.
;*****************************************************************
;LABEL      DIRECTIVE   VALUE       COMMENT
            AREA        sdata, DATA, READONLY
            THUMB
CTR1        DCB         0x10
```

```
MSG         DCB         "Copying table..."
            DCB         0x0D
            DCB         0x04
;*************************************************************
; Program section
;*************************************************************
;LABEL      DIRECTIVE   VALUE       COMMENT
            AREA        main, READONLY, CODE
            THUMB
            EXTERN      OutStr      ; Reference external subroutine
            EXPORT      __main      ; Make available

__main
start       MOV         R0,#0
            LDR         R1,=FIRST
            LDR         R2,=CTR1
            LDRB        R2,[R2]
loop1       STRB        R0,[R1]
            ADD         R0,R0,#1
            ADD         R1,R1,#1    ; Store table
            SUBS        R2,R2,#1
            BNE         loop1
            LDR         R0,=MSG
            BL          OutStr      ; Copy message
            LDR         R1,=FIRST
            MOV         R2,#0x10
loop2       LDRB        R0,[R1]
            STRB        R0,[R1,#OFFSET]
            ADD         R1,R1,#1    ; Copy table
            SUBS        R2,R2,#1
            BNE         loop2
            B           start
;*************************************************************
; End of the program  section
;*************************************************************
;LABEL      DIRECTIVE       VALUE                       COMMENT
            ALIGN
            END
```

In Program_Directives.s above, 'loop1' creates a table in the address range 0x2000.0400 - 0x2000.040F. These address locations contain 0x00 - 0x0F at the end of 'loop1'. The second loop, ('loop2') copies the table in the address range 0x2000.0400 - 0x2000.040F to the address locations 0x2000.0410 - 0x2000.041F.

1.9 Branching and Relative Addressing:

Relative addressing is used only by branch instructions, and branch instructions use only the relative addressing mode. A branch instruction makes a decision on whether or not to alter program flow based on information in the Program Status Register PSR. Refer to the lecture notes for a discussion of the condition code bits and their meaning.

The branch instruction you will probably use the most is BEQ or branch on equal to zero. In a program you will frequently test a number to see if it equals another number. One way you could do this is by subtracting one number from another and leaving the result in a register. This can update the status bits. Compare instructions do the same thing, but do not change the value in the register. If the numbers are equal, the 'Z' flag, or zero bit, will be set. In other words, if the result of an operation is zero, the zero bit will be 1. The BEQ instruction can then be used to either jump somewhere else in the program

or continue with the next instruction. Refer to the following code.

```
start   LDR R0,=0x20000900
        LDR R1,=0x20000910
        LDR R0,[R0]
        LDR R1,[R1]
        CMP R0,R1
        BEQ done
        ADD R0,R0,#0x25
done    WFI
        B   done
```

This code loads a number into register R0 from memory location 0x2000.0900, then compares it to a number in memory location 0x2000.0910, then tests the result. If they are the same, the program jumps to the label 'done'. If not, the next instruction is executed. The program relative address jumped to is defined as 'done' when you assemble the program. The instruction B, branch always is an unconditional jump, i.e., you always branch on this instruction.

Many times a program will perform an operation a certain number of times in a loop and then continue. It would be nice to just say 'for i = 1:100 ...', but this is assembly language, so it is not so easy. One way to implement a loop is to initialize a counter; then decrement it each time the loop is executed. The decrement instruction is followed with a BNE, branch on not equal to zero, which causes the program to jump back to the top of the loop. When the counter hits zero, the program executes the next instruction after the loop.

There are several branch instructions in the TM4C instruction set. Refer to the Instruction Set table and make sure you understand how they work before you use them. Also, make sure that the flags are being set the way you think they are by the instruction before the branch.

1.10 Monitor Utility Subroutines

The monitor utility subroutines are a set of routines that allow communication between the TM4C board and a serial window. We will introduce three of them this week. The InChar subroutine inputs the ASCII character from the keyboard of the PC to register R0. For example, the upper case letter 'A' is loaded into register R0 as 0x41. The subroutine Inchar loops until you press a key. Similarly, OutChar subroutine outputs an ASCII character from register R0 to Termite screen.

The program in below shows how InChar and OutChar could be used. Type up this program and run it. What happens? We will go into more details of subroutines in the next lab, but for now think of BL as an unconditional jump with a way to get back where it came from. The CMP instruction in Line 2 compares the character input to 0x20, which is the ASCII code for the SPACEBAR. This program echoes every key you hit using OutChar until you hit SPACEBAR. Refer to Appendix C for the ASCII table.

```
get     BL      InChar
        CMP     R0,#0x20
        BEQ     done
        BL      OutChar
        B       get
done    B       done
```

The other useful subroutine is OutStr. This subroutine outputs a string of ASCII bytes pointed to by the address in register R0 until the end of transmission character, 0x04, is reached. If you want to display different messages to the terminal screen when certain conditions are met, you can have the string stored in memory with the 0x04 after the last character. Remember that we have already used this subroutine in the program in *Program_Directives.s*. This subroutine prints 'Copying table...' on the screen. The program stores this message in the memory and the starting address of this message has the label 'MSG'. On the following line is 0x0D, which is ASCII for a Carriage Return (new line). The end of transmission

character, 0x04, follows the actual message. In loop 2, the address of 'MSG' is loaded in index register R0 and in the following line, this message is printed on the screen using OutStr subroutine.

1.11 Setting up Termite Terminal Emulator

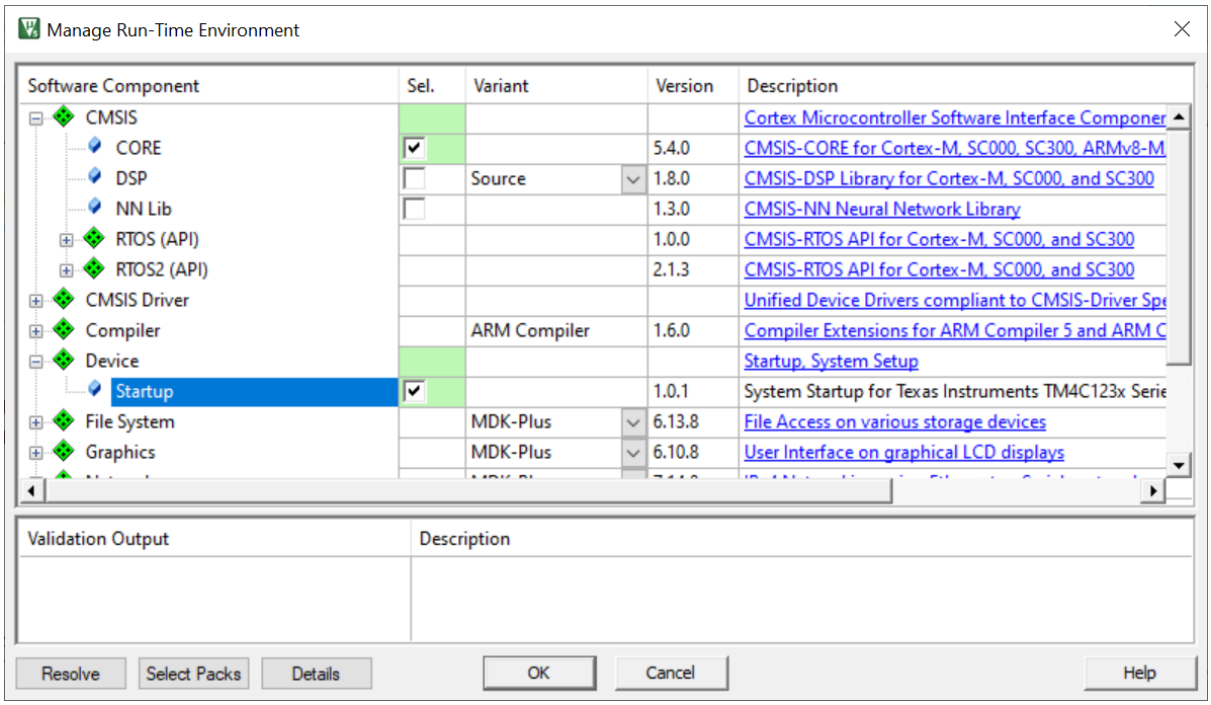
To see the output of the OutStr subroutine, or to send characters to the board using InChar, you will need a terminal window. While there are many terminal emulators out there (HyperTerminal, PuTTy), we have chosen to use Termite from CompuPhase due to its simplicity.

First find the communication port (COM Port) your board is using on your PC.

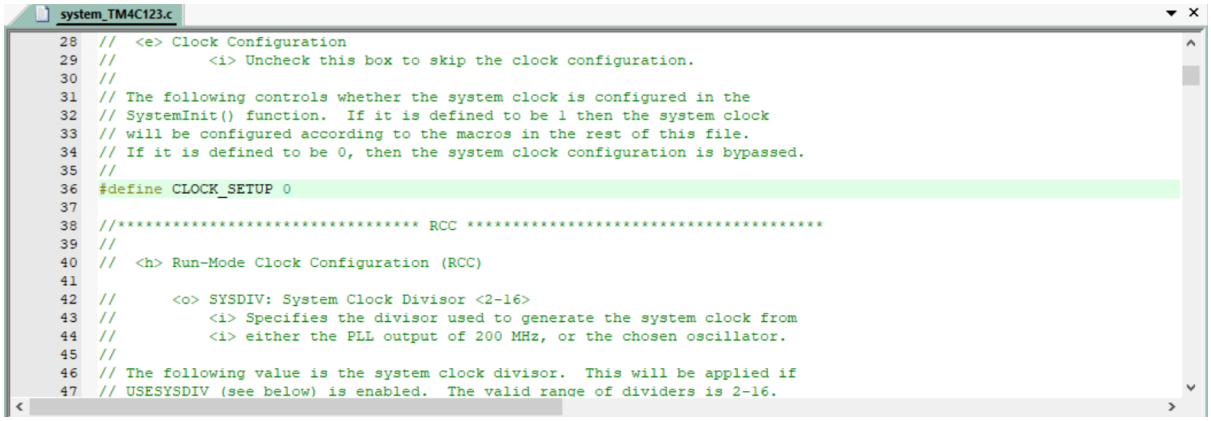
1. Connect your board to the PC using the supplied USB cable.
2. Go to Start, Right Click Computer >Manage This will bring up the Computer Management Window. Select Device Manager under System Tools.
3. Expand Ports (COM & LPT) and find Stellaris Virtual Serial Port (COM x) where x is the number of the communication port. Remember this number. Next open Termite and change the Settings to use the correct COM port.
4. Click Settings and under Port, select the number your board is using.
5. Set Transmitted text to append nothing.
6. The rest of the settings should remain default at 9600 baud, 8-bit, 1-stop, no parity.

1.12 Programming in C Language

To program in C language, you need to create a μVision project as explained in Section 1.3. However, this time on Manage Run-Time Environment screen, you will check CORE box under CMSIS and Startup box under Device.



Keil will add two files to the project, startup_TM4C123.s and system_TM4C123.c. **You wil not manually add Startup.s file given in Odtuclass.** Now, open system_TM4C123.c. Find the line that starts with #define CLOCK_SETUP and change it to #define CLOCK_SETUP 0.



Entry point to a C program is “int main()” function, and unlike the C programs running on a computer, main function should not finish execution on the microcontroller. You can use the following code template.

```
int main(){
    // place here the codes that should
    // run once

    while(1){
        // place here the codes that should
        // run continuously
    }
}
```

To be able to use utility subroutines, you still need to add the .s files distributed on Odtuclass to the project. Moreover, you need to reference the subroutines with full signature. To do this, add the following declarations above the main function.

```
/* InChar does not take an argument */
/* and returns a char (byte)        */
extern char InChar(void);

/* OutChar takes a char (byte) as an      */
/* argument and does not return any value */
extern void OutChar(char);

/* OutStr takes a pointer (address) as an  */
/* argument and does not return any value  */
extern void OutStr(char*);
```

2 Preliminary Work

The following items are expected to be exercised in order to be prepared for the demonstration of your comprehension of this laboratory work.

1. (15 %) Build, run and understand *Practice_Lab.s* Put a screenshot of Keil in your report, showing the contents of the modified memory locations in Memory Window in the debugger.
2. (15 %) Build, run and understand *Program_Directives.s*. You have to add *OutStr.s* to your project folder. Put a screenshot of Keil in your report, showing the contents of the modified memory locations in Memory Window in the debugger.
3. (25 %) Make the following modifications on *Program_Directives.s*. This time you will create and copy a different table.
   - The table is created starting at 0x2000.0680
   - The table starts with 0x00 and goes like this:

0x2000.0680	0x00
0x2000.0681	0x00
0x2000.0682	0x01
0x2000.0683	0x01
0x2000.0684	0x02
0x2000.0685	0x02
...	...
0x2000.0—	0x0A
0x2000.0—	0x0A

     Which means, the numbers are repeated twice (Until 0x0A).
   - Copy the contents of this new table and paste to the end of itself. Note that the length of this table is different from that in *Program_Directives*.

   Put a screenshot of Keil in your report, showing the contents of the modified memory locations in Memory Window in the debugger.
4. (15 %) Write the program given in 1.10. You will have to add *InChar.s*, *OutChar.s* to your project folder.
5. (15 %) Build, run and understand *Program_Directives.c*. You have to add *OutStr.s* to your project. Put a screenshot of Keil in your report, showing the contents of the modified memory locations in Memory Window in the debugger.
6. (15 %) Rewrite the program given in 1.10 in C language. You will have to add *InChar.s*, *OutChar.s* to your project.