Assembler Directives and Address Modes

Name: __Conor Roberts #1056167__

Mark: _____/131

Overview

In this lab, we will take a closer look at the most common assembler directives. As well, we will start to explore some of the address modes that are part of the 68000 ISA.

Objectives

Upon completion of this lab you will be able to:

- Understand assembly-language format.
- Understand how the main assembler directives, DC, DS, EQU, and ORG, affect the assembler process.
- Understand how assembler expressions can be used to compute numeric values at assembly time, and how these values can be used in the assembly process.
- Understand the difference between data registers and address registers.
- Understand the main address modes including absolute, immediate, register indirect, post-increment, pre-decrement, indirect with offset, and indirect with index and offset.

Preparation

Prior to starting the lab, you should review your course notes and perform the following reading assignments from your textbook (if you have not already done so):

- Section 4.3 (Format of Assembly-Language Program)
- Section 4.4 (Assembler symbols)
- Section 4.5 (Assemble-Time expressions)
- Section 4.6-4.6.5 (ORG, EQU, END, DC, DS)

- Section 2.4-2.4.9 (Data Register Direct, Absolute Short, Absolute Long, Register Indirect, Post-Increment Register Indirect, Pre-decrement Register Indirect, Register Indirect with Offset, Register Indirect with Index and Offset)

As explained in class, address modes are concerned with the way in which data are *accessed* rather than the way in which data are *processed*. Address modes are an important part of any ISA, because they tell instructions where to look to find the data they need to carry out their operations. The actual location of the data is referred to as the *effective address*, and the effective address of data can be in a register, a memory location, or even inside the instruction itself!

In this lab, we will consider two of the three address modes that we have been using all along: absolute addressing and immediate addressing. Before proceeding, **read about these address modes in Section 2.4 of your textbook**.

Introduction

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 machine instructions, and is translated into the appropriate binary code by the assembler. (We have already encountered a number of typical instructions, like ADD and MOVE.) Assembler directives, on the other hand, 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 (or labels) 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**.

Part 1: Define Storage (DS) Directive

The *Define Storage (DS)* directive is used to *reserve an uninitialized section of memory* at assemble time to hold one or more bytes, words or long words. The reserved memory is available to the program to access the moment that the program is loaded into memory and starts running. Compilers for the 68000 typically use the DS directive to implement *global variables* in high-level languages, where the variables are *uninitialized*. For example, consider the program illustrated in Table 1. Column 1 shows high-level code, while column 2 shows the assembly language equivalent.

Table 1: DS example.

High-Level Code	68000 Assembler		
`int A, B, C;`	`A`	`DS.W`	`1`
	`B`	`DS.W`	`1`
	`C`	`DS.W`	`1`
`C = A + B;`		`MOVE.W`	`A,D0`
		`ADD.W`	`B,D0`
		`MOVE.W`	`D0,C`

The first three DS assembler directives reserve memory for the three (uninitialized) global variables `A`, `B`, and `C`. Notice that in this case, each `int` variable is treated as a 16-bit (i.e., word) value, and that each variable can be referred to by its label (i.e., `A`, `B`, `C`). In general, the DS directive can be qualified with .B, .W, or .L depending on the data size required for the variable.

Step 1

Download the sample program called **Lab3a.X68** from the course website.

Step 2

Start Easy68K. Once running, load the file Lab3a.X68 using the File->Open File menu choice. You should see something similar to below. (Remember to properly comment your code.)

```
*-------------------------------------------------------
* Title     : Lab3a
* Written by : Your name goes here
* Date      : dd/mm/yyyy
* Description: Examples of Define Storage (DS)
*-------------------------------------------------------

        ORG     $8000
START:  MOVE.W  A,D0          ;move word at A to D0
        ADD.W   B,D0          ;add word at B to D0
        MOVE.W  D0,C          ;store sum in D0 at C

        SIMHALT               ; halt simulator


* Reserve storage for variables A, B, C;
        ORG     $9000
A       DS.W    1             ;reserve 1 word in memory
B       DS.W    1             ;reserve 1 word in memory
C       DS.W    1             ;reserve 1 word in memory

        END     START         ; last line of source
```

Step 3

Assemble the program. What memory address is associated with each of the 3 labels A, B, and C? Print the (32-bit hexadecimal) memory address in the table below. **[1.5 points]**

Label	Address (32-bit)
A	00009000
B	00009002
C	00009004

Step 4

Invoke the Easy68K simulator. Before running the program, use the *memory window* to determine the *initial* value of the words at locations A, B, and C. Print the 3 words (i.e., 16-bit values) in the table below. **[1.5 points]**

Label	Value of word at label (16-bit)
A	FFFF
B	FFFF
C	FFFF

Are the previous words (i.e., 16-bit values) all zero? Explain. **[1 point]**

These values are not all zero. Their values represent an uninitialized value.

Step 5

Based on the list of values in column 2 of the previous table, what 16-bit value should appear in memory at address 0x00009004 once the program runs and completes execution? **[1 point]**

Adding FFFF and FFFF will result in 1FFFE. This address will only represent FFFE due to it being 16-bits.

Run the program, and verify that your previous answer above is correct.

Step 6

What happens with the *size indicator* is missing from the DS directive? To answer this question, remove the size indicator (i.e., `.w`) from the *second* DS directive in the original program. Assemble the program. Does the assembler make an assumption about the size of the data when the size indicator is missing? Explain. (Hint: Compare the address of label `c` in both the original and new program.) **[3 points]**

> Nothing happens when the size indicator is removed. The assembler will assume the size of "WORD" (W) when the size indicator is missing.
>
> Comparing the result of both:
> Before - $00009004 = FFFE
> After - $00009004 = FFFE

Now, run the program to completion, and demonstrate to yourself that both programs produce the same result.

Part 2: Define Constant (DC) Directive

The *Define Constant (DC)* directive is used to *initialize a section of memory* with one or more constants (i.e., bytes, words or long words) at assemble time. (Multiple values can appear on the same line, separated by commas.) Compilers for the 68000 typically use the DC directive to implement initialized global variables in high-level languages. For example, consider the program illustrated in Table 2. Column 1 shows high-level code, while column 2 shows the assembly language equivalent.

Table 2: DS example.

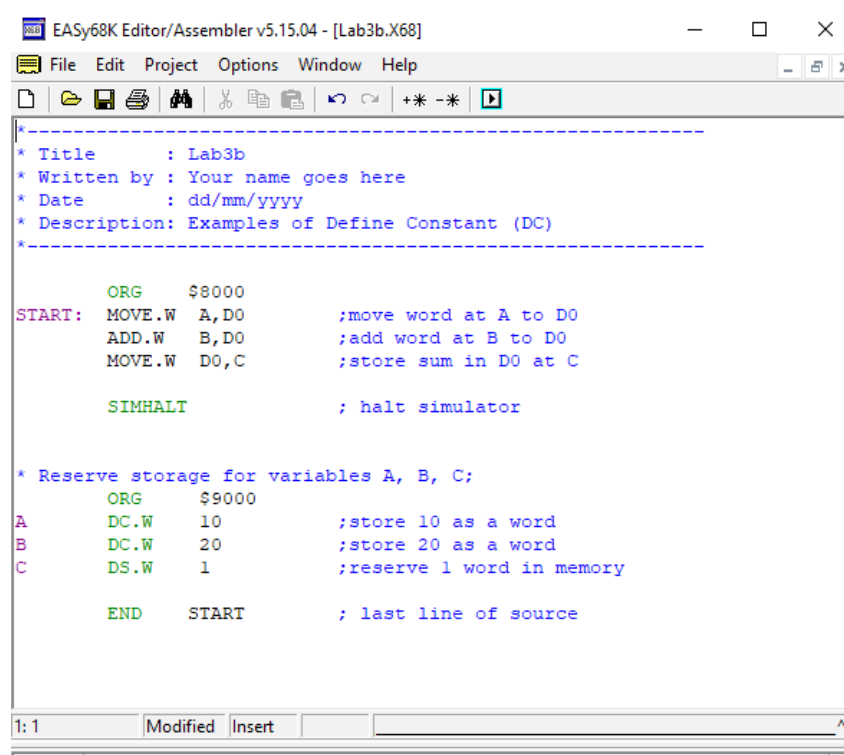| High-Level Code	68000 Assembler

`int A=10, B=20, C;`	`A        DC.W     10`
	`B        DC.W     20`
	`C        DS.W     1`

`C = A + B;`	`         MOVE.W   A,D0`
	`         ADD.W    B,D0`
	`         MOVE.W   D0,C`

The first two DC directives cause the two 16-bit constants 10_{10} and 20_{10} to be loaded into memory at the current location during the assembly process. The remaining DS directive reserves space, again at assemble time, for a single word (i.e., 16-bit variable), but no information is stored in memory. Here, the expectation is that the variable (in this case c) will be assigned a value at some point during the execution of the program, as illustrated in the example code (C = A + B;).

Step 1

Download the sample program called **Lab3b.X68** from the course website.

Step 2

Start Easy68K. Once running, load the file Lab3b.X68 using the File->Open File menu choice. You should see something similar to below. (Remember to properly comment your code.)


```
EASy68K Editor/Assembler v5.15.04 - [Lab3b.X68]                    —  □  ×
File  Edit  Project  Options  Window  Help                          _ 日 ×

*----------------------------------------------------------------
* Title     : Lab3b
* Written by : Your name goes here
* Date      : dd/mm/yyyy
* Description: Examples of Define Constant (DC)
*----------------------------------------------------------------

        ORG    $8000
START:  MOVE.W  A,D0           ;move word at A to D0
        ADD.W   B,D0           ;add word at B to D0
        MOVE.W  D0,C           ;store sum in D0 at C

        SIMHALT                ; halt simulator


* Reserve storage for variables A, B, C;
        ORG    $9000
A       DC.W   10              ;store 10 as a word
B       DC.W   20              ;store 20 as a word
C       DS.W   1               ;reserve 1 word in memory

        END    START           ; last line of source

1: 1          Modified  Insert
```

Step 3

Assemble the program and then invoke the Easy68K simulator. Before running the program, use the memory window to determine the initial word (i.e., 16-bit) value located at each memory address associated with the three labels A, B, and C. Write the (32-bit hexadecimal)

memory address and the (16-bit) value at that memory address in the table below. **[1.5 points]**

Label	Address (32-bit)	Value (16-bit)
A	00009000	000A
B	00009002	0014
C	00009004	FFFF

Notice how the constants 10_{10} and 20_{10} are padded to the left with zeros when stored in memory. In general, if the number of bytes required to encode the constant is less than the number of bytes of memory allocated for the constant, the assembler will always pad the data to the left with zeros.

Step 4

Based on the list of values in column 3 of the previous table, what 16-bit value should appear in memory at address 0x00009004 once the program runs and completes execution? **[1 point]**

> The specified address should contain the hex representation of 30 (10 + 20).
> This value is represented as 001E in hex.

Run the program, and verify that your answer above is correct.

Part 3: Using DC with Character Data
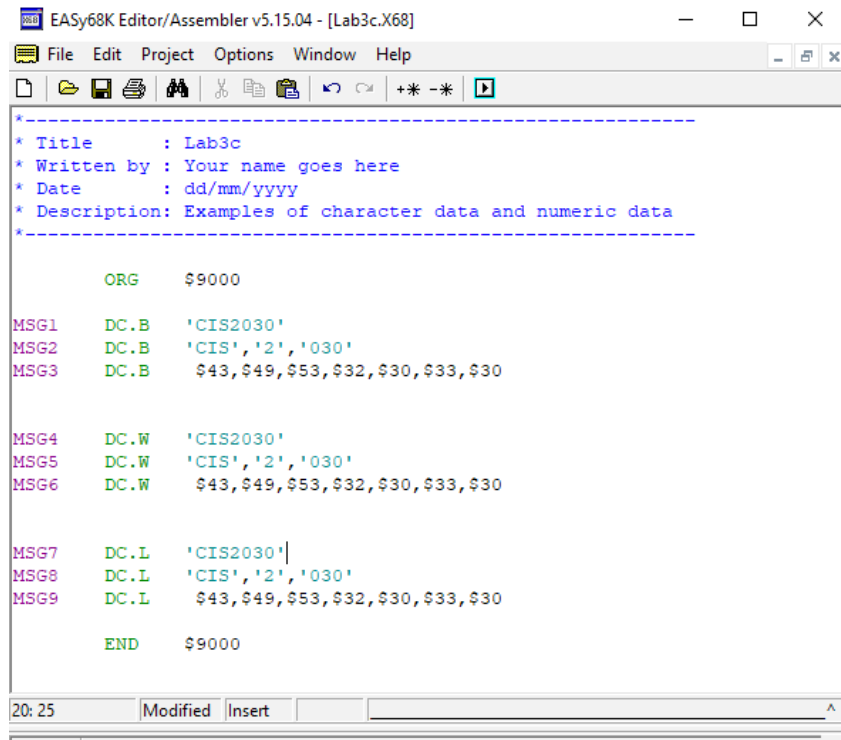
The DC directive can also be used to store ASCII characters in memory. ASCII codes can be defined as a series of *bytes* by including the characters within single quotation marks. However, if a single ASCII character is defined as a word (or a long word), the character is *left justified*.

Step 1

Download the sample program called **Lab3c.X68** from the course website.

Step 2

Start Easy68K. Once running, load the file Lab3c.X68 using the File->Open File menu choice. You should see something similar to below. (Remember to properly comment your code.)

```
EASy68K Editor/Assembler v5.15.04 - [Lab3c.X68]                  —   □   ×
File  Edit  Project  Options  Window  Help                          _ 🗗 ×

*----------------------------------------------------------
* Title      : Lab3c
* Written by : Your name goes here
* Date       : dd/mm/yyyy
* Description: Examples of character data and numeric data
*----------------------------------------------------------

         ORG    $9000

MSG1    DC.B   'CIS2030'
MSG2    DC.B   'CIS','2','030'
MSG3    DC.B    $43,$49,$53,$32,$30,$33,$30


MSG4    DC.W   'CIS2030'
MSG5    DC.W   'CIS','2','030'
MSG6    DC.W    $43,$49,$53,$32,$30,$33,$30


MSG7    DC.L   'CIS2030'
MSG8    DC.L   'CIS','2','030'
MSG9    DC.L    $43,$49,$53,$32,$30,$33,$30

         END    $9000


20: 25          Modified  Insert
```

Step 3

Assemble the program, and then invoke the simulator. Now, open the memory window and use it to help answer the questions below.

Questions

1. MSG1, MSG4, and MSG7 specify the characters *CIS2030* as a single string. How is the string stored in memory? **[1.5 point]**

   > The string is stored in memory with each character (represented as a byte) stored at a separate, but adjacent, location.

2. Does it make any difference if the strings associated with MSG1, MSG4, and MSG7 are shorter, longer or the same length as the length of the size indicator (i.e., .B, .W, .L)? **[2 points]**

> When stored as a byte, the string will be stored using additional extension words.
>
> When stored as a long, the string will be stored using zeroes to fill the empty space.

3. MSG2, MSG5, and MSG8 specify the characters *CIS2030* as individual characters and small groups of characters. Does this result in the same representation in memory as putting all characters in a single string, as in the case of MSG1, MSG4, and MSG7? Explain. **[2 points]**

> This results in a different representation. Each string requries it's own set of words. Meaning that any string of odd length will result in extra zeroes to fill the empty space.

4. MSG3, MSG6, and MSG9 specify the characters *CIS2030* in hexadecimal. For example, the ASCII character 'C' has the hexadecimal value 0x43. Does this result in the same representation in memory as in the previous two cases? Explain. **[2 points]**

> This results in a different representation. Since each character is treated as its own string, the result is that each character is represented in memory as its own long word. Zeroes are added to fill the space.

Part 4: ORG and EQU Directives

As explained in class, assemblers employ a variable known as the *location counter* to keep track of the next available location in memory. (This value of this variable can be obtained using the asterisk * character.) As each line of code in the source file is assembled, the resulting machine instruction or data is assigned memory space starting at the address in the location counter; then the location counter is incremented by the amount of spaced used in order to point to the address of the next available memory location.

The ORG directive can be used to initialize the location counter. The directive is followed by a single argument, which is the address to be assigned to the location counter. By using multiple ORG directives throughout a source file, it is possible to store machine instructions or data structures at specific memory locations. Normally, at least two ORG directives are used in any program: one to specify the location of machine instructions, and another to specify the location of data. Also, the argument (i.e., address) is typically specified symbolically rather than numerically to improve the readability of the source code. To assign a name (i.e., label) to a numeric value, the EQU directive can be used. It is important to note that neither the ORG directive nor the EQU directive generate any machine code.

Step 1

Download the sample program called **Lab3d.X68** from the course website.

Step 2

Start Easy68K. Once running, load the file Lab3d.X68 using the File->Open File menu choice. You should see something similar to below. (Remember to properly comment your code.)

```
EASy68K Editor/Assembler v5.15.04 - [Lab3d.X68]                    —   □   ×
File   Edit   Project   Options   Window   Help                        _ ₆ ×

* Title      : Lab3d
* Written by : Your name goes here
* Date       : dd/mm/yyyy
* Description: Examples of ORG and EQU
*-----------------------------------------------------------

CODE    EQU    $8000          ;start of source code
DATA    EQU    $9000          ;start of data

        ORG    CODE
START:  MOVE.W  A,D0          ;move word at A to D0
        ADD.W   B,D0          ;add word at B to D0
        MOVE.W  D0,C          ;store sum in D0 at C

        SIMHALT               ; halt simulator


* Reserve storage for variables A, B, C;
        ORG    DATA
A       DS.W   1              ;reserve 1 word in memory
B       DS.W   1              ;reserve 1 word in memory
C       DS.W   1              ;reserve 1 word in memory

        END    START          ; last line of source

20: 21                Insert
```

Functionally, this program is identical to the first program contained in the file Lab3a.X68. Only cosmetic differences exist. First, the EQU directive has been used to link the starting addresses of the code ($8000) and the data ($9000) to the labels CODE and DATA. Second, the two labels have been used as arguments in the two ORG directives, which are responsible for locating the instructions and data at the previous two addresses, respectively. The whole point of using the labels is to improve the readability of the code.

Step 3

Examine the listing files of this program (Lab3d.X68) and the original program (Lab3a.X68), and in so doing demonstrate to yourself that there is no functional difference between the two programs at the machine code level. Run both programs to confirm their equivalence.

Part 5: DS, DC and Memory Alignment

As explained in class, the 68000's ISA requires that word and long word boundaries be maintained. This means that although bytes can be stored at any address, both words and long words must be stored only at *even* addresses. This restriction causes the assembler to always ensure that both words and long words are aligned on even address boundaries when the DC and DS directive are used with mixed data sizes. This is accomplished through use of the assembler's location counter. Whenever the location counter contains an odd address,
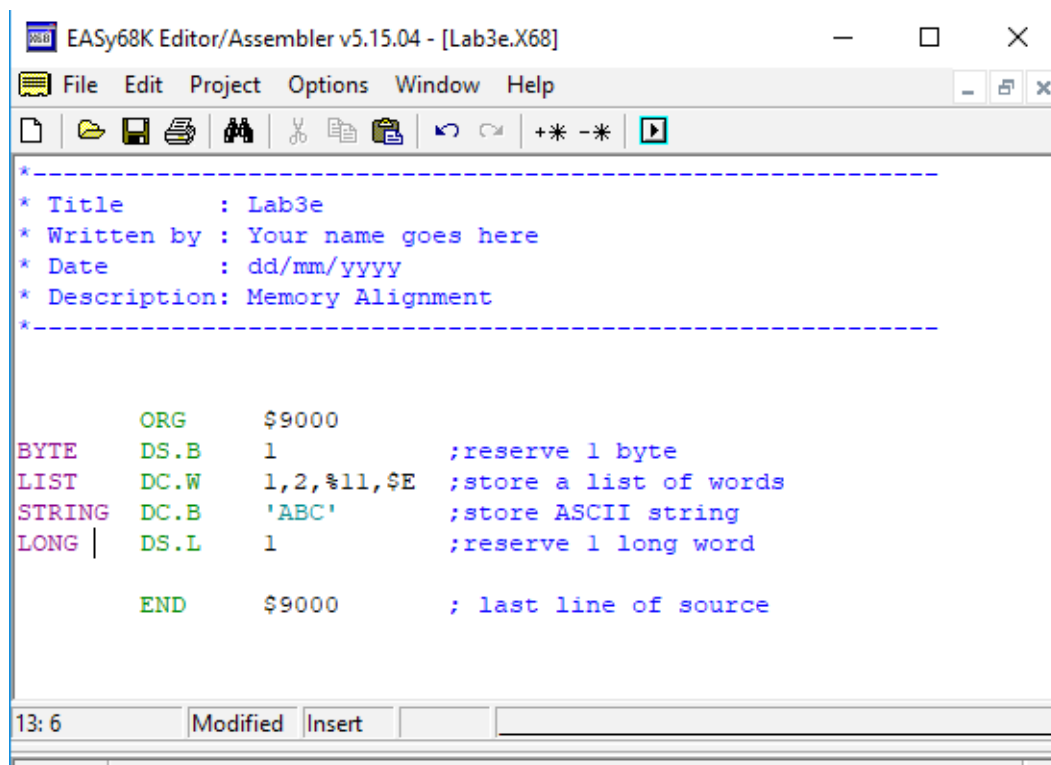
and the next item to be stored in memory is a word or a long word, the assembler first increments the location counter by 1 so that it contains an even address.

Step 1

Download the sample program called **Lab3e.X68** from the course website.

Step 2

Start Easy68K. Once running, load the file Lab3e.X68 using the File->Open File menu choice. You should see something similar to below. (Remember to properly comment your code.)

Step 3

Review the source code (do not assemble the program) for the previous program, and complete the memory map below. Remember to show the label beside the corresponding address, as illustrated in class. Also, show the contents of memory in hexadecimal. If you skip any bytes, you can leave the contents of those locations (in the map) blank. **[4 points]**

```
0x90000E FF
0x00900E FF
0x00900D FF
0x00900C 43
0x00900B 42
0x00900A 41
0x009009 0E
0x009008 00
0x009007 03
0x009006 00
0x009005 02
0x009004 00
0x009003 01
0x009002 00
0x009001 FF
0x009000 FF
```

Step 4

Now, assemble the program and invoke the simulator. Use the symbol table and the memory window to verify that your memory map is correct.

Part 6: Assemble-Time Expressions

As explained in class, assemble-time expressions are evaluated by the assembler (at assemble time) and produce a numeric value, which can then be used as a constant value in the source code. For example, the code below uses 3 assembler directives to define the length, width, and area of a rectangle.

```
LENGTH   EQU     25
WIDTH    EQU     17
AREA     EQU     LENGTH*WIDTH
```

An examination of the symbol table after assembly reveals that the value of the label AREA is 0x1A9 or 425 in decimal.

```
SYMBOL TABLE INFORMATION
Symbol-name          Value
--------------------------
AREA                 1A9
LENGTH               19
WIDTH                11
```

The resulting number can now be used in the assembly process; that is, whenever the programmer refers to the label AREA the assembler will replace the label with its numeric value, 0x1A9. This not only makes the program code easier to read, but also easier to maintain, as the programmer never needs to manually compute the value of AREA.

Step 1

Using Easy68K, create a source file (called **Lab3f.X68**) that contains the previous assembler directives. Assemble the program, and then examine the symbol table to verify that the assembler, at assembly time, computes the value of AREA.

Part 7: Absolute Long Addressing

Absolute-long addressing is one of the address modes that we have already been using. With this address mode, the data (byte, word, or long word) is contained in memory, and the effective address of the data is specified as a 32-bit address. At the assembler level, the effective address is specified either numerically or using a label. At the machine-code level, the effective address is stored in *two* extension words following the 16-bit operation word.

Step 1

Download the sample program called **Lab3g.X68** from the course website.

Step 2

Start Easy68K. Once running, load the file Lab3g.X68 using the File->Open File menu choice. You should see something similar to below. (Remember to properly comment your code.)

In the program above, LIST is a label with value 0x00009000. Four consecutive words are defined starting at this memory address. The first word has address 0x00009000, the second word has address 0x000090002, etc. Notice that although the program contains a single label (LIST) with the address of the first word, there are no other labels for accessing the remaining 3 words in the list. However, since the relative positions of the three remaining words in the list with respect to word 1 are known, the addresses of words 2, 3, and 4 can be expressed as assemble-time expressions, and computed by the assembler at assemble time, as illustrated in the code section of the program.

Step 3

Assemble the program, and then invoke the simulator. Now examine the listing file and write down the machine code for the MOVE and ADD instructions on lines 12 through 15 in the table below. Notice that each instruction consists of a 16-bit operation word, followed by *two* 16-bit extension words. **[2 points]**

Assembly Instruction	Operation Word	Extension words
MOVE.W LIST,D0	3039	0000 9000
ADD.W LIST+2,D0	D079	0000 9002
ADD.W LIST+4,D0	D079	0000 9004
ADD.W LIST+6,D0	D079	0000 9006

What do each of the four extension word pairs refer to? Be specific. **[4 points]**

> Each of the four extension words refer to an address in memory.
>
> Each of these memory addresses refers to the location from which the data is being pulled.
>
> LIST is stored in memory starting at location 00009000. Each word takes up 2 bytes of data in memory. Meaning that The second word is stored at 00009002, the next at 000090004, and the last at 00009006.

Step 4

Run the program in trace mode, and make sure that you understand exactly what the program is doing.

Part 8: Absolute Short Addressing
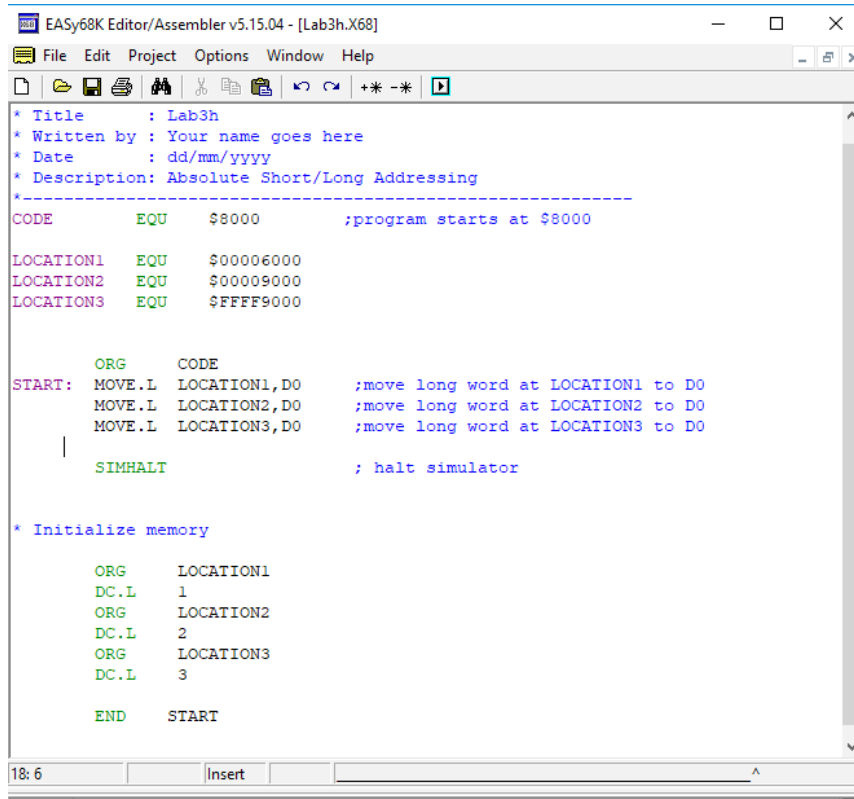
Absolute-short addressing is similar to absolute long addressing, but the effective address of the data (byte, word, or long word) contained in memory is specified as a 16-bit address, which is automatically sign-extended to 32-bits by the processor before it is used. At the assembler level, the effective address is specified either numerically or using a label. At the machine-code level, the effective address is stored in *one* extension word following the 16-bit operation word. This address mode results in shorter instructions and which execute faster than instructions that use absolute-long addressing. A drawback with this form of addressing is that the addressable memory range is limited to the lower (0x000000 – 0x0007FF) and upper (0xFF8000 – 0xFFFFFF) 32K bytes of memory.

Step 1

Download the sample program called **Lab3h.X68** from the course website.

Step 2

Start Easy68K. Once running, load the file Lab3h.X68 using the File->Open File menu choice. You should see something similar to below. (Remember to properly comment your code.)

```
EASy68K Editor/Assembler v5.15.04 - [Lab3h.X68]                    —  □  ×
File  Edit  Project  Options  Window  Help                              _ ₧ ×

* Title      : Lab3h
* Written by : Your name goes here
* Date       : dd/mm/yyyy
* Description: Absolute Short/Long Addressing
*-------------------------------------------------------
CODE        EQU    $8000          ;program starts at $8000

LOCATION1   EQU    $00006000
LOCATION2   EQU    $00009000
LOCATION3   EQU    $FFFF9000


        ORG     CODE
START:  MOVE.L  LOCATION1,D0     ;move long word at LOCATION1 to D0
        MOVE.L  LOCATION2,D0     ;move long word at LOCATION2 to D0
        MOVE.L  LOCATION3,D0     ;move long word at LOCATION3 to D0
    |

        SIMHALT                  ; halt simulator


* Initialize memory

        ORG     LOCATION1
        DC.L    1
        ORG     LOCATION2
        DC.L    2
        ORG     LOCATION3
        DC.L    3

        END     START

18: 6           Insert                                          ^
```

Step 3

Assemble the program, and then invoke the simulator. Now, examine the listing file and answer the questions below:

5. Is an absolute short or absolute long address mode used in the MOVE.L instruction on line 15 when accessing LOCATION1? Explain. **[2 points]**

> It is absolute short addressing because it is moving from a memory location (LOCATION1) to a data register (D0)

6. Is an absolute short or absolute long address mode used in the MOVE.L instruction on line 16 when accessing LOCATION2? Explain. **[2 points]**

This line will use absolute long addressing because this is outside of the first 32K bits of memory and not enough to be within the last 32K bytes.

7. Is an absolute short or absolute long address mode used in the MOVE.L instruction on line 17 when accessing LOCATION3? Explain. **[2 points]**

Line 17 uses absolute short addressing because it is within the last 32K bytes of memory.

Step 4

The Easy68K simulator displays the total cycle time required for (all) previously executed instructions, as illustrated below. (In actual hardware, each 68000 instruction takes a specific number of clock cycles to execute. Instructions that require more clock cycles have longer execution times compared to instructions with fewer clock cycles.) By tracing through a program one instruction at a time, it is possible to determine the number of clock cycles for each instruction (by observing the current cycle time, executing the next instruction, and then subtracting the new cycle time from the previous cycle time).

```
     T S   INT    XNZVC        Cycles
SR= 0010000000001000               52
```

Trace through the program and determine the number of clock cycles required to execute the following instructions: **[1.5 points]**

Assembly Instruction	Number of Clock Cycles in Instruction
MOVE.L LOCATION1,D0	16
MOVE.L LOCATION2,D0	20
MOVE.L LOCATION3,D0	16

8. Which of the previous three instructions has the longest execution time? Can you explain why this is so? **[2 points]**

The second instruction has the longest execution time because absolute long addressing is slower than absolute short addressing.

Part 9: Immediate Addressing

Immediate addressing is another address mode that we have already encountered. With immediate addressing the data (byte, word or long word) is stored in the instruction itself. At the assembler level, immediate addressing is specified by placing a # in front of the data. At the machine-code level, the data is stored in *one* or possibly *two* extension words following the 16-bit operation word, depending on the size of the data. Immediate addressing can only be used for the *source* operand.

Step 1

Download the sample program called **Lab3i.X68** from the course website.

Step 2

Start Easy68K. Once running, load the file Lab3i.X68 using the File->Open File menu choice. You should see something similar to below. (Remember to properly comment your code.)

Step 3

Assemble the program, and then invoke the simulator. Trace through the program step-by-step, and complete the table below showing the number of bytes in each instruction, the number of extension words used to store the immediate data in each instruction, the number of clock cycles required to execute each instruction, and the contents of D0 after each instruction executes. **[9 points]**

Instructions	Bytes	Extension Words	Clock Cycles	Contents of D0
MOVE.L #'JAYS',D0	6	2	12	4A415953
MOVE.B #VALUE,D0	4	1	8	4A415964
MOVE.W #VALUE,D0	4	1	8	4A410064
MOVE.L #VALUE,D0	2	0	4	00000064
MOVE.B #$64,D0	4	1	8	00000064
MOVE.B #%1100100,D0	4	1	8	00000064
MOVE.L #$FFFFFFFF,D0	2	0	4	FFFFFFFF
MOVEQ #$FF,D0	2	0	4	FFFFFFFF
MOVE.B #VALUE*2,D0	4	1	8	FFFFFFC8

9. After assembly, do the assembly instructions `MOVE.B #VALUE,D0`, `MOVE.B #$64,D0`, and `MOVE.B #%1100100,D0` all result in the same machine code being produced? Explain. **[2 points]**

> These all result in the same machine code being produced. Hex value $64 and binary value %1100100 both convert to decimal as 100. Therefore, all of the instructions will result in the same thing.

10. The assembly language instructions `MOVE.L #$FFFFFFFF,D0` and `MOVE.L #'JAYS',D0` both employ a long word data type, but do they have the same number of extension words? Explain. (*Hint: Pay close attention to the MOVEQ instruction on line 19, and described on pages 79 and 326 of your textbook.*) **[2 points]**

> They do not have the same number of extension words. By default, a MOVEQ instruction will sign extend any value. Thus, sign extending $FFFFFFFF will result in 00000000 which does not require an extension word.

The previous table in step 3 provides a sense of how the number of extension words associated with an instruction affects the number of clock cycles required to execute the instruction. In the 68000's ISA, reading/writing a byte/word from/to memory takes 4 clock cycles, while reading or writing a long word takes 8 clock cycles. An instruction like `MOVE.L #'JAYS',D0` consists of three memory-read cycles. As the instruction is three words long, the first memory-read cycle retrieves the operation word (`203C`) for the instruction. Then, the long word data (`4A415953`) is read. This takes two more memory read-cycles. The total number of clock cycles required to read, decode and execute the instruction is 12. In general, (shorter) instructions that use address modes that result in fewer or even no extension words execute faster than (longer) instructions that use address modes that result in one or more extension words.

Part 10: Address Registers

The 68000 ISA contains eight address registers referred to as A0 through A7. Unlike data registers, address registers do not contain data; rather they contain the addresses of memory locations that contain data. If this reminds you of a pointer in a high-level language, good, because address registers are used to implement pointers. And, just like pointers in high-level languages, address registers can be de-referenced in order to access the data they point to in memory.

Before proceeding, you are encouraged to review Section 2.2.2 in your textbook, which gives details about address registers and the addresses stored in them. Also, you are encouraged to review the unique instructions that operate only on the contents of address registers, including LEA (pages 80 and 312), MOVEA (pages 79 and 318), ADDA (page 269) and SUBA (page 354).

Part 11: Address Registers Indirect

As illustrated in Table 3, address register indirect is used to de-reference a pointer.

Table 3: Address Register Indirect

High-Level Code	68000 Assembler

`int C, data, *ptr;`
 	
`ptr = &data;`
 	
`C = *ptr;`
	`data   DS.L    1`
	`C      DS.L    1`
	
	`       LEA     data,A0`
	`       MOVE.W  (A0),C`

In the code sample above, two DS assembler directives reserve memory for the two variables C, and data. In this case, each variable of type int is treated as a 32-bit (i.e., long word) value. Address register A0 is used to implement *ptr, while the instruction LEA is used to initialize A0 with the value (i.e., address) of data. The pointer A0 is then de-referenced by enclosing the address register in round parentheses: (A0). The nature of the move operation is obvious – copy the long word whose (effective) address is contained in address register A0 to memory location C.
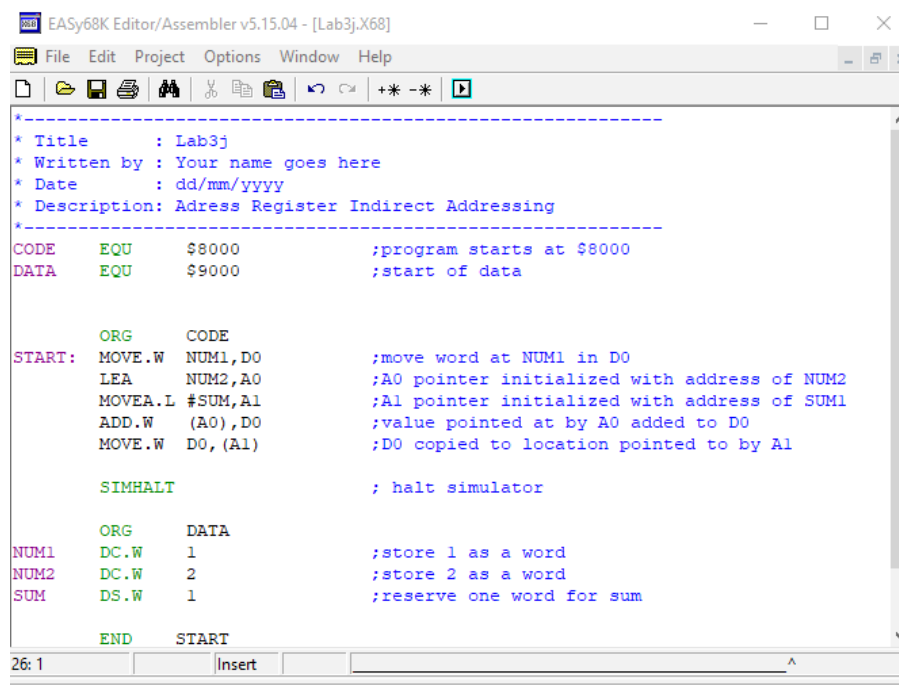
It should be clear from the previous example why indirect addressing is referred to as *indirect*. It is because accessing data requires two steps. The processor must first go to the address register to get the memory address of the data, and second it must go to memory and get the actual data.

Step 1

Download the sample program called **Lab3j.X68** from the course website.

Step 2

Start Easy68K. Once running, load the file Lab3j.X68 using the File->Open File menu choice. You should see something similar to below. (Remember to properly comment your code.)

The previous program illustrates two alternate ways of initializing an address register: LEA and MOVEA. The former instruction first computes the effective address of its source operand (e.g., the value of NUM2); then loads the value into the specified address register (e.g., A0). The latter instruction is a variant of the traditional move instruction, and is solely used to move values (e.g., the value of SUM) into an address register (e.g., A1). The difference between the two instructions is that LEA loads the *address* of the source operand into an address register, whereas MOVEA loads the *value* of the source operand into the address register. In some situations, like the one described here, it is possible to obtain the desired effect using either instruction. However, this is not always the case, as we will see later in the course.

Step 3

Assemble the program, and then invoke the simulator. Trace through the program step-by-step, and complete the table below showing the contents of the registers *after* each instruction executes. **[4 points]**

Instruction	Contents of D0	Contents of A1
`MOVE.W  NUM1,D0`	00000001	00000000
`LEA     NUM2,A0`	00000001	00000000
`MOVEA.L #SUM,A1`	00000001	00009004
`ADD.W   (A0),D0`	00000003	00009004

Step 4

Once the program finishes executing, use the memory window to determine the word value at address 0x00009004. What does this value represent? **[1 point]**

> The word value at 00009004 is 0003. This value represents D0

Step 5

Examine the listing file, and then write down the machine code for the two `MOVE.W` instructions on lines 12 and 16, respectively. What addressing mode does each instruction use when accessing memory? Which instruction do you think takes fewer clock cycles to execute? Why? **[4 points]**

> Line 12: 3039 00009000
> Line 16: 3280
>
> Line 12 uses absolute long addressing. Line 16 uses data register direct addressing. Data register direct takes fewer clock cycles because data is transferred between data registers. It is the fastest address mode.

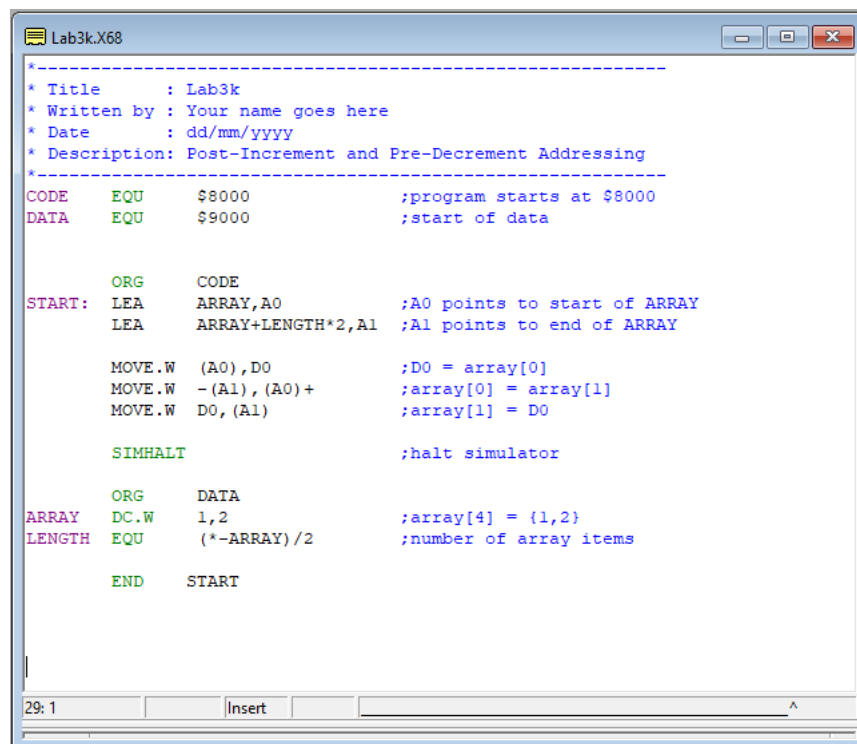## Part 12: Address Register Indirect with Post-Increment and Pre-Decrement

As explained in class, the *post-increment* address mode works the same way as indirect addressing, except that the contents of the address register are incremented to point at the next data item *after* execution of the instruction. (The 68000 automatically increments the address in the address register by 1, 2, or 4 depending on the data size specified in the instruction.) This address mode is useful when traversing arrays in a forward direction, as once the pointer (i.e., address register) is initialized with the address of the first item in the array it is automatically advanced with each access. The *pre-decrement* address mode operates in a similar way, but the contents of the address register are decremented to point to the previous data item *before* the instruction executes.

Step 1

Download the sample program called **Lab3k.X68** from the course website.

Step 2

Start Easy68K. Once running, load the file Lab3k.X68 using the File->Open File menu choice. You should see something similar to below. (Remember to properly comment your code.)

```
*-------------------------------------------------------------
* Title     : Lab3k
* Written by : Your name goes here
* Date      : dd/mm/yyyy
* Description: Post-Increment and Pre-Decrement Addressing
*-------------------------------------------------------------
CODE    EQU    $8000               ;program starts at $8000
DATA    EQU    $9000               ;start of data


        ORG    CODE
START:  LEA    ARRAY,A0            ;A0 points to start of ARRAY
        LEA    ARRAY+LENGTH*2,A1   ;A1 points to end of ARRAY

        MOVE.W (A0),D0             ;D0 = array[0]
        MOVE.W -(A1),(A0)+         ;array[0] = array[1]
        MOVE.W D0,(A1)             ;array[1] = D0

        SIMHALT                    ;halt simulator

        ORG    DATA
ARRAY   DC.W   1,2                 ;array[4] = {1,2}
LENGTH  EQU    (*-ARRAY)/2         ;number of array items

        END    START
```

Notice that in this program the * operator is *overloaded*. When used on line 27 to help define the value for label LENGTH, * refers to the current value of the assembler's location counter (see part 4). When used in the LEA instruction on line 12 to help compute the address of the end of the array, * acts as a multiplication operator.

Step 3

Before assembling the program, complete the memory map below for the data section of the code. Show the contents of memory in hexadecimal. **[3 points]**

```
0x009005 | 01 |
0x009004 | 00 |
0x009003 | 02 |
0x009002 | 00 |
0x009001 | 01 |
0x009000 | 00 |     ARRAY
```

Step 4

Assemble the program, and then invoke the simulator. Trace through the program step-by-step and complete the table below showing the contents of the registers *after* each instruction executes. **[7.5 points]**

Instruction	Contents of A0	Contents of A1	Contents of D0
LEA ARRAY,A0	00009000	00000000	00000000
LEA ARRAY+LENGTH*2,A1	00009000	00009004	00000000
MOVE.W (A0),D0	00009000	00009004	00000001
MOVE.W -(A1),(A0)+	00009002	00009002	00000001
MOVE.W D0,(A1)	00009002	00009002	00000001

Step 5

Once the program finishes executing, use the memory window to complete the memory map below. (Show the contents of memory in hexadecimal.) **[3 points]**

```
0x009005  FF
0x009004  FF
0x009003  01
0x009002  00
0x009001  02
0x009000  00          ARRAY
```

What does the program do? Be precise. **[2 points]**

> This program sets the value of D0 to the first value in the array. The program
> then sets the first value of the array equal to the second. The program then sets
> the value of the second item in the array to D0.

Step 7

Using the previous program as a starting point, make the following changes and save the program under the file name Lab3l.X68. The program must now work correctly on an array of four long words, using the same post-increment and pre-decrement addressing technique used in the original program. **[20 points]**

- You will need to change the data size indicator, where appropriate, to accommodate long words
- You will need to change the assemble-time expression to compute the number of array items.
- You will need to modify the assemble-time expression used to compute the address at the end of the array.
- You will need to write code to first operate on array elements 0 and 3, and then array items 1 and 2.

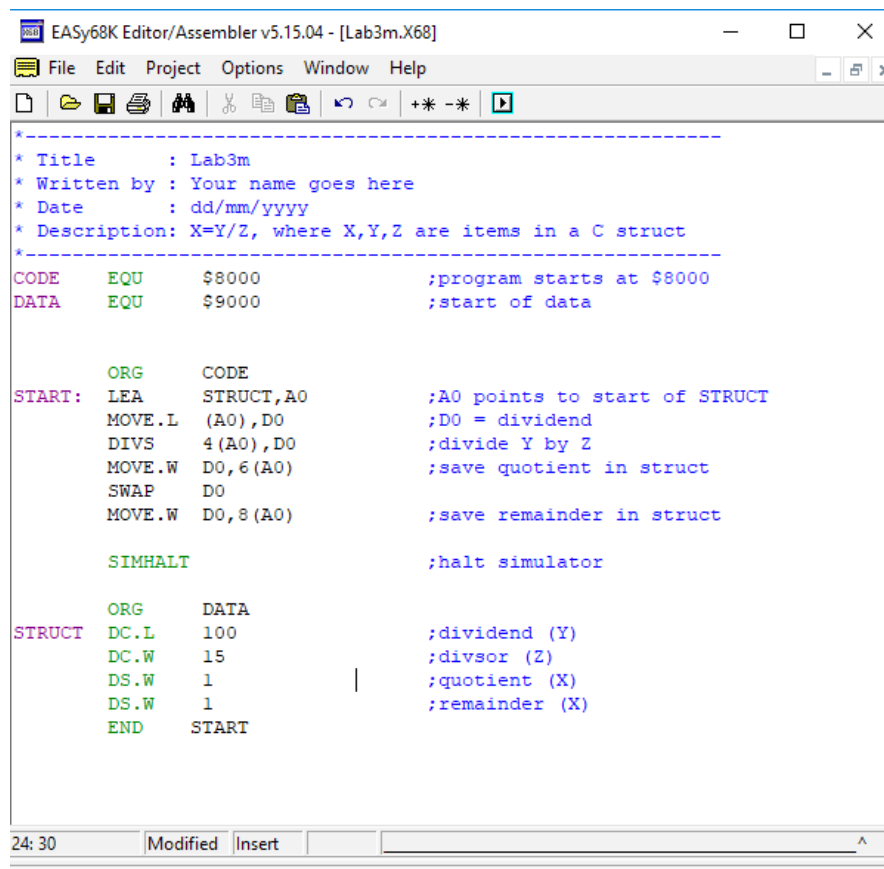## Part 13: Address Register Indirect with Displacement

As explained in class, the *displacement* address mode works the same way as indirect addressing, except that a 16-bit signed value is added to the contents of an address register to form the effective address of the data. Typically, this address mode is used in situations where a pointer (i.e., address register) is set to point to the start of a list of data (or structure) in memory, and then the displacement is used as an offset from the start of the list (or structure) in order to access a specific item.

Step 1

Download the sample program called **Lab3m.X68** from the course website.

Step 2

Start Easy68K. Once running, load the file Lab3m.X68 using the File->Open File menu choice. You should see something similar to below. (Remember to properly comment your code.)

The previous program computes the function X = Y/Z. Each of the previous variables is contained in a 10-byte C structure. The first 4 bytes in the structure contain the dividend, the next two bytes contain the divisor, the next 2 bytes are reserved for the quotient, and the remaining 2 bytes are reserved for the remainder.

Step 3

Before assembling the program, complete the memory map below for the data section of the code. Show the contents of memory in hexadecimal. **[5 points]**

```
0x009009  FF
0x009008  FF
0x009007  FF
0x009006  FF
0x009005  0F
0x009004  00
0x009003  64
0x009002  00
0x009001  00
0x009000  00          STRUCT
```

Step 4

Assemble the program, and then invoke the simulator. Trace through the program step-by-step and complete the table below showing the contents of the registers *after* each instruction executes. **[6 points]**

Instruction	Contents of A0	Contents of D0
LEA STRUCT,A0	00009000	00000000
MOVE.L (A0),D0	00009000	00000064
DIVS 4(A0),D0	00009000	000A0006
MOVE.W D0,6(A0)	00009000	000A0006
SWAP D0	00009000	0006000A
MOVE.W D0,8(A0)	00009000	0006000A

Make sure that you understand why the registers contain the values that they do.

Step 5

Once the program finishes executing, use the memory window to complete the memory map below. (Show the contents of memory in hexadecimal.) **[5 points]**

```
0x009009  0A
0x009008  00
0x009007  06
0x009006  00
0x009005  0F
0x009004  00
0x009003  64
0x009002  00
0x009001  00
0x009000  00        STRUCT
```

Part 14: Address Register Indirect with Index and Displacement

The *index with displacement* address mode combines some of the previous address modes into a single address mode. The effective address of the data is computed by adding the address in an address register, the 16-bit or 32-bit value in an index register, and an 8-bit displacement. The index register can either be an address register or a data register. As discussed in class, this address mode is extremely useful when seeking to access elements of a multi-dimensional array.

Step 1

Download the sample program called **Lab3n.X68** from the course website.

Step 2

Start Easy68K. Once running, load the file Lab3n.X68 using the File->Open File menu choice.

Step 3

Examine the code carefully. Notice that the program contains a statically allocated 4 x 4 array, called MATRIX, which contains the following 8-bit values:

$$\text{MATRIX}_{ij} = \begin{bmatrix} 0 & 1 & 2 & 3 \\ 4 & 5 & 6 & 7 \\ 8 & 9 & 10 & 11 \\ 12 & 13 & 14 & 15 \end{bmatrix}$$

Once running, the program prompts the user to enter a row number *i* and a column number *j*. (Both *i* and *j* should be in the range of 0 to 3.) The program then displays `MATRIX[i][j]` – the value of the byte at row *i* and column *j*, as shown below.


```
Sim68K I/O                                    —    □    ✕
Enter row (0-3): 2
Enter column (0-3): 3
Array Element: 11
```

Step 4

Run the program multiple times, each time trying different values for *i* and *j*. Make sure you understand how the program functions, before proceeding.

Step 5

Using the previous program as a starting point, modify the program so that each matrix element is now a long word rather than a byte. Then, update the program so that it works correctly with long words. (Note: You do not have to change the input and output code – it will continue to function correctly.) You will need to modify the code on lines 35 through 39 to take into account that each array element is 4 bytes rather than a single byte. Save the new program under the file name Lab3o.X68. **[20 points]**