# Laboratory 1

# Intro to CodeWarrior

**Concepts:**

* Debugging and simulating with CodeWarrior
* Basic assembly program structure

**Objectives:**

* Simulate the 9S12 microcontroller using CodeWarrior
* Use trace and breakpoints to monitor a program’s execution.
* Read and modify memory and registers during debugging.

**Files Needed:**

* Lab\_1.zip from Blackboard

**Introduction:**

This week’s lab is intended to familiarize you with the CodeWarrior IDE. The assignment in the following section uses CodeWarrior to examine some existing programs. It will demonstrate techniques for debugging programs and help you gain some familiarity with the processor.

**Procedure:**

1. Copy Lab01.zip from Blackboard to your USB Flash drive and unzip the project.
2. Open CodeWarrior by clicking on the desktop icon, and open the lab01.mcp file in the folder you just unzipped (*mcp* is the extension for a CodeWarrior project).

You should now see a file hierarchy under a tab labeled Lab\_1.mcp.

Main.asm contains the assembly code. In today’s lab, the program has already been written. In future labs, this is where your code will go.

Mc9s12dg256.inc contains declarations for all the named bits and control registers in the processor. We won’t use them yet, but we will reference this file quite a bit when we start working with physical I/O and peripheral devices.

Project.prm tells the assembler where different memory types are located for this particular chip’s memory map. Open this file. Notice that the RAM segment is listed as READ\_WRITE memory from 0x1000 to 0x2FFF. The assembler will automatically put internal variables into this RAM space. The range 0x3000 to 0x3FFF is unaccounted for in this file. We will manually place things in this range in our assembly code throughout the term, and the assembler will avoid it so that multiple items don’t get allocated to the same memory address.

Also note that ROM\_C000 is specified as READ\_ONLY. The memory at this range is Flash. As far as an executing microprocessor program is concerned, this space contains only program code and constants and cannot be written to. Special steps are required to write to Flash, so it cannot be used for variables using *store* instructions like RAM.

Near the bottom of the file, notice the “VECTOR 0 Entry” line. This configures the microprocessor to start executing at the address corresponding to *Entry* when the chip is powered up or reset.

1. Open main.asm and read through it.

You should see a brief header comment section, followed by an “;export symbols” section needed by the assembler. Next, there is an “ORG $3000” section. ORG tells the assembler to start placing items in memory starting at that address. Later, we will use this section to specify program inputs and outputs, since we have control over which addresses are used.

The “MY\_EXTENDED\_RAM: SECTION” will be used for internal variables (those that the program uses for storage, but the user never needs to look at). The assembler will find RAM for items in this section, but we are not concerned about which specific addresses are used. Do not put internal variables in this section.

The “MyCode: SECTION” is where the instructions are. There is a small header that you should not modify. As part of this header, the stack pointer register is initialized. Although we won’t use the SP in this program, it appears in the header so this file can be used as a template, but it would need to be uncommented.

1. In the Lab\_1.mcp tab, make sure “Full Chip Simulation” is selected from the drop-down list, shown in Figure 1. Then, click on the “debug” button, shown with a green arrow and a tiny bug. This should assemble the program and open a second window. When using a development board, CodeWarrior will also download the machine code into the Dragon12+ board.

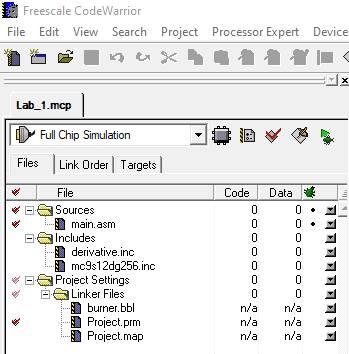


Figure 1

1. One of the windows in the True-Time Simulator and Real Time Debugger, which we’ll call just the debugger, is the Memory window. Go to memory address 3000h by right-clicking in the window, and selecting “Address..”. As shown in Figure 2, you will get a pop-up box that allows you to type in an address, shown below.

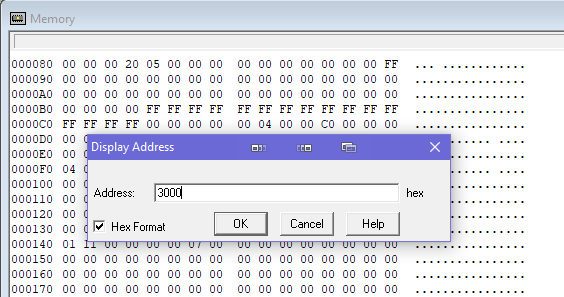


Figure 2

Once you click on OK, you’ll see a section of memory with all dashes, shown in Figure 3.

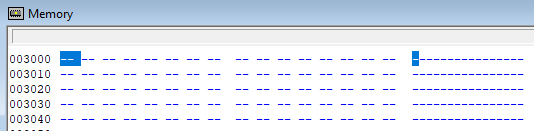


Figure 3

Note that the leftmost location is 3000h. The second location from the left is 3001h, and so on.

The dash in memory means that memory does not exist at that address. Reads and writes to memory locations that don’t exist can be configured to stop the simulation with errors. Let’s demonstrate this. First, notice that the LDAA 3000h line is highlighted in the Source window shown in Figure 4.

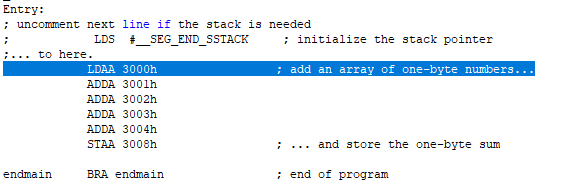


Figure 4

The highlighted line is the next one that will execute (this is sometimes not true when branching is involved.) Run the program by clicking on the green arrow button at the top of the window shown in Figure 5, which runs the program until a breakpoint is or until a halt instruction is reached. Our program has an infinite loop at the bottom. If it runs correctly, we should manually have to stop the execution.

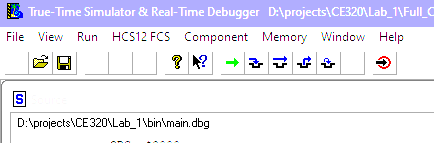


Figure 5

Instead of executing continuously, the simulation halts after one line with the error “No memory at (3000:1)”. There are several ways to fix this:

* The simulator can create memory when the microcontroller first writes to it.
* The code can declare space at that address, and the simulator will honor the allocation.
* The memory zone can be manually declared.

1. We will use the second method, so that we can manually enter some values in memory for the processor to read (i.e. it won’t have to write to the addresses first). In the CodeWarrior window with *main.asm*, add the “ds.b 16” line in the ORG $3000 section. This says that we want 16 bytes worth of space. Then, click the debug button. The debugger will refresh.

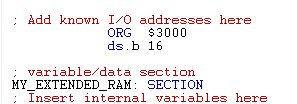


Figure 6

Resetting the debugger also resets the Memory window, so again go to memory address 3000h. You will now see sixteen bytes with “uu” in them. This means that memory locations exist, but they have not been written to yet, so the contents are undefined. The processor can now read and write without errors, and we can manually change them by double-clicking on the location.

Also note that pressing “enter” changes the value and ends the data entry mode, but pressing the spacebar moves the cursor to the next location for faster entry.

Enter the six values from

1. Table 1 into memory at the addresses shown.

|  |  |
| --- | --- |
| **Memory Address** | **Contents** |
| 3000h | 3Ah |
| 3001h | 5Eh |
| 3002h | FEh |
| 3003h | 8Bh |
| 3004h | 56h |
| 3008h | 00h |

Table 1. Initial Data (Note: addresses are not consecutive)

Figure 7 shows the assembly language program from *main.asm*. It adds a list of five one-byte numbers beginning at address 3000h, and it stores the one-byte result in address 3008h.

1. ***Before*** running the program, add a breakpoint at the “endmain BRA endmain” instruction by right-clicking on the instruction and selecting “Set Breakpoint”. Notice that the breakpoint appears in both the Source and Assembly panes.

|  |  |  |  |
| --- | --- | --- | --- |
| **Code Line** | **Memory Address** | **Contents** | **Assembly Code** |
| 1: | C000h | B6 | LDAA 0x3000 |
| C001h | 30 |
| C002h | 00 |
| 2: | C003h | BB | ADDA 0x3001 |
| C004h | 30 |
| C005h | 01 |
| 3: | C006h | BB | ADDA 0x3002 |
| C007h | 30 |
| C008h | 02 |
| 4: | C009h | BB | ADDA 0x3003 |
| C00Ah | 30 |
| C00Bh | 03 |
| 5: | C00Ch | BB | ADDA 0x3004 |
| C00Dh | 30 |
| C00Eh | 04 |
| 6: | C00Fh | 7A | STAA 0x3008 |
| C010h | 30 |
| C011h | 08 |
| 7: | C012h | 20 | BRA endmain |
| C013h | FE |

Figure 7 – Sequential Program to Add Five Numbers

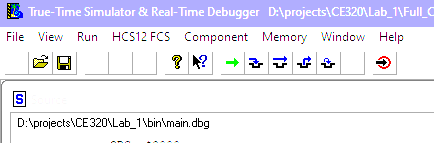
The Assembly Pain may hide the machine code by default. Right-click in the pane and select “Display”, then “Code”, and you should see the contents above. Note that the figure shows the byte-by-byte storage of the instruction vs the instruction-by-instruction format in the Assembly window. Finally, just to show that consistency is not important to some people, the Assembly window displays hexadecimal using a *0x* prefix instead of a *$* prefix or *h* suffix. You CANNOT use the *0x* notation when writing assembly.

1. Click on the green arrow button to run the program. The debugger should halt with the “endmain” line highlighted.

**Question 1**: What answer does the program calculate?

**Question 2**: If the numbers were added as unsigned bytes and the number of digits in the answer was not limited, what is the correct sum of the five numbers?

1. Reset the S12 to the beginning of the program by clicking the “Reset Target” button. Note that this DOES NOT reset memory addresses, so you may need to restore some of them according to the table above.
2. Fill in the values for Figure 8 by tracing through the program instruction by instruction. The command for this is the “Single Step” button, or press F11.



Each line in the figure should be completed with the values ***after*** the line has been executed. The first row shows the initial value of the PC (set to the first instruction’s address), with all other registers unknown since the program has not affected them yet. *Note that the CCR bits are black if they are 1, and they are light gray if they are 0.*

**Question 3**: The running sum in accumulator A may experience overflow. Which trace lines show unsigned overflow after adding a value to A?

**Question 4**: Which trace lines contain signed overflow after adding a number to A?

**Question 5**: Which condition code register bits correspond to the unsigned overflows and signed overflows in the lines from Questions 3 and 4?

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **Trace**  **Line** | **Code**  **Line** | **PC** | **A** | **N** | **Z** | **V** | **C** |
| 0 | - | C000 | - | - | - | - | - |
| 1 |  |  |  |  |  |  |  |
| 2 |  |  |  |  |  |  |  |
| 3 |  |  |  |  |  |  |  |
| 4 |  |  |  |  |  |  |  |
| 5 |  |  |  |  |  |  |  |
| 6 |  |  |  |  |  |  |  |
| 7 |  |  |  |  |  |  |  |

Figure 8 - Program Trace

Figure 9 shows an assembly language program. This program adds five numbers like the first program, but it uses a loop instead.

1. Close the debugger. Comment out the six lines of code from the first program, and enter the assembly code for the second program from Figure 9.

Run the debugger, and then reenter the values from

1. Table 1 into memory.

**Question 6:** Using the Assembly window, complete the Machine Code column from Figure 9.

1. Run the new program from the beginning, remembering to set a breakpoint at the “endmain BRA endmain” line.
2. Verify that the second program generates the same answer in 3008h as the first. If it doesn’t, fix it before you move on.
3. Reset the simulator to the beginning of the program, and complete the program trace in Figure 10 for the first 15 executed instructions. (It will not reach the end.)
4. Reset the processor to the beginning of the second program again and add a breakpoint to line C006h.
5. Run the program from the beginning. The address displayed in the PC should be C006h when the processor halts. Note that when the processor stops at a breakpoint, the instruction that the PC points to has not yet been executed.
6. A breakpoint trace is like a program trace, except that instead of recording the register values after each line of code is executed, we will record the register values when the processor stops at the breakpoint. Complete the breakpoint trace in Figure 11, given that you are currently at iteration 0, and the first time that the program returns to line C006h is considered iteration 1. Clicking on the “Start/Continue” button (i.e. the green arrow button) will resume program execution and halt at the next breakpoint.

|  |  |  |  |
| --- | --- | --- | --- |
| **Code Line** | **Memory Address** | **Machine Code** | **Assembly** |
| 1: | C000h |  | CLRA |
| 2: | C001h |  | LDX #3000h |
| C002h |  |
| C003h |  |
| 3: | C004h |  | LDAB #5 |
| C005h |  |
| 4: | C006h |  | BEQ \*+8 |
| C007h |  |
| 5: | C008h |  | ADDA 0,X |
| C009h |  |
| 6: | C00Ah |  | INX |
| 7: | C00Bh |  | DECB |
| 8: | C00Ch |  | BRA \*-6 |
| C00Dh |  |
| 9: | C00Eh |  | STAA 3,X |
| C00Fh |  |
| 10: | C010h |  | BRA \* |
| C011h |  |

Figure 9 – Looping Program to Add Five Numbers

**Question 7**: Which lines of code in Figure 9 must be changed to write the sum to memory location 3500h?

**Question 8**: Which lines of code in Figure 9 must be changed to add numbers beginning at memory location 3200h?

**Question 9**: Which lines of code in Figure 9 must be changed to add 10010 numbers instead of 5?

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Trace**  **Line** | **Code**  **Line** | **PC** | **IX** | **A** | **B** | **N** | **Z** | **V** | **C** |
| 1 |  |  |  |  |  |  |  |  |  |
| 2 |  |  |  |  |  |  |  |  |  |
| 3 |  |  |  |  |  |  |  |  |  |
| 4 |  |  |  |  |  |  |  |  |  |
| 5 |  |  |  |  |  |  |  |  |  |
| 6 |  |  |  |  |  |  |  |  |  |
| 7 |  |  |  |  |  |  |  |  |  |
| 8 |  |  |  |  |  |  |  |  |  |
| 9 |  |  |  |  |  |  |  |  |  |
| 10 |  |  |  |  |  |  |  |  |  |
| 11 |  |  |  |  |  |  |  |  |  |
| 12 |  |  |  |  |  |  |  |  |  |
| 13 |  |  |  |  |  |  |  |  |  |
| 14 |  |  |  |  |  |  |  |  |  |
| 15 |  |  |  |  |  |  |  |  |  |

Figure 10 – Looping Program Trace

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Iteration** | **PC** | **IX** | **A** | **B** | **N** | **Z** | **V** | **C** |
| 0 |  |  |  |  |  |  |  |  |
| 1 |  |  |  |  |  |  |  |  |
| 2 |  |  |  |  |  |  |  |  |
| 3 |  |  |  |  |  |  |  |  |
| 4 |  |  |  |  |  |  |  |  |
| 5 |  |  |  |  |  |  |  |  |

Figure 11 - Breakpoint Trace

**Deliverables/Scoring:**

* 20 points - Compliance with posted lab report guidelines.
* 20 points - Answers to questions.
* 20 points - Completed Figure 8 – Program trace
* 20 points - Completed Figure 10 – Program trace
* 20 points - Completed Figure 11– Breakpoint trace

Submit the deliverables according to the lab report guidelines posted on Blackboard. Note that since you did not write any unique code, there is no code submission for this lab.