



Khalifa University of Science, Technology and Research

Electronic Engineering Department

ELCE333 Microprocessor Systems laboratory

Laboratory Experiment 1

MICROCONTROLLER ASSEMBLY PROGRAM DEVELOPMENT

Lab Partners

Dana Bazazeh, 100038654

Moza Al Ali 100038604

Date Experiment Performed: 28/01/2015

Date Lab Report Submitted: 04/02/2015

Lab Instructor: Dr. Mohamed Al Zaabi/ Dr. Mahmoud Khonji

SPRING 2015

Table of Contents

Table of Contents	2
List of Figures and Tables	3
Abstract	<i>Ошибка! Закладка не определена.</i>
1. Introduction	5
1.1 Aim	5
1.2 Objectives	5
2. Design and results	6
<u>TASK - 1: Adding Numbers</u>	6
<u>TASK - 2: Subtracting Numbers</u>	7
<u>TASK-3: Simple Program:</u>	9
3. Assignment Questions	11
4. Conclusions and Recommendations	13

List of Figures and Tables

List of Figures:

Figure 1: microcontroller HCS12.....	5
Figure 2: Code for Adding Numbers.....	6
Figure 3: Code for Subtracting Numbers.....	8
Figure 4: content of main.lst file for Task 1.....	9
Figure 5: unsigned 8 bit multiplication code.....	12
Figure 6: multiplication final result.....	12

List of Tables:

Table 1: Registers contents while single stepping in adding program.....	7
Table 2: Registers contents while single stepping in subtracting program.....	8
Table 3: The memory contents for the developed program.....	10
Table 4: The CCR flags for the developed program.....	11

Abstract

In this lab experiment students were asked to edit, assemble, simulate, download and execute an assembly program. Mainly, there were 3 different tasks in this session. students were asked in first task to edit a given code so that it should be able to perform an addition of numbers.

Moreover, the same procedure of task 1 was taken in the second task but with certain modification in order to subtract byte A from B. Finally in the last task students created a List file which was produced by the assembler rather than the binary file.

All the three tasks were performed using CodeWarrior IDE software.

1. Introduction

Basically, A typical microcontroller includes a processor, memory, and peripherals. It is an integrated circuit with 4 main components and 2 buses, these are CPU, ROM, RAM and the Input/output (I/O) Interface with data bus and address bus.

Its main function is to facilitate the operation of the electromechanical systems found in everyday convenience items. Moreover, it is designed to perform arithmetic and logic operations that make use of small number-holding areas called registers. Typical microcontroller operations include adding, subtracting, comparing two numbers, and fetching numbers from one area to another. These operations are the result of a set of instructions that is part of the microcontroller design.

This mean tool for this experiment is the CodeWarrior IDE software which will help the students to edit, develop, debug, download and execute the assembled code for the microcontroller HCS12.



Figure 1: microcontroller HCS12

1.1 Aim

The main goal of this lab is to get students introduced to the CodeWarrior IDE software and learn how to to edit, assemble, simulate, download and then execute an assembly program.

1.2 Objectives

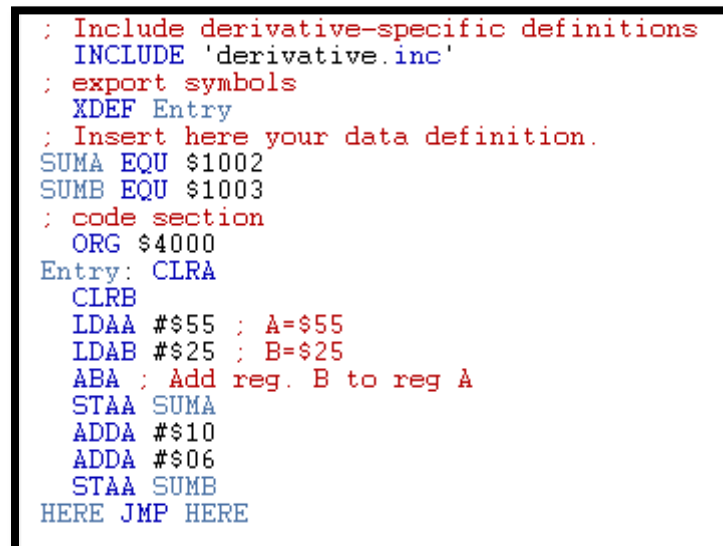
- Access a microcontroller HCS12 program
- Check the program execution using single step debugging
- Examine the system registers and memory contents.
- Write and recognize a list file and binary file using the assembly code
- Use the HCS12 instruction manual as appropriate

2. Design and results

Now, we will move on to discuss three different tasks, adding numbers, subtracting numbers and finally creating a simple program and examining the List File.

TASK - 1: Adding Numbers

Using the CodeWarrior IDE and following the same procedure we did in the introductory session, we create a new project and use the code provided in the labsript to examine the line-by-line debugging and examine how the memory addresses and registers change with each instruction. Therefore, we start by pasting the code in the main.asm file, followed by the making and debugging of the code. The code is shown in figure 2 below.

A screenshot of assembly code displayed in a text editor with a black border. The code is written in a mix of red and blue text on a white background. It includes comments in red and instructions in blue. The code defines two memory locations, SUMA and SUMB, and performs a series of operations: clearing registers, loading values into accumulators A and B, adding B to A, and storing the results in SUMA and SUMB.

```
; Include derivative-specific definitions
INCLUDE 'derivative.inc'
; export symbols
XDEF Entry
; Insert here your data definition.
SUMA EQU $1002
SUMB EQU $1003
; code section
ORG $4000
Entry: CLRA
      CLRB
      LDAA #$55 ; A=$55
      LDAB #$25 ; B=$25
      ABA ; Add reg. B to reg A
      STAA SUMA
      ADDA #$10
      ADDA #$06
      STAA SUMB
      HERE JMP HERE
```

Figure 2: Code for Adding Numbers

The above code uses 2 labels SUMA and SUMB to refer to memory addresses 1002 and 1003 respectively. SUMA will hold the summation result of adding accumulator A and B while SUMB will hold the summation result after adding the immediate values to the accumulator A. We begin by clearing both accumulators and then loading them with a constant hexadecimal using immediate addressing mode. Next, we add both accumulators together where the result is stored in accumulator A. Therefore, we use the label SUMA to hold the result of this summation. Then, we add immediate values to existing content of accumulator A and store the final result in SUMB.

.Now that we have analyzed the code, the next step would be to debug it and record the changes that occur in accumulators A, B and D, where D is just the 2 byte combination of A and B. We record the memory contents after executing each instruction as can be seen in Table 1 below.

Table 1: Registers contents while single stepping in adding program

Instruction	A	B	D	Comments
CLRA	00	CB	00CB	Accumulator A cleared 0 \rightarrow A
CLRB	00	00	0000	Accumulator B cleared 0 \rightarrow B
LDAA #\$55	55	00	5500	\$55 \rightarrow A
LDAB #\$25	55	25	5525	\$25 \rightarrow B
ABA	7A	25	7A25	Add Reg. B to A \rightarrow A
STAA R1	7A	25	7A25	A \rightarrow (R1)
ADDA #\$10	8A	25	8A25	\$10+A \rightarrow A
ADDA #\$06	90	25	9025	\$06+A \rightarrow A
STAA R2	90	25	9025	A \rightarrow (R2)

We can clearly see that the contents of the accumulators changed with each instruction as we expected and therefore we can conclude that the code is working properly. Although we did not record the contents of memory addresses 1002 and 1003, we did examine how their contents changed after each instruction in the debugger. Therefore, we can clearly see that

TASK - 2: Subtracting Numbers

Now that we are done with addition, we can try to manipulate the code to subtract the registers from each other. Here we need to subtract the byte in A from the byte in B. In order to find out which instruction to use, we look at the instruction set manual of the HCS12 microcontroller. We find that the instruction SUBB will have the following effect $B - (M) \rightarrow B$ and therefore we can use it for this purpose. The final code is shown below in figure 3.

```

; Include derivative-specific definitions
INCLUDE 'derivative.inc'
; export symbols
XDEF Entry
; Insert here your data definition.
AA EQU $1003
RESULT EQU $1004

; code section
ORG $4000

Entry: CLRA
      CLRB
      LDAA #$55
      LDAB #$25
      STAA AA
      SUBB AA
      STAB RESULT

HERE JMP HERE

```

Figure 3: Code for Subtracting Numbers

A closer look at the code above represents the step taken for subtracting numbers. The instruction SUBB which subtract the operand saved in LOC1 from the value stored in accumulator B. following this, the step in task 1 is repeated in recording the contents of registers before and after the simulations. As we can see from the code above, we again use labels, AA to store the value of accumulator A because the SUBB instruction cannot take a register, only a memory address. The other label RESULT will store the result of subtracting A from B. AA and RESULT refer to memory addresses \$1003 and \$1004 respectively.

Now that we have analyzed the code, we repeat the same process as task 1 where we debug the code line by line and record the changes that occur in accumulators A, B and D. The results are shown below in Table 2.

Table 2: Registers contents while single stepping in subtracting program

Instruction	A	B	D	Comments
CLRA	00	CB	00CB	Accumulator A cleared 0→ A
CLRB	00	00	0000	Accumulator B cleared 0→ B
LDAA #\$55	55	00	5500	\$55 → A (load accumulator A)
LDAB #\$25	55	25	5525	\$25→ B (load accumulator B)
STAA AA	55	25	5525	A → (AA)
SUBB AA	55	D0	55D0	B-(AA)→B
STAB RESULT	55	D0	55D0	B→(RESULT) store result

Again, we can clearly see that the contents of the accumulators changed with each instruction as we expected and therefore we can conclude that the code is successfully subtracting the 2 accumulators as we need it. We can see from the table that in some instructions, no change occurs in the contents of any of the accumulators and this is because these instructions are storing to a memory address, either \$1003 or \$1004 in our case. The final answer of subtracting B-A (\$55-\$25) is given as D0 which is correct.

TASK-3: Simple Program:

For this task, we study another type of file which is the (LST) List File. This file can also be produced by the assembler just like the binary file (object file) which is the main.asm. The LST file contains:

- The source-code program.
- The assembled code in hexadecimal format (including memory locations).
- Any error messages.
- A list of user-defined symbols used and their values.

Therefore, the list file is very important in terms of debugging the source code. To obtain the list file of the code in task 1, we follow the steps given in the labscript, making sure we are on Full Chip Simulation mode before doing so. These steps will create a main.lstfile in our directory. Opening this file, we can see the content in figure 4 below.

```

Freescale HC12-Assembler
(c) Copyright Freescale 1987-2010

Abs. Rel.  Loc  Obj. code  Source line
-----
   1    1
   2    2
12817   3
12818   4
12819   5
12820   6      0000 1002  SUMA EQU $1002
12821   7      0000 1003  SUMB EQU $1003
12822   8
12823   9
12824  10 a004000 87      Entry: CLRA
12825  11 a004001 C7      CLRB
12826  12 a004002 8655    LDAA #$55 ; A=$55
12827  13 a004004 C625    LDAB #$25 ; B=$25
12828  14 a004006 1806    ABA ; Add reg. B to reg A
12829  15 a004008 7A10 02  STAA SUMA
12830  16 a00400B 8B10    ADDA #$10
12831  17 a00400D 8B06    ADDA #$06
12832  18 a00400F 7A10 03  STAA SUMB
12833  19 a004012 0640 12  HERE JMP HERE
12834  20

```

Figure 4: content of main.lst file for Task 1

As we can see from the file contents, the “obj. code” refers to the assembled code while the “Loc” refers to the memory location or address. In order to identify the components and size of each instruction, we fill table 3 below with the starting address, no. of bytes and the instruction code or opcode.

Table 3: The memory contents for the developed program

Instruction	Start Address	No. of Bytes	Instruction Code	Comments
CLRA	\$4000	1	87	No operand, no memory access, inherent addressing. 1B opcode
CLRB	\$4001	1	C7	No operand, no memory access inherent addressing. 1B opcode
LDAA #\$55	\$4002	2	8655	Immediate addressing 1B opcode and 1B value
LDAB #\$25	\$4004	2	C625	Immediate addressing 1B opcode and 1B value
ABA	\$4006	2	1806	Immediate addressing 2B opcode
STAA R1	\$4008	3	7A10 02	Immediate addressing 1B opcode and 2B operands
ADDA #\$10	\$400B	2	8B10	Immediate addressing 1B opcode and 1B value
ADDA #\$06	\$400D	2	8B06	Immediate addressing 1B opcode and 1B value
STAA R2	\$400F	3	7A10 03	Immediate addressing 1B opcode and 2B operands

From the above table we notice that the first instruction starts at the address \$4000 because we specified that in the code using ORG \$4000. Also, we notice that the number of bytes vary depending on the type of instruction and on the addressing mode. We can see that the storing instructions have the most number of bytes (3 bytes). Moreover, we see that the starting address of the instruction depends on the number of bytes occupied by the previous instruction. The difference between 2 consecutive instruction addresses will give us the no. of bytes of the first instruction.

3. Assignment Questions

1. Find the status of CCR flags - C, V, H, N, Z and Acc A after executing A, B, C and D individually and comment on the results.

We write the following instructions in the main and then using the debugger, we can notice which flags are 1 and which are 0 in the CCR register under the register window. The flag status is 1 when it is will be highlighted. Our results are shown below.

Table 4: The CCR flags for the developed program

Instructions		C	V	H	N	Z	Acc A	Comment
A	LDAA #\$A5	0	0	0	1	0	A5	A5 is negative no
	ADDA #\$89	1	1	0	0	0	2E	Add results in carry and overflow
B	LDAA #\$80	1	0	0	1	0	80	80 is negative no. and there is carry
	ADDA #\$80	1	1	0	0	1	0	Result is zero with carry and overflow
C	LDAB #\$A5	1	0	0	1	0	0	A5 is negative and there is carry
	SUBB #\$68	0	1	0	0	0	0	Result contains overflow
D	LDAB #\$65	0	0	0	0	0	0	No flags
	SUBB #\$65	0	0	0	0	1	0	The result is 0

2. Write an instruction sequence to multiply two 8-bit numbers using MUL instruction (unsigned). Initialize Accumulator A and Accumulator B with \$F4 and \$F2 respectively using the load instructions. Highlight the result in the D register and the CCR flags.

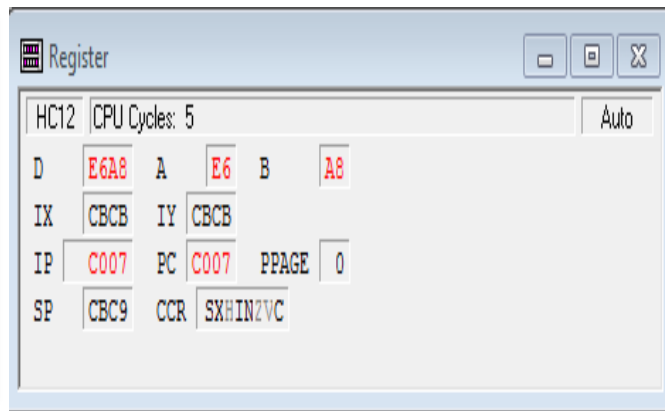
We refer to the instruction manual. There we find that the instruction MUL will carry out the following: Unsigned $A * B \rightarrow D$. It will multiply accumulators A and B and store the result in accumulator D since an 8 bits * 8 bits = 16 bits. Figure 5 below shows the complete code.

```
INCLUDE 'derivative.inc'
; export symbols
XDEF Entry

Entry:
  CLRA
  CLRB

  LDAB #$F4
  LDAA #$F2
  MUL
```

Figure 5: unsigned 8 bit multiplication code



Register					
HC12 CPU Cycles: 5 Auto					
D	E6A8	A	E6	B	A8
IX	CBCB	IY	CBCB		
IP	C007	PC	C007	PPAGE	0
SP	CBC9	CCR	SX	H	N
			Z	V	C

Figure 6: multiplication final result

After executing the entire code, we see that the result of $F4 * F2 = E6A8$ and that this is stored in accumulator D as can be seen in figure 6 above. We can also see from the CCR flags that the result is negative and has carry.

4. Conclusions and Recommendations

In conclusion, students have accomplished all the 3 tasks successfully and they are now able to able to edit, assemble, simulate, download and then execute an assembly program through assembly source code of microcontroller HCS12. In addition, all the objectives and aims of the experiment were met. Also, students now understand what different CCR flags represent. Apart from that, they can create and understand what a list file is and what it contains from information about memory locations and Op-Code of the instructions.