MIE438 – Microprocessors and Embedded Microcontrollers

Laboratory Experiment 1

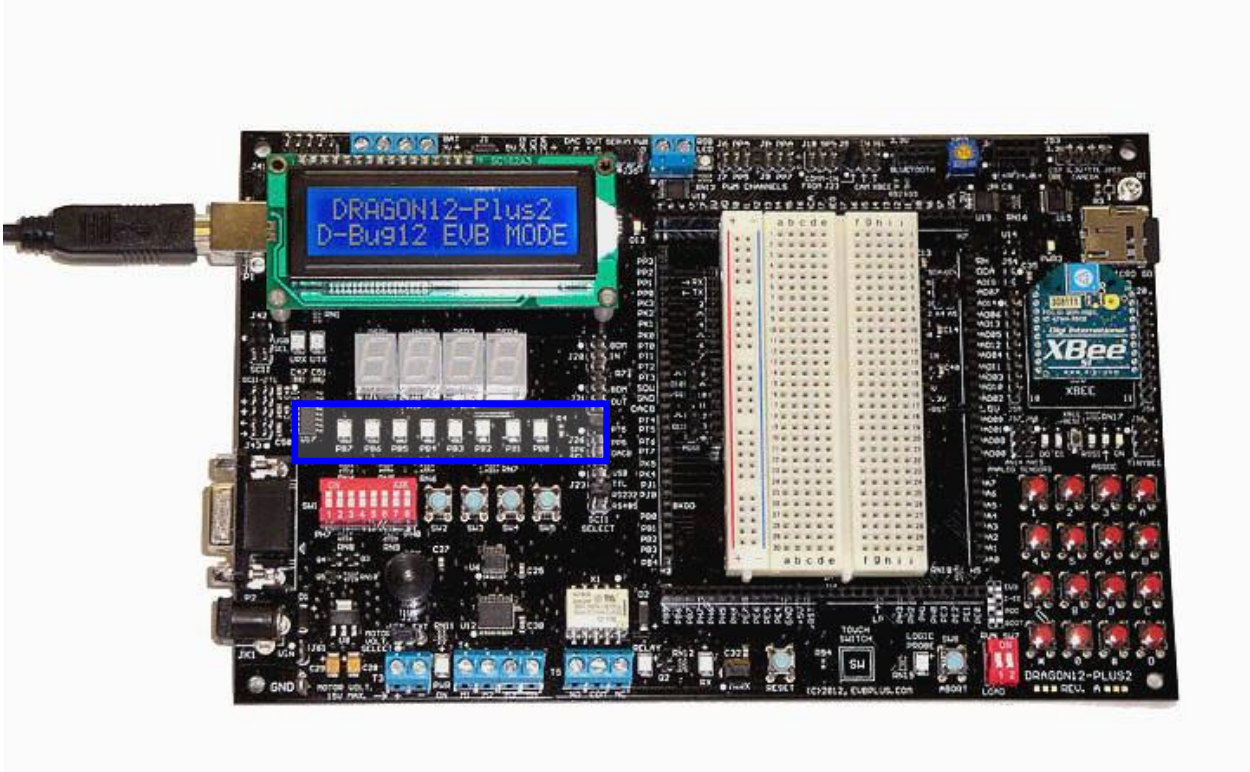
Lab Guide

Department of Mechanical and Industrial Engineering

University of Toronto

Introduction

The goal of this lab is to familiarize you with the basics of the Freescale Dragon12-Plus2 Evaluation Board. You will write, simulate, and run several simple programs using the provided evaluation board and software. You will also investigate basic arithmetic functionality.



To begin any experiment, we need to provide power to the board. The Dragon12-Plus2 (shortened to Dragon12) can be powered from either the USB connector or the power plug, however all peripherals cannot be operated from USB power. For this reason, we will use the supplied power adapter at all times.

Be certain that you plug in the power cord FIRST before connecting the USB plug to the computer. Failure to do so may result in the driver installation failing when additional components are connected.

We will introduce this new board in stages, as it is complex with many features. Included are integrated input devices (pushbuttons, DIP switches, touch sensors, matrix keypad, etc.) and output devices (speaker, motor drivers, relays, etc.), as well as advanced peripherals (XBee and Bluetooth expansion boards, MicroSD slot, LCD screen, etc.). This lab will use only the LEDs highlighted with the blue rectangle above (labelled PB0-PB7), as most of our work will be virtual to begin with.

Let's begin the lab by loading a new project and investigating the IDE functionality first.

Introduction to Using the CodeWarrior IDE

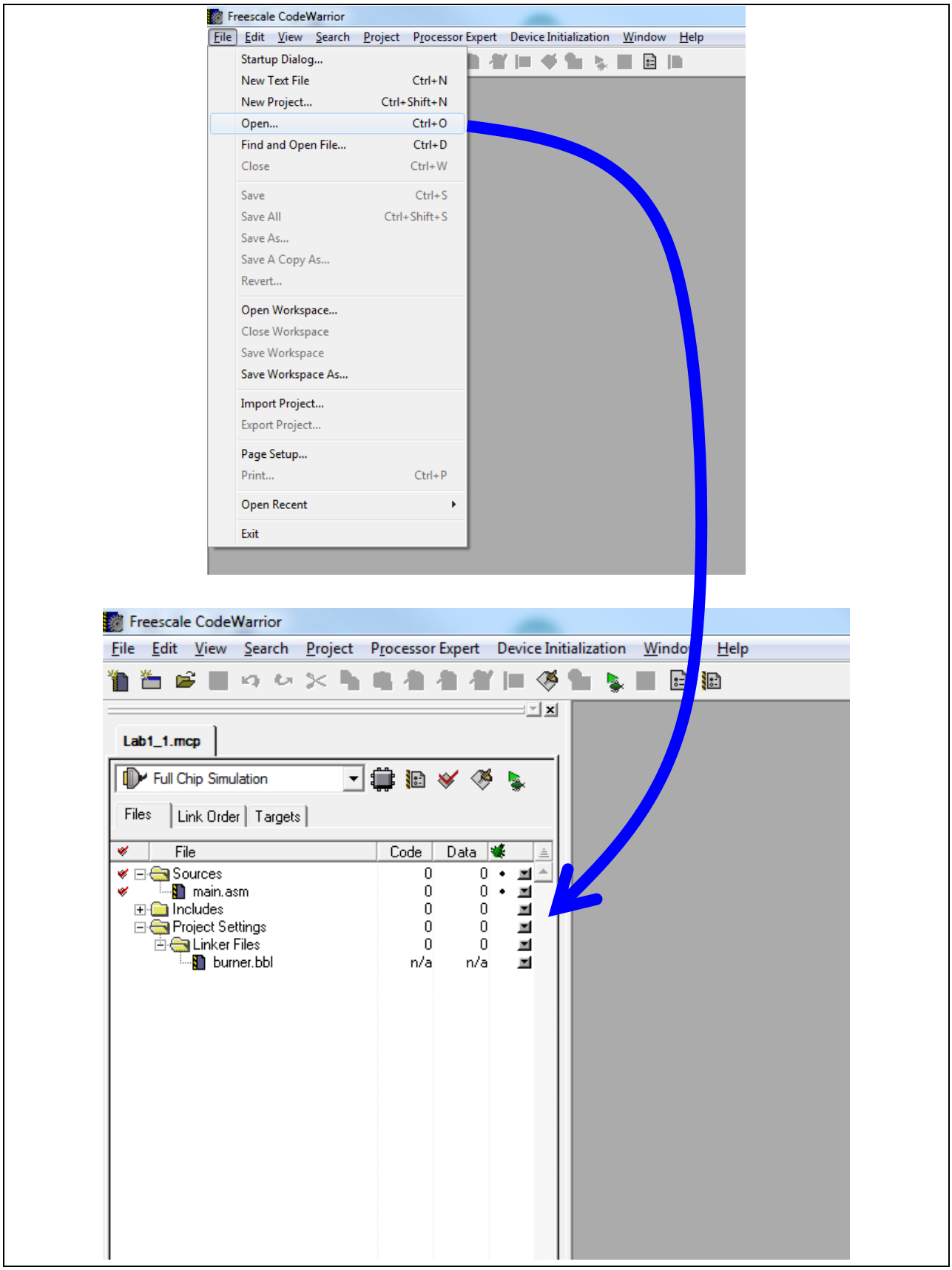
1. Start the CodeWarrior IDE by clicking on START → Programs → Software → MC402 → FreeScale CodeWarrior → CodeWarrior IDE.

2. When the CodeWarrior IDE is started, the 'Startup' window appears. For now, let's begin with a simple project – select the 'Start Using CodeWarrior' option.

3. The main CodeWarrior IDE window is now visible. You can close the 'Tip of the Day' window, you won't be needing it where we are going. Instead, go to the file menu and selected 'File' → 'Open'. Navigate to your folder containing the file 'Lab1_1.mcp'.

This will open the first project containing some simple assembly code. We are beginning our instruction at the assembly level to gain an appreciation for the complexity behind writing programs on an embedded platform. As we learned in class, due to the limited resources available in such systems, normal programming assumptions often do not apply. In particular, even when we use a high-level programming language such as C or C++, we will need to be much more aware of the hardware than we would for a general computing application.

For now, open the file 'main.asm' by double-clicking in the project list.


Freescale CodeWarrior
File Edit View Search Project Processor Expert Device Initialization Window Help
Startup Dialog...
New Text File Ctrl+N
New Project... Ctrl+Shift+N
Open... Ctrl+O
Find and Open File... Ctrl+D
Close Ctrl+W
Save Ctrl+S
Save All Ctrl+Shift+S
Save As...
Save A Copy As...
Revert...
Open Workspace...
Close Workspace
Save Workspace
Save Workspace As...
Import Project...
Export Project...
Page Setup...
Print... Ctrl+P
Open Recent
Exit
Freescale CodeWarrior
File Edit View Search Project Processor Expert Device Initialization Help
Lab1_1.mcp
Full Chip Simulation
Files Link Order Targets
File Code Data
Sources 0 0
main.asm 0 0
Includes 0 0
Project Settings 0 0
Linker Files 0 0
burner.bbl n/a n/a

Program 1 – Simulating an Assembly Language Program

4. The code window that opens should look similar to that shown below. Look over the code, paying particular attention to the regions that are highlighted and their function.

```
;*****************************************************************
;* MIE 438 - Microprocessors and Embedded Microcontrollers *
;*---------------------------------------------------------------*
;* File:     main.asm                                            *
;* Author:   Andrio Renan Gonzatti Frizon                        *
;*           andrio.frizon@gmail.com                             *
;*---------------------------------------------------------------*
;* Created on:           May 21, 2014                            *
;* Last modified on:     June 05, 2014                           *
;*****************************************************************

; export symbols
            XDEF Entry, _Startup            ; export 'Entry' symbol
            ABSENTRY Entry        ; for absolute assembly: mark this as application entry point

; Include derivative-specific definitions
            INCLUDE 'derivative.inc'

ROMStart    EQU  $4000  ; absolute address to place my code/constant data

; variable/data section

            ORG RAMStart
 ; Insert here your data definition ------------------------------------------------------------------

; code section ------------------------------------------------------------------------------------------
            ORG   ROMStart

Entry:
_Startup:
            LDS   #RAMEnd+1     ; initialize the stack pointer

            CLI                 ; enable interrupts
main:       ;-------------------------------------------------------------------------------------------
            LDAA    #16         ; load accumulator a
            LDAB    #$10        ; load accumulator b
            ABA                 ; add accumulator b to a
            ADDA    #$10        ; add to accumulator a

;*****************************************************************
;*                 Interrupt Vectors                             *
;*****************************************************************
            ORG   $FFFE
            DC.W  Entry           ; Reset Vector
```

A

B

C

D

E

Function of each code segment:

A – These symbols are automatically generated and are part of the assembler settings; they can be ignored for now.

B – This section contains an **EQU** directive. This is an assembler directive that tells the assembler to **equate** the label 'ROMStart' with the value of hexadecimal 4000. CodeWarrior uses the prefix '$' to denote hexadecimal numbers; we will use this interchangeably with the more general '0x' prefix from the lectures. The **ORG** directive that follows tells the assembler to locate any code that follows at the absolute location in memory given – in this case at whatever location 'RAMStart' represents. No code follows, as this section is being reserved as space for later variables. We do not currently have any, so we ignore it for now.

C – The previous section, B, will be used later as a storage location for variables in RAM; this section, C, (starting at memory location 0x4000) will contain our code. Again, an **ORG** directive indicates the **origin** or location of the code.

D – This section contains our actual code; we will analyze its function in detail shortly.

E – The final section contains a special initialization sequence which allows the program debugger and serial to function; we can ignore it too.

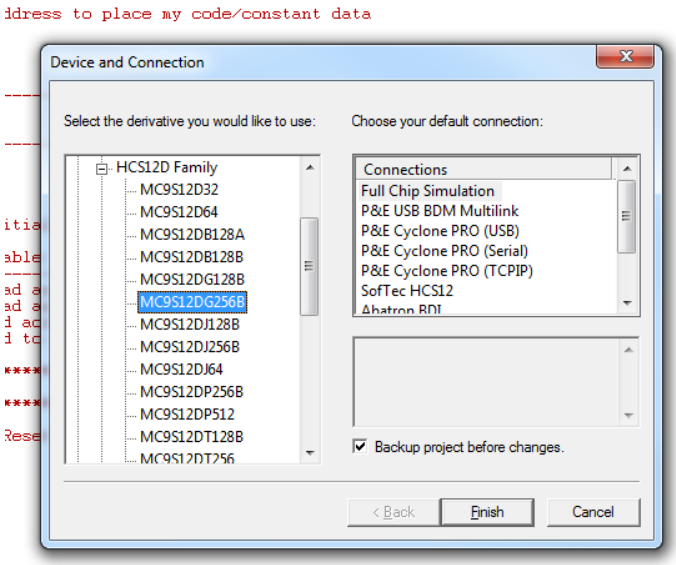
Program Function

Let's try to determine the function of the program that has been loaded. We have the following assembly code:

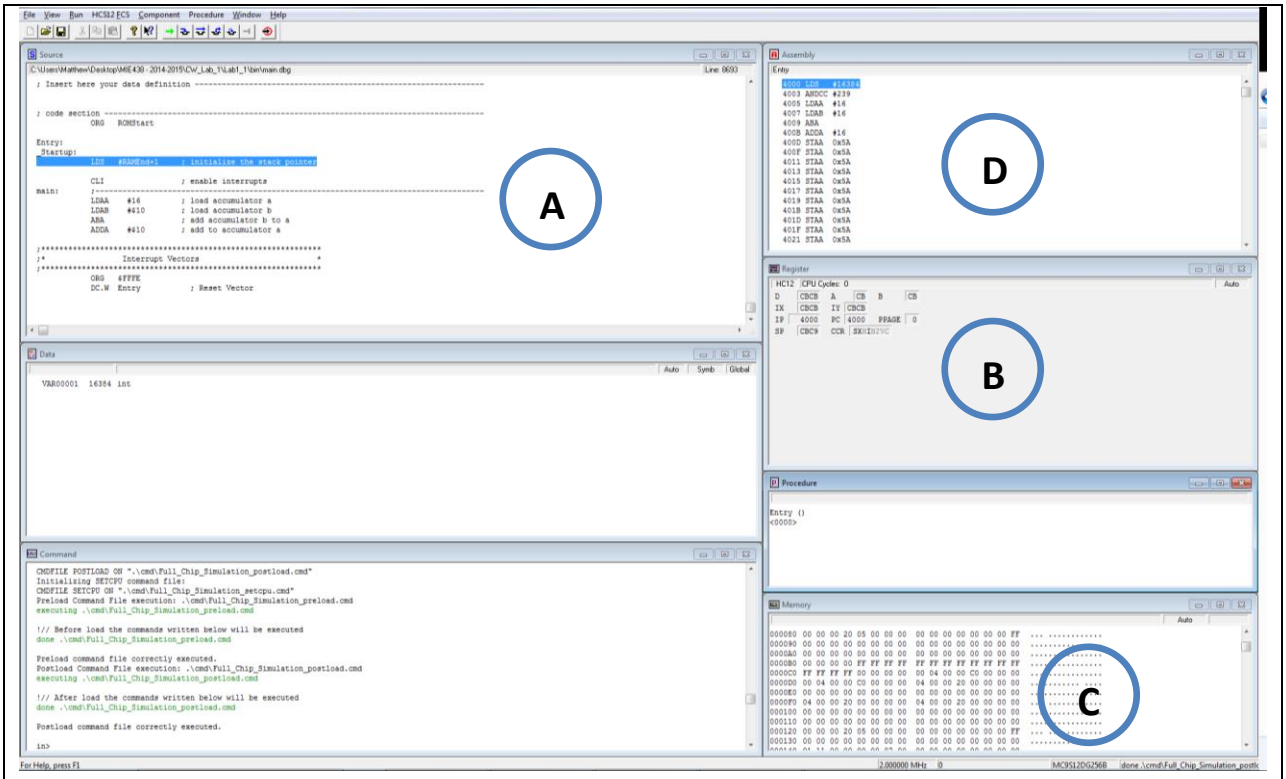
LDS #RAMEnd+1	This is a special instruction which initializes the **stack pointer**, which refers the microcontroller to a special memory in area called the stack. We will look at the stack in detail in later labs, but for now we only need to remember to include the instruction as shown.
CLI	This instruction clears the **interrupt mask**. Interrupts will be investigated in later labs as a method to deal with unexpected, sudden, or otherwise **asynchronous** input without having the microcontroller constantly wait. We are enabling interrupts to allow the built-in serial monitor to function.
LDAA #16	This instruction loads **accumulator A** with a **constant decimal** value 16 (constant denoted by the '#' and decimal denoted by the **absence** of any other prefix). The accumulator is a special **register**, the smallest and fastest form of storage on the microcontroller. Most instructions (commands) modify the contents of registers directly. Others allow us to load and store data to and from registers and the slower memory. The accumulators in particular are special-function; they are involved with most **arithmetic** and **input/output (I/O)** operations.
LDAB #$10	This function loads the second accumulator, B, with another constant value – hexadecimal 0x10, or decimal 16.
ABA	This instruction tells the microcontroller to **add** the contents of accumulators A and B, with the result replacing the contents of accumulator A.
ADDA #$10	This instruction adds a further **constant** 0x10 value to the current contents of accumulator A. Note that both this and the previous instruction are inherently **unsigned** operations; they do not interpret the results or input in any particular way such as two's complement.

5. From the description on the previous page, answer **Question 1** of the deliverable.

6. Let's simulate the program. The CodeWarrior IDE is capable of fully simulating the function of the chip without actually using the board. This allows you to write and debug programs at home. This project is already set to target the board simulator, but you can always modify the target of existing projects by selecting 'Project' → 'Change MCU/Connection…'. The derivative (left list) should always be set to MC9S12DG256B, which is the actual microcontroller on the boards we use. The default connection (right) can be selected as either 'Full Chip Simulation' which uses the simulator, or 'HCS12 Serial Monitor' which will have CodeWarrior load the program onto the physical trainer board. If you have opened the window, be sure Full Chip Simulation is still selected and click 'Finish.'



7. We can run the program by selecting 'Project' → 'Debug' or by pressing **F5**. Please note that in the case of a full-chip simulation as we are doing here, the debugging window should load directly. If we are loading the program onto the board an additional window may appear now requiring us to select the connection – we will cover this process later. If the window appears now, your project target is not set to simulation.



The debugging window contains important views of all the internal information for our evaluation board. Note that the current window is purely simulated, but it will appear the same for the real board. Note the following information sections:

A – This is the **code window**; it shows the program code (assembly or high-level) from the previous window along with a highlighted bar indicating the current execution position. We can fully control execution using the following controls:

Run → Start/Continue (F5): Starts the program running at maximum speed; it will continue until the program terminates on its own or until **Run → Halt (F6)** is pressed.
Run → Single Step (F11): This steps the program through one instruction at a time, updating the result in each window. Very useful for debugging a program; we will use it shortly.
Run → Step Over/Out (F10/Shift+F11): These are more advanced stepping controls that will be used when debugging high-level programs where one line may implement a subroutine or multiple instructions.

B – The **register window** shows you the current value of the registers, including special registers such as accumulator A and B. Notice that they values are quasi-random on start-up, meaning these registers should be initialized in programs before using them.

C – The **memory window** shows the current contents of memory. In later labs, where we use the memory for long-term storage of variables, this window will allow you to track their contents. For now we can ignore it.

D – The **assembly window** shows the contents of memory interpreted as instructions; useful for seeing the low-level instructions actually executed by the processor when debugging high-level code.

8. Use the **single step** command (F11) **twice**; the code highlight line should highlight the first instruction (LDAA) at this point. Note the starting value of accumulator A (ACCA). Press single step once more and observe the change in A. In particular, notice that in the register window numbers are shown in hex.

9. Step through the program until you reach the end. Watch the **assembly window** (D) as you continue; notice that you can keep stepping past the end of the program. In this case, the semi-random garbage in memory after your program will be executed as valid instructions – in some cases, this may overwrite the final value in accumulator A with useless numbers. Later programs will need a command to terminate execution. If you step too far, you can use 'HCS12 FCS' → Reset or Control+R to reset the execution to the start.

10. Step until after the final instruction has executed. Observe the values of accumulators A and B as the program runs, and answer **Question 2** in the deliverable.

11. Observe the program in the assembly window (D). Notice the numbers to the left of each instruction; these are the locations in code memory of each instruction. For example, the LDS instruction is located at 0x4000 (as would be correct, given the ORG ROMStart directive from earlier). Observe the locations of the other program instruction and answer **Question 3** in the deliverable.

Program 2 – Running a Real Assembly Language Program

12. It is time to run a program using the actual hardware. Close the debugger window and return to the main CodeWarrior IDE window. Close the Lab1_1.mcp project by clicking the small 'x' to the right of the name tab. Close any code windows that are open as well. Use File → Open to open the Lab1_2.mcp project. Open the new main.asm for Program 2.

13. Notice that this new program is much more complex than the previous one. Let's step through the new parts:

```
;*****************************************************
;* MIE 438 - Microprocessors and Embedded Microcontrollers *
;*-----------------------------------------------------*
;* File:     main.asm                                  *
;* Author:   Andrio Renan Gonzatti Frizon              *
;*           andrio.frizon@gmail.com                   *
;*-----------------------------------------------------*
;* Created on:          May 21, 2014                   *
;* Last modified on:    May 21, 2014                   *
;*****************************************************

; export symbols
            XDEF Entry, _Startup            ; export 'Entry' symbol
            ABSENTRY Entry        ; for absolute assembly: mark this as application entry point


; Include derivative-specific definitions
   INCLUDE 'derivative.inc'

ROMStart    EQU  $4000  ; absolute address to place my code/constant data

; variable/data section

 ifdef _HCS12_SERIALMON
            ORG $3FFF - (RAMEnd - RAMStart)
 else
            ORG RAMStart
 endif
; Insert here your data definition ------------------------------------------------------------------


; code section ------------------------------------------------------------------------------------
            ORG   ROMStart                                                  |


Entry:
_Startup:
            ; remap the RAM &amp;amp; EEPROM here. See EB386.pdf
 ifdef _HCS12_SERIALMON
            ; set registers at $0000
            CLR   $11                   ; INITRG= $0
            ; set ram to end at $3FFF
            LDAB  #$39
            STAB  $10                   ; INITRM= $39

            ; set eeprom to end at $0FFF
            LDAA  #$9
            STAA  $12                   ; INITEE= $9

            LDS   #$3FFF+1          ; See EB386.pdf, initialize the stack pointer
 else
            LDS   #RAMEnd+1         ; initialize the stack pointer
 endif

            CLI                     ; enable interrupts
main:       ;-------------------------------------------------------------------------------------------
            LDAA    #$FF
            STAA    DDRJ            ; make port J an output port (necessary to enable LED, dragon12-plus2_hcs12_manual.pdf page 10)
            STAA    DDRB            ; make port B an output port (LEDs, dragon12-plus2_hcs12_manual.pdf page 10)
            STAA    DDRP            ; make port P an output port (necessary to disable 7-segment displays, dragon12-plus2_hcs12_manual.pdf page 11)
            LDAA    #$0F
            STAA    PTP             ; turn off 7-segment LED display and RGB
            CLR     PTJ             ; make PJ1 low to enable LEDs
            CLR     PORTB           ; turn off PB0

back:
            LDAA    #155
            LDAB    #$21
            SBA                     ; subtract B from A
            STAA    PORTB           ; turn on PB0
            JMP     back
;*****************************************************************
;*                 Interrupt Vectors                             *
;*****************************************************************
            ORG     $FFFE
            DC.W    Entry           ; Reset Vector
```

A – The first potion of the code begins with 'Entry:' and '_Startup:'. These are examples of **labels**; they are used to refer to locations in code memory. Unlike the assembler directive ORG, we can refer to and access labels in our code to allow **addressing** of various locations. We will see one use of labels shortly with the **jump** instruction. Otherwise, for this program our general initialization is minimal – we are mostly telling the onboard debugger where to locate our code.

B – The 'main' section of the code begins with our program-specific initialization. Most of the code here is the start-up code necessary for the real hardware platform. Our goal is to control the LEDs on PB0 to PB7; these LEDs are wired to physical pins on the 68HCS12 microcontroller. However, other peripherals and devices may share these lines; other features of the 68HCS12 may also use these I/O pins on the microcontroller itself. The code here is used to turn off all features that may interfere with our use of **Port B**, which is the output port (pins) that the LEDs PB0 to PB7 are connected to. We also configure Port B as an output port – most ports are multi-function, and can be inputs or outputs.

C – The 'back' section is where the actual program of interest is located. Notice that is contains similar basic math to what we saw in Program 1. The addition is replaced with subtraction (unsigned, SBA). There is also one additional instruction, STAA (**store** accumulator A) which writes the value in accumulator A somewhere. In particular, it sends it to Port B, which is connected to the LEDs mentioned earlier. Lastly, the section ends with a JMP (**jump**) instruction. This tells the microcontroller to go back to the code at the label 'back,' creating an infinite loop. This prevents the program from executing junk data past the end of our program.

14. Run the program by pressing F5 or selecting Project → Debug. The debug window will appear as before. If all settings are correct, the program will be automatically downloaded onto the board. Execution does not begin until you tell the program to run. However, one of two issues may also occur:

a. If a message that says 'Loading a new application will stop execution of the current one.' appears, simply click OK. This ends any program currently running on the board.

b. A message that the board could not be found ('Could not connect to hardware' followed by 'Fatal Error') may also appear. Double-check the USB and power connection first of all. Most likely, however, this is due to an incorrect communication setting. In particular, you must select the correct virtual COM port from the drop-down list 'HOST Serial Communication Port': {COM1, COM2, etc.}. The TAs will let you know the correct setting. Once you select it, close the debug window and reopen it from CodeWarrior to ensure the program downloads properly.

When ready, your debugging window should look like it did before with the simulation.

15. Use Run → Start (F5) to run the program on the platform. Notice that the LEDs on Port B should change. Answer **Question 4** and **Question 5** in the deliverable.

16. Reset the platform using Reset or Control+R. Use the step commands to walk through the program, observing the 'jump' back to earlier code at the end of the program. Notice that the result shown on Port B is actually being constantly updated by the infinite loop, although the number shown with each iteration is the same.

Program 3 – Writing a Real Assembly Language Program

17. Close the debugger, project, and any remaining code windows. Open the project Lab1_3.mcp and open the main.asm file. Examine the arithmetic code located at the 'back' label.

18. When ready, use the debugger to run the new code on the hardware platform. Observe the new output on the Port B LEDs. Answer **Question 6** and **Question 7** in the deliverable.

19. Close the debug window and return to the CodeWarrior IDE. Modify the program in main.asm (the section starting at 'back:') to implement the subtraction using a **manual implementation of two's complement** of the value loaded by the LDAB instruction. In other words, you should still load #89 into B, but then take the two's complement of the number and add the result. Ensure the answer shown on the LEDs is correct when interpreted as a two's complement number. Record your modified program for **Question 8**.

Hint: You will need new instructions to accomplish the above; check the provided instruction set. In particular, the NEG commands (NEGA/NEGB) and logic commands (ORA, ANDA, etc.) may be useful. Ask the TAs for help if you get stuck.

20. Once you are done the above, try loading and running Lab1_4.mcp (labelled 'just for fun'). It will produce an interesting effect.

***** End of Laboratory 1 *****

MIE438 – Microprocessors and Embedded Microcontrollers

Laboratory Experiment 1

Deliverable

Department of Mechanical and Industrial Engineering

University of Toronto

Last Name, First Name	Student Number

1. From the code description, predict the result that will be stored in accumulator A at the end of the program.

2. What was the final value in accumulator A? Does this match the result from Q1?

3. How many bytes of code memory does the LDS instruction use? What about the others (LDAA, LDAB, ABA, ADDA)? Speculate on why they are different.

4. What is the output of the LEDs for Port B? Interpret the result as an unsigned binary number.

5. Is the result shown on the LEDs correct when interpreted as unsigned? Why or why not?

6. What is the output of the LEDs for Port B?

7. Is the result shown on the LEDs correct under **any** number representation (i.e., sign and magnitude, one's complement, two's complement)? Speculate why or why not?

8. Record the program changes (starting with the 'back:' label) used to manually implement a two's complement subtraction.

This lab hand-in is due at the start of the subsequent lab section.