ECE 362 Lab Verification / Evaluation Form

Experiment 2

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	Data Transfer Group Instructions	3	
2	Arithmetic Group Instructions	6	
3	Logical Group Instructions	5	
4-A	Code Segment A	2	
4-B	Code Segment B	2	
4-C	Code Segment C	2	
4-D	Code Segment D	2	
4-E	Code Segment E	2	
5	Thought Questions	1	
	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 2: Assembly Language Programming Techniques – Part 1

Instructional Objectives:

- To learn more about the HC(S)12 Instruction Set
- To learn about basic assembly language programming techniques by examining several program modules, testing their operation using appropriate data, and modifying these modules.

References:

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

Pre-lab Preparation:

- Read this document in its entirety
- Review the description of the HC(S)12 instruction set provided in Chapter 3 of the Meyer text
- Fill in all the "Before Execution" boxes for Steps 1-3 and complete the ASM skeleton file provided for Step 4

Introduction:

A clear understanding of a machine's instruction set (the "tools in the toolbox") is essential to successful assembly language programming. The primary objective of this laboratory exercise is therefore to provide practical experience with the HC(S)12 instruction set and its plethora of addressing modes – material that is covered in Chapter 3 of the text. To accomplish this goal, you will complete a number of exercises dealing with commonly-used HC(S)12 instructions and addressing modes. Using the Simulator integrated into Code Warrior, you will execute the code segments you have written with appropriate test data you have chosen. This will help reinforce your understanding of how instructions and data are stored in memory, plus help prepare you for future debugging of more complex programs.

In the first part of this experiment you will examine representative instructions from the following general groups: data transfer, arithmetic, and logical. There will be a number of short exercises for which you will be asked to write a code segment and assemble the statements into machine code. For each exercise, you will also pick appropriate test data and perform any memory and/or register initializations necessary to carry out the exercise. You will then execute the code segments using "Full Chip Simulation" mode in Code Warrior and record the results of the execution in the provided tables. The primary objectives of these exercises is to enhance your understanding of how instructions and data are stored in memory, provide practice in determining *meaningful* test data, provide practice in examining the results of instruction execution, and help prepare you for future debugging of more complex program.

In the second part of the experiment, you will combine instructions together to create "code segments" that perform some meaningful and perhaps socially-redeeming tasks (e.g., extended precision arithmetic). The same tools used to analyze the execution of isolated instructions will be used to examine the behavior of the code segments you write.

Step 1: Data Transfer Group Instructions

This group consists of instructions that move data between registers or between memory locations and registers. A list of some of the HC(S)12 instructions that comprise this category is shown below.

Mnemonic	*Description*
EXG	Exchange register values
LD	Load register
MOV	Move memory
ST	Store register
TFR	Transfer register to register
LEA	Load Effective Address
PSH	Push data onto stack
PUL	Pull data from stack

STEP 1-A: `MOVW` instruction using immediate/extended addressing modes.

Assembly Instruction(s)	Machine Code	Relevant Memory	Relevant Registers
		Before execution:	Before execution:
		After execution:	After execution:

STEP 1-B: `LEA` instruction using accumulator offset indexed addressing mode.

Assembly Instruction(s)	Machine Code	Relevant Memory	Relevant Registers
		Before execution:	Before execution:
		After execution:	After execution:

STEP 1-C: PSHA instruction, followed by PSHB and PULD.

Assembly Instruction(s)	Machine Code	Relevant Memory	Relevant Registers
		Before execution:	Before execution:
		After execution:	After execution:

Step 2: Arithmetic Group Instructions

This group consists of instructions which add, subtract, increment, or decrement data in registers or memory. A list of some of the HC(S)12 instructions that comprise this group appears below.

Mnemonic	*Description*
ADC	Add with carry
ADD	Add
DEC	Decrement by 1
INC	Increment by 1
DAA	Decimal adjust accumulator A
MUL	Multiply
SUB	Subtract
SBC	Subtract with carry (borrow)

STEP 2-A: ADCB instruction using immediate addressing mode.

Assembly Instruction(s)	Machine Code	Relevant Memory	Relevant Registers
		Before execution:	Before execution:
		After execution:	After execution:

STEP 2-B: ADDD instruction using extended addressing mode.

Assembly Instruction(s)	Machine Code	Relevant Memory	Relevant Registers
		Before execution:	Before execution:
		After execution:	After execution:

STEP 2-C: ADDA instruction using immediate addressing, followed by DAA instruction.

Assembly Instruction(s)	Machine Code	Relevant Memory	Relevant Registers
		Before execution:	Before execution:
		After execution:	After execution:

STEP 2-D: SUBA instruction using immediate addressing, followed by DAA instruction (what goes wrong?).

Assembly Instruction(s)	Machine Code	Relevant Memory	Relevant Registers
		Before execution:	Before execution:
		After execution:	After execution:

STEP 2-E: `EMUL` instruction using inherent addressing mode.

Assembly Instruction(s)	Machine Code	Relevant Memory	Relevant Registers
		Before execution:	Before execution:
		After execution:	After execution:

STEP 2-F: `FDIV` instruction using inherent addressing mode.

Assembly Instruction(s)	Machine Code	Relevant Memory	Relevant Registers
		Before execution:	Before execution:
		After execution:	After execution:

Step 3: Logical Group Instructions

This group includes instructions which perform logical ANDs, ORs, XORs, compares, rotates, or complements of data in registers or memory. Some of the HC(S)12 instructions that comprise this group are listed below.

Mnemonic	*Description*
AND	Bit-wise AND
OR	Bit-wisel OR
EOR	Bit-wise XOR
ASL or LSL	Arithmetic (logical) shift left
ASR	Arithmetic shift right
CMP	Compare
TST	Test for zero or minus
ROL	Rotate left through carry
ROR	Rotate right through carry

STEP 3-A: `LSLA` instruction using inherent addressing mode.

Assembly Instruction(s)	Machine Code	Relevant Memory	Relevant Registers
		Before execution:	Before execution:
		After execution:	After execution:

STEP 3-B: `RORB` instruction using inherent addressing mode.

Assembly Instruction(s)	Machine Code	Relevant Memory	Relevant Registers
		Before execution:	Before execution:
		After execution:	After execution:

STEP 3-C: `CPX` instruction using extended addressing mode.

Assembly Instruction(s)	Machine Code	Relevant Memory	Relevant Registers
		Before execution:	Before execution:
		After execution:	After execution:

STEP 3-D: `ANDCC` instruction using immediate addressing mode.

Assembly Instruction(s)	Machine Code	Relevant Memory	Relevant Registers
		Before execution:	Before execution:
		After execution:	After execution:

STEP 3-E: `ORCC` instruction using immediate addressing mode.

Assembly Instruction(s)	Machine Code	Relevant Memory	Relevant Registers
		Before execution:	Before execution:
		After execution:	After execution:

Step 4: Writing Code Segments

In the previous experiment you were introduced to various embedded system software development tools. You learned about the translation of assembly language instructions to and from machine code in the assembly and disassembly exercises. Finally, you executed a specific instruction using a variety of different addressing modes. Given those tools, the next objective is to learn how instructions "work together". To achieve this, you will be asked to write several small segments of code. Each segment of code will have a defined function it is to perform and constraints will be given on how it is done or what it can modify in the process. It is important to learn to write code within those constraints since you will often be writing a portion of a program that will interface with other code in a specific way.

A "skeleton file" to facilitate completion of this step is available on the course web site (reproduced below for your reference). The description of each program is contained in the header section. You should download a copy of the file and "fill in" the code segments where indicated. To test the code, you will assemble and load the program in to the Code Warrior debugger, initialize any required register or memory locations, then run the program using Full Chip Simulation mode. You should demonstrate each code segment to your lab instructor as you complete it.

Skeleton file listing:

```
;*******************************************************************************
; Step 4-A:
;
; Write a program that loads two 16-bit numbers from the memory location
; "ops", divides the first number by the second, then stores the quotient
; in the memory location "quot" and the remainder in memory location
; "remain".   Note that the space for the operands and results has
; already been allocated for you below.  Each has a label associated with
; the memory location.  You can use the labels "ops", "quot", and "remain"
; when writing your code, and the assembler will convert it to the
; appropriate memory address.
;
; Note that the instructions for the next step begin at memory location
; $0820.  Check your assembled source listing (".lst") file to make sure
; that your code does not pass that location.

step4a    org     $0800

; Put your code here

          stop    ; use a placeholder for breakpoint

ops       rmb     4
quot      rmb     2
remain    rmb     2


;*******************************************************************************
; Step 4-B:
;
; Write a program that tests whether the unsigned value contained
; in the A register is higher than value stored at the memory location
; "tval".  If it is, the program sets the variable "higher" to $FF,
; and if not, the program sets the variable "higher" to $00.
;
;*******************************************************************************

step4b    org     $0820

; Put your code here

          stop    ; use a placeholder for breakpoint

tval      fcb     100
higher    rmb     1
```



```
;*************************************************************************
; Step 4-C:
;
; Write a program that performs the addition of the two
; 3-byte numbers located at the memory location "adds" and stores
; the result in the memory location "sum".
;
; Note:  You may not change the values of any of the registers.
;
; Therefore, you should push any registers used in the program at the
; beginning of the program, and then pull (pop) them off at the end
; of the program.
;
; NOTE: The operands are stored MSB to LSB.
;
;*************************************************************************
step4c    org     $0840

; Put your code here

          stop    ; use a placeholder for breakpoint

          org     $0870
adds      rmb     6         ; Addends
sum       rmb     3         ; Sum


;*************************************************************************
; Step 4-D:
;
; Write a program that will transfer a specified number of bytes of data
; from one memory location (source) to another memory location (destination).
; Assume that the source address is contained in the Y register, the
; destination address is contained in the X register, and the A register
; contains the number of bytes of data to be transferred.  The X, Y and A
; registers should return with their original values, and the other registers
; should be unchanged.
;
; Note: For this program, you should use a FOR loop. The basic
; structure of a FOR loop is:
;
;         loop    check counter
;                 branch out of loop if done (here, to label "done")
;                 perform action
;                 branch back to "loop"
;         done    next instruction
;
; NOTE: When testing this program, make sure that you are not transferring
; data to memory locations where your program is located!!!  Check your
; assembled listing file to see where your programs are located.
;*************************************************************************

step4d    org     $0890

; Put your code here

          stop    ; use a placeholder for breakpoint
```



```
;*****************************************************************
; Step 4-E:
;
; Write a program that determines how many bits of the number passed
; in the A register are 1's.  The A register should return its original
; value, and the number of 1 bits should be returned in the B register.
;
; Note: For this program, you should use a DO loop. The basic
; structure of a DO loop is:
;
;                initialize counter
;         loop   perform action
;                update and check counter
;                branch back to "loop" if not done
;
; You will need to maintain three pieces of data for this program:
;    (a) initial value (in A register)
;    (b) number of 1 bits (returned in B register)
;    (c) counter for the loop
;
; Since we only have two accumulators available in the HC(S)12, you will
; need to use an index register, a local variable (stored in memory), or
; the stack to implement this.  A memory location with the label "count"
; has been reserved below if you would like to use it.
;
;*****************************************************************

step4e    org     $0920

; Put your code here

          stop    ; use a placeholder for breakpoint

count     rmb     1

          end
```

Step 5: Thought Questions

Place your answers to the following thought questions in the space provided, below.

(a) Why are the (legacy) ABX and ABY instructions not included in the native HC(S)12 instruction set?

_____

_____

(b) Why are ORCC and ANDCC instructions provided in addition to ones like SEC, CLC, SEI, CLI, SEV, and CLV?

_____

_____

