CIS*2030 - Lab#5

Writing and Assembling 68000 Programs

The purpose of this lab is to gain a better understanding of how to write and assemble 68000 assembly-language programs.

Upon completion of this lab, students will be able to:

- Read and interpret a *listing* file.
- Read and interpret a Motorola S-Record,
- Understand how the assembler's main directives (EQU, ORG, DC, DS, and END) affect the assembly process,
- Illustrate the effects of assembler directives by creating a memory map,
- Understand the difference between an assembler directive and a machine instruction, and
- Write simple programs using both the assembler directives and arithmetic and data movement instructions.

During this lab you will make use of the following 68000 assembly-language programs which are available from course website:

- *L5_1.X68* provided
- *L5_2.X68* provided
- *L5_3.X68* provided
- *L5_4.X68* To be written by you
- *L5_5.X68* To be written by you

Part A: Listing Files

Recall that in Lab 4 we learned how to take pre-written assembly-language programs and automatically convert them into machine-language programs using a program called an *assembler*. (Strictly speaking, the assembler that we used is called a *cross-assembler*. This is because the assembler program generates machine language for a different type of computer than the one that it is running on. In our case, the assembler is running on an Intel (x86) architecture, but producing machine language for the Motorola 68000.) Using the assembler is easy. Any ASCII text editor can be used to create a source file; that is, an assembly language program. This source file is then assembled to produce a *binary* file (the 68000 machine language program to be executed) and a *listing* file. The listing file contains the source program together with the assembled code in hexadecimal format and any error messages. If errors are found during the assembly phase, you must return to the source file and edit it to correct them. When the program has been assembled and no syntax errors have been made, only then will the assembler produce the binary file that can be run on the 68KMB.

To see how the assembler creates a listing file, we will work with the source program illustrated in Fig. 1 (and also given in class).

```
* L5_1.x68: Program given in class

    ORG $8000               ;program starts at $8000
START   MOVE.B  VALUE1,D0   ;load first byte from $9000
        MOVE.B  VALUE2,D1   ;load second byte from $9001
        ADD.B   D0,D1       ;sum bytes
        MOVE.B  D1,RESULT   ;store sum at $9002
        TRAP    #14         ;return to MON68K

    ORG $9000       ;data starts at $9000
VALUE1  DC.B    11  ;initialize RAM with    11
VALUE2  DC.B    22  ;initialize RAM with    22
RESULT  DS.B    1   ;reserve one byte of    RAM
        END START
```

Figure 1: Simple program presented in class (L5_1.x68).

The previous program loads two bytes stored in memory (with predefined values 11 and 22, respectively) into data registers D0 and D1, respectively. The (byte) contents of both registers are then added together, and the resulting sum is stored back in memory at hexadecimal address 9002. The program terminates by passing control back to the monitor program.

To assemble and generate a *listing file* for the previous program, follow the two steps:

1. After logging in, create the directory structure *H:\2030\L5* under the *H:* directory. Download the **L5_1.x68** file from the course website and save it as **L5_1.x68** under the *H:* directory. **Note: The text in the file should be left justified; that is, it should start at column 0. Also, be sure to explicitly type the *.x68* extension when saving!**

2. To create a listing file, simply assemble the source program; that is, first open a DOSBox window from **Start Menu>All Programs>DOSBox…,** then change into the directory H:\2030\L5 and enter the following command:

   H:\2030\L5>asm68 L5_1

   The previous command causes the assembler (asm68) to take the ASCII encoded source file (L5_1.x68) and convert it into a binary (or object) file (L5_1.bin) that contains the binary machine-language code produced by the assembly process; but it also causes the assembler to generate a listing file that can be viewed using a text editor (L5_1.lis). **Note: the assembler will only produce a binary file if there are no syntax errors in the original source file.**

Now use a text editor to examine the listing file. Your listing file should look like the one given below in Fig. 2. Notice that the listing file shows the original source program before (right-hand side) and after (left-hand side) assembly.

```
Source file: L5_1.X6
1
2 *  L5_1.x68
3
4  00008000                  ORG     $8000    ;program starts at
5  00008000 103900009000     MOVE.B           ;load first byte from
6  00008006 123900009001     MOVE.B           ;load second byte from
7  0000800C D200             ADD.B   D0,D1    ;sum bytes
8  0000800E 13C100009002     MOVE.B           ;store sum at $9002
9  00008014 4E4E             TRAP    #14      ;return to MON68K
1
1   00009000                 ORG              ;data starts at $9000
1   00009000 0B              DC.B      11     ;initialize RAM with
1   00009001 16              DC.B      22     ;initialize RAM with
1   00009002 00000001        DS.B      1      ;reserve one byte of
1
1           000080           END  START

   Lines: 16, Errors: 0, Warnings: 0.
```

Figure 2: List file produced by assembler (L5_1.LIS).

Using your knowledge of listing files and instruction formats from class, answer the following questions related to the listing file above.

a) At what hexadecimal address in memory does the instruction ADD.B D0,D1 start?

b) At what hexadecimal address in memory does the MOVE.B D1,RESULT end?

c) What is the hexadecimal value of the *operation word* for the MOVE.B D1,RESULT instruction?

d) What are the hexadecimal addresses of the first and second *extension words* for the instruction MOVE.B D1,RESULT?

e) Using the instruction format for the ADD instruction given on page 267 of your textbook and the listing file in Fig. 2, identify which bits of the instruction ADD.B D0,D1 represent the following components of the assembly-language instruction:

ADD _____ .B _____ Do _____

Now Consider the following S-Record:

S113800010390000900012390000900 1D20013C111

Identify the following five fields in the S-Record:

a) Record Type: _____

b) Byte Count: _____

c) Load Address: _____

d) Data Bytes: _____

e) Checksum: _____

Part B: Motorola S-Records

You will recall from lab 4 that the (binary) machine-language program produced by the assembler must be converted into Motorola S-Records before it can be downloaded to the 68KMB. The previous S-Records are a common format for transferring binary data between devices. The binary data are encoded as ASCII characters which can then be transferred between two devices using a simple serial protocol, like the one that exists between the SunRay and the 68KMB.

1. To convert the binary file created in step 2 of Part A (L5_1.bin) to Motorola S-Records (L5_1.txt) **open a Windows' Command Prompt** from **Start Menu>All Programs>Accessories**, change into directory H:\2030\L5 and enter the following command:

   H:\2030\L5>srec L5_1

   This will cause the file "L5_1.txt" to be created. This file can be downloaded to the 68KMB and is illustrated in Fig. 3. The file contains a total of five records: one of type S0, one of type S9, and three of type S1. The latter type of record is known as a data record. In general, data records can appear in any order in the file, and each record's subfields are composed of ASCII bytes whose associated characters, when paired, represent one-byte hexadecimal values. **Review the description of Motorola S-Records on page 50 of your textbook**.

```
S113800010390000900012390000900 1D20013C111
S1098010000090024E4E38
S10590000B1649
S90380007C
```

Figure 3: Machine-language program L5_1.BIN encoded using Motorola S-Records.

Part C: Assembler Directives

As discussed in class, an assembly language is made up of two types of statements: executable instructions and assembler directives. An executable instruction is one of the processor's valid instructions and is translated into the appropriate machine-language code by the assembler. We have already encountered a number of typical instructions (e.g., ADD, MOVE, SUB, DIV, etc.). Assembler directives cannot be translated into machine code; they simply tell the assembler things it needs to know about the program and its environment. Basically, assembler directives link symbolic names to actual values, allocate storage for data in RAM, set up pre-defined constants, and control the assembly process. The assembler directives to be covered in this section are: EQU, DC, DS, ORG, and END. **Before proceeding review each of the previous directives by reading Sections 4.6.1 through 4.6.5 of your textbook**.

In Fig. 4 (below) you will find a 68000 assembly-language program that employs all five of the previous assembler directives. Download the program (called L5_2.x68) from the CourseLink and load it into the 68000's memory. If you forget how to do this, review the procedure in Part A of Lab 4.

```
*L5_2 sample program

CODE    EQU $8000
DATA    EQU $9000

        ORG CODE
START   MOVE.W  $9000,D0
        MULU    #7,D0
        DIVU    #3,D0
        ADD.W   $9004,D0
        MOVE.W  D0,$9002
        TRAP    #14
        END CODE

ORG DATA
    DC.W    $10
    DS.W    1
    DC.W    10
    END DATA
START
```

Figure 4: Sample program - L5_2.X68.

1. Execute the program in trace mode and answer the questions below.

a) What does the program do?

b) What is the effect of the assembler directive 'ORG CODE'?

c) What is the effect of the assembler directive 'DS.W 1'?

d) What is the effect of the assembler directive 'DC.W $10?

e) What is the effect of the assembler directive 'DC.W 10'?

f) What is the effect of the assemble directive 'CODE EQU $8000'?

2. Given the DC and DS directives below, fill in the contents of the memory map (on the next page) and give the starting address that each symbol represents.

```
        ORG     $9000
list    DC.B    $A, 2, %110
val1    DC.W    $CB
val2    DS.B    3
val3    DC.L    list + 2
team    DC.B    'Jays'
```

Symbol	Address
list	
val1	
val2	
val3	
team	

RAM

9011	
9010	
900F	
900E	
900D	
900C	
900B	
900A	
9009	
9008	
9007	
9006	
9005	
9004	
9003	
9002	
9001	
9000	

Part D: Assemble-Time Expressions

Most 68000 assemblers permit the programmer to employ an expression as a valid name. The assembler, to yield a numeric value, evaluates expressions. For example, you can write the following:

```
FRAME   EQU     128
FRAME2  EQU     FRAME+
FRAME3  EQU     FRAME2
```

In this example, the first assembler directive equates the value 128 to the symbolic name FRAME. The next directive equates the symbolic name FRAME2 to the value FRAME+16. Since FRAME has already been equated to 128, the assembler equates FRAME2 to 128+16 or
144. As explained in class, the following code is illegal.

```
FRAME   EQU     128
FRAME2  EQU     FRAME3+
FRAME3  EQU     FRAME*4
```

The code is illegal because the second line attempts to equate FRAME2 to the FRAME3 plus 16, but the value of FRAME3 has not yet been evaluated. The following listing file shows how the assembler reports the error.

```
1  0000900                      OR  $9000
2           00000080   FRAME:   EQ  128
3           00000000   FRAME2   EQ  FRAME3+
**** Illegal forward reference - operand
4           00000200   FRAME3   EQU FRAME*4
5           00009000            END $9000
```

It is almost natural to regard arithmetic operations (e.g., FRAME+16) as part of the executable program. They are not. Arithmetic expressions are evaluated by the assembler and the resulting numbers used in the assembly. Since the expressions are evaluated when the program is assembled, they are called assemble-time expressions. There are two reasons to use assemble- time expressions. The first is to make your code easier to maintain. The second is to make your code easier to read. **Review Section 4.5 of your textbook to learn more about the assemble- time expressions supported by the assembler used in this course**.

In Fig. 5 (below) you will find a 68000 assembly-language program that employs a single assemble-time expression. Download the program (called L5_3.x68) from the course website and load it into the 68000's memory. If you forget how to do this, review the procedure in Part A of Lab 4.

```
*L5_3.x68 example of program with assemble-time expression

         ORG    $8000

START  MOVE.B   $9001,D0   ;numeric memory address
       MOVE.B VAR1+1,D0     ;assemble-time expression
         MOVE.B     VAR2,D0     ;label
         TRAP   #14
       ORG $9000
VAR1   DC.B    $01
VAR2   DC.B    $02
VAR3   DC.B    $03
VAR4   DC.B    $04

         END    START
```

Figure 5: Example of assemble-time expression.

a) Examine the listing file for the previous program. Is there any difference between the machine-language code for the three MOVE instructions? Explain your findings.

b) Trace each MOVE instruction. What value is being placed into D0 each time a move operation is performed?

c) If the instruction "MOVE.B VAR1+1,D0" was replaced with "MOVE.B VAR3-1,D0", how would this effect the result in D0?

d) If the instruction "MOVE.B VAR2,D0" was changed to "MOVE.B #VAR2,D0", what would happen?

Part E: Write Your Own Program

Write a program called evaluate to compute the following algebraic expression: Z = 3Y + 2X
Assume that variables X, Y and Z are treated as unsigned 16-bit values stored in memory starting at hexadecimal address 9000. Use the assembler directives DS and DC to create the variables in memory; you can initialize these variables to any values that you consider appropriate. However, you must use labels in your program to access the variables. Save your program as **H:\2030\L5\L5_4.X68**. Verify that your program runs correctly by running it on the
68KMB.

Now re-write the previous program (save it as **H:\2030\L5\L5_5.X68**), but this time assume that only the first variable (e.g., X) stored in memory starting at hexadecimal address 9000 is identified by a label. Use an **assemble-time expression** to compute the addresses of the remaining variables. Examine the listing files for the programs in questions 12 through 13. Is there a difference when using labels *and* assemble-time expressions? Explain.

As a result of completing this lab you should be able to read and interpret a listing file, read and interpret a Motorola S-Record, understand how the assembler's main directives (EQU, ORG, DC, DS, and END) affect the assembly process, be able to illustrate the effects of assembler directives by drawing a memory map, understand the difference between an assembler directive and a machine instruction, and write (simple) programs using both the assembler directives and arithmetic and data movement instructions discussed thus far in the course.

Looking ahead to Lab 6

Next time we will introduce a new type of register, called an address register. Up to now we have assumed that addresses are encoded as part of the instruction itself. Address registers give us another, more efficient option. Assuming that our data is in memory, we can store the memory address of the data that we wish to access in and address register, then, when we want to access the data, we can tell the instruction to look inside the address register to find the address of the data it needs to fetch. If this reminds you of pointers in C, good, because address registers are used for just that – implementing pointers in high-level languages.