

ECE 362 Lab Verification / Evaluation Form

Experiment 1

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	Demonstration of Learning the Tools	1	
2	Verification of Hand Assembly Using Full Chip Simulation	2	
3	Verification of Hand Disassembly Using Full Chip Simulation	2	
4	Exploring Addressing Modes (10 Exercises)	20	
	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 1: Freescale HC(S)12 Instruction Set Overview

Instructional Objectives:

- To learn how to use your Freescale M68DKIT912C32 Microcontroller Kit
- To learn how to use HC(S)12 instructions and addressing modes

References:

- *S12CPUv2 Reference Manual*, Sections 2, 3, 5, and Appendix A (on *References* page)
- *CPU12 Reference Guide* (on *References* page)
- *Using Your M68DKIT912C32 Microcontroller Kit* (on *References* page)
- *M68DKIT912C32 Quick Start Guide* (on *References* page)

Pre-lab Preparation:

- Complete the assigned homework set and carefully read through the experimental procedure.

Introduction:

Chapter 3 of the preliminary text provides an introduction to the architectural and programming model of the “target” microcontroller that will be used in all the laboratory exercises: the Freescale HC(S)12. As you go through this lab, you will see many similarities between the HC(S)12 and the simple computer covered in ECE 270) – for example, the bus structure, program counter, stack operation, subroutine linkage mechanism, and the actual machine instructions themselves (LDA, STA, ADD, AND, etc.). As you might guess, though, the HC(S)12 has many more capabilities than our “home grown” simple computer: its instruction set is much more powerful, it has several additional registers, its data types are more diverse, and its addressing modes are more extensive. The purpose of this lab, then, is to introduce you to the basic features and characteristics a “real” microprocessor generally possesses.

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 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 the “Full Chip Simulation” mode in Code Warrior, and record the results of the execution in the provided tables. The primary objectives of these exercises are to: (a) enhance your understanding of how instructions and data are stored in memory, (b) provide practice in determining *meaningful* test data, (c) provide practice in examining the results of instruction execution, and (d) help prepare you for future debugging of more complex program.

Code Warrior Tutorial

CodeWarrior is the integrated development environment (IDE), which we will use to edit, load, and debug our code. This is a quick tutorial that will give you an introduction to the tools we will be using for the duration of the semester.

Start CodeWarrior on the lab computers as follows:

Start > Programs > ECE Software > CodeWarrior IDE

When the Startup window comes up, select **Create New Project**.




When the *New Project* window appears, select:

HCS12 > HCS12C Family > MC9S12C32

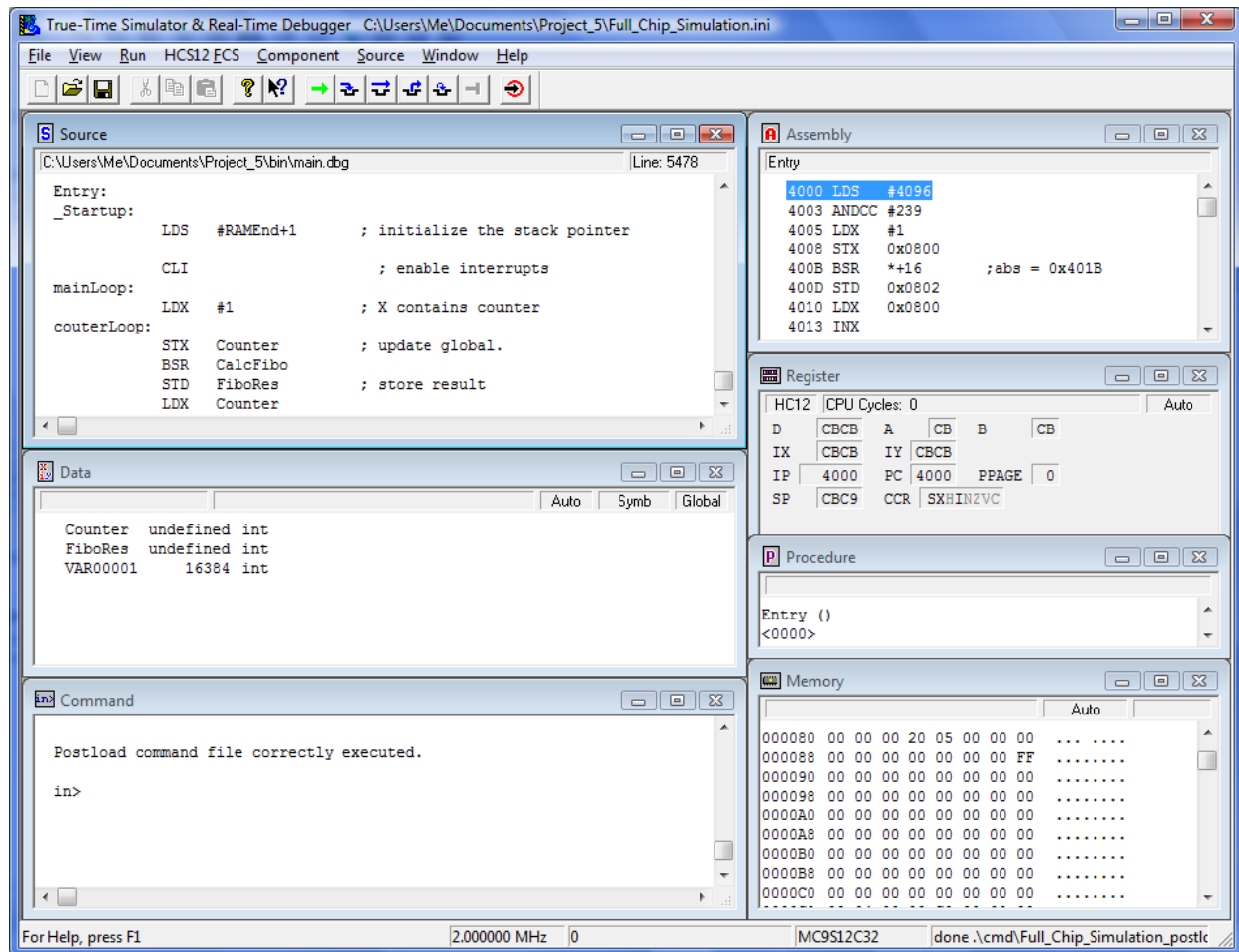
In the right window, select either **Full Chip Simulation** (to simulate code execution without a target microcontroller board attached) or **TBDML** (to operate an attached microcontroller board in background debug mode – requires a **USBDMILT** pod, available in lab or separate/optional purchase). When finished, press **Next**. Deselect the check by **C** and select **Absolute Assembly**. Press **Finish**.

NOTE: For the first three experiments, **Full Chip Simulation** mode can be used. Starting with Experiment 4, applications will be loaded in flash memory, which will require use of the **USBDMILT** pod. These experiments can also be completed using the Microcontroller Kit.

In the left frame, double-click **main.asm** to open the assembly source file. You will edit the contents of this file later, but for the purposes of this tutorial, you can leave the preloaded program as it is.

Now, click the **Debug** button () to start the debugger. On the dialog box that appears, click OK. The debugger will erase the flash memory on the microcontroller chip (if using the TBDMLT pod) and load your **main.asm** program.

This is the **Debugger window**, which you will use to run and debug your code.

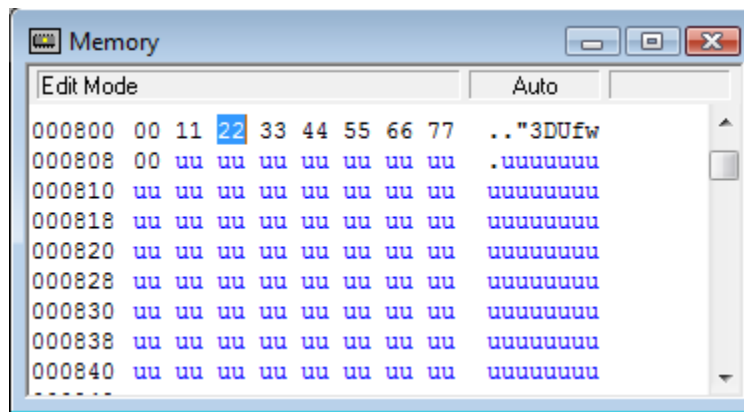


The **Source window** is where you can view the progress of your program's execution. Each time you **start** () or **single step** (), you will notice that a new line will be highlighted in the source window. The highlighted line is the next instruction to be run.

If you want to set a **Breakpoint** in your code, you can right-click the line in the source window and select **Set Breakpoint**. This means that when the debugger encounters this address, it will halt the processor before running that instruction. Get comfortable with this feature and use this often when debugging code. It will also come in handy if you want to run a set of instructions and stop before running another. You can delete a breakpoint by right-clicking again and selecting **Delete Breakpoint**.

The **Assembly window** allows you to view memory as a sequence of disassembled instructions. Be careful though: this does not mean that any given address contains data that was meant to be interpreted as instruction. For example, a value in memory of \$FF (0xFF) will be interpreted as an LDS instruction, even if this data wasn't meant to be an instruction.

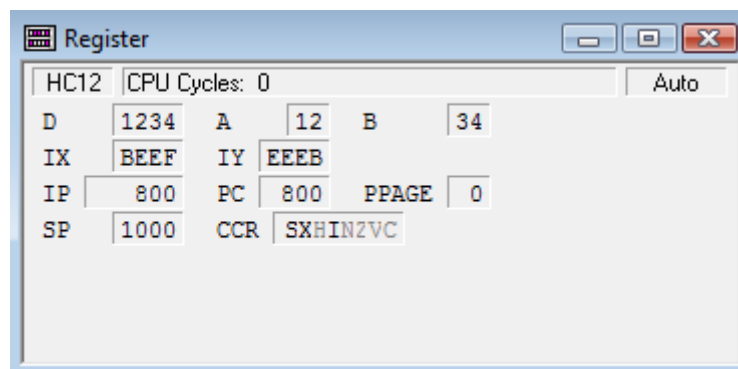
Note: Hexadecimal numbers will sometimes be notated by a "0x" or "\$" prefix or an "h" suffix. For example, you might see 20₁₆ written either as "0x20", "\$20" or "20h". It should also be noted that **CodeWarrior will only allow the "\$" notation for hex numbers in your code.**



The **Memory window** allows you to view and edit the contents of memory (Note: All of the values are displayed in hexadecimal format). You can view a specific address by right-clicking inside the window, selecting **Address**, entering the address in the text box, and pressing **Okay**. The leftmost number in each row is the base address for that row.

Each grouping of two hexadecimal digits located to the right of the row base address is a byte of data in memory. The address of each byte of data corresponds to the row base address plus the zero-based column number. For example, in the above figure, the first row base address is \$800 (0x0800). The first column corresponds to address \$800 (0x0800) and contains data \$00 (0x00); the second column corresponds to address \$801 (0x0801) and contains data \$11 (0x11); etc.

If you double-click a given byte of data, it will become highlighted, and you can edit it. Once you edit the data, if the address of the data you changed is also visible in the **Assembly window**, you will see it update as well.



The **Register window** contains the values of the various registers on the microcontroller. To edit any register's value, you can double-click on it and enter a new value. To toggle any value in the CCR (condition code register), you can double-click on the letter of the bit you want to change. If you modify the value in the PC (program counter) register, you will change the address of the next instruction your microcontroller will execute.

If you have any variables in your code, you can watch them by right-clicking in the **Data window**, clicking on **Add Expression...**, and typing in the variable name. Make sure that you also change **Mode** under the right-click menu to **Periodical**. Enter “1” into the text box for the fastest data refresh rate. (This will allow the values to automatically update.)

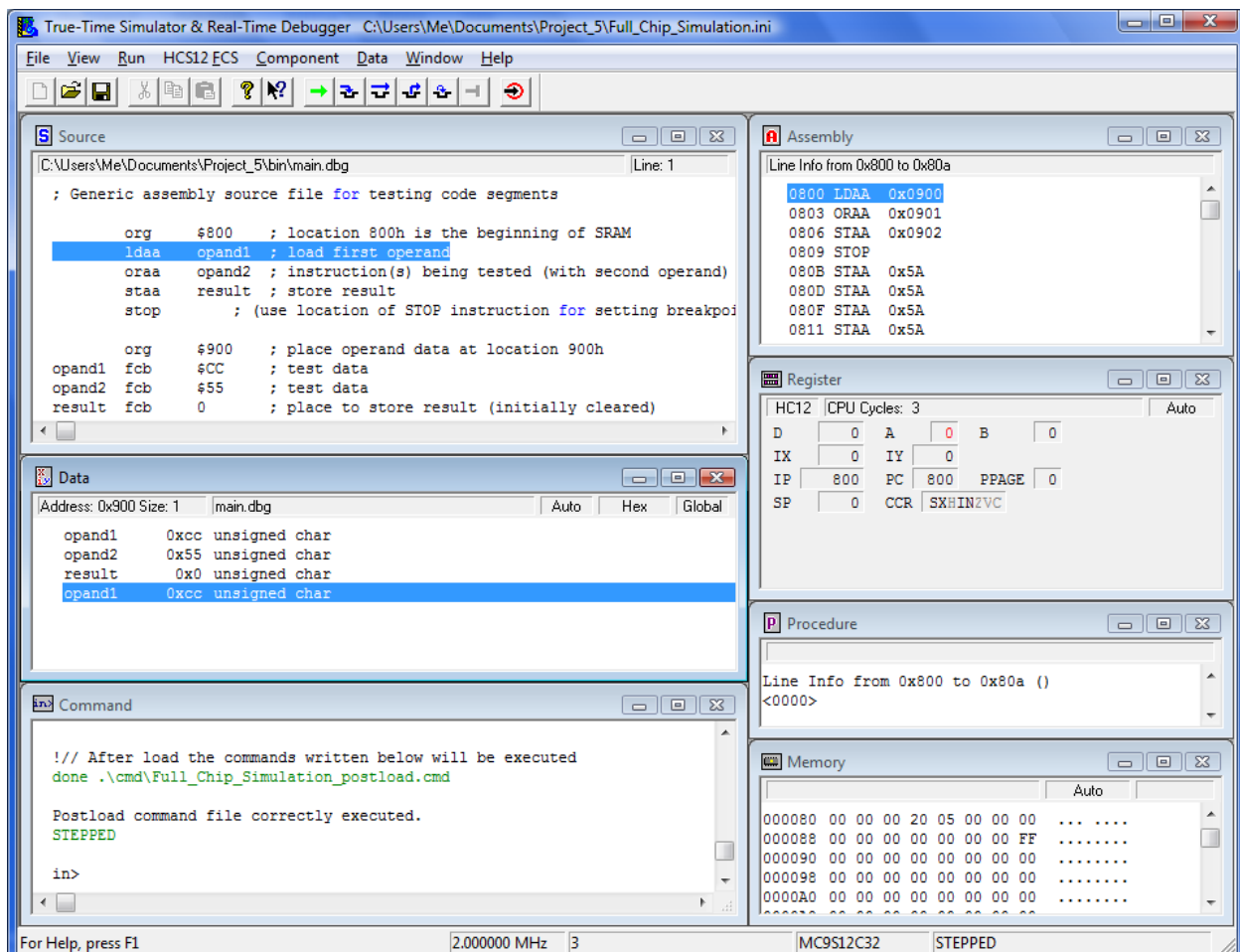
To examine the effect of executing code segments using the development environment in lab, use the following “generic” assembly source file for each test case (available on the course website):



```
; Generic assembly source file for testing code segments

    org     $800      ; location $800 is the beginning of SRAM
    ldaa    opand1     ; load first operand
    oraa    opand2     ; instruction(s) being tested (with second operand)
    staa    result     ; store result
    stop    ; SET BREAKPOINT HERE

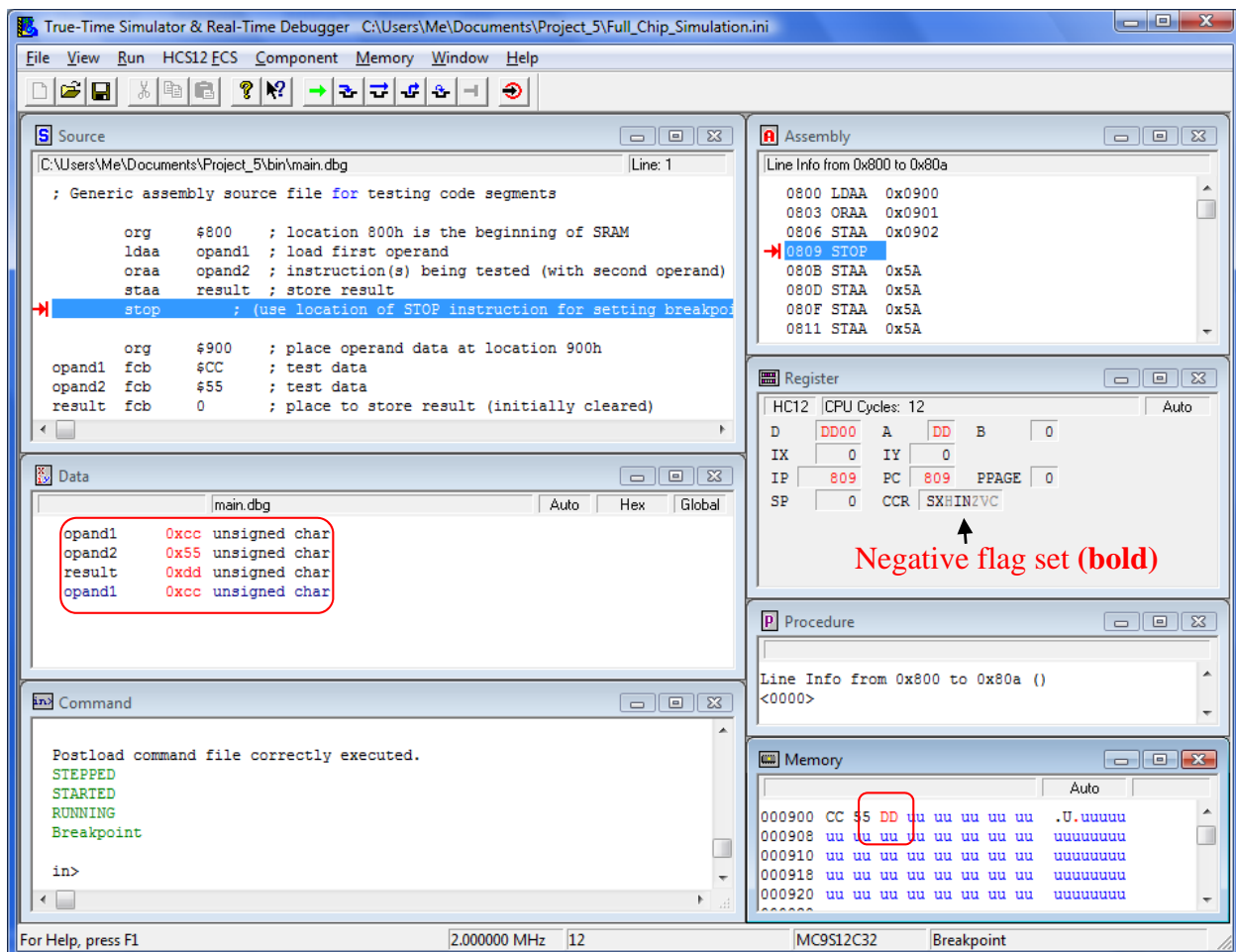
    org     $900      ; place operand data at location $900
opand1 fcb     $CC     ; test data
opand2 fcb     $55     ; test data
result fcb     0       ; place to store result (initially cleared)
end
```

Loading this code into the debugger should yield the following (after setting PC to \$800 and clearing the other registers). If your code is not displayed in the source window after you change the PC, press the single step button, change the PC back to \$800 and re-clear the registers.



At this point, you can either “single step” () through the code or “Start” () to execute the code up to a “breakpoint”. (A convenient place to set a breakpoint is the STOP instruction.) The memory and register values will change as the code executes. Note that the assembly window can be used to display the machine code as it is stored in memory, while the data and memory windows can be used to monitor how memory locations are being modified as the code executes.

After execution of the code segment is complete, the debugger display should be as follows:



Note that the value \$DD (0xDD) has been written to memory location \$902 (0x0902, “result”), that the A register also (still) contains the value \$DD (0xDD), that the Negative Flag (N) has been set by virtue of the result generated, and that the Zero Flag (Z) remains cleared.

Experimental Procedure

You are to write and assemble the instruction sequence requested for each exercise that follows. You must also choose appropriate test data, and show initialization data for any memory locations and/or registers necessary to perform the exercise. Note that test and initialization data is to be placed in the *before execution* boxes of each table. In addition, no memory locations or registers you choose as relevant should be left as *don't cares*. If no memory locations are required to perform a particular exercise, write "NONE" in the before execution box under relevant memory. The following sample exercise illustrates how a table should be completed.

SAMPLE EXERCISE: "ORAA" instruction using extended addressing mode.

Assembly Instruction(s)	Machine Code	Relevant Memory	Relevant Registers
ORAA \$900 Effective Address: \$900	BA 09 00	Before execution: (0900) = 55	Before execution: (A) = CC N = 0, Z = 1
		After execution:	After execution:

In the table above, memory location \$900, along with the test data \$55 and \$CC, have been chosen arbitrarily. However, note that the chosen memory locations reside in *user* memory space, and that \$55 and \$CC are *meaningful* test data (i.e., ORing \$55 and \$CC tests all possible bit combinations and produces a change in the A register and flags which will clearly indicate the code segment works.)

You will then execute your code segments using Full Chip Simulation Mode in Code Warrior (after having initialized any relevant memory locations and/or registers). Execution results will then be recorded in the *after execution* boxes provided in the tables. For example, after executing the ORAA instruction given above, the sample exercise table would be completed (based on memory and register observations) as shown below.

SAMPLE EXERICSE: "ORAA" instruction using extended addressing mode.

Assembly Instruction(s)	Machine Code	Relevant Memory	Relevant Registers
ORAA \$900 Effective Address: \$900	BA 09 00	Before execution: (0900) = 55	Before execution: (A) = CC N = 0, Z = 1
		After execution: (0900) = 55	After execution: (A) = DD N = 1, Z = 0

Note that the effective address for this example is \$900, since extended addressing mode was used. Note also that if you choose to put your data at \$900, you will need to place your code at *different* memory locations – a suggested location would be \$800.

Step 1: Learning the Tools

Using the generic test file available on the course website (listed on page 5 of this document), verify that you get the same results as given for the SAMPLE EXERCISE. Demonstrate to your lab instructor that you can assemble a source file, load the assembled machine code into the target system using the debugger, trace through the code execution, and document the results obtained from executing the code sequence.

Step 2: Verification of Hand Assembly Using the Microcontroller Kit

Enter the machine code from **Homework 1 Problem 1** using the memory window in the debugger. Execute the program to verify its operation. Try two additional values for `bcдин` by modifying the location in which it is stored, and record the “after execution” results found in memory location `binout`. Explain what happens if “illegal” (i.e., non-BCD) data is used – *try it!*

Explanation: _____

Step 3: Verification of Hand Disassembly Using the Microcontroller Kit

Enter the machine code from **Homework 1 Problem 2** using the memory window in the debugger. Store the opcodes beginning at location \$900, initialize (A) to 0, (B) to 0, (SP) to \$C00, (PC) to \$900, and the condition code (CCR) register to 0 (i.e., all flags cleared). Then, single step through the program to verify its operation and fill out the table below. Compare these results to the values you “predicted” when you completed the assignment by hand.

Execution Step	(PC)	(A)	(B)	CCR bits set
Initial Values	0900	00	00	–
After Single Step 1				
After Single Step 2				
After Single Step 3				
After Single Step 4				
After Single Step 5				
After Single Step 6				
After Single Step 7				
After Single Step 8				
After Single Step 9				
After Single Step 10				

Step 4: Exploring Addressing Modes

Using the ADDA instruction, you are going to explore the various addressing modes. For each addressing mode, write the instruction in the appropriate format, and assemble the instruction. You must also choose appropriate test data, and show initialization data for any memory locations and/or registers necessary to perform the exercise. Note that test and initialization data is to be placed in the *before execution* boxes of each table. In addition, no memory locations or registers you choose as relevant should be left as *don't cares*. If no memory locations are required to perform a particular exercise, write "NONE" in the before execution box under relevant memory.

Step 4-A: ADDA instruction using **immediate** addressing mode.

Assembly Instruction	Machine Code	Relevant Memory	Relevant Registers
Effective addr:		Before execution:	Before execution:
		After execution:	After execution:

Step 4-B: ADDA instruction using **extended** addressing mode.

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

Step 4-C: ADDA instruction using **indexed with 5-bit constant offset** addressing mode.

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

Step 4-D: ADDA instruction using **indexed with 8-bit constant offset** addressing mode.

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

Step 4-E: ADDA instruction using **indexed with 16-bit constant offset** addressing mode.

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

Step 4-F: ADDA instruction using **indexed with (byte) accumulator offset** addressing mode.

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

Step 4-G: ADDA instruction using **indexed with auto post-increment by 1** addressing mode.

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

Step 4-H: ADDA instruction using **indexed with auto pre-decrement by 1** addressing mode.

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

Step 4-I: ADDA instruction using **indirect indexed with constant offset** addressing mode.

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

Step 4-J: ADDA instruction using **indirect indexed with accumulator offset** addressing mode.

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