MIE438 – Microprocessors and Embedded Microcontrollers

Laboratory Experiment 2

Lab Guide

Spring 2016

Department of Mechanical and Industrial Engineering

University of Toronto

Introduction

The goal of this lab is to introduce basic assembly language programming topics to you and allow you to begin writing some more complex and useful programs. For this lab, we will be comparing the assembly language programs we write to equivalent high-level C code. In later labs, we will move away from assembly language in order to write longer and more functional programs quickly. However, when working with embedded systems you may occasionally have to write small sections of embedded assembly code. Assembly code is often used in control loops and other similar structures where fast, predictable, and/or controlled execution time is valued above other metrics. Also, knowledge of program structure at the low-level will help you to write *better* embedded high-level code, as we will see in later lectures.

Registers Introduction

In the last lab we briefly introduced the concept of a **register**. All microcontrollers contain a set of registers which are used for storing data; they are the **smallest and fastest** elements of storage. Typically, registers are small in capacity, but have a high spatial cost of integration, and are thus a scarce resource. Microcontrollers and microprocessors also typically have a specified **word length** which is common to most or all of the instructions available in the **instruction set architecture** for that chip. As such, most registers will typically be of a size equal to the word length (i.e., the MCS-51/8051 is part of an 8-bit microcontroller family, and primarily contains 8-bit registers). Note that some microcontrollers, such as the 68HC12, are **mixed word-length** – they contain instructions that span more than one word length, such as 8-bit **and** 16-bit. In these microcontrollers, it is common for the instruction set to contain a few commands which combine multiple registers automatically into larger bit widths, allowing the microcontroller to work more easily with larger numbers.

General Purpose Registers

As their name suggests, most registers in a microcontroller are **general purpose (GP)** – they have no special function or limitations. For the 68HC12, the base specification provides only two general purpose registers – X and Y (and even these have some limitations). This was a deliberate design choice with the 68HC12, as the use of the **accumulators** (from Lab 1) is heavily favoured. Other microcontrollers do not have any accumulators, and perform arithmetic and other math directly on general-purpose registers – such microcontrollers typically offer more GP registers to the programmer. In general, one can consider each register to be a potential 'variable' in code; here, X and Y can each store a single 16-bit number. We have instructions that can work directly with registers; for example:

LDX	#5101	–	$X = [5101]_{10} = [13ED]_{16} = [0001\ 0011\ 1110\ 1101]_2$
ABX		–	Accumulator $B = [B] + [X]$ **
EXG	A, X	–	Switch values in Accumulator A and X **

Note that the final two commands have **mixed** operands; A and B are **8-bit** accumulators, and X is 16-bits wide. To complete these instructions, A and B are automatically extended by placing zeros in the missing 8 most-significant bit (MSB) positions. You will need to be mindful of such automatic functionality – the 68HC12 uses the **complex instruction-set computing** (**CISC**) design principle, and thus each instruction may implement complex, multi-effect functionality.

Accumulators and Register D

Two special registers are present in the 68HC12 (and many similar microcontrollers) called accumulators. For the 68HC12 they are referred to as Accumulator A and B. These registers are used by almost all math, logic, and I/O instructions. In general, the result for any such operation will be stored in an accumulator. These two accumulators, unlike other GP registers, are 8-bits in size. We also have available a third register, called D, although it is not a strictly separate entity. Instead, register D combines the contents of A and B as a 16-bit number (typically, the contents of A are treated as the most-significant bits, MSBs, of D). This allows us to easily work with both 8-bit and 16-bit numbers. The following are some examples of commands using the accumulators and D:

LDAB #12	–	$B = [12]_{10}$
TBA	–	$A = B$
MUL	–	$D = A \times B$ (8-bit unsigned multiply)
ABA	–	$A = A + B$
ADDD #1000	–	$D = D + [1000]_{10}$

Data Pointers and Index Pointers

Some microcontrollers contain a special register called a **data pointer** (DPTR). As the name suggests, this register is used as a data pointer, holding an address which points to data in memory. In other architectures, these are used strictly to allow access to external/extended memory by providing a means to **address** the memory (more on this later). The 68HC12 implements index pointers – they have the same functionality as above, but are shared. For the 68HC12, the general-purpose registers X and Y fill this function. In contrast to a strict data pointer found on another microcontroller, there are many instructions on the 68HC12 which can work with or modify X and Y directly, saving time and code length. In other architectures, one must often laboriously transfer the data pointer contents into a general purpose register first before modifying the value, and only then putting the result back in the data pointer. The benefit for this second method is that the chip can generally be made faster due to the reduced digital logic complexity– this is the basis for **reduced instruction-set computing** (**RISC**), as we learned in class.

Other Special Function Registers (SFRs)

The 68HC12 contains additional registers which are even more focused in their application; each SFR typically has only a handful of instructions that affects it. More accurately, the special function registers are single-task, specialized registers that are used to control or monitor internal or external functions for the microcontroller. We have already learned about several of these registers in class, and in Lab 1.

Program Counter (PC) – The program counter is a 16-bit register which stores the address of the **next instruction** that will be executed by the 68HC12. It is automatically incremented whenever an instruction completes. Note that it will not always increment by one. As we learned in class, different instructions require different amounts of memory to store. At a minimum, most instructions require between one and two bytes to identify the instruction itself. Operands (such as A, B, X, etc.) may be encoded as a portion of these one to two bytes. Immediate values will require additional bytes following the instruction identifier.

CLRA	–	[87]	Clears accumulator, A = [0]
LDAA #12	–	[86] [0C]	Red highlights value stored as part of instruction
NOP	–	[00]	**N**o **Op**eration; CPU waits one cycle

Note that there is no way to directly increment the program counter or change its contents, such as 'LDPC #1234.' Instead, we will use **branch**, **jump**, and **call** instructions in later labs to control the flow of the program (indirectly changing PC in the process). We have already seen unconditional branching in Lab 1 (BRA; it is used to form an endless loop at the end of the main code). Other commands will allow us to branch on certain conditions (such as an overflow condition, or an arithmetic result being zero). Further commands will allow us to call and return from **subroutines** (functions in high-level programming).

Stack Pointer (SP) – The stack pointer is another special function register that points to the **top of the stack**. This means it points to the memory address '**after**' the last item pushed onto the stack – the next free stack space. When you execute a **PSH** (push) instruction, the value or register to be pushed is placed in the available memory location pointed to by SP. Then, the SP is **decremented** by one or two (depending on the size of what is pushed) to point to the next open location. When you **POP** a value from the stack, SP is incremented first, and then the data is read back from the stack. We will make extensive use of the stack later to implement **subroutines**, **functions**, and **interrupts** – it is a convenient system for passing variables, parameters, and results to and from other portions of our program.

Condition and Carry Register (CCR) – Almost all microcontroller implement one or more SFRs like the **condition and carry register**. In essence, these are status registers that track important occurrences when running your program. For the CCR, each bit represents a different issue or status.

CCR Bit Meanings

N – Negative; set if MSB of operation result is 1
Z – Zero; set if result of operation is 0
V – Set if operation produced an overflow (interpretation-dependent)
C – Set if operation produced a carry-out (or borrow-in on subtraction)
H – Half-carry flag; set when there is a carry at bit position 3
X – Extended flag; set like C during extended-precision arithmetic
I – Set when an interrupt (asynchronous event) has occurred

I/O Ports – We have seen the use of Port B to display data on the LEDS in Lab 1. For the 68HC12, ports are **memory-mapped –** they are accessed in an identical way to reading or writing data from memory. The port is referred to by a fixed location in memory (or, possibly a range of locations). Thus, any instruction that accesses memory (such as LDAA, STAA, etc.) can work directly with these ports. Other microcontrollers use SFRs to implement input and output control; these will be inherently specific to the particular microcontroller. Still others have dedicated instructions that send data to and from the I/O pins; there are advantages to each approach.

Other SFRs – The 68HC12 (like other microcontrollers) has multiple other SFRs which are used to control and communicate with special circuitry and functions within the chip. For example, there are registers to set up **interrupts**, **timers**, and other special features. We will come back to study these at a later time.

Programming and Arithmetic

In this lab, we will continue our binary arithmetic work from last lab. By the end, you should be comfortable writing short programs in assembly code, as well. In the next lab, we will look at addressing modes and begin working with high level code.

Our task is to create a program that we can use to perform addition and subtraction **without** having to rewrite the code to change the numbers being added or subtracted. We will begin by writing a simple C-like pseudo-code program, which you will then translate into assembly language.

It is important to note that while the 68HC12 has multiple output and input ports available, on this particular trainer board there are many peripherals and devices already connected to these ports. While we can turn off **some** of these devices easily, we will structure our program to make use of the two easiest-to-access ports, Port B and Port H. In particular, we will use Port B (which has LEDs connected) to display operands and results, and we will use Port H (which has eight on/off switches connected) to input the numbers / operands. We will also make use of the keypad for some additional controls, although this code will be provided to you for now.

Goal: Write a program that accepts and stores two numbers from Port H (entered using the red row of small switches). It will also accept a third input from Port H that will control the **calculation function**: if Bit 7 is set (1) the program adds, otherwise it subtracts. This same input will also control the **display**: if Bit 0 is set (1) the program displays the result of the add or subtract on Port B, otherwise it displays the CCR contents.

Let us begin by writing out our program using a high-level language. We can translate this to assembly language later. In terms of general structure, the above goal will require some form of **loop** which continuously updates the addition/subtraction and displayed data, depending on user input. It will also require somewhere to store operands / results. Lastly, it will require condition checking and logic to add/subtract and display the correct result. To make things simple, we will provide a **keypad** function that handles the input of numbers for you. Thus, the set of steps in pseudo-code could look something like the following:

0. Keep a copy of the current operand 1, operand 2, result, CCR, and desired display function
1. Check if Bit 7 has changed; if so, update the result as addition or subtraction
2. Check if Bit 0 has changed; if so, update what is shown on Port B
3. Ask the external keypad function to update operand 1, operand 2, and/or desired display function if the user has pressed any keys
4. Repeat Steps 1 to 4 forever.

The above will continually update the math function and displayed value. We can now translate from pseudocode into something closer to a high-level (C-like) programming language. This intermediate step is not strictly necessary, but helps in our understanding.

```
void main
{
   uint8 operand1;       // Space to hold operand 1
   uint8 operand2;       // Space to hold operand 2
   uint8 result;         // Space to hold the result
   uint8 ccr;            // Space to hold the PSW after add/sub
   uint8 display;        // Holds the control word which determines the math
                            function and displayed data
   do
   {
      if (Bit 7 of display set)                  // Bit 7 is set
      {
         result = operand1 + operand2;           // Add function
         ccr = _CCR;                             // Save the current CCR
      }
      else
      {
         result = operand1 - operand2;           // Subtract function
         ccr = _CCR;                             // Save the current CCR
      }
```

```
      if (Bit 0 of display set)          // Bit 0 is set, so...
         _PORTB = result;                // ...show A+B or A-B (result).
      else
         _PORTB = ccr;                   // Otherwise, show CCR

      updateKeypad();                    // Call external function to
                                            update operand1, operand2,
                                            and/or display if user presses
                                            appropriate keys
   }
   while (true);
}
```

The overall structure of the above code is relatively straight-forward. We start with an endless do/while loop to ensure that we keep updating the result and display continuously until the program is forcefully terminated. The program defines space to save the result, operands, display function, and CCR. We continuously check Bit 7 to see if the result/CCR needs updating, and Bit 0 to see which result to display. Each loop iteration concludes by asking the external function to check once to see if there has been any user input (button presses on the keypad); if so, the function handles updating operand1, operand2, etc.

Hint: We can use **bitwise AND** to determine if Bit 7 in 'status' is set – 0x80 corresponds to $[1000\ 0000]_2$, so a non-zero result when AND-ing means that Bit 7 is set. In this case, we add A and B.

There is one important thing to notice above – we already have an intrusion of low-level issues into our high-level code. As part of our specification, we want to display the CCR contents after the addition or subtraction. However, **every** assembly instruction has the potential to change CCR. Unless we save a copy **immediately** after the actual add or subtract instruction, the true content of CCR for that operation may be changed or lost by the time we get around to displaying the result on Port B. Basically, we have no way of knowing (at the high level) what instructions will execute between 'result = operand1 + operand2' and '_PORTB = ccr.' Small hardware-level nuances like this have the potential to cause major debugging headaches later, and are not something we normally deal with or are aware of when writing high-level code. The above code gets around this problem by explicitly saving (in the variable 'ccr') a **copy** of the CCR **immediately** after adding or subtracting.

We are now ready to try translating the C-like code into an assembly language program.

Lab Procedure

Step 1. To begin this lab, connect the microcontroller board to its power source and USB plug, making sure to connect the power plug **first**. Open CodeWarrior and chose the option to open an existing project: "Lab2_1.mcp". Once the project is loaded, open the file "main.asm."

Step 2. We will now begin the process of converting our high-level code into assembly. As we will see, even this simple program is relatively lengthy in assembler; we will move to high-level programming very soon to make things more efficient. Recall from earlier that in our code, we have a total of five variables:

Variables

```
uint8 operand1    uint8 operand2    uint8 result    uint8 ccr    uint8 display
```

For simplicity, one might want to assign each of the above to its own register or accumulator. On other platforms this would be possible, but the 68HC12 has a limited number of suitable registers. Instead, we will assign a specific space in memory for each variable to reside at. This is accomplished through the directive DS:

```
operand_1    DS    1
```

The above directive reserves **1 byte** of space (DS 1) which can be referred to (**addressed**) using the label 'operand_1.' Recall from lab 1 that memory is shared; we want to be able to modify the contents of this memory, so we will place it in the memory area specifically reserved for variables. This is denoted by the origin (ORG) directive:

```
ORG RAMStart
… Variables go here …
```

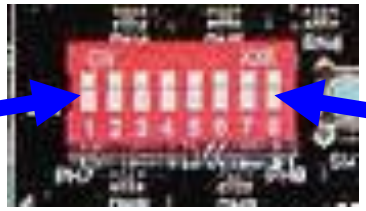
For now we will not be concerned with the efficiency of this approach. We can also create the endless DO/WHILE loop using the BRA (**br**anch **a**lways) instruction we saw in some of the code from Lab 1. As we will see in the branching and flow control lecture, BRA is a form of 'always jump' instruction. In this case, it will set the PC to the code address representing the label 'MAIN.' Lastly, we can also call the external function which updates the variables operand1, operand2, etc. using the BSR (**B**anch to **S**ub**r**outine) instruction. Our skeleton code thus far is shown on the following page.

```
void main
{
   uint8 operand1;
   uint8 operand2;
   uint8 result;
   uint8 ccr;
   uint8 display;
   do
   {

      updateKeypad();
   }
   while (true);
}
```

```
ORG RAMStart

operand_1     DS 1
operand_2     DS 1
result        DS 1
ccr_status    DS 1
display       DS 1

ORG ROMStart
              ; *** Some initializing code
                will go here to disable
              various board functions ***
MAIN:
              ; *temporary display code*
              LDAA operand1
              STAA PORTB
              ; *temporary display code*

              ; Call the external keypad f'n
              BSR KEYPAD
              ; Loop back to start
              BRA MAIN
```

In the above, the red highlighted code is temporary display code. This will allow us to verify that our simple program is working. Right now, we will constantly display the contents of variable operand_1 on PORTB. You can modify this line to display the content of variables 'operand_2' and 'display,' both of which can already be changed by the function 'KEYPAD'.

Step 3. To test functionality, build the program and download to the board. Once running, test your program by flipping some switches on PORTH (the red DIP switch bank):

Bit7 of PORTH  Bit0 of PORTH

Then, press 'A' on the keypad to load the switch values into 'operand1.' For reference, pressing 'B' will load 'operand2,' and pressing 'C' will load 'display.'

The PORTB LEDs should mirror the switch positions on PORTH. You can try changing the display code (LDAA/STAA) to show 'operand2' and/or 'display.' If you feel so inclined, you can also examine the keypad function in the code; it is well-commented and is conceptually understandable, even if the exact operation of the keypad itself hasn't been explained yet.

Step 4. Now that we have the basics complete, let's set up the arithmetic operation. We can notice that we will need to check two conditions in the IF/ELSE statements – one for the arithmetic section, and one for the display. To begin, convert the IF/ELSE into assembly code for **Question 1** of the hand-in. Specifically, you should do the following:

a. Close the previous project and open Lab2_2.mcp → main.asm. This project contains the previous main loop and keypad code, plus a skeleton for the two IF/ELSE statements.
b. In this skeleton, write the assembly language instructions (at the two indicated locations) that would check Bit7 and Bit0. Specifically, you will need to (i) load the appropriate bitmask for each IF condition (i.e., Bit 7 for Add/Subtract and Bit 0 for Display CCR/Result) in one of the accumulators, and then logical AND this mask with the 'display' variable.
c. If the result of the AND operation is zero, then the bit being checked is not set and the ELSE clause should be executed. Otherwise, we execute the THEN clause and branch **over** the ELSE clause.
d. This functionality is already implemented for you via the use of a **conditional branch** instruction: BEQ (or, **Br**anch if result **EQ**ual to zero). Thus, if you simply complete the AND instruction, the CCR will be updated and either the 'THEN' or 'ELSE' clause will be executed, depending on whether the result of AND is zero or not zero. Note that you could set up the IF/ELSE statement in the opposite order by using a BNE instruction (**B**ranch if **N**ot **E**qual to zero). Although functionally equivalent, good coding dictates that we would choose the order for the THEN/ELSE clauses to minimize the average time taken – in other words, on the average is the 'THEN' clause executed more often, or the 'ELSE' clause? In this case, however, we do not need to worry about such details, as execution time is not important.

Step 5. Once the condition checking code is complete, download and load the program onto the board, but do not run it yet. It should follow a structure like the code below:

```
MAIN:
IF1:        ; Your code to check Bit7 goes here
            ; Your code to check Bit7 goes here
            BEQ ELSE1          ; If display.Bit7 is zero, jump to the ELSE condition
THEN1:      <Display 0xF0>     ; Otherwise, execute the THEN condition
            BRA ENDIF1         ; End of THEN condition
ELSE1:      <Display 0x00>     ; Start of ELSE condition
ENDIF1:                        ; End of first IF statement
IF2:        ; Your code to check Bit0 goes here
            ; Your code to check Bit0 goes here
            BEQ ELSE2          ; If display.Bit0 is zero, jump to the ELSE condition
THEN2:      <Display 0x0F>     ; Otherwise, execute the THEN condition
            BRA ENDIF2         ; End of THEN condition
ELSE2:      <Display 0xFF>     ; Start of ELSE condition
ENDIF2:                        ; End of first IF statement
            BSR KEYPAD
            BRA MAIN
```

Step 6. Set the switches on PORTH to **all zero**. Step through the program line-by-line until you reach the label IF1. This is where your Bit7 checking code should be. Step through this code to verify that the **Z** flag (zero) in the CCR is set after the logical AND (just before BEQ). As you continue, you should see microcontroller jump to and execute the ELSE1 clause code, which will leave all PORTB LEDs off (0x00 output displayed).

Step 7. Next, continue stepping until the second BEQ instruction (after your Bit0 check code). Again, you should notice the ELSE2 clause execute, this time setting all PORTB LEDs to the on state (0xFF output).

Step 8. Now, allow the program to run freely by pressing Run/F5. Set the PH7 and PH0 switches to one (up position) and press the 'C' key on the keypad to load the new value into the 'display' variable. Once complete, press Halt/F6. Step the program until you reach the 'BRA main' instruction, which takes you back to the start of the loop.

Step 9. At this point, we have loaded the 'display' variable to contain [1000 0001], which should mean that both 'THEN' clauses are executed (Bit7 and Bit0 are set). Step through the two IF statements again and verify that this is the case – you should see the pattern [1111 0000] appear for THEN1 and the pattern [0000 1111] appear for THEN2.

We will study conditional branching in more detail in the next lab, but this should demonstrate the basics required to set up structures like IF/ELSE. Answer **Question 2** in the hand-in.

Step 10. Close the previous project and open the final skeleton, Lab2_3.mcp → main.asm. Complete the conversion of the high-level code into assembly using the template given; you can start by copying over the code you wrote to check Bit0 and Bit7. As a hint, CCR can be accessed using multiple instructions – look at TPA and TFR in the instruction set handout. Note that there are also instructions which can load data **into** the CCR – this is typically bad practice, however, as it can lead to unpredictable behaviour. Also, keep in mind that when you retain a copy of CCR in the variable 'ccr_status,' this should be done **immediately** after the math operation (add/subtract) to avoid another instruction changing the CCR.

```
MAIN:
IF1:        ; Your code to check Bit7 goes here
            ; Your code to check Bit7 goes here
            BEQ ELSE1          ; If Bit7 is zero, jump to the ELSE condition
THEN1:      ; Your code for result = operand1 + operand2; goes here
            ; Your code for ccr_status = CCR; goes here
            BRA ENDIF1         ; End of THEN condition
ELSE1:      ; Your code for result = operand1 – operand2; goes here
            ; Your code for ccr_status = CCR; goes here
ENDIF1:                        ; End of first IF statement
IF2:        ; Your code to check Bit0 goes here
            ; Your code to check Bit0 goes here
            BEQ ELSE2          ; If Bit0 is zero, jump to the ELSE condition
THEN2:      ; Your code to display ccr_status on PORTB goes here
            BRA ENDIF2         ; End of THEN condition
ELSE2:      ; Your code to display result on PORTB goes here
ENDIF2:                        ; End of first IF statement
            BSR KEYPAD
            BRA MAIN
```

Write your completed code listing for the main loop in **Question 3** of the lab hand-in.

Step 11. Now that you have completed your program, you may use it to carry out simple arithmetic. To this end, complete the following additions and subtractions using the program. Answer **Question 4** and **Question 5** in the lab hand-in.

(a) 12 + 67
(b) 190 + 51
(c) 81 – 23
(d) 15 – 65

Step 12. (BONUS – Complete for extra 0.5 out of 1.5 pts) – The code as written may still quite inefficient in many aspects. For instance, we may make inefficient use of registers and accumulators. There may instances of duplicated code or inefficient code structure/order (as we have essentially converted 1:1 from high-level code). Answer **Question 6** by changing the previous code in one small way. You may either (i) make the code faster or more efficient, or (ii) add a small feature of your choosing. Test and verify your addition. Note that this step could be done after the lab using the simulator, if desired. Your goal here should be to try something new using the microcontroller.

Here are a few suggestions if you do not know where to start:

(a) The code performs the math operation every loop iteration, even if the operands have not changed. You could try to move the math code into the 'keypad' routine so it only runs when operand1/2 change. Look for where the variables operand1 / operand2 are actually changed by STAA.

(b) You can try to make your code shorter / faster. A general rule of programming and compilation is that short code is fast code (not always true, but a reasonable guideline nonetheless). Look for ways to remove duplication and wasted effort.

(c) You are also free to add a feature to the program. For instance, you could add a way to view the operands using the remaining switches on PORTH/display.

***** End of Laboratory 2 *****

MIE438 – Microprocessors and Embedded Microcontrollers

Laboratory Experiment 2

Deliverable

Spring 2016

Department of Mechanical and Industrial Engineering

University of Toronto

Last Name, First Name	Student Number

Q1. Record the code for the bitmasks / AND here:

Q2. Speculate on the function of the BRA ENDIF1 and BRA ENDIF2 instructions in this code. Why are they needed?

Q3. Record your code listing for the completed arithmetic program (main loop only, no initialization or other code needed). You can also attach a printout.

Q4. For each addition and subtraction, record the result as outputted by your program. Also record the CCR register for each, again using your program.

Q5. Was the overflow flag set correctly when calculating the results for Question 4, assuming a two's complement math operation? If not, list how to change your code to correct this issue (**Hint**: Any instruction executed between the math operation, say ABA, and copying the CCR to a register may cause flags to be reset, particularly overflow).

Q6. (BONUS) Attach your modified code listing as a printout.

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