

# 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
Pre-Lab 1	Hand Assembly	5	
Pre-Lab 2	Hand Disassembly	5	
1	Learning the Tools	2	
2	Verification of Hand Assembly Using Microcontroller Kit	2	
3	Verification of Hand Disassembly Using Microcontroller Kit	2	
4	Exploring Addressing Modes	9	
	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 pre-lab exercises and have them available to check off when your lab begins

### 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 your Microcontroller Kit, 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 on your Microcontroller Kit, 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.

**Pre-lab:**

The pre-lab exercises for this experiment address how *assembly language mnemonics* are translated to/from *machine (or “object”) code* – processes referred to as program *assembly* and *disassembly*. These processes can be performed either “by hand” or by using a software development tool called an *Integrated Development Environment (IDE)*.

**Hand Assembly Example**

While not something you would want to do every day, the process of “hand-assembling” a program provides insights concerning how instructions are encoded and stored in memory; also, it provides you with a much greater appreciation of all the tedious work the IDE does for you in a matter of microseconds! The following short program will load the A and B accumulators with 8-bit unsigned integer operands and multiply them to produce a 16-bit result. A description of each step in the hand-assembly process is provided below.

Addr	[~]	Obj Code	Line No.	Assembly	Mnemonics
0800			1	org	\$800
0800	[02]	CE0900	2	ldx	#opands
0803	[03]	A630	3	ldaa	1,x+
0805	[03]	E600	4	ldab	0,x
0807	[03]	12	5	mul	
0808	[03]	7C0902	6	std	prod
080B	[09]	183E	7	stop	
			8		
0900			9	org	\$900
0900		04	10	fcdb	4
0901		03	11	fcdb	3
0902			12	rmb	2
0904			13	end	

**Step-by-Step Description of Hand-Assembly for Example Program**

- (1) The ORG pseudo-op simply tells the assembler that the next opcode byte or data byte should be placed at the address indicated by the ORG statement. Also, the label “start” is assigned the value \$800. *Note that no “object code” is assembled for this statement!*
- (2) The opcode for the first instruction – “LDX” – can be found by examining either page 13 of the *CPU12 Reference Guide* or Table A-1 of the *S12CPUv2 Reference Manual*. Since *immediate* addressing mode is used (as indicated by “#”), the opcode is CEh. But because “opands” is a *forward reference*, we can not as of yet determine the two-byte value (here, an *address*) which should be placed following this opcode. On the “first pass” of the assembly process we will therefore reserve the requisite two bytes and mark “opands” as an *undefined symbol*. As the “first pass” of the assembly proceeds, all labels/symbols used in the program *should* be assigned a value; to keep track of all the symbols used in a program and their assigned values, a *symbol table* is utilized. Once the “first pass” is complete (and each entry in the symbol table is assigned a value), a “second pass” is used to “fill in the blanks” – i.e., to fill in the forward reference values which were left unassigned during the “first pass” of the assembly process. At this point, then, our symbol table is as follows:

SYMBOL	VALUE
start	\$800
opands	<undefined>

- (3) Because three bytes were reserved for the previous instruction (“LDX”), the opcode of the next instruction – “LDAA” – will be stored at location \$803. Since an indexed addressing mode is used, the opcode is A6h and a “postbyte” must be included following the opcode to indicate the *particular type* of indexed addressing mode used. The format of the requisite postbyte can be determined using the “Indexed Addressing Mode Postbyte Encoding” table on page 21 of the *Guide* or in Table A-3 of the *Manual*; here we find the postbyte should be \$30 (the actual encoding scheme used to derive this can be found on page 22 of the *Guide* or in Table A-4 of the *Manual*).
- (4) Since two bytes of program memory were consumed by the previous instruction (“LDAA”), the opcode of the next instruction – “LDAB” – will be stored at location 0805h. Since an indexed addressing mode is used, the opcode is \$E6 and a “postbyte” must be included following the opcode. Using a procedure similar to that described above, a post byte of \$00 is obtained.
- (5) Since two bytes were consumed by the previous instruction (“LDAB”), the opcode of the next instruction – “MUL” – will be stored at location \$0807. The opcode for this single-byte instruction is \$12.
- (6) The next instruction, located at \$808, will store the double-byte result obtained in a pair of consecutive memory locations (in “big endian” format, i.e., *high order byte first*). Here, extended addressing mode is used; the symbol that identifies the storage location address is “prod” – again, a *forward reference*. We must therefore reserve two bytes following the “STD” opcode (\$7C) to store the 16-bit value associated with the symbol “prod”. Our symbol table is now as follows:

SYMBOL	VALUE
start	\$800
opands	<undefined>
prod	<undefined>

- (7) The final instruction, “STOP” (a two-byte instruction stored at locations \$80B and \$80C), halts the execution of the processor (we will use STOP instructions as a “placeholder” for setting breakpoints).

- (8) Following an “ORG” at location \$900 to establish the location of the operand and result data, the next line of assembly code uses the “FCB” (form constant byte) psuedo-op to *assign an initial value* to the memory location indicated *when the object code is downloaded into the target system’s memory*. The address at which this constant byte is to be stored is \$900. Also, since the label “opands” is associated with this statement, it is now assigned the value \$900. Our symbol table is correspondingly modified as follows:

SYMBOL	VALUE
start	\$800
opands	\$900
prod	<undefined>

- (9) On the next line line, an “FCB” pseudo-op is used to initialize the second operand, located at \$901.
- (10) At location \$902, an “RMB” (reserve memory block) pseudo-op is used to *reserve* two bytes of memory for storing the result. *Note that these locations are NOT initialized when the object file is downloaded into the target system’s memory*. Also, the symbol “prod” is encountered and assigned the value \$902. Our symbol table is correspondingly modified as follows:

SYMBOL	VALUE
start	\$800
opands	\$900
prod	\$902

- (11) The “END” pseudo-op indicates the source file is complete – *note that no object code is assembled for this statement!* We are not finished yet, however, because on the “first pass” we left some forward references undefined (in the “partially assembled” code). Thus, a “second pass” (i.e., second scan over the partially assembled source file) is necessary to “fill in” the previously undefined forward references.

Question: What if the symbol table still has some undefined entries at the end of the “first pass”? Will the “second pass” be successfully completed?

Answer: \_\_\_\_\_  
 \_\_\_\_\_

**Pre-lab Exercise 1:** The following program takes the BCD number located at the memory location “bcdin”, converts it to its binary equivalent, then stores the converted value at the memory location “binout”. For example, if the BCD number “10” was stored at “bcdin”, it would be converted to the binary number “00001010” (which equals 0Ah), and store it at “binout”. Hand-assemble this BCD-to-binary (bcdb) program and complete the symbol table.

Address	Object Code	Assembly Instructions
		bcd b org \$800 ; BCD to Binary Conversion Program
_____	_____	ldaa bcdin ; (A) ← packed BCD input data
_____	_____	anda #\$f0 ; mask off lower nibble
		; => (A) = 16 * (upper nibble)
_____	_____	lsra
_____	_____	tfr a,b ; (B) = 8 * (upper nibble)
_____	_____	lsra
_____	_____	lsra ; (A) = 2 * (upper nibble)
_____	_____	psha ; store temporarily on stack
_____	_____	addb 1,sp+ ; (B) = (2 + 8) * (upper nibble)
_____	_____	ldaa bcdin ; reload initial BCD data
_____	_____	anda #\$0f ; mask off upper nibble
_____	_____	psha ; store temporarily on stack
_____	_____	addb 1,sp+ ; (B) = 10 * (un) + (ln)
_____	_____	stab binout ; store converted binary value
_____	_____	stop
_____	_____	bcdin fcb \$34 ; BCD input data
_____	_____	binout rmb 1 ; location to store converted binary
_____	_____	end

*Symbol Table:*

### Hand Disassembly Example

As one might guess, program *disassembly* is just the inverse of program assembly – only *less* fun to do on a regular basis. Unfortunately, however, it is sometimes necessary when trying to debug a system based on a “memory dump.” For the purpose of illustration, the following example program (that performs BCD subtraction) will be disassembled step by step.

<i>Address</i>	<i>Contents</i>	<i>Disassembled Instruction</i>
0900	CE	ldx    #\$90C
0901	09	
0902	0C	
0903	89	adca    #0
0904	00	
0905	A0	suba    1,x+
0906	30	
0907	AB	adda    0,x
0908	00	
0909	18	daa
090A	07	
090B	18	stop
090C	3E	
090D	34	<i>data</i>
090E	49	<i>data</i>

### Step-By-Step Description of Disassembly For Example Program

- (1) Using the numerically-listed S12CPU Opcode Map (Page 28 of the *Guide* or Table A-2 of the *Manual*) as a reference, opcode (\$CE) – stored at location \$900 – is an “LDX” instruction using immediate addressing mode. The two data bytes, immediately following the opcode, are \$09 and \$0C.
- (2) The next opcode (\$89), stored at location \$903, is an “ADCA” instruction using immediate addressing mode. The single data byte, immediately following the opcode, is \$00.
- (3) The next opcode (\$A0), stored at location \$905, is a “SUBA” instruction using some form of indexed addressing mode. The post byte (\$30) following the opcode indicates the particular form of indexed addressing utilized; using page 21 of the *Guide* or Table A-3 of the *Manual* as a guide, the particular form of indexed addressing is found to be “auto post-increment by one” with “X” as the index register.
- (4) The next opcode (\$AB), stored at location \$907, is an “ADDA” instruction using some form of indexed addressing mode. The requisite post byte (here, \$00) is translated as in (3), above, into “zero offset indexed” with “X” as the index register.
- (5) The next opcode (\$18), stored at location \$909, references “page 2” of the Opcode Map; this tells us that more than one byte is used to represent the machine code for the instruction stored here. The second opcode byte (\$07), combined with the first, indicates that this is the “DAA” instruction.
- (6) The final opcode (\$183E), stored at locations \$90B and \$90C, is the “STOP” instruction.
- (7) The remaining two bytes, stored at locations \$90D and \$090E, represent data used by the program.

**Pre-lab Exercise 2.** Disassemble the HC(S)12 machine code listed below and “single step” through it by hand, completing the chart below. Write the disassembled instructions under the *Disassembled Instruction* heading, clearly indicating the instructions associated with the specific memory contents. Each “step” refers to the execution of one instruction. Assume the first opcode byte is at location \$900, and that all CCR bits are initially cleared.

er

<i>Address</i>	<i>Contents</i>	<i>Disassembled Instruction</i>
0900	86	
0901	17	
0902	8B	
0903	D3	
0904	18	
0905	07	
0906	C6	
0907	84	
0908	37	
0909	AB	
090A	80	
090B	18	
090C	07	
090D	86	
090E	19	
090F	A0	
0910	B0	
0911	18	
0912	07	
0913	18	
0914	3E	

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				

## 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**.




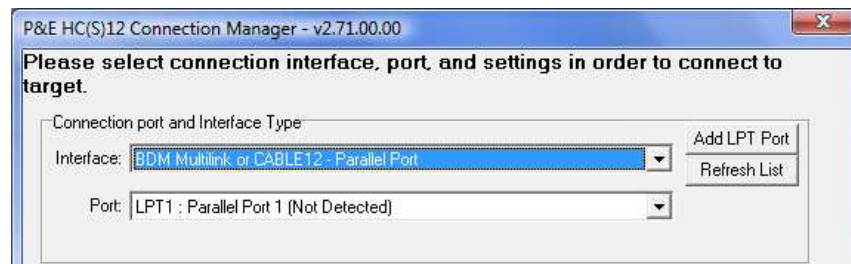
When the *New Project* window appears, select:

HCS12 > HCS12C Family > MC9S12C32

In the right window, select **P&E Multilink/Cyclone Pro**. When finished, press **Next**. Deselect the check by **C** and select **Absolute Assembly**. Press **Finish**.

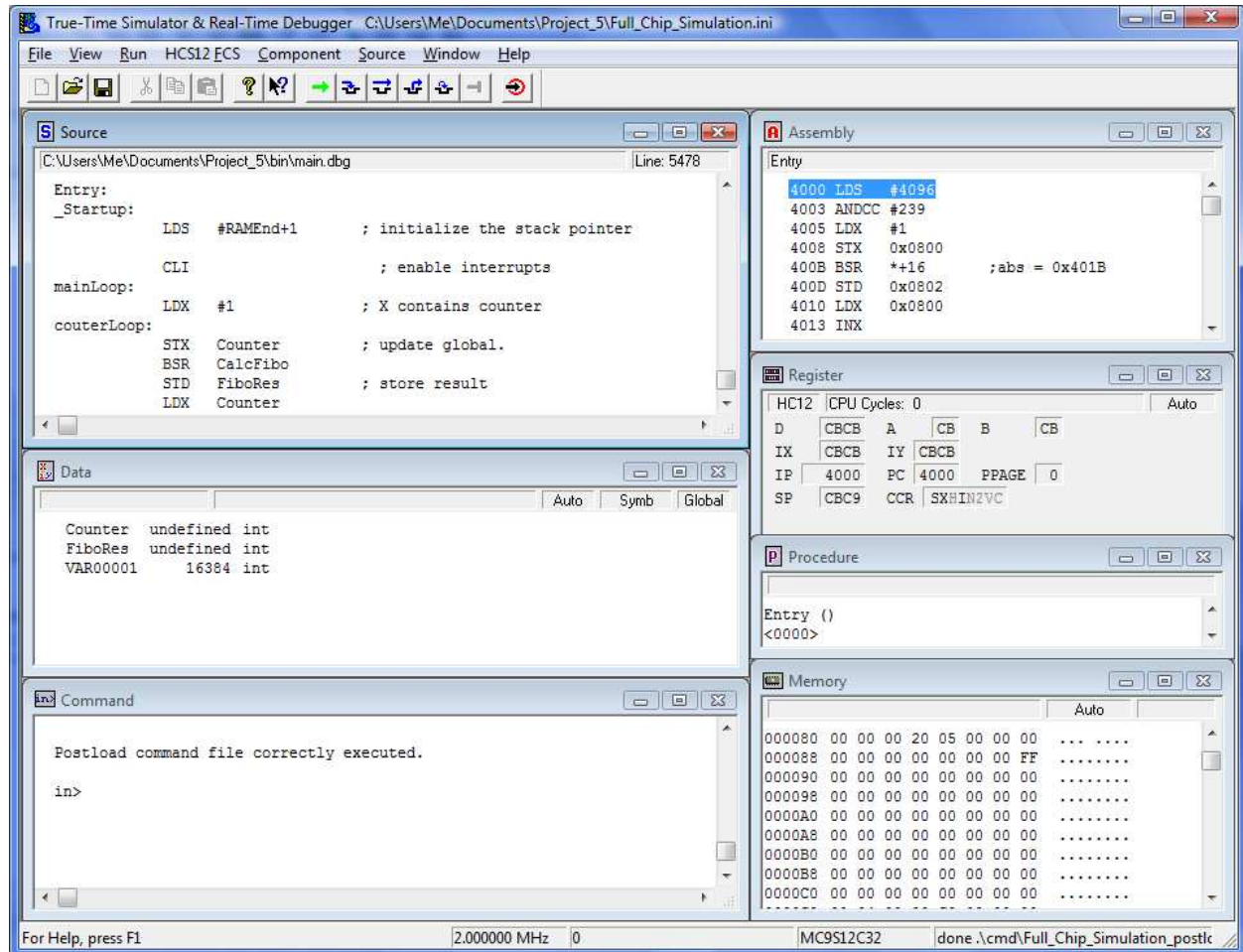
In the left frame, double-click **main.asm** to open the assembly source file. You will edit the contents of this file later, but for the purposes of this tutorial, you can leave the preloaded program as it is.

Now, click the **Debug** button (  ) to start the debugger. The window shown below will pop up. Make sure that the **Connection and Interface Type** drop-down box has **BDM Multilink or Cable 12 – Parallel Port** selected to connect to the BDM on the parallel port.



Click **Connect**. An update message may appear; wait until it is done, and it will proceed automatically.

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

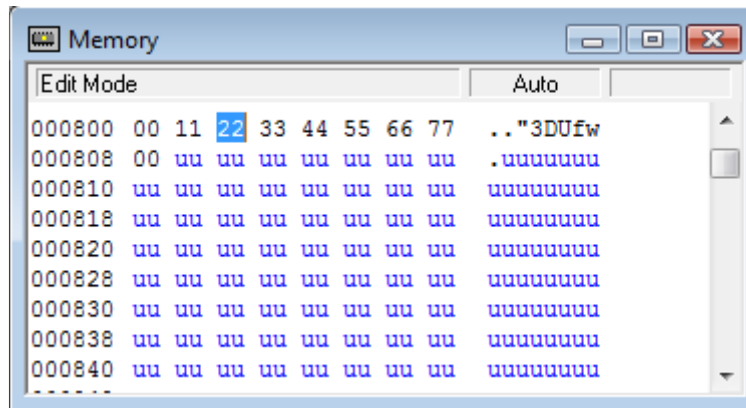


The **Source window** is where you can view the progress of your program's execution. Each time you **start** (→) or **single step** (⏏), you will notice that a new line will be highlighted in the source window. The highlighted line is the next instruction to be run.

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.

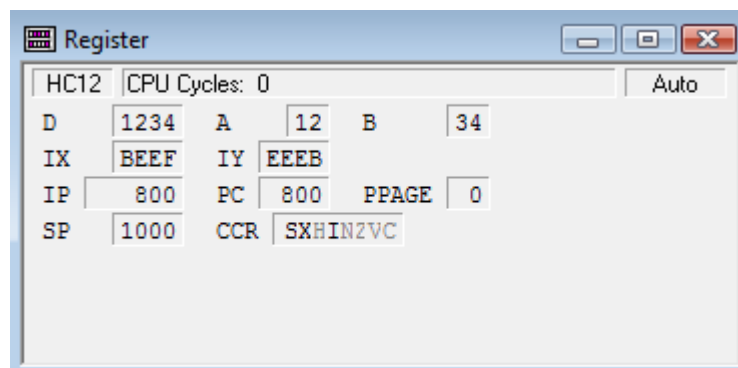
**Note:** Hexadecimal numbers will sometimes be notated by a "0x" or "\$" prefix or a "h" suffix. For example, you might see 20<sub>16</sub> written either as "0x20", "\$20" or "20h". It should also be noted that **CodeWarrior will only allow the "\$" notation for hex numbers in your code.**



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.)

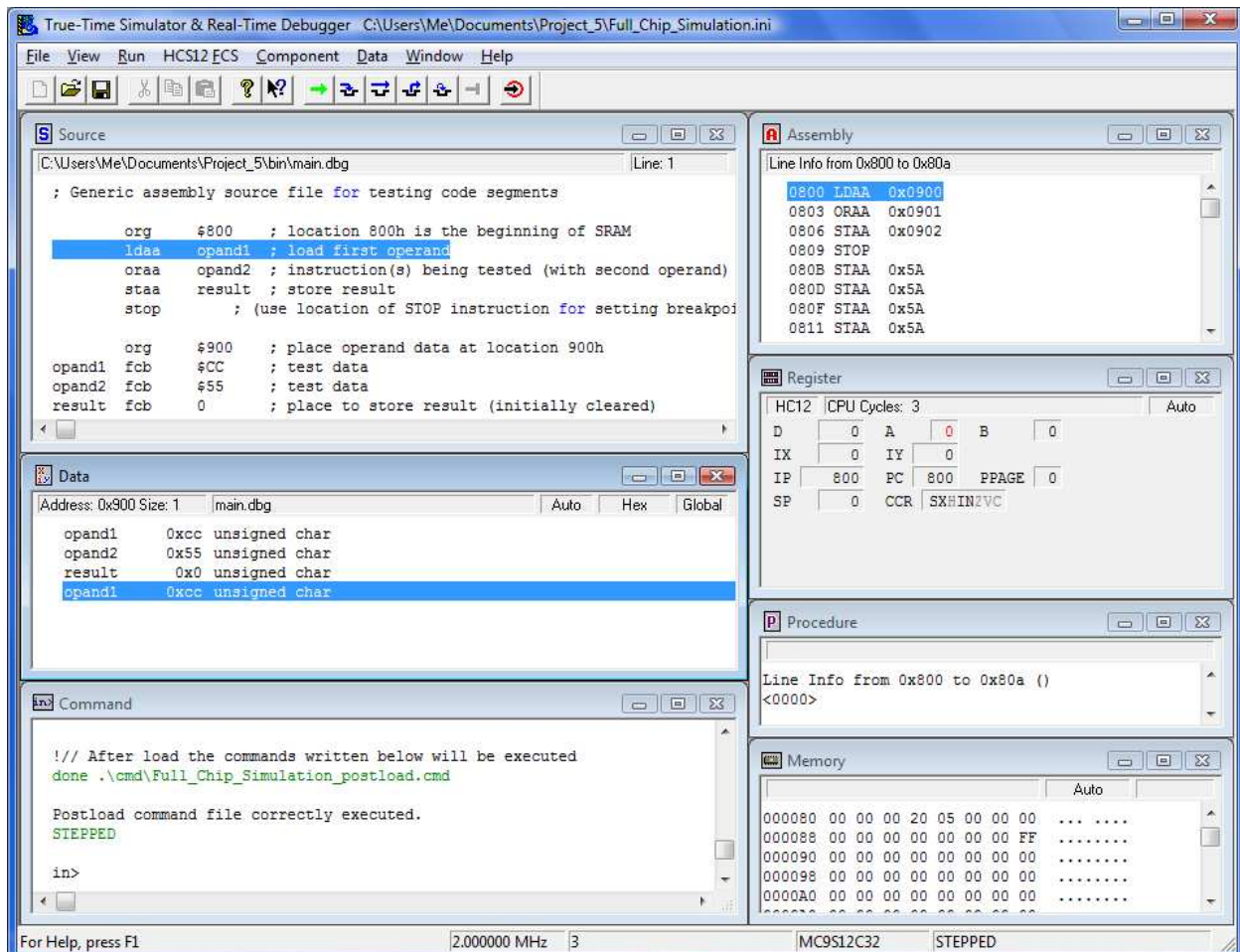
To examine the effect of executing code segments using the development environment in lab, use the following “generic” assembly source file for each test case:

```
; 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
```

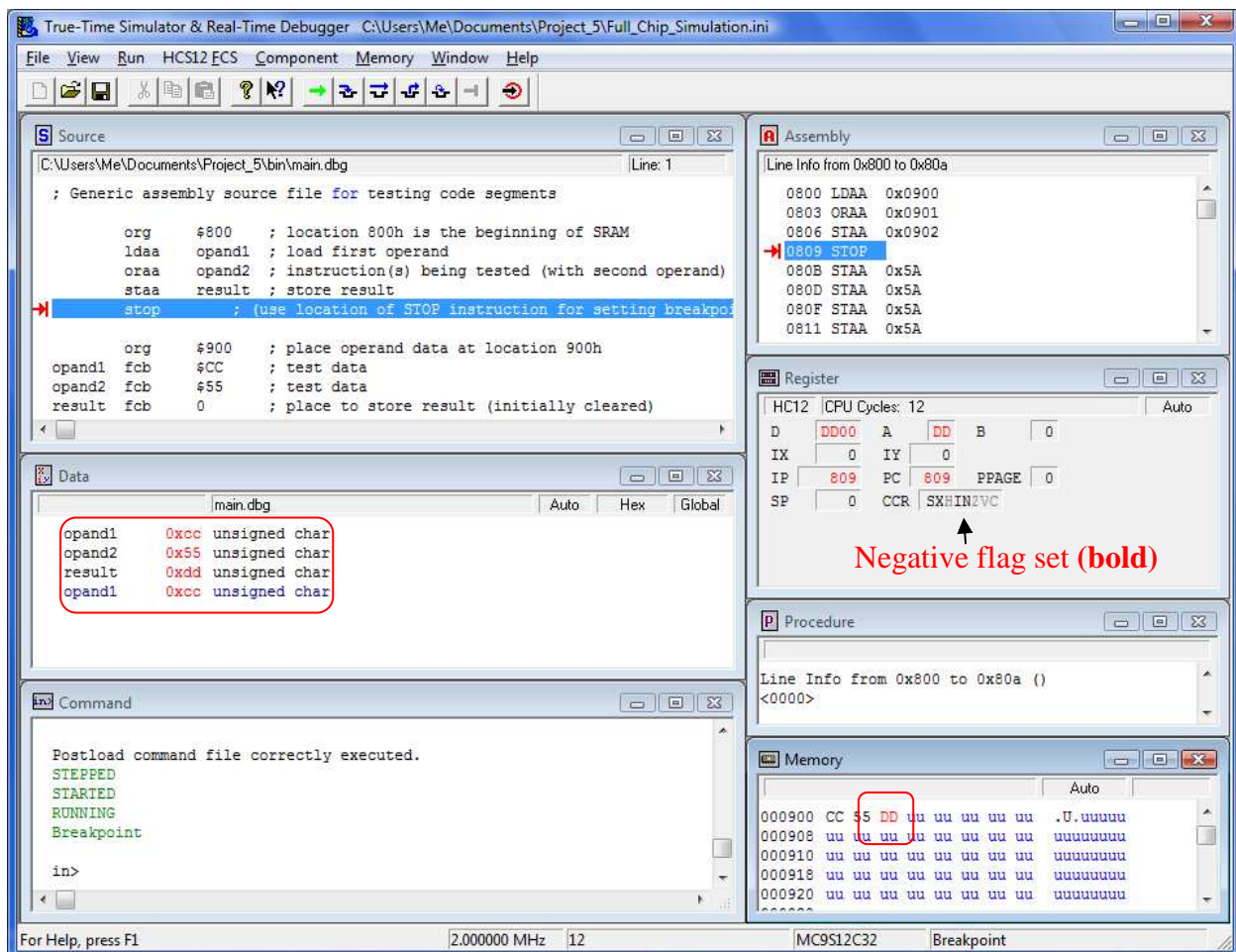
Loading this code into the debugger should yield the following (after setting PC to \$800 and clearing the other registers). If your code is not displayed in the source window after you change the PC, press the single step button, change the PC back to \$800 and re-clear the registers.





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
<b>ORAA \$900</b>  Effective Address: <b>\$900</b>	<b>BA 09 00</b>	Before execution: <b>(0900) = 55</b>	Before execution: <b>(A) = CC</b> <b>N = 0, Z = 1</b>
		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 on the Microcontroller Kit (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: <b>(0900) = 55</b>	After execution: <b>(A) = DD</b> <b>N = 1, Z = 0</b>

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 described under **Experimental Procedure** (available on the course website), 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 **Pre-lab Exercise 1** using the memory window in the debugger. Execute the program to verify its operation. Try two additional values for `bcdin` 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 **Pre-lab Exercise 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 Prelab Exercise 2.

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 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 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-Indirect 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-I:** “ADDA” instruction using Indexed-Indirect with Accumulator Offset addressing mode.

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