EE 308 – LAB 2

ASSEMBLY LANGUAGE PROGRAMMING AND 9S12 PORTS

In this sequence of three labs, you will learn how to write simple assembly language programs for the MC9S12 microcontroller, and how to use general purpose I/O (input/output) ports.

Week 1

Introduction and Objectives

This laboratory will give you more experience with the tools we will use this semester — the Dragon12 Plus evaluation board (EVB), the DBug12 monitor, the as12 assembler and the AsmIDE. Be sure to read through the entire lab and do the prelab for each section before coming to lab

1. Consider the program in Figure 1:

```
prog:   equ     $2000     ; put program at address $2000
data:   equ     $1000     ; put data at address $1000

        org     prog
        ldab    #29       ; Immediate (IMM) addressing mode
        ldaa    #235
        sba               ; Inherent (INH) adressing mode
        std     result    ; Extended (EXT) addressing mode
        swi

        org     data
result: ds.w    1         ; Reserve one word (two bytes) for result
```

Figure 1. Demo program for Part 1 of Lab 2.

PreLab

- Hand assemble this program, i.e., determine the opcodes the MC9S12 will use to execute this program.
- How many cycles will this take on the MC9S12? (Do not consider the `swi` instruction.)
- How long in time will this take? (Note: the MC9S12 executes 24 million cycles per second.)
- What will be the state of the N, Z, V and C bits after each instruction has been executed? (Ignore the `swi` instruction.)
- What will be in address 0x1000 and 0x1001 after the program executed?

a) Assemble the program using **as12**. Look at the **lst** and **s19** files. You should be able to relate the *opcodes* from the prelab to the data in the s19 file. Verify that they agree.

b) Load the program onto your Dragon12 Plus board. Trace through the program. Verify that the Z, N, V and C bits are what you expect after each instruction.

c) Look at the contents of addresses 0x2000 and 0x2002. Do the values agree with your answers from the prelab?

d) You could change this program to add rather than subtract by changing the **`sba`** instruction to an **`aba`** instruction. Modify the program, assemble it and load the new program into the MC9S12.

a. Find the address for the `sba` instruction. (You can find this from the .lst file or by using the asm command in DBug-12.)

b. In the **MC9S12 Core Users Guide**, find the op code for the `aba` instruction.

c. Go to the address of the `sba` instruction, and change the op code to that of the `aba` instruction.

d. Run the program again, and verify that the program now adds rather than subtracts.

2. Consider the program in Figure 2, which is moves data from one table into another.

```
; MC9S12 program to copy a table of data from one location to another
; January 27, 2009

prog:   equ     $2000       ; put program at address $2000
data:   equ     $1000       ; put data at address $1000
count:  equ     16          ; 16 elements in the table

        org     prog
        ldab    #count      ; ACCB keeps count of number to transfer
        ldx     #table_1    ; X points at table_1
        ldy     #table_2    ; Y points at table_1
repeat: ldaa    1,X+        ; get data from table_1, X points to next element
        staa    1,Y+        ; save into table_2, Y points to next element
        decb                ; Decrement counter
        bne     repeat      ; If not done, continue with next element
        swi                 ; Done -- return to DBug-12

        org     data
; Initialize data in table
table_1: dc.b   $44,$61,$74,$61,$20,$54,$61,$62
         dc.b   $6c,$65,$00,$c2,$a3,$5a,$31,$ff
table_2: ds.b   count           ; Reserve count bytes of memory for result
```

Figure 2. Demo program for part 2 of lab 2

a) Use the text editor to enter this program, and assemble it into an s19 file.

b) How many cycles will it take to execute the program take on the MC9S12?

c) How long will it take to execute the program?

d) Load the program into your MC9S12. Use MD to verify that the data is in the table at address 0x1000. Run the program, and verify that the table has been copied into `table_2`.

d) Use the Block Fill option of DBug-12 to change the values in addresses 0x1000 through 0x1FFF to 0xff. Reload the s19 file.

e) Set a breakpoint at the label `repeat`. (Look at the .lst file to find the address of the label.)

f) Execute the program again. The program should stop the first time it reaches the `repeat` label, with `0x10` in ACCB, and 0x1000 in X.

g) Continue running the program. It should stop each time it gets to the repeat label, B should be decremented by one, X should be incremented by one, and there should be a new entry in table2. Use the RD and MD commands of DBug-12 to verify this.

3. Consider the code fragment of Figure 3. Do parts (a) and (b) below before coming to lab

```
        ldy     #1000
loop1:  ldx     #25000
loop2:  dbne    x,loop2
        dbne    y,loop1
        swi
```

Figure 3. Demo program for part 3 of lab 2.

Question to answer before lab:

a) How many cycles will this program take on the MC9S12? (Again, ignore the swi instruction.)

b) How long will it take to execute this program?

c) Use a text editor to enter the code into a program – you will have to add org statements and other assembler directives to make the program work.

d) Assemble the program and run it on the HC12. How long does it take to run?
This time should match your answer to part (b)

WEEK 2

Introduction and Objectives

The purpose of this laboratory is to write a few assembly language programs and test them on your MC9S12. You will learn how to create delays with software loops, and how to do simple I/O.

PreLab

Make sure you have the programs written and clearly thought out before you come to the lab. You should put all your code starting at memory location 0x2000. You are encouraged to bring the programs in on a flash drive.

The Lab

The Dragon12 Plus has several ways to display data. It has an LCD display, four seven-segment LED displays, and eight individual LEDs. Programming the LCD display is rather complicated, and will not be discussed at this time. The easiest display to start with is the individual LEDs. You will write some programs to display patterns on the LEDs.

You can display patterns on the LEDs by writing to the I/O port at address 0x0001 (called **PORTB**). Because of the way the LEDs are connected, it is necessary to do some other setup of I/O ports as well. The first five instructions in the program below set up the MC9S12 hardware so that you can write patterns to the LEDs. (For now, we will give you the code you need to set up the I/O system properly. In later labs, you will learn how to do the setup yourself.) The following program will flash the LEDs:

```
PORTB   equ     $____       ; Port B data register
DDRB    equ     $____       ; Port B direction register
PTP     equ     $____       ; Port P data register
DDRP    equ     $____       ; Port P direction register
PTJ     equ     $____       ; Port J data register
DDRJ    equ     $____       ; Port J direction register

        bset    DDRP,#$0F   ; Make PP0-PP3 outputs
        bset    PTP,#$0F    ; Turn off seven-seg LEDs
        bset    DDRJ,#$02   ; Make PJ1 output
        bclr    PTJ,#$02    ; Turn on individual LEDs
        bset    DDRB,#$FF   ; Activate control lines for LEDs

        movb    #$55,PORTB  ; Turn on every other LED
loop:   com     PORTB       ; Toggle LEDs
        bra     loop        ; Repeat
```

Figure 1. Demo program for week 2 of Lab 2.

1) Complete the above program by adding the necessary addresses and: assembler directives. Assemble the program.

a) Test your program on the MC9S12. Trace through the loop to see what is happening.
Note: The program should cause the LEDs to flash on and off.

b) Run your program by typing "g 2000". When you do this, the LEDs flash so quickly that it looks like they are all lit. Add some code to create a 100 ms delay between the last two instructions of the program. When you do this, you will be able to see the lights flash.

2) Write a program which will start with all the LEDs off, then increment the LEDs, implementing a binary counter. Run your program to verify that it works. Make sure you use a delay so you can see the LEDs incrementing.

3) Write a program to count the number of odd bytes in memory from address 0xff00 to 0xffff. Display this number on the LEDs. (Note: You need to set up the hardware to display on the LEDs as in the program of Figure 1.)

4) Write a program which puts the exclusive OR of the eight-bit numbers from memory locations 0x8000 through 0x8FFF and display the result on the LEDs. (This operation is often used to generate a check sum to verify data transmission. The sending computer generates and transmits the check sum along with the data. The receiving computer calculates the check sum for the received data and compares it with the check sum sent by the sending computer. If the two values do not match, then there was an error in the transmission.)

WEEK 3

Introduction and Objectives

In this week's lab you will write an assembly language program to display various patterns on the eight individual LEDs on your Dragon12-Plus board. The displayed pattern will be based on the state of two bits of the onboard DIP switch. You will also start using subroutines, and investigate the stack and stack pointer, and learn how to load your program into EEPROM so the program will remain on your board after a power cycle.

PreLab

The program for this lab will display four different patterns on the LED display connected to Port B. You will use the state of bits 1 and 0 of the onboard DIP switch to select which of the four patterns to display.

Write a program to set up Port B as an eight bit output port (be sure to disable the seven-segment displays, and to enable the individual LEDS), and to implement (i) a binary up counter, (ii) a Gray code counter, (iii) a Johnson counter, and (iv) a Ford Thunderbird style turn signal based on the state of the DIP switches. Insert a 100 ms delay between updates of the display. Write the delay as a subroutine. Be sure to initialize the stack pointer in you program.

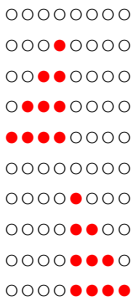
(Note: you will be referring to a number of MC9S12 registers in this and future programs. It is tedious and error-prone to look up and enter the addresses of the registers each time you write a new program. There is a file on the EE 308 web page called "hcs12.inc" which has a list of all registers and their addresses for the 9S12DP256 version of the MC9S12 microcontroller. If you include that file in your program (by including the line `#include hcs12.inc` as the first line of your program), you can refer to all registers by name rather than having to look up their addresses.)

(i) For the binary up counter, have the LEDs count 0, 1, 2, 3, 4, …. It should take 256 counts from the time all LEDs are off until the next time all are off.

(ii) An eight-bit Gray code counter generates a count where only one bit changes at a time. A simple method to generate a Gray code count is to take the exclusive OR of the binary count with the binary count shifted right by one. (E.g., the number 79_{10} is 01001111_2. To find the Gray code for this, take the XOR of 01001111_2 with 00100111_2, which gives 01101000_2.) When you shift the binary count on the MC9S12, use a logical shift rather than an arithmetic shift, because the MSB of the Gray code counter should be the same as the MSB of the binary counter.

(iii) A Johnson counter works as follows: To generate the next count from the present count, shift the present count right by one bit. The most significant bit of the next count will be the inverse of the least significant bit of the present count. The starting count for a Johnson counter is 00000000_2

(iv) The TBird style turn signal looks like this:



The easiest way to implement the TBird style turn signal is to put the above pattern into a table in memory, and cycle through the table.

Set up Port H as an input port, and use bits 1 and 0 to control which of the LED patterns are displayed as shown in Fig. 1. When you switch between functions, the new function should start up where it ended when it was last activated, so set aside variables to save the states of the various patterns. In your program, be sure to mask out the other bits of Port H so that only PH1 and PH0 are used to determine the pattern to display.

PH1	PH0		Display
0	0		Binary Up Counter
0	1		Gray Code Counter
1	0		Johnson Counter
1	1		TBird Turn Signal

Figure 1. Port B inputs to control the Port A functions.

The Lab

Assemble your program, and run it on the MC9S12. If you have difficulty getting your program to work, start by trying to implement one function only – say, the binary counter. Once this works, start working on your next functions.

Verify that all the functions work correctly. In particular, make sure that the Gray code counter counts by changing one bit only. You may have to slow the counter down, or put in a breakpoint, to verify this.

Set a break point at the first line of your delay subroutine after you save the registers used by the subroutine on the stack. When the breakpoint is reached, check the value of the stack pointer, and the data on the stack. Make sure you understand what these mean. (What value is in the SP register? What data is on the stack? What do the data on the stack represent? Make sure you document this in your lab book.)

When you get your program to work, have your lab instructor or TA verify the program operation.

The MC9S12 has EEPROM (Electrically Erasable Programmable Read Only Memory) functionality. If you put your program into EEPROM the program will remain there when you turn off power.

The **EEPROM** is located at address **0x400**. You can just change the origin statement of your assembly language program, then reassemble, and reload your program. (Loading programs into EEPROM takes a longer time than loading programs into RAM. DBug-12 needs to tell the ASMIDE to wait while it programs some EEPROM bytes before ASMIDE sends the next set of bytes to program. It uses a protocol called Xon/Xoff to do this. Make sure ASMIDE is set up to use Xon/Xoff. To do this, go to the View/Options drop-down menu, click on "Set COM Parameters", and make sure "Flow Control" is set to Xon/Xoff.) Use the MD command to verify that your program was correctly loaded into EEPROM. Type "**G 400**" to run your program out of EEPROM. It should work the same as it did when you ran it out of **RAM**. (Try it.) You can power cycle your board, and then type "**G 400**", and again your program will run correctly. (Try it. The TBird pattern may not work correctly. The reason for this and the solution is discussed below.)

For some applications it would be nice if you could run your program without having to type "**G 400**" – if your board is controlling a robot, and no computer is connected to it, it would be impossible to start the program by typing "**G 400**". DBug12 has a special mode to allow you to run a program out of EEPROM without having to type "**G 400**". If you set the two switches on the LOAD DIP switch to "Jump to EEPROM" mode (Switch 2 on, Switch 1 off), and power cycle the board (or push the reset button), the program will run immediately out of EEPROM. (Try it.) You will notice that the program runs much slower – actually, six times slower than it did when you ran it by typing "**G 400**". This is because DBug12 does some system initialization which is bypassed when you run your program directly from EEPROM. In particular, the Dragon12-Plus board has an 8 MHz clock, and the MC9S12 runs at half the clock frequency, or 4 MHz. The MC9S12 has a built in phases lock loop (PLL) which allows the chip to generate a faster clock internally, and run with a 24 MHz E-clock frequency. In order to get the chip to run at the higher frequency, you must do the initialization which enables the PLL. Here is some code which will do that initialization (adapted from the Dragon12-Plus Reference Manual):

```
; PLL code for 24MHz bus speed from an 8 crystal
             sei                          ; disable interrupts
             bclr    CLKSEL,%10000000     ; clear bit 7, clock derived from oscclk
             bset    PLLCTL,%01000000     ; Turn PLL on, bit 6 = 1 PLL on, bit 6 = 0 PLL off
             movb    #$05,SYNR            ; 5+1=6 multiplier
             movb    #$01,REFDV           ; divisor=1+1=2, 8*2*6 /2 = 48MHz PLL freq,
                                          ; for 8 MHz crystal
wait_b3:     brclr   CRGFLG,%00001000,wait_b3 ; Wait until bit 3 = 1
             bset    CLKSEL,%10000000     ; derive clock from PLL
```

Add the above code to your program, right after the "org $400" line and before the first line of your program. Load this new code into **EEPROM**. (Be sure to **move SW1 of the LOAD DIP switch down** in order to get back to the DBug12 monitor so you can load new code into memory.) Now **move SW1 of the LOAD DIP SWITCH to the up position**, power cycle your board, and your program should run at the same speed it did when running out of RAM.

Another problem with running out of EEPROM is that data which is loaded when you load your program is not present when you start your program out of EEPROM after a power cycle. For example, if the TBird pattern is put into RAM, when you turn power off that pattern is lost, and when you run turn power back on and start running the program from EEPROM, an incorrect pattern is displayed. To fix this, put the table into the program section of memory rather than the data section. In this way, the table is programmed into EEPROM as well your program. Now if you power cycle the board, the table with the TBird pattern is still

there. When you put a program into EEPROM, only variables which change should be put into the data section. Also, you need to initialize these variables in the program rather than using a "**`dc.b`**" directive.

Note: The document "**readme_EEPROM.pdf**" which came on the DRAGON12-Plus CD says that you need to convert your S1 code (in the S19 file) to S2 code to successfully load a program into EEPROM. This is because the **MC9S12 EEPROM** must be programmed with an even number of bytes, and must be programmed starting at an even address. However, I have had no problem loading a program which starts on an odd address or has an odd number of bytes. I think that, when DBug12 sees that a user wants to load a program which starts on an odd address or contains an odd number of bytes, into EEPROM, it automatically adds the bytes needed to make the program start on an even address or to contain an even number of bytes. If you have trouble getting an EEPROM program to load correctly, you should try converting your S1 code to S2 code as discussed in the "**readme_EEPROM.pdf**" document.