ECE 362 Lab Verification / Evaluation Form

Experiment 3

Evaluation:

IMPORTANT! You must complete this experiment during your scheduled lab period. All work for this experiment must be demonstrated to and verified by your lab instructor *before the end* of your scheduled lab period.

STEP	DESCRIPTION	MAX	SCORE
1	`nchars`	3	
2	`ncmain`	3	
3	`reverse`	5	
4	`pmsgx`	2	
5	`message`	6	
6	`eval`	6	
	TOTAL	25	

Signature of Evaluator: _____

Academic Honesty Statement:

IMPORTANT! Please carefully read and sign the Academic Honesty Statement, below. *You will not receive credit for this lab experiment unless this statement is signed in the presence of your lab instructor.*

"In signing this statement, I hereby certify that the work on this homework and experiment is my own and that I have not copied the work of any other student (past or present) while completing them. I understand that if I fail to honor this agreement, I will receive a score of ZERO and be subject to possible disciplinary action."

Printed Name: _____ Class No. __ __ __ __ - __

Signature: _____ Date: _____

Experiment 3: Assembly Language Programming Techniques – Part 2

Instructional Objectives:

- To learn to write more complex assembly language programs which include various subroutine parameter passing techniques.
- To learn about macros and their appropriate use in assembly language programs.

Reference:

- ***S12CPUV2 Reference Manual***, Sections 2, 3, 5, and Appendix A (on CD included with kit)
- ***CPU12 Reference Guide*** (on course web site)

Prelab Preparation:

- **Read this document and write the code for *all* steps prior to coming to lab**

Parameter Passing Techniques

Writing modular code will be heavily emphasized in this course. Requisite to creation of modular code is a thorough understanding of how parameters may be passed from one procedure to another. You were introduced to this concept in Part 1, and will further explore it in Part 2. There are four basic ways parameters may be passed: (a) via registers ("call by value"), (b) via pointers ("call by address") to a global parameter area, (c) via a private parameter area (here, following the "call" instruction), and (d) via the stack.

(a) Via Registers

Any time a small number of parameters is involved, passing them via the CPU's registers represents the most effective technique. Conversion routines and so-called "I/O device driver" routines are good examples of applications where passing parameters via registers is highly desirable. There are two character I/O routines that utilize this technique which we will use considerably in the remaining of the course: inchar and outchar. The subroutine outchar transmits the ASCII character passed to it in the A register to the terminal screen using the on-chip HCS12 Serial Communications Interface (SCI). The subroutine inchar waits to receive a character from the keyboard (also using the SCI), and when it does, returns its ASCII value in the A register.

(b) Via Pointers

Often times a *structure* must be passed to a subroutine. Because structures (e.g., arrays or linked lists) are often large and of variable size, it is not feasible to pass them via registers. Therefore, a *pointer* to the *starting address* is often passed to the subroutine – typically, via an *index register*. Since the actual values are not passed (rather, they reside in *globally accessible* locations in memory), we refer to this technique as "call by address" or "call by name".

(c) Via Private Parameter Area

In some instances it is appropriate to associate a *private parameter area* with each call to a given subroutine. A good example is a message printing routine, where a different string is to be printed each time the routine is called. One way this can be accomplished in assembly language programming is to *place the parameters following the "call" (i.e.,* `jsr` *or* `bsr` *instruction).*

Use of a private parameter area is illustrated in the following example:

```
      jsr     pmsg
      fcb     'Print this string'
      fcb     NULL    ; ASCII null (string termination character)
      {next instruction}
```

Note the data (here, an *ASCII string of characters*) is "sandwiched" between the "JSR" instruction and the instruction that immediately follows. As indicated in the listing of `pmsg`, below, two things must be accomplished: (1) a pointer to the start of the string must be determined, and (2) the return address (on top of the stack) must be *corrected* so it points to the "next instruction." Also note the `pmsg` subroutine continues to output characters until it reaches the `NULL` character. It is therefore important to remember to terminate your string with the `NULL` character or you might be printing some interesting and unwanted characters.

```
NULL    equ     0       ;ASCII null (string termination character)
pmsg    pulx            ;return address (top stack item)
                        ;points to start of string
ploop   ldaa    1,x+    ;access next character of string
        cmpa    #NULL   ;test for ASCII null (termination)
        beq     pexit   ;exit if encounter ASCII null
        jsr     outchar ;print character on terminal using
                        ; outchar routine
        bra     ploop
pexit   pshx            ;place corrected return address back
        rts             ;on top of stack and return
```

(d) Via the Stack

The most "generic" way to pass parameters is via the stack. Most modern high-level language compilers (e.g., "C") utilize this technique every time a procedure call is generated – use of the stack to pass parameters facilitates *recursion* as well as *arbitrary nesting* of subroutine calls.

Writing More Complex Assembly Language Programs

In the following exercises, you will have the opportunity to utilize the various parameter-passing techniques while learning to write more complex programs. They will also introduce you to the character I/O routines that we will be using the rest of the semester (and eventually writing ourselves!) as well as the use of macros.

Step 1:

Write a subroutine `nchars` that will print N consecutive ASCII characters, beginning with the specified character. The A register will contain the ASCII value of first character to be printed, and the B register will contain the number of consecutive characters to be printed. For instance, if (A) = $41 (ASCII code of "A") and (B) = $09, a string of nine consecutive letters will be printed, producing the following output:

```
ABCDEFGHI
```

An ASCII table can be found on page 30 of the ***CPU12 Reference Guide***

You should write your code in the Lab 3 skeleton file (available from the course website). The character I/O routines `inchar` and as well as have already been written and are included in the file, along with a "main" program to test the `nchars` subroutine. A routine to initialize the HC12's serial interface (`sinit`) is also provided. Use **Tera Term** on the PC to communicate with the target system's serial port.

Step 2:

Write a program `main` that will prompt for a character, store it in the A register, prompt for a number, store it in the B register, then call the subroutine `nchars` (written for Step 1) to print N consecutive ASCII characters, beginning with the specified character. It should keep repeating these steps until the character "q" is entered. You should use the macro `print` to format the prompt strings for the code.

An example session is as follows:

```
Enter character: N
Enter number: 8
NOPQRSTU
Enter character: N
Enter number: 0

Enter character: ;
Enter number: 7
;<=>?@A
```

Add your code for Step 2 to the Lab 3 skeleton file. Note that the `pmsg` subroutine (used in the macro `print`) is also provided in this file.

Step 3:
Write a program that reads up to 30 characters from the keyboard and stores them in a globally defined array, `marray`. When the user enters either 30 characters or a carriage return, print back the character string in reverse order.

An example session:

```
Hello, how are you?<CR>
?uoy era woh ,olleH
```

You should add your code to the Lab 3 skeleton file from the previous steps.

Step 4:
Write a subroutine `pmsgx` that prints a NULL-terminated ASCII string whose address is passed to it in the X register. You should write your code in the Lab 3 skeleton file from the previous steps. The character I/O routines `inchar` and `outchar` have already been written and are included in the file.

Step 5:
Using the subroutine `psmgx`, write a program message that prompts the user for a number in the range 0 to N. Note that the number N is predetermined, but can be changed. If a number in the range of 1 to N is entered, it should print the corresponding message then repeat the prompt. If the number 0 is entered, the program should print the exit message and then stop. If a number outside of the range is entered, it should print an error message and re-prompt. You should add your code to the Lab 3 skeleton file from the previous steps. Note that the value of N and the messages have been defined, but you may modify them if you wish. An example session is as follows (assume N = 4):

```
Enter number: 1
Message 1.
Enter number: 3
Message 3.
Enter number: 5
Invalid message number. Try again!
Enter number: 0
Good-bye!
```

Step 6:
Write a program `eval` that evaluates the expression that is contained on the stack, and places the result back on the stack. Then write a main program that prompts the user for the two single-digit operands and the function code, then calls eval to print a message of your choice.

The program should be capable of implementing add, subtract, multiply and XOR operations, specified by the following "function codes":

Add:	ASCII code for "A"
Subtract:	ASCII code for "S"
Multiply:	ASCII code for "M"
XOR:	ASCII code for "X"

The operation implemented should be Op1 Function Op2. To simplify the program, assume that the operands and results are single-digit values (the result of the multiply can be truncated to the lower byte). You do not need to implement any error checking.

An example session would be as follows:

```
Op 1: 3
Op 2: 4
Function: A
Result: 7
```

Note that when you enter the subroutine `eval`, the stack will look like the following for that example (assuming an initial value of SP = $1000):

`SP = 0FFB`

0FFA	
0FFB	Return Addr: High Byte
0FFC	Return Addr: Low Byte
0FFD	41
0FFE	04
0FFF	03

The stack will look like the following at end of `eval` (before the return):

`SP = 0FFD`

0FFA	
0FFB	Return Addr: High Byte
0FFC	Return Addr: Low Byte
0FFD	Return Addr: High Byte
0FFE	Return Addr: Low Byte
0FFF	07

You should add your code to the Lab 3 skeleton file. The subroutine `eval` must only modify the CCR (correctly for specified function) and no other register! The character I/O routines `inchar`, `outchar`, and `psmg` have already been written and are included in the file. In addition, you have been provided a function `htoa` which will convert a single digit hexadecimal number to its ASCII character (useful for printing out the result!)

