Assembler Directives and Address Modes

CIS*2030
Lab Number 3

Name: Ankush Madharha

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`

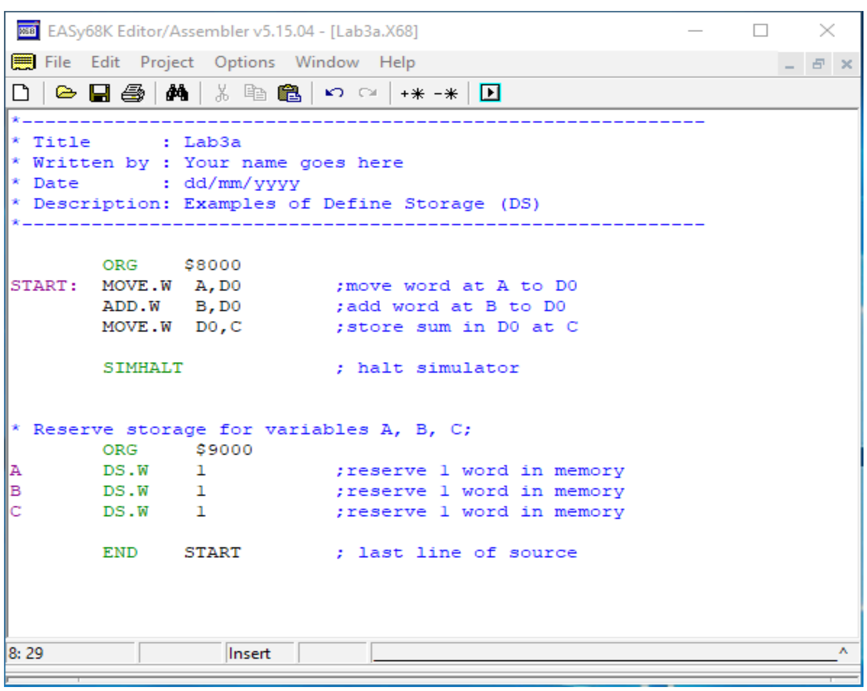
The first three DS assembler directives reserve memory for the three (uninitialized) global variables `A`, `B`, and `C`. Notice that in this case, each `int` variable is treated as a 16-bit (i.e., word) value, and that each variable can be referred to by its label (i.e., `A`, `B`, `C`). In general, the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



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`	0x0000 9000
`B`	0x0000 9002
`C`	0x0000 9004

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`	0xFF FF
`B`	0xFF FF
`C`	0xFF FF

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

The previous words are not all zero. DS does not initialize the memory it is reserving, and thus the default values FF are still the value of the words.

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]**

Once the program runs and completes execution, the value 0xFF FE should appear at address 0x0000 9004. This is because the following is happening: A (0xFF FF) + B (0xFF FF) = C ((1)FF FE).

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 changes when the size indicator is missing from the second DS directive. The assembler assumes the size indicator to be word if there is no size indicator. If we compare the address of label C with and without the size indicator W, it is the exact same (0x0000 9004).

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`

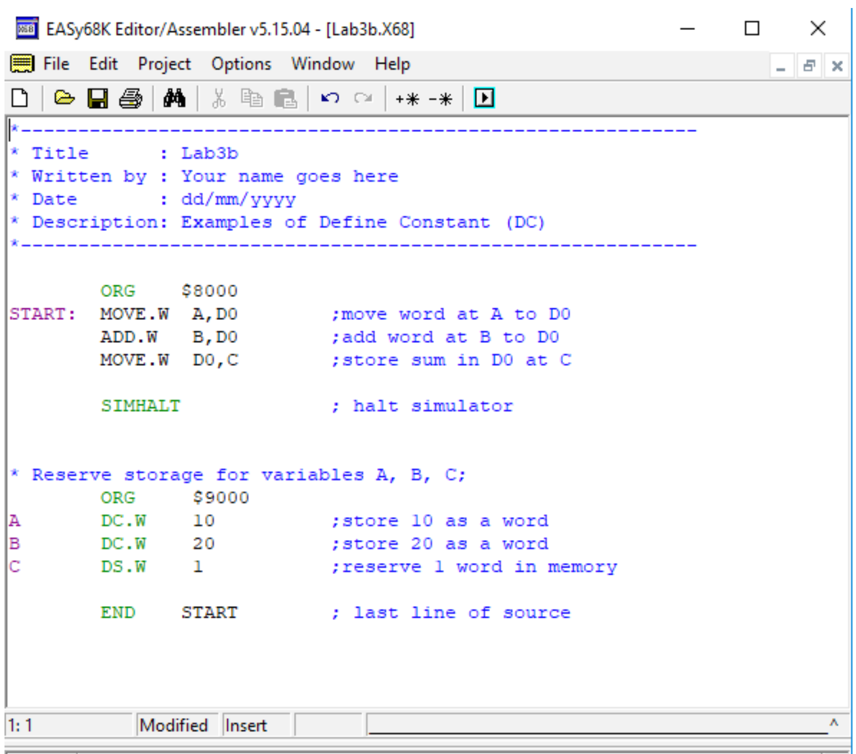
The first two DC directives cause the two 16-bit constants 10_{10} and 20_{10} to be loaded into memory at the current location during the assembly process. The remaining DS directive reserves space, again at assemble time, for a single word (i.e., 16-bit variable), but no information is stored in memory. 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.)



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	0x0000 9000	00 0A
B	0x0000 9002	00 14
C	0x0000 9004	FF FF

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]**

Once the program runs and completes execution, the value 0x00 1E should appear in memory at address 0x0000 9004. 000A + 0014 = 001E

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

Part 3: Using DC with Character Data

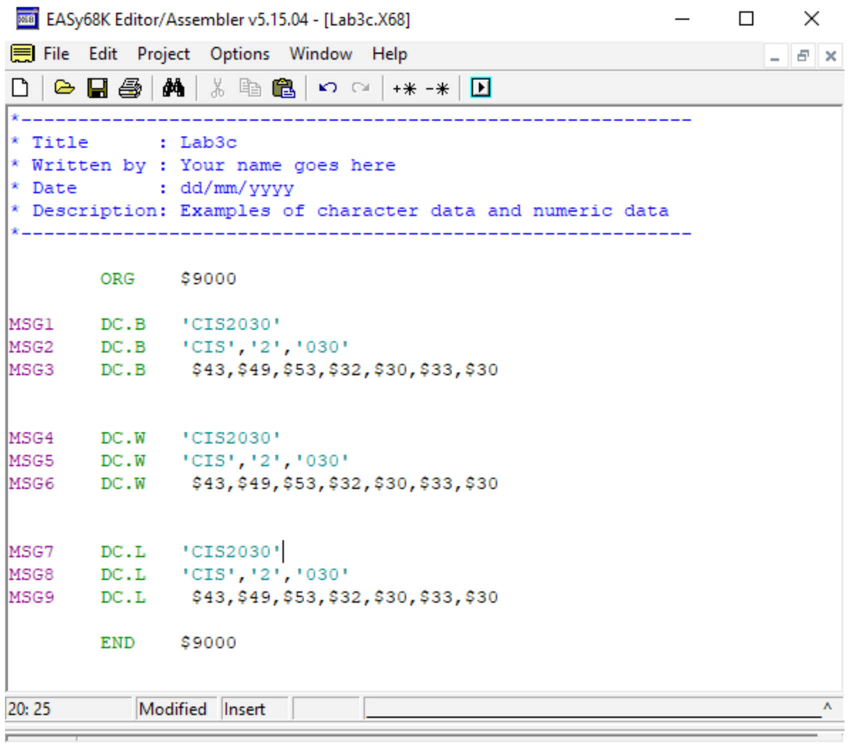
The DC directive can also be used to store ASCII characters in memory. ASCII codes can be defined as a series of *bytes* by including the characters within single quotation marks. However, if a single ASCII character is defined as a word (or a long word), the character is *left justified*.

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.)



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]**

MSG1 stores single ASCII values as 6 separate bytes in hex. MSG4 stores pairs of ASCII values as 4 separate words in hex. MSG7 stores four ASCII values as 2 separate long-words in hex.

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]**

If a single ASCII character is defined as a word (or a long word), the character is padding with zeros from the left. If the byte size indicator is used then the ASCII codes will be defined as a series of bytes.

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]**

MSG1 and MSG2 have the same representation in memory. MSG4 and MSG5 have different representations, since MSG5 has groups of characters, some of those group are shorter than a word and are padded with zeros. MSG7 and MSG8 have different representation for the same reason, the groups are shorter than a long-word are are padded with zeros (from the right). Note, MSG8 and MSG5 take up more memory than their counterparts MSG4 and MSG7.

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]**

MSG3 has the same representation in memory as the previous two. MSG6 has different representation as EACH ASCII character is padded with 2 zeros from the left because of the W size indicator. MSG9 has different representation as each ASCII character is padded with 6 zeros from the left because of the L size indicator.

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.

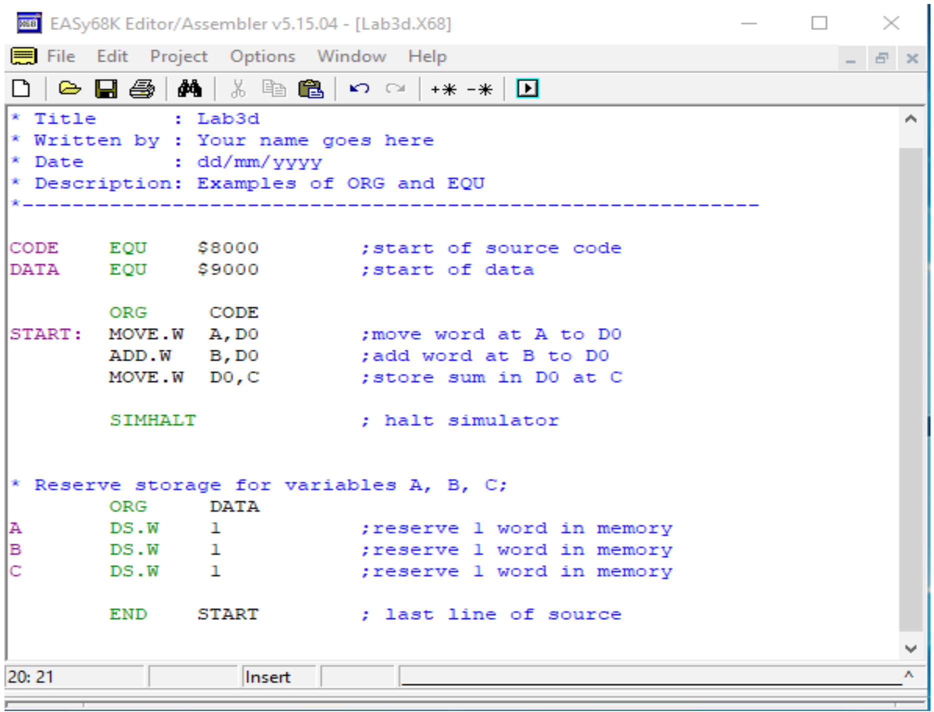
The ORG directive can be used to initialize the location counter. The directive is followed by a single argument, which is the address to be assigned to the location counter. By using multiple ORG directives throughout a source file, it is possible to store machine instructions or data structures at 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.)



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

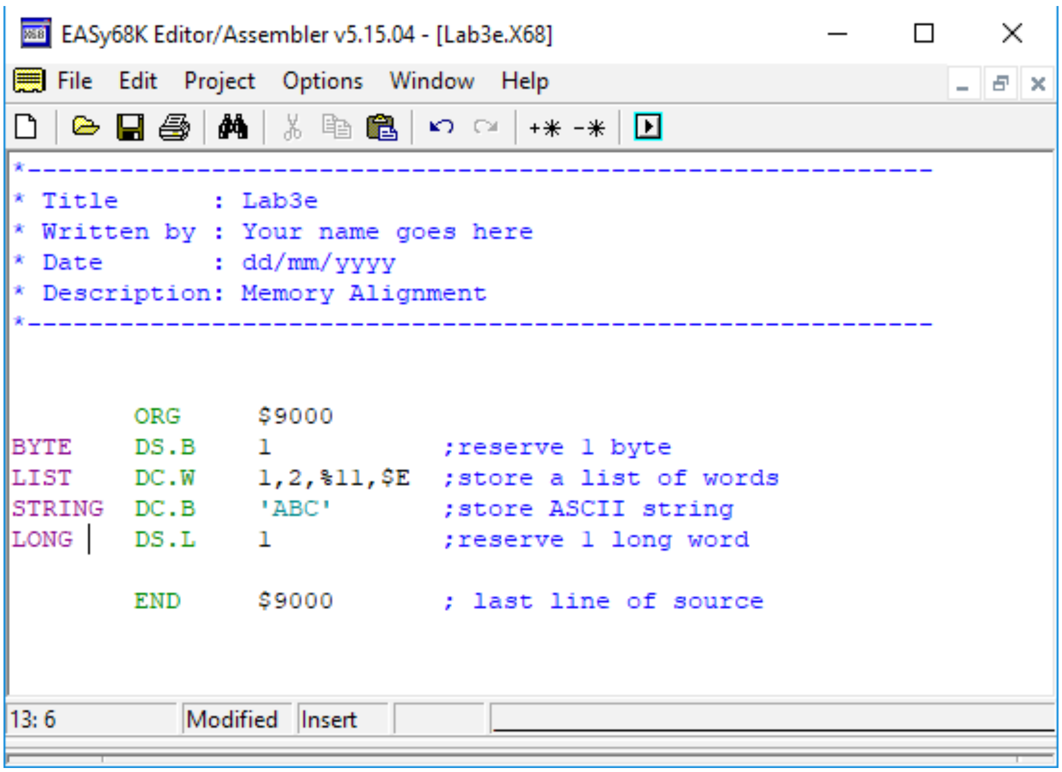
As explained in class, the 68000's ISA requires that word and long word boundaries be maintained. This means that although bytes can be stored at any address, both words and long words must be stored only at *even* addresses. This restriction causes the assembler to always ensure that both words and 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	0xFF	
0x00900E	0xFF	LONG
0x00900D		
0x00900C	0x43	
0x00900B	0x42	
0x00900A	0x41	STRING
0x009009	0x0E	
0x009008	0x00	LIST[3]
0x009007	0x03	
0x009006	0x00	LIST[2]
0x009005	0x02	
0x009004	0x00	LIST[1]
0x009003	0x01	
0x009002	0x00	LIST[0]
0x009001		
0x009000	0xFF	BYTE

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

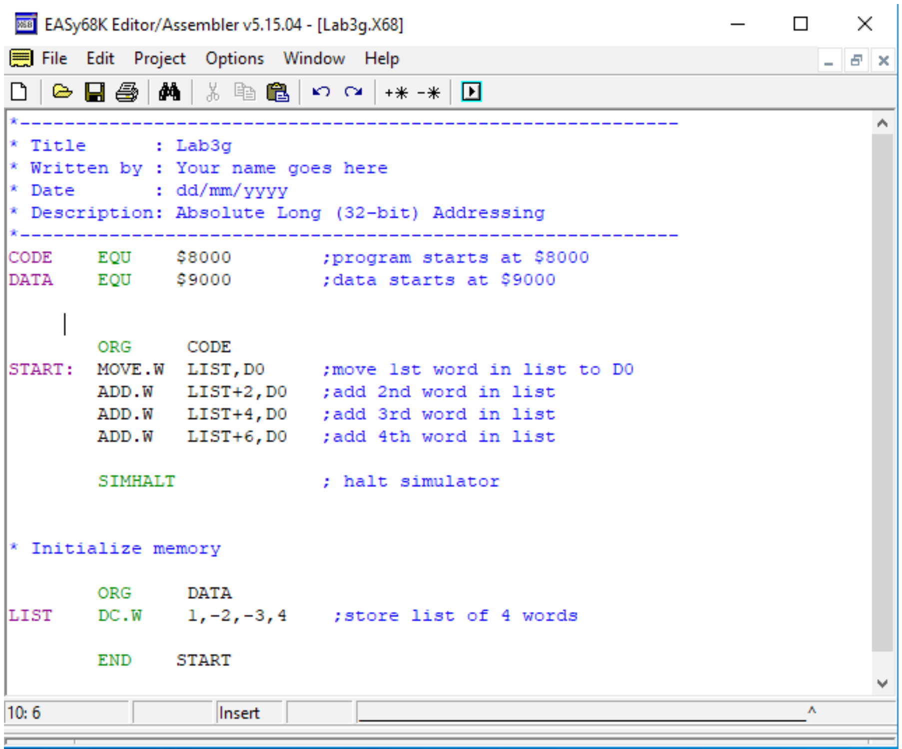
Absolute-long addressing is one of the address modes that we have already been using. With this address mode, the data (byte, word, or long word) is contained in memory, and the effective address of the data is specified as a 32-bit address. At the assembler level, the effective address is specified 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.)




```
*-----------------------------------------------------------
* Title      : Lab3g
* Written by : Your name goes here
* Date       : dd/mm/yyyy
* Description: Absolute Long (32-bit) Addressing
*-----------------------------------------------------------
CODE    EQU     $8000           ;program starts at $8000
DATA    EQU     $9000           ;data starts at $9000


        ORG      CODE
START:  MOVE.W   LIST,D0        ;move 1st word in list to D0
        ADD.W    LIST+2,D0      ;add 2nd word in list
        ADD.W    LIST+4,D0      ;add 3rd word in list
        ADD.W    LIST+6,D0      ;add 4th word in list

        SIMHALT                 ; halt simulator


* Initialize memory

        ORG      DATA
LIST    DC.W     1,-2,-3,4       ;store list of 4 words

        END     START
```

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 word pairs refer to the address of the source operand.
LIST: 0000 9000
LIST+2: 0000 9002
LIST+4: 0000 9004
LIST+6: 0000 9006

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

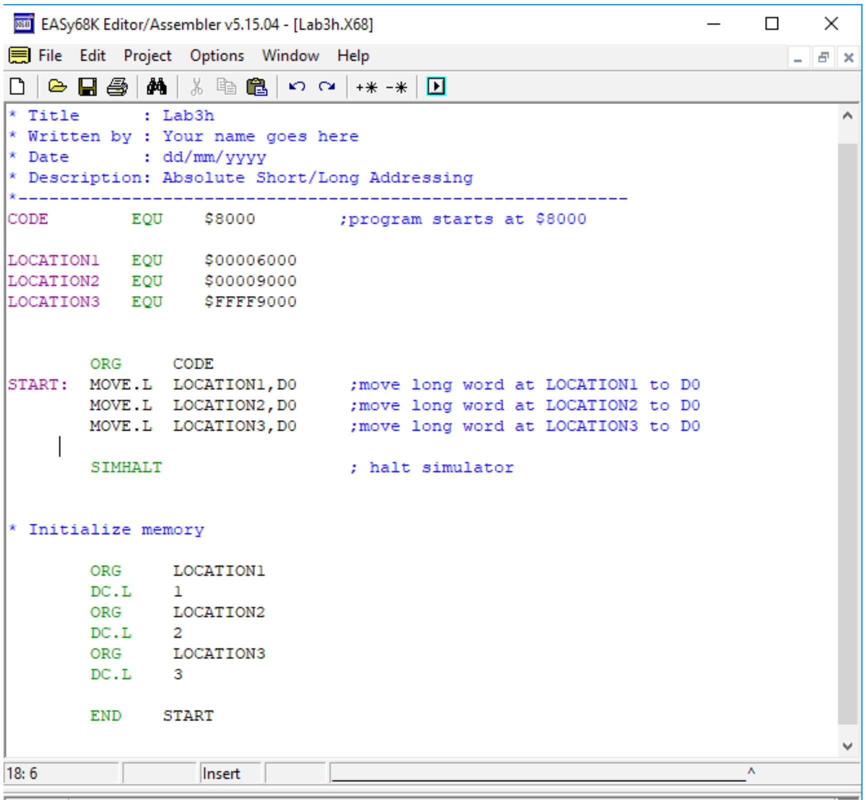
Absolute-short addressing is similar to absolute long addressing, but the effective address of the data (byte, word, or long word) contained in memory is specified as a 16-bit address, which is automatically sign-extended to 32-bits by the processor before it is used. At the assembler level, the effective 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.)




```
* 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
```

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]**

An absolute short address mode is used, this can be seen as the extension word is comprised of a single 16-bit value.

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]**

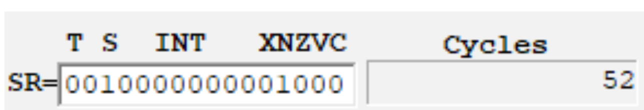
An absolute long address mode is used, this can be seen as the extension word is comprised of two 32-bit values.

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]**

An absolute short address mode is used, this can be seen as the extension word is comprised of a single 16-bit value.

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).



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. This is because absolute long addressing is used, an extra read occurs due to the extra word in the extension word.

Part 9: Immediate Addressing

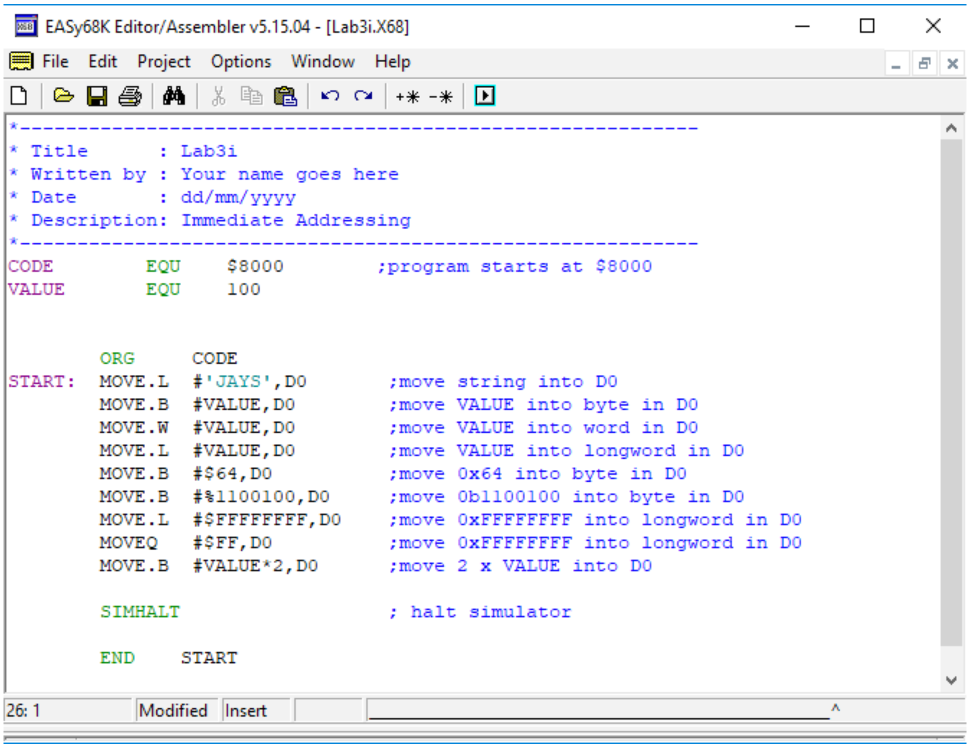
Immediate addressing is another address mode that we have already encountered. With immediate addressing the data (byte, word or long word) is stored in the instruction itself. At the assembler level, immediate addressing is specified by placing a # in front of the data. At the machine-code level, the 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.)



EASy68K Editor/Assembler v5.15.04 - [Lab3i.X68]

File Edit Project Options Window Help

```
*-----------------------------------------------------------
* Title      : Lab3i
* Written by : Your name goes here
* Date       : dd/mm/yyyy
* Description: Immediate Addressing
*-----------------------------------------------------------
CODE        EQU    $8000        ;program starts at $8000
VALUE       EQU    100


        ORG     CODE
START:  MOVE.L  #'JAYS',D0       ;move string into D0
        MOVE.B  #VALUE,D0        ;move VALUE into byte in D0
        MOVE.W  #VALUE,D0        ;move VALUE into word in D0
        MOVE.L  #VALUE,D0        ;move VALUE into longword in D0
        MOVE.B  #$64,D0          ;move 0x64 into byte in D0
        MOVE.B  #%1100100,D0     ;move 0b1100100 into byte in D0
        MOVE.L  #$FFFFFFFF,D0    ;move 0xFFFFFFFF into longword in D0
        MOVEQ   #$FF,D0          ;move 0xFFFFFFFF into longword in D0
        MOVE.B  #VALUE*2,D0      ;move 2 x VALUE into D0

        SIMHALT                  ; halt simulator

        END    START
```

26: 1 Modified Insert

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	1	12	4A41 5953
`MOVE.B #VALUE,D0`	4	1	8	4A41 5964
`MOVE.W #VALUE,D0`	4	1	8	4A41 0064
`MOVE.L #VALUE,D0`	2	0	4	0000 0064
`MOVE.B #$64,D0`	4	1	8	0000 0064
`MOVE.B #%1100100,D0`	4	1	8	0000 0064
`MOVE.L #$FFFFFFFF,D0`	2	0	4	FFFF FFFF
`MOVEQ #$FF,D0`	2	0	4	FFFF FFFF
`MOVE.B #VALUE*2,D0`	4	00C8	8	FFFF FFC8

Questions

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]**

   Yes, they all result in the same machine code being produced. The operation word is the same because we are using the MOVE instruction with the same size indicator B, and the source operand are all the same (64 when converted to hex) and the destination operand is the same (D0).

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. This is because the assembler is actually using MOVEQ when executing the first instruction. This is because MOVEQ encodes an 8-bit immediate source operand in the low-byte of the instruction word, with the destination being a data register. The 8-bit value FF can be sign extended to FFFF FFFF. The assembler converts MOVE to MOVEQ depending on the specification of the source and destination operands.

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`.

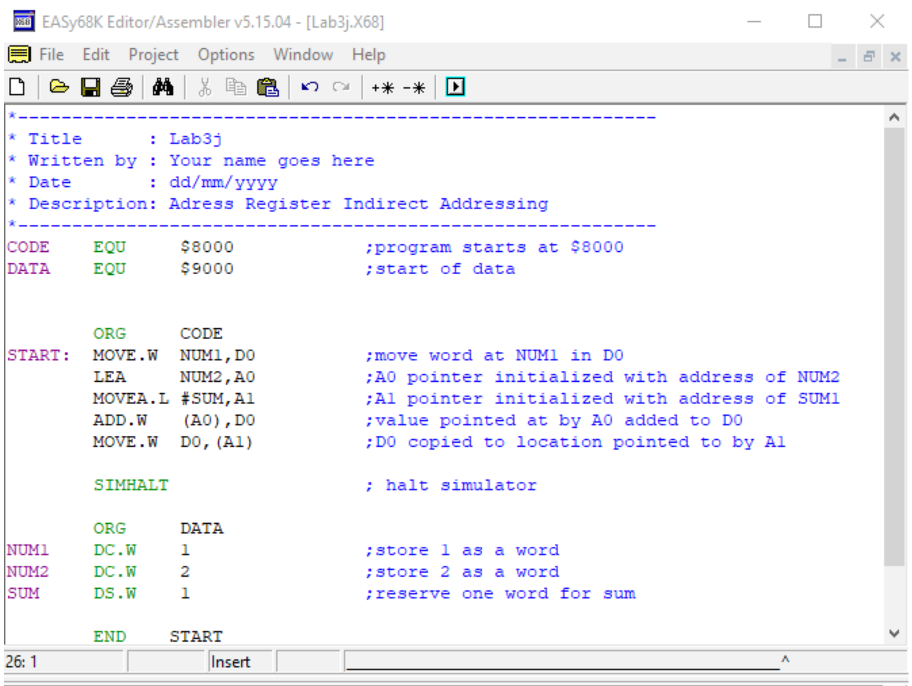
It should be clear from the previous example why indirect addressing is referred to as *indirect*. It is because accessing data requires two steps. The processor must first go to the address register to get the memory address of the data, and second it must go to memory and get the actual data.

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.)




```
*-----------------------------------------------------------
* Title      : Lab3j
* Written by : Your name goes here
* Date       : dd/mm/yyyy
* Description: Adress Register Indirect Addressing
*-----------------------------------------------------------
CODE    EQU     $8000           ;program starts at $8000
DATA    EQU     $9000           ;start of data


        ORG     CODE
START:  MOVE.W  NUM1,D0         ;move word at NUM1 in D0
        LEA     NUM2,A0         ;A0 pointer initialized with address of NUM2
        MOVEA.L #SUM,A1         ;A1 pointer initialized with address of SUM1
        ADD.W   (A0),D0         ;value pointed at by A0 added to D0
        MOVE.W  D0,(A1)         ;D0 copied to location pointed to by A1

        SIMHALT                 ; halt simulator

        ORG     DATA
NUM1    DC.W    1               ;store 1 as a word
NUM2    DC.W    2               ;store 2 as a word
SUM     DS.W    1               ;reserve one word for sum

        END    START
```

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`	0000 0001	0000 0000
`LEA      NUM2,A0`	0000 0001	0000 0000
`MOVEA.L #SUM,A1`	0000 0001	0000 9004
`ADD.W    (A0),D0`	0000 0003	0000 9004

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 this address is 0x00 03. This value represents the sum of the labels NUM1 and NUM2.

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 mode and line 16 uses data register direct. Line 16 will use fewer clock cycles to execute because data register direct addressing mode is much faster than absolute long.

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.)

```
Lab3k.X68
*-----------------------------------------------------------
* 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

29: 1          Insert
```

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]**

Address	Contents	
0x009005	0xFF	
0x009004	0xFF	
0x009003	0x02	
0x009002	0x00	
0x009001	0x01	
0x009000	0x00	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`	0000 9000	0000 0000	0000 0000
`LEA ARRAY+LENGTH*2,A1`	0000 9000	0000 9004	0000 0000
`MOVE.W (A0),D0`	0000 9000	0000 9004	0000 0001
`MOVE.W -(A1),(A0)+`	0000 9002	0000 9002	0000 0001
`MOVE.W D0,(A1)`	0000 9002	0000 9002	0000 0001

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]**

Address	Value	
0x009005	0xFF	
0x009004	0xFF	
0x009003	0x01	
0x009002	0x00	
0x009001	0x02	
0x009000	0x00	ARRAY

Step 6

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

> The program is swapping the two words "in the array". It switches the word at memory 0000 9000 with the word at memory location 0000 9002.

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

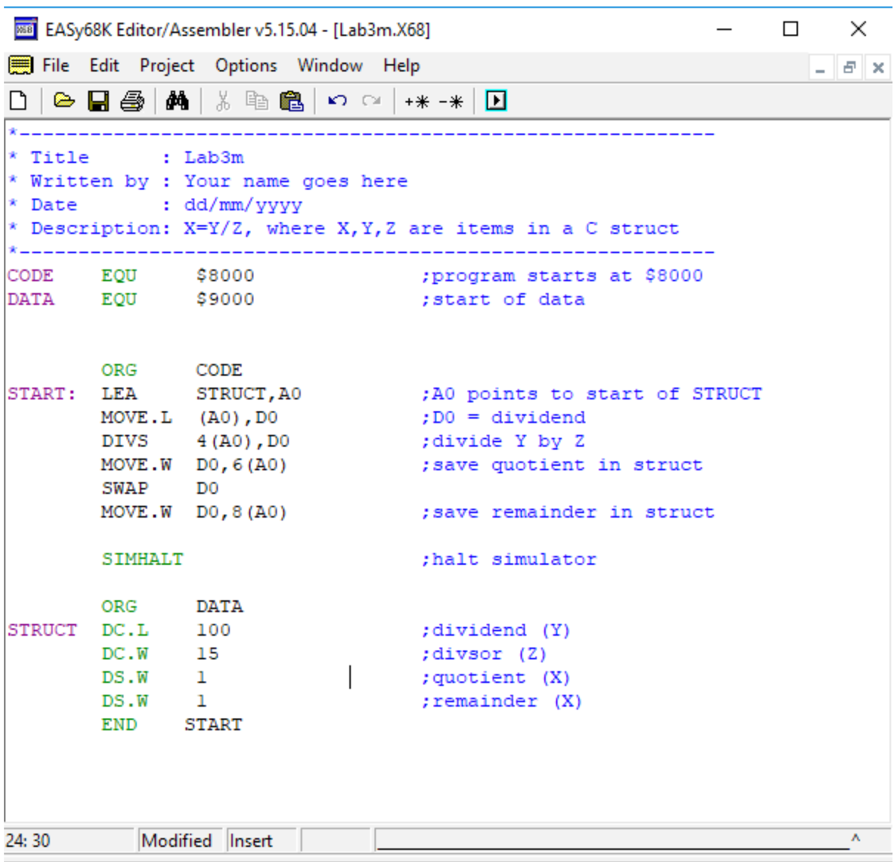
As explained in class, the *displacement* address mode works the same way as indirect addressing, except that a 16-bit signed value is added to the contents of an address register to form the effective address of the data. Typically, this address mode is used in situations where a pointer (i.e., address 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.)




```
*-----------------------------------------------------------
* Title      : Lab3m
* Written by : Your name goes here
* Date       : dd/mm/yyyy
* Description: X=Y/Z, where X,Y,Z are items in a C struct
*-----------------------------------------------------------
CODE    EQU     $8000                   ;program starts at $8000
DATA    EQU     $9000                   ;start of data


        ORG     CODE
START:  LEA     STRUCT,A0               ;A0 points to start of STRUCT
        MOVE.L  (A0),D0                 ;D0 = dividend
        DIVS    4(A0),D0                ;divide Y by Z
        MOVE.W  D0,6(A0)                ;save quotient in struct
        SWAP    D0
        MOVE.W  D0,8(A0)                ;save remainder in struct

        SIMHALT                         ;halt simulator

        ORG     DATA
STRUCT  DC.L    100                     ;dividend (Y)
        DC.W    15                      ;divsor (Z)
        DS.W    1                       ;quotient (X)
        DS.W    1                       ;remainder (X)
        END    START
```

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	0xFF	
0x009008	0xFF	
0x009007	0xFF	
0x009006	0xFF	
0x009005	0x0F	
0x009004	0x00	
0x009003	0x64	
0x009002	0x00	
0x009001	0x00	
0x009000	0x00	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`	0000 9000	0000 0000
`MOVE.L  (A0),D0`	0000 9000	0000 0064
`DIVS    4(A0),D0`	0000 9000	000A 0006
`MOVE.W  D0,6(A0)`	0000 9000	000A 0006
`SWAP    D0`	0000 9000	0006 000A
`MOVE.W  D0,8(A0)`	0000 9000	0006 000A

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	0x0A	
0x009008	0x00	
0x009007	0x06	
0x009006	0x00	
0x009005	0x0F	
0x009004	0x00	
0x009003	0x64	
0x009002	0x00	
0x009001	0x00	
0x009000	0x00	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:

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

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.

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]**