University of Victoria

Department of Electrical & Computer Engineering

CENG 255 - Introduction to Computer Architecture

Laboratory Manual

Laboratory Experiment #1

By

Farshad Khunjush, Nikitas J. Dimopoulos, and Kin F. Li

The laboratory experiments are developed to provide a hands-on introduction to the ColdFire architecture. The labs are based on CodeWarrior Development Studio, an integrated development environment (IDE) for embedded systems.

You are expected to read this manual carefully and prepare **in advance** of your lab session. Pay attention to the parts that are **<u>bolded and underlined</u>**. You are required to address these parts in your lab report. In addition, be prepared to answer the questions listed in the **Prelab** section. Your lab instructor will ask you these questions during the lab. Your answers to these questions and your preparation for the lab will be graded.

Laboratory 1: Using the CodeWarrior Integrated Development Environment

1.1 Goal
In this introductory lab, you will learn the procedure for developing assembly language programs and the basic features of the ColdFire architecture.

1.2 Objectives
Upon completion of this lab, you will be able to:
- Assemble and link ColdFire architecture assembly programs.
- Load program onto a ColdFire development board.
- Run programs.
- Examine / Modify memory locations (including program and data) and registers.
- Set breakpoints and single step execution.

1.3 Prelab (Your Lab Instructor will ask you these questions during the lab and your answers will be graded.)
- **What is a cross-assembler?**
- **What is the difference between big-endian and little-endian?**
- **What is an exception?**
- **Comment on the program in 1.4.2 Part 2 (Explain what it does).**
- **Comment on the program in 1.4.3 Part 3 (Explain what it does).**

1.4 Hardware Introduction
The **ColdFire M52233 DEMO** board consists of a **ColdFire** MCF52233 processor, a ROM that stores the monitor program, 256 KB Flash, 32KB SRAM, 3 UARTS, 32-bit Timers, a general purpose timer, 4 LEDS, and 3 push-buttons. The CPU clock rate for this board is 60 MHz, and the user programming model is the same as the M68000 family microprocessors.

For this lab, we employ **CodeWarrior** Development Studio (CodeWarrior IDE 5.7) for ColdFire V6.3.

1.4.1 Part 1
1. Login to one of the lab computers.

2. On the desktop you will find a shortcut to **CodeWarrior** Development Studio. Open this application.

3. Next, we use **CodeWarrior** to assemble a simple program. The Lab Instructor will provide you with an URL where you can download a folder containing the project **BareAsm_0**. This project contains the assembly code along with the supporting files for this part of the lab.

Unzip the folder on the M: drive. When you have unzipped the folder, you need to assemble and link your source program using **CodeWarrior,** before running it on the **ColdFire** board. The assembler translates assembly language programs into object programs. Then, the linker transforms object programs into executable programs that you can run on **ColdFire**.

The following shows the contents of the **main.s** file in the **BareAsm_0** project.

```
# This program stores the result of "3 + 4" into register d4.

/*
 * File: main.s
 */
         .text

         .global  _main
_main:
         link     a6,#0
         lea.l    -12(a7),a7

         move.l   #3,d3
         move.l   #4,d4
         add.l    d3,d4
loop:    bra      loop
         unlk a6
         rts
.end
```

Figure 1. Assembly Code for Lab 1.4.1

4. To assemble the **main.s** program, use the following command from the main menu:

Click Project -> Compile

This operation generates the object file for the main.s file. Use the following command from the main menu to observe the object file:

Click Project -> Disassemble

This command shows the object file for main.s as shown below.

```
             *** ELF HEADER ***

ident[EI_CLASS] = 1
ident[EI_DATA]  = 2
ident[EI_VERS]  = 1
type            = 1
machine         = 04 (EM_68K)
version         = 1
entry           = 0x00000000
phoff           = 0x00000000
shoff           = 0x0000039C
flags           = 0x00000000 ( )
ehsize          = 52
```

```
phentsize       = 0
phnum           = 0
shentsize       = 40
shnum           = 12
shstrndx        = 1
```

*** SECTION HEADER TABLE ***

no	offset / link	size / info	flags / addralign	addr / entsize	type	name
1	0x00000224 0	0x0000005F 0	0x00000000 1	0x00000000 0	STRTAB	.shstrtab
2	0x00000284 0	0x0000004F 0	0x00000000 1	0x00000000 0	STRTAB	.strtab
3	0x000002D4 2	0x00000080 7	0x00000000 4	0x00000000 16	SYMTAB	.symtab
4	0x00000034 0	0x0000009E 0	0x00000000 0	0x00000000 0	PROGBITS	.debug
5	0x00000354 3	0x0000000C 4	0x00000000 4	0x00000000 12	RELA	.rela.debug
6	0x000000D4 0	0x0000009E 0	0x00000000 0	0x00000000 0	PROGBITS	.debug
7	0x00000174 0	0x0000002F 0	0x00000000 0	0x00000000 0	PROGBITS	.debug
8	0x00000360 3	0x00000030 7	0x00000000 4	0x00000000 12	RELA	.rela.debug
9	0x000001A4 0	0x00000062 0	0x00000000 0	0x00000000 0	PROGBITS	.line
10	0x00000390 3	0x0000000C 9	0x00000000 4	0x00000000 12	RELA	.rela.line
11	0x00000208 0	0x0000001C 0	0x00000006 4	0x00000000 0	PROGBITS	.text

*** SYMBOL TABLE (.symtab) ***

no	value	size	bind	type	other	shndx	name
1	0x00000000	0x0000009E	LOCAL	SECTION	0x00	.debug	.debug
2	0x00000000	0x0000009E	LOCAL	SECTION	0x00	.debug	.debug.empty
3	0x00000000	0x00000062	LOCAL	SECTION	0x00	.line	.line._@DummyFn1
4	0x00000000	0x0000002F	LOCAL	SECTION	0x00	.debug	.debug._@DummyFn1
5	0x00000000	0x0000001C	LOCAL	SECTION	0x00	.text	.text
6	0x00000000	0x0000001C	LOCAL	FUNC	0x00	.text	_@DummyFn1
7	0x00000000	0x00000000	GLOBAL	NOTYPE	0x00	.text	_main

*** STRING TABLE (.shstrtab) ***

```
0x00000000:
0x00000001:  .shstrtab
```

```
0x0000000B:  .strtab
0x00000013:  .symtab
0x0000001B:  .debug
0x00000022:  .rela.debug
0x0000002E:  .debug
0x00000035:  .debug
0x0000003C:  .rela.debug
0x00000048:  .line
0x0000004E:  .rela.line
0x00000059:  .text

             *** STRING TABLE (.strtab) ***

0x00000000:
0x00000001:  .debug
0x00000008:  .debug.empty
0x00000015:  .line._@DummyFn1
0x00000026:  .debug._@DummyFn1
0x00000038:  .text
0x0000003E:  _main
0x00000044:  _@DummyFn1


             *** EXECUTABLE CODE (.text) ***

Address     ObjectCode        Label  Opcode   Operands              Comment

;
;    5:  .text
;    6:
;    7:  .global      _main
;    8: _main:
;    9:           link     a6,#0
;

0x00000000                    _main:
;                             main:
0x00000000                    _@DummyFn1:
;                             @DummyFn1:
0x00000000  0x4E560000               link     a6,#0
;
;   10:       lea.l    -12(a7),a7
;   11:
;
0x00000004  0x4FEFFFF4               lea      -12(a7),a7
;
;   12:       move.l   #3,d3
;
0x00000008  0x263C00000003           move.l   #3,d3                 ; '....'
;
;   13:       move.l   #4,d4
;
0x0000000E  0x283C00000004           move.l   #4,d4                 ; '....'
;
;   14:       add.l    d3,d4
;
0x00000014  0xD883                   add.l    d3,d4
;
;   15: loop: bra      loop
;   16:
;
0x00000016  0x60FE                   bra.s    *+0                   ;0x00000016
;
```

```
;   17:       unlk a6
;
0x00000018  0x4E5E                 unlk     a6
0x0000001A  0x4E75                 rts
```

Figure 2. Listing (object file) for main.s

Figure 2 shows the ColdFire machine language of the main.s assembly file. In this part, search for **`*** EXECUTABLE CODE (.text) ***`** string. At this location you can observe the machine language.

The first column shows the absolute address of each translated instruction. It should be noted that this address is different from the final location where the program will be loaded. The second column shows the translated code of each instruction. Finally, the next columns represent the original assembly codes of the translated program.

As can be seen, this program has only one section, which is the program section. It starts with the ***".text"*** keyword. It should be noted that the sizes of instructions in the ColdFire architecture are not identical.

It is worth mentioning that the ColdFire architecture is a big-endian architecture. As can be seen, each instruction is converted into its corresponding object code. For example, the instruction ***add.l d2,d3*** is translated to ***0xD883***. You can find more details about assembly language and different addressing modes in the ColdFire Family Programmer's Reference Manual.

5. In this section, we build the executable file using the following command:

Click Project -> Make

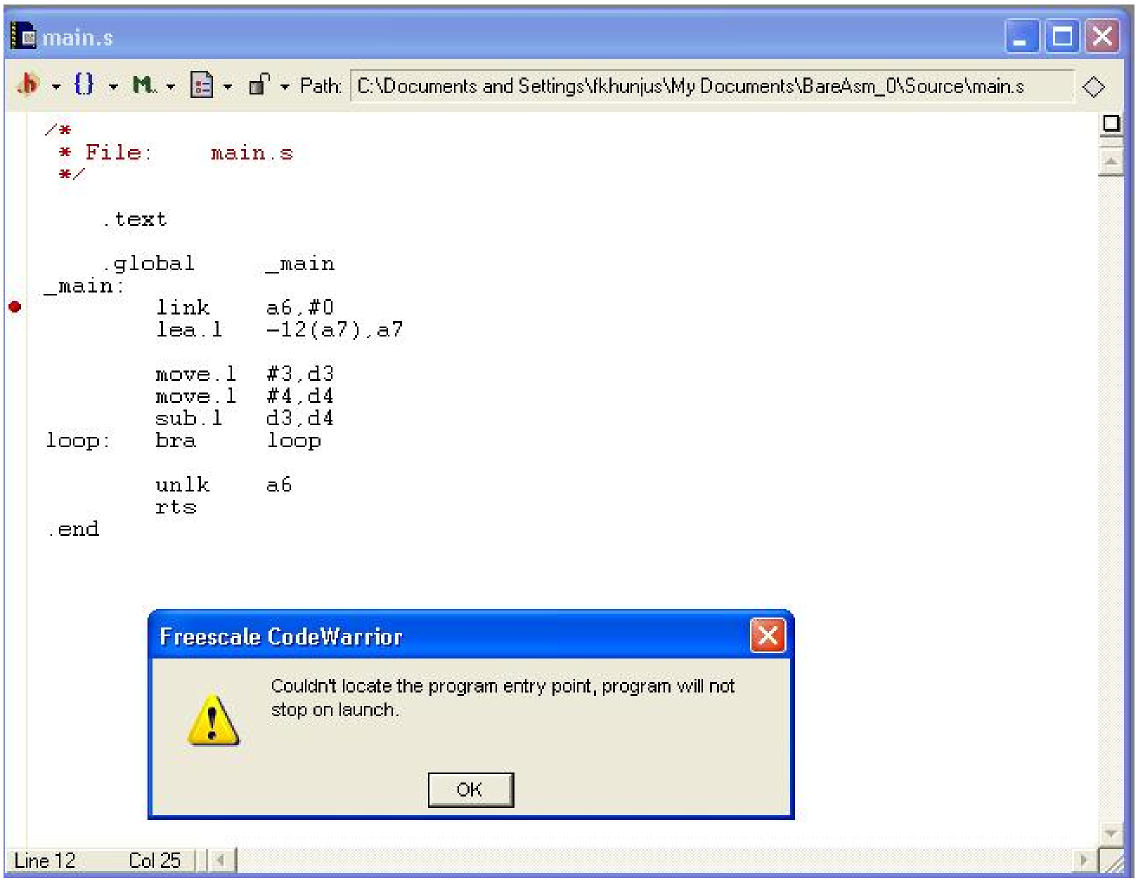
This command also writes the address mapping of the program segments into the ***M52233DEMO Console Debug.elf*** file in the **bin** folder. The filename is an acronym for Executable Linking and Formatting and is the file format that is used in the GNU Linux/UNIX world for object files and executables (it is used in more than just GNU Linux/UNIX but it originated in the UNIX world). It is also the format that the CodeWarrior IDE uses to store the code that is to be downloaded and debugged on the M52233DEMO board. Use a text editor (e.g., TextPad, etc.) and check the address mapping of each section. In addition, the final program in which the addresses are resolved can be found in the ***M52233DEMO Console Debug.elf.S19*** file in the **bin** folder.

6. Once you have assembled and linked your assembly language program, you can execute it through **CodeWarrior Development Studio**. Before execution, the executable program has to be loaded onto the board.

Ensure that the CodeFire board is properly connected to the workstation and is powered on.

- Go to the first line of the program (**link a6,#0**).
- Right click on the mouse and select the **set breakpoint** option.
- **Click Project -> Debug**

You will see the following warning for running the program; press OK and continue to the next step.



Now the program has been loaded into the memory and can be executed. As shown in Figure 3, the blue arrow points to the starting location of the loaded program. At the bottom of the page you can find Line and Col Number of the current line. Beside these numbers you will find a pop-up window which allows you to select the representation style of the program. The normal representation is the source program that you wrote. At this point, try to view other representations. For example, the assembly representation illustrates the final machine code and the assembly language of the program. It shows the program's starting location (***$20000500***) in the **M52233Demo** board's memory. In addition, all addresses are resolved in this representation. Discuss this representation with your Lab Instructor.

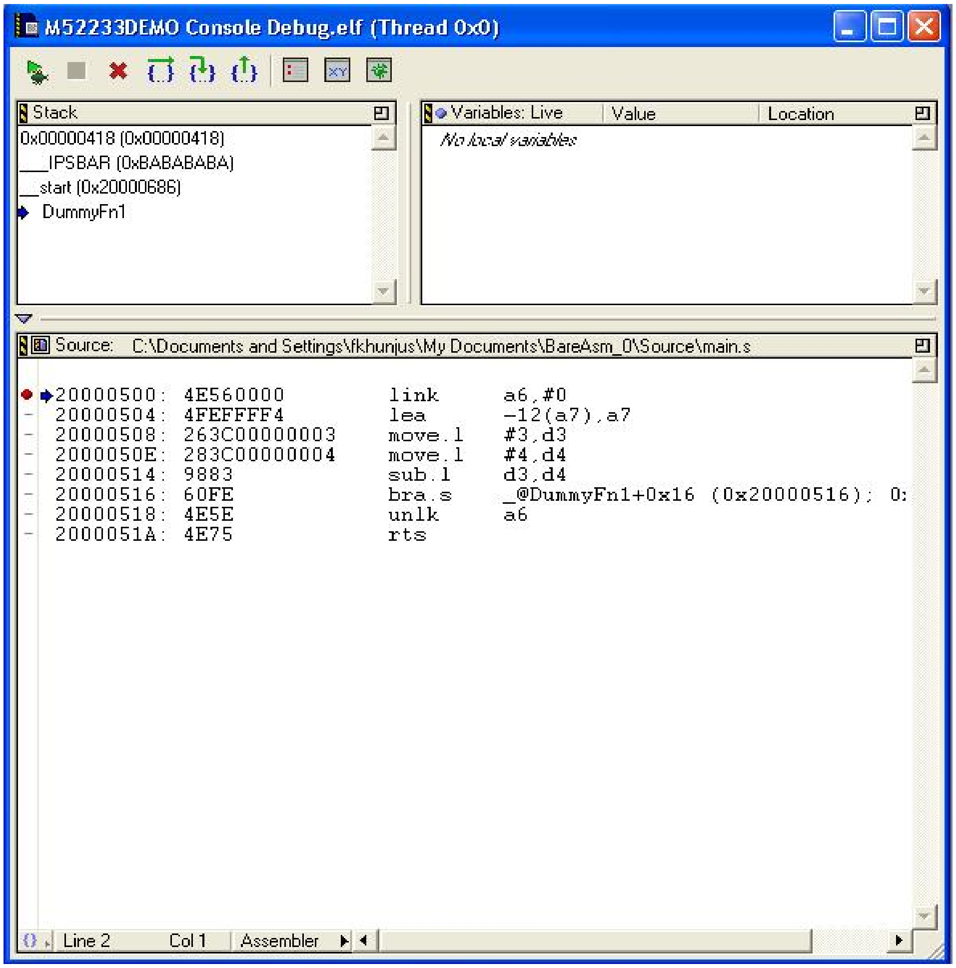
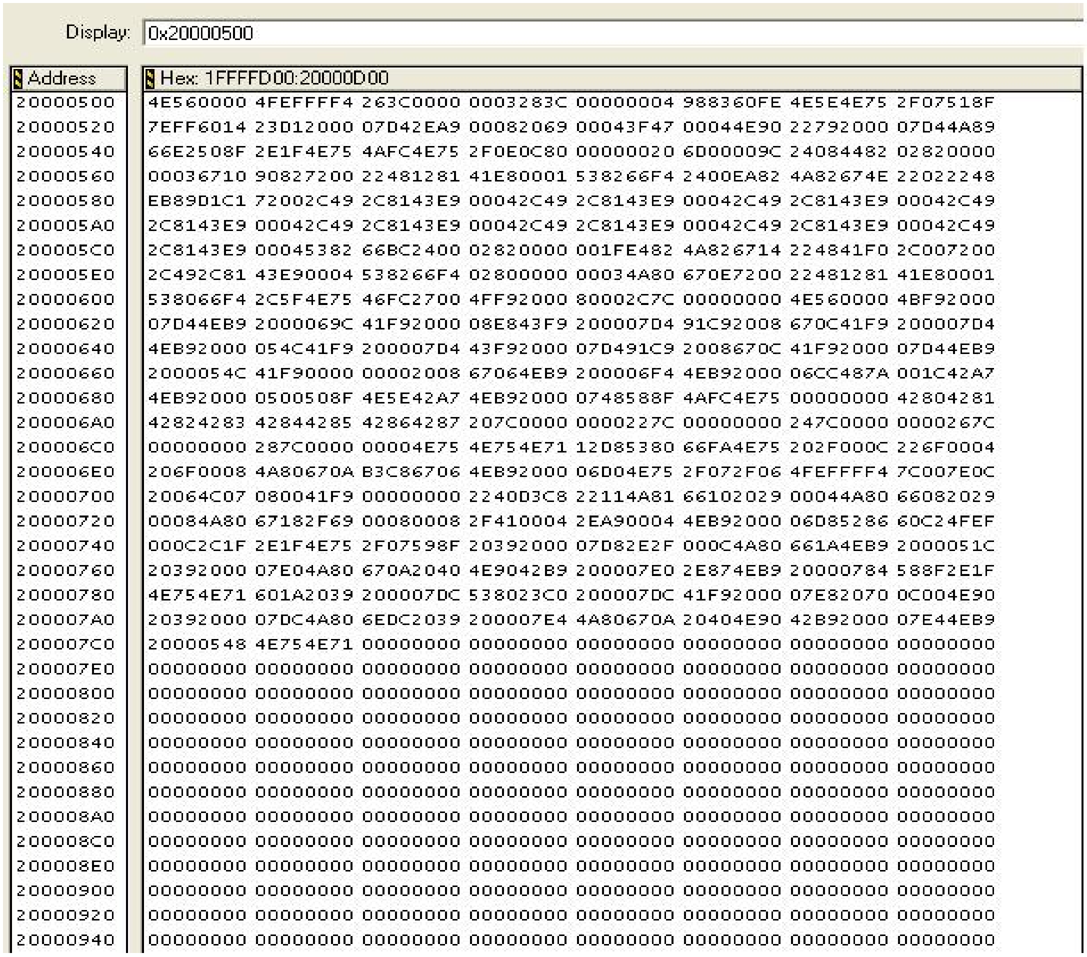


Figure 3. Downloaded Program

Before running the program, examine the contents of the memory and the registers.

Figure 4 shows the content of the memory after loading the program onto the board. As mentioned before, the ***M52233DEMO Console Debug.elf*** file in the **bin** folder describes the memory mapping of the program after the linking process. As can be seen from the file's contents, the code section starts at the location with address **0x20000500**. Therefore, we expect the first byte of the program is located in address **0x20000500.** To verify this, use the **Data -> View *Memory*** option shown in the menu. If you click on this option, you will see Figure 4 where the contents of the memory are displayed.



Display: 0x20000500

Address	Hex: 1FFFFD00:20000D00							
20000500	4E560000	4FEFFFF4	263C0000	0003283C	00000004	988360FE	4E5E4E75	2F07518F
20000520	7EFF6014	23D12000	07D42EA9	00082069	00043F47	00044E90	22792000	07D44A89
20000540	66E2508F	2E1F4E75	4AFC4E75	2F0E0C80	00000020	6D00009C	24084482	02820000
20000560	00036710	90827200	22481281	41E80001	538266F4	2400EA82	4A82674E	22022248
20000580	EB89D1C1	72002C49	2C8143E9	00042C49	2C8143E9	00042C49	2C8143E9	00042C49
200005A0	2C8143E9	00042C49	2C8143E9	00042C49	2C8143E9	00042C49	2C8143E9	00042C49
200005C0	2C8143E9	00045382	66BC2400	02820000	001FE482	4A826714	224841F0	2C007200
200005E0	2C492C81	43E90004	538266F4	02800000	00034A80	670E7200	22481281	41E80001
20000600	538066F4	2C5F4E75	46FC2700	4FF92000	80002C7C	00000000	4E560000	4BF92000
20000620	07D44EB9	2000069C	41F92000	08E843F9	200007D4	91C92008	670C41F9	200007D4
20000640	4EB92000	054C41F9	200007D4	43F92000	07D491C9	2008670C	41F92000	07D44EB9
20000660	2000054C	41F90000	00002008	67064EB9	200006F4	4EB92000	06CC487A	001C42A7
20000680	4EB92000	0500508F	4E5E42A7	4EB92000	0748588F	4AFC4E75	00000000	42804281
200006A0	42824283	42844285	42864287	207C0000	0000227C	00000000	247C0000	0000267C
200006C0	00000000	287C0000	00004E75	4E754E71	12D85380	66FA4E75	202F000C	226F0004
200006E0	206F0008	4A80670A	B3C86706	4EB92000	06D04E75	2F072F06	4FEFFFF4	7C007E0C
20000700	20064C07	080041F9	00000000	2240D3C8	22114A81	66102029	00044A80	66082029
20000720	00084A80	67182F69	00080008	2F410004	2EA90004	4EB92000	06D85286	60C24FEF
20000740	000C2C1F	2E1F4E75	2F07598F	20392000	07D82E2F	000C4A80	661A4EB9	2000051C
20000760	20392000	07E04A80	670A2040	4E9042B9	200007E0	2E874EB9	20000784	588F2E1F
20000780	4E754E71	601A2039	200007DC	538023C0	200007DC	41F92000	07E82070	0C004E90
200007A0	20392000	07DC4A80	6EDC2039	200007E4	4A80670A	20404E90	42B92000	07E44EB9
200007C0	20000548	4E754E71	00000000	00000000	00000000	00000000	00000000	00000000
200007E0	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000
20000800	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000
20000820	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000
20000840	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000
20000860	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000
20000880	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000
200008A0	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000
200008C0	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000
200008E0	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000
20000900	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000
20000920	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000
20000940	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000

Figure 4. Contents of the Memory

The first line of the program, which is the machine code for the ***link*** instruction and equal to ***4E560000***, resides in the location with the address ***0x20000500,*** as expected.

NOTE: We can change the contents of the memory at any location. For this, the memory option is used. Enter the address of the location you want to change, for example ***0x20000514***, in the **Display** window. Then, enter the new contents of the memory location by typing the value in the memory window.

7. At this point, we are ready to run the loaded program. For this, we use the **step over (F10)** option inside the ***debug*** window. This option allows us to run the program line by line and to observe the new contents of memory locations and registers resulting from executing the loaded program. In this section, we run the first instruction to see how the contents of the registers change. Before running the program, we check the contents of the register using **View -> Registers** option from the main menu. You might need to resize the windows to see both the program and register windows at the same time. You can also change each register's value, as needed.

- Check the PC register (program counter), the Data registers (D0-D7), Address registers (A0-A7), and the SR (Status Register).
- Run the program using ***Step Over*** for two instructions.
- Check the PC register.
- Check the contents of D3 register. It is also possible to change the content of a register by selecting it and typing the new value.

8. At this point, we run the program to termination and check the register contents.
   - Set a breakpoint at the **bra loop** instruction.
   - Now press the F5 button to resume execution.
   - Check the contents of D4 register (you should have 7 in this register).

Exercise: Change the contents of the memory with the address "*0x20000514*" to "SUBTRACT" and run the program. What are the contents of the registers and why?

It is obvious that altering the contents of the program memory can change a program's functionality.

Exercise: Discuss the advantages and disadvantages of changing the contents of the program memory during execution time?

9. The contents of some special registers are critical and changing them might result in exceptions (e.g., access to protected areas, or the ability to access non-instruction sections in the memory). Thus, we have to be careful when changing these registers. One of these registers is the PC register.

1.4.2 Part 2

1. The Lab Instructor will provide you with another URL where you can download a small zip folder that contains the project **BareAsm_1.mcp**. The **main.s** file in the source folder contains the assembly code for this part of the lab.

Unzip this folder and put it into your M: drive. You need to assemble and link your source program before running it on the **ColdFire** board.

To assemble the **lab1a.s program**, follow the same procedure mentioned in 5 and 6 of 1.4.1 Part 1.

Figure 5 shows the listing for ***main.s.*** It can be obtained by using **Project-> Disassemble** option of the main menu.

```
    *** EXECUTABLE CODE (.text) ***

Address   ObjectCode      Label  Opcode   Operands          Comment


Source file:        C:\Documents and Settings\fkhunjus\My Documents\BareAsm_1\Source\main.s
```

```
Producer:        Metrowerks Assembler - ColdFire ELF Assembler

;
;    8:  .text
;    9:
;   10: .global               _main
;   11: _main:
;   12:               link    a6,#0
;
0x00000000               _main:
;                    main:
0x00000000               _@DummyFn1:
;                    @DummyFn1:
0x00000000  0x4E560000           link    a6,#0
;
;   13:          lea.l   -12(a7),a7
;
0x00000004  0x4FEFFFF4           lea     -12(a7),a7
;   14:          move.l       #x,a3
0x00000008  0x267C00000000        movea.l  #.data,a3
;   15:          move.l       #y,a4
0x0000000E  0x287C00000002        movea.l  #.data+4,a4
;   16:          move.l       #sum,a5
0x00000014  0x2A7C00000004        movea.l  #.data+8,a5
;   17:          move.w  (a3),d2
0x0000001A  0x3413               move.w   (a3),d2
;   18:          move.w       (a4),d3
0x0000001C  0x3614               move.w   (a4),d3
;   19:          add.l        d3,d2
0x0000001E  0xD483               add.l    d3,d2
;   20:          move.w       d2,(a5)
0x00000020  0x3A82               move.w   d2,(a5)
;   21: loop:   bra          loop
0x00000022  0x60FE               bra.s    *+0              ; 0x00000022
;   22:          unlk   a6
;
0x00000024  0x4E5E               unlk     a6
0x00000026  0x4E75               rts


                  *** RELOCATIONS (.rela.text) ***

no    type            offset      addend      symbol

0     R_68K_32        0x00000016  0x00000004  .data
1     R_68K_32        0x00000010  0x00000002  .data
2     R_68K_32        0x0000000A  0x00000000  .data


                  *** INITIALIZED DATA (.data) ***

Header:
 Section Alignment : 4
 Section Size      : 6
0x00000000:  00 11 00 22 00 00                   '..."..'
```

Figure 5. Listing for the main.s file

<u>Exercise: Describe the function of this program.</u>

This file contains the corresponding ColdFire machine language. As can be seen, the program has two sections, data and program. The data section includes the variables used in the program. This section starts with the ***.data*** keyword. The program section starts with the ***".text"*** keyword.

The data section contains the initial value of the variables, if there are any. For example, in this program we define ***x*** as a short variable which takes 2 bytes and its initial value is 0x11 (17). The code section consists of the translation of the source code. As can be seen, each instruction is converted into its corresponding object code. For example, the instruction ***add.l d3,d2*** is translated into 0xD483, which represents the opcode ***D4*** and the parameters for the instruction. Other instructions need more attention. For example, instruction **move.l #x,a3** in line 14 is translated into **0x267C00000000**. The first two bytes represent the op-code and the destination of the instruction, respectively. However, the address of the source operand is equal to zero. The reason is that the source operand is a variable, whose final address is not known before the linking process.

3. In this section, we build the executable file using the **Project -> Make** option in the main menu as explained in 1.4.1 Part 1. It also generates the address mapping of the data and text segments into the **M52233DEMO Console Debug.elf** file in the bin directory.

Use a text editor and check the address mapping of each section in this file because we will use them in the subsequent sections. The file shows the final address of the program and data sections. Using this information, the linker patches the object file, which is generated by assembler, at the locations where the final destinations have not yet been resolved.

4. As explained, the executable program has to be loaded into the board before execution. For this, make sure that the **ColdFire** board is properly connected to the workstation and powered on.

- Go to the first line of the program (**link a6,#0**).
- Right click on the mouse and select the **set breakpoint** option.
- **Click Project -> Debug**

You will see a warning, just press OK and continue to the next step.

Now the program has been loaded into the memory and can be executed. As shown in Figure 6, the blue arrow points to the starting location of the loaded program. Change the representation of the program from source to assembly, then you should see the following codes.

```
20000500: 4E560000       link     a6,#0
20000504: 4FEFFFF4       lea      -12(a7),a7
20000508: 267C200007E4   movea.l  #536872932,a3
2000050E: 287C200007E6   movea.l  #536872934,a4
20000514: 2A7C200007E8   movea.l  #536872936,a5
2000051A: 3413           move.w   (a3),d2
2000051C: 3614           move.w   (a4),d3
2000051E: D483           add.l    d3,d2
```

```
20000520: 3A82      move.w   d2,(a5)
20000522: 60FE      bra.s    _@DummyFn1+0x22 (0x20000522); 0x20000522
20000524: 4E5E      unlk     a6
20000526: 4E75      rts
```

Figure 6. Assembly code after linking

Before running the program using **step over**, examine the contents of the memory and the registers.

Figure 7 shows the content of the memory after loading onto the ColdFire board. As explained, the file **M52233DEMO Console Debug.elf** in the bin folder describes the memory mapping of the program after the linking process. As can be seen from the file's contents, the code section starts at the location with address **0x20000500** and the data section starts from address **0x200007E4**. Therefore, we expect the first byte of the program is located at address ***0x20000500*** and the first byte of data at address ***0x200007E4.*** To verify this, we use the **Data -> View *Memory*** option. If we click on this option we will see Figure 7.

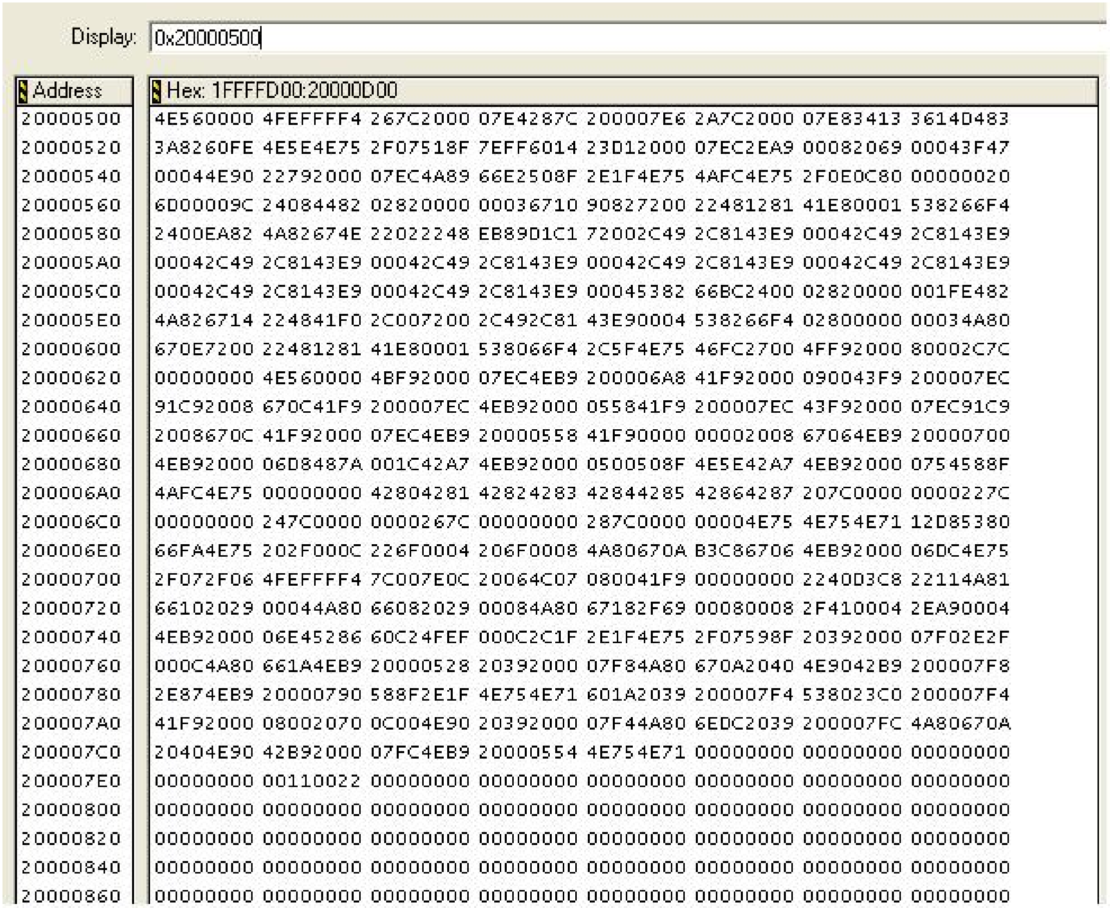
The first byte of the program, which is the op-code for the ***link*** instruction and equal to ***4E56***, resides in the location with the address ***0x20000500***, as expected. The figure also shows the contents of the data section. As all the variables are defined as ***short***, they need two bytes for their representation. The first variable, ***x***, has a value equal to 0x0011 and is located at the address ***0x200007E4***. The second variable, ***y***, has a value equal to 0x0012 and is located at the address ***0x200007E8***. The next variable, ***sum***, is located immediately after the variable ***y*** and its value is equal to 0. All this information can be observed in Figure 7 .

Using this mapping, the addresses in the object file, which are set to 0000 by the assembler, are inserted by the linker. To investigate this problem, **compare the first four instructions in the listing with the contents of the program memory that begins at the address *0x20000500***.

5. At this point we are ready to run the loaded program. For this, we use the ***step over (F10)*** option inside the ***debug*** window. This option allows us to run the program line by line and to observe the updated contents of memory locations and registers as the result of executing the loaded program. In this section, we run seven instructions to see how the contents of the registers change. Before running the program, we check the content of the registers.

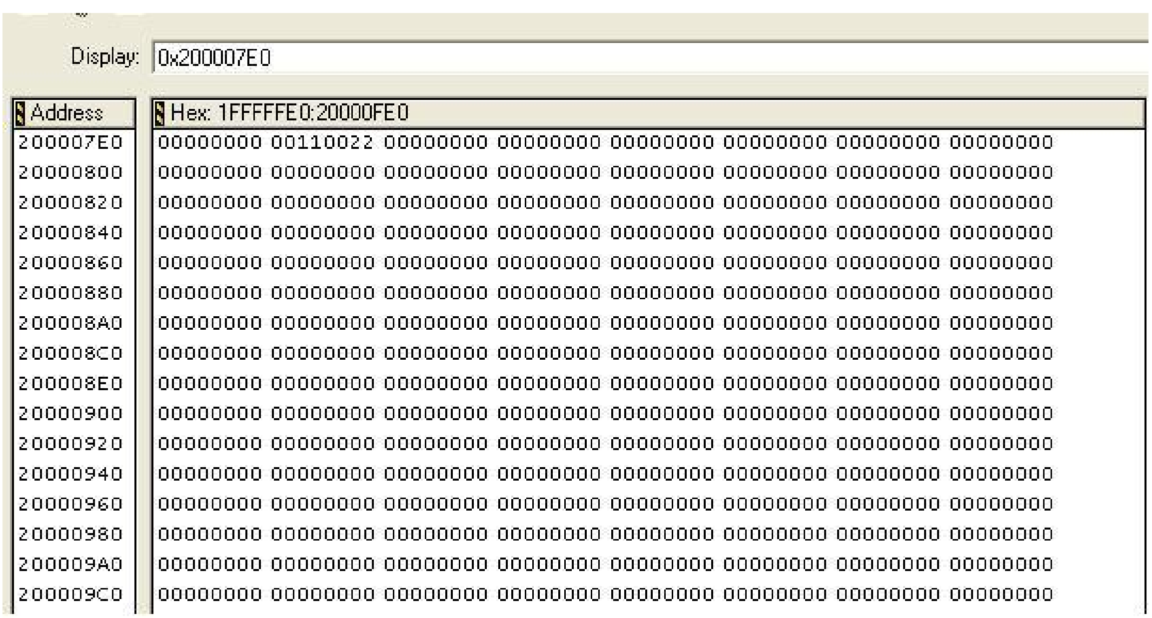
To observe the contents of the registers use the **View -> Registers** option from the main menu. You can also change each register's value, as needed.

- Check the PC register (program counter), the Data registers (D0-D7), the Address registers (A0-A7), and the SR (Status Register).
- Run the program using ***Step Over*** for one instruction.
- Check the PC register.
- Set a breakpoint at the **bra loop** instruction.



Display: 0x20000500

Address	Hex: 1FFFFD00:20000D00							
20000500	4E560000	4FEFFFF4	267C2000	07E4287C	200007E6	2A7C2000	07E83413	3614D483
20000520	3A8260FE	4E5E4E75	2F07518F	7EFF6014	23D12000	07EC2EA9	00082069	00043F47
20000540	00044E90	22792000	07EC4A89	66E2508F	2E1F4E75	4AFC4E75	2F0E0C80	00000020
20000560	6D00009C	24084482	02820000	00036710	90827200	22481281	41E80001	538266F4
20000580	2400EA82	4A82674E	22022248	EB89D1C1	72002C49	2C8143E9	00042C49	2C8143E9
200005A0	00042C49	2C8143E9	00042C49	2C8143E9	00042C49	2C8143E9	00042C49	2C8143E9
200005C0	00042C49	2C8143E9	00042C49	2C8143E9	00045382	66BC2400	02820000	001FE482
200005E0	4A826714	224841F0	2C007200	2C492C81	43E90004	538266F4	02800000	00034A80
20000600	670E7200	22481281	41E80001	538066F4	2C5F4E75	46FC2700	4FF92000	80002C7C
20000620	00000000	4E560000	4BF92000	07EC4EB9	200006A8	41F92000	090043F9	200007EC
20000640	91C92008	670C41F9	200007EC	4EB92000	055841F9	200007EC	43F92000	07EC91C9
20000660	2008670C	41F92000	07EC4EB9	20000558	41F90000	00002008	67064EB9	20000700
20000680	4EB92000	06D8487A	001C42A7	4EB92000	0500508F	4E5E42A7	4EB92000	0754588F
200006A0	4AFC4E75	00000000	42804281	42824283	42844285	42864287	207C0000	0000227C
200006C0	00000000	247C0000	0000267C	00000000	287C0000	00004E75	4E754E71	12D85380
200006E0	66FA4E75	202F000C	226F0004	206F0008	4A80670A	B3C86706	4EB92000	06DC4E75
20000700	2F072F06	4FEFFFF4	7C007E0C	20064C07	080041F9	00000000	2240D3C8	22114A81
20000720	66102029	00044A80	66082029	00084A80	67182F69	00080008	2F410004	2EA90004
20000740	4EB92000	06E45286	60C24FEF	000C2C1F	2E1F4E75	2F07598F	20392000	07F02E2F
20000760	000C4A80	661A4EB9	20000528	20392000	07F84A80	670A2040	4E9042B9	200007F8
20000780	2E874EB9	20000790	588F2E1F	4E754E71	601A2039	200007F4	538023C0	200007F4
200007A0	41F92000	08002070	0C004E90	20392000	07F44A80	6EDC2039	200007FC	4A80670A
200007C0	20404E90	42B92000	07FC4EB9	20000554	4E754E71	00000000	00000000	00000000
200007E0	00000000	00110022	00000000	00000000	00000000	00000000	00000000	00000000
20000800	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000
20000820	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000
20000840	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000
20000860	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000



Display: 0x200007E0

Address	Hex: 1FFFFFE0:20000FE0							
200007E0	00000000	00110022	00000000	00000000	00000000	00000000	00000000	00000000
20000800	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000
20000820	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000
20000840	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000
20000860	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000
20000880	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000
200008A0	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000
200008C0	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000
200008E0	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000
20000900	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000
20000920	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000
20000940	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000
20000960	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000
20000980	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000
200009A0	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000
200009C0	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000

Figure 7. Memory Contents (Program & Data Sections)

- Now press the F5 button to resume execution.
- Check the contents of the ***sum*** variable at the address ***0x200007EA***.

The memory dump after the execution (i.e., memory content) is shown in Figure 8.

Important Note:
As can be seen from the Memory and Register Windows in this experiment, you can change the contents of any memory or register during the execution of the loaded program. This method can be used to debug and check the correctness of the program execution.

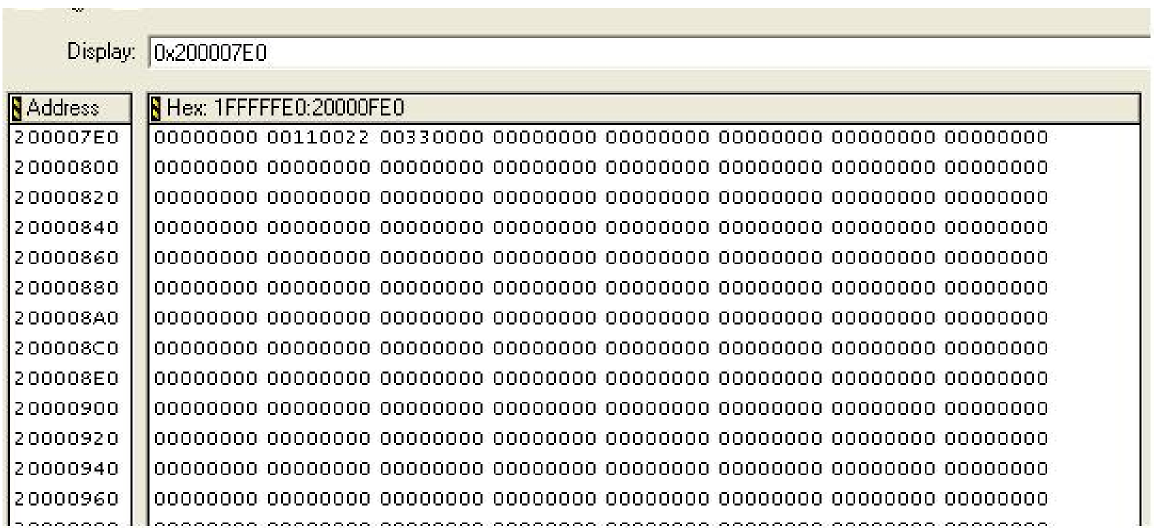


Figure 8. Memory Contents after Execution

1.4.3 Part 3

The following source code is considered for this portion of the lab.

```
/*
 * File:       main.s
 */
##################################################################
# Data Section starts here
##################################################################
          .data
x_array:       .short 0x24,0x12,0x05,0x66,0x12,0x01,0x08,0x14
y_array:       .short 0x12,0x33,0x21,0x0A,0x15,0x11,0x25,0x99
sum_array:     .space  16
###################################################################
# End of data section
###################################################################
#
###################################################################
# Code Section starts here
```

```
################################################################
                        .text
                        .global         _main
_main:
                        link   a6,#0
                        move.l      #x_array,a3
                        move.l      #y_array,a4
                        move.l      #sum_array,a5
                        moveq       #8,d0
loop:                   move.w      (a3),d2
                        move.w      (a4),d3
                        add.l       d3,d2
                        move.w      d2,(a5)
                        addq.l      #2,a3
                        addq.l      #2,a4
                        addq.l      #2,a5
                        subi.l      #1,d0
                        cmpi.l      #0,d0
                        bne         loop
endloop:                bra         endloop
                        unlk   a6
                        rts
.end
```

You are required to assemble and link this source file, and run the executable file.

Deliverable for Part 3:

- **Description of the program.**
- **Program Listing.**
- **Memory Map Assignment by the linker.**
- **Snap shots of the data section before and after execution of the program.**
- **The contents of the memory and a3, a4, a5, and d0 registers at the *bra endloop* breakpoint.**
- **Change the program in such a way that it performs its operation for 32 elements. Is there any limitation in the provided code?**

Summary:

In this lab we have described the steps in the translation process and introduced the basic functions provided by the **CodeWarrior Development Studio** environment.

The assembler translates an assembly language program into object code. The linker transforms an object code program into an executable program.
Figure 9 summarizes the steps used to translate an assembly language program into an executable program.

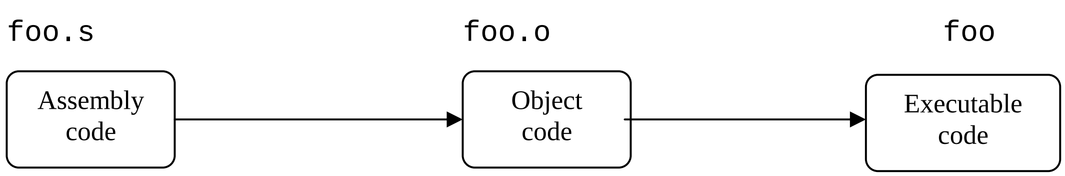


Figure 9. From Assembly to Executable