Data Types and Operations

Name: _____

Mark: _____/70

Overview

In this lab, we will continue our exploration of the 68000's ISA by considering some of its data types, data-movement instructions, and arithmetic instructions.

Objectives

Upon completion of this lab you will be able to:

- Understand how data-movement and arithmetic instructions (MOVE, SWAP, ADD, SUB, MUL, DIV, and EXT) function with different size data, and
- Understand how the flags in the condition-code register (CCR) are updated based on the result produced by different program instructions execution.

Preparation

Prior to commencing with 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 2.2.1 (Byte, Word, Longword Operations)
- Section 3.2.1 (MOVE, LEA)
- Section 3.2.2 (ADD, SUB, MULS, MULU, DIVU, DIVS, EXT)
- Section 3.2.4 (SWAP)
- Section 2.2.5 (Condition Code Bits)

Introduction

Now that we know how to edit, assemble, run and debug an assembly-language program using Easy68K, we will employ this knowledge as we explore the mechanics of some of the more commonly used data-movement and arithmetic instructions that are part the ISA of the 68000.

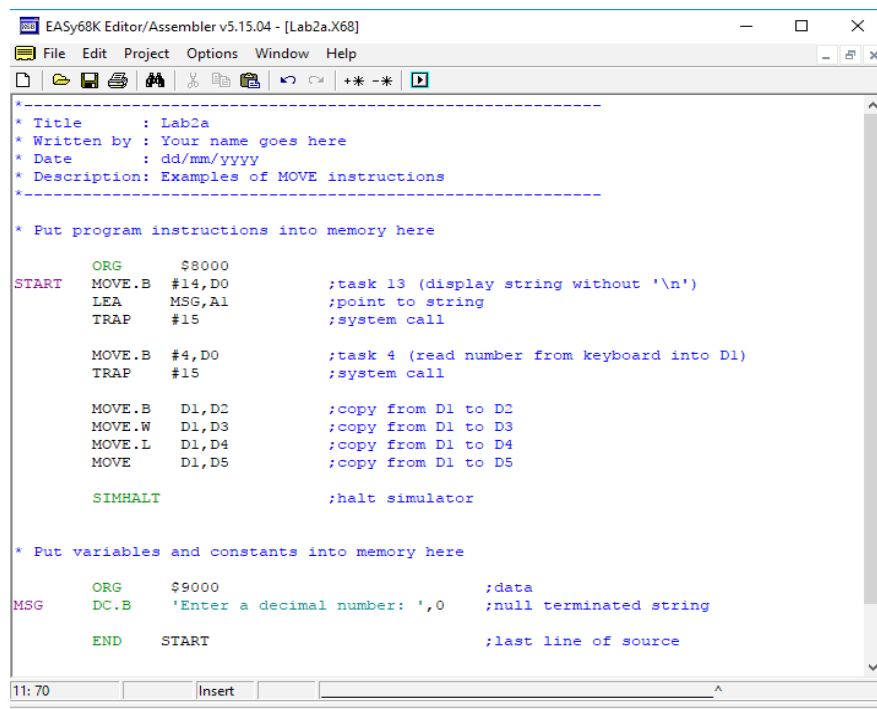## Part 1: Effects of different data types when moving data between registers

Step 1

Review the MOVE instruction described on page 316 of your textbook along with the examples given in Section 2.2.1.

Step 2

Download the sample program called **Lab2a.X68** from the course website.

Step 3

Start Easy68K. Once running, load the file Lab2a.X68 using the File->Open File menu choice. You should see something similar to below. (Remember to properly comment your code.)

```
*-----------------------------------------------------------
* Title      : Lab2a
* Written by : Your name goes here
* Date       : dd/mm/yyyy
* Description: Examples of MOVE instructions
*-----------------------------------------------------------

* Put program instructions into memory here

        ORG     $8000
START   MOVE.B  #14,D0          ;task 13 (display string without '\n')
        LEA     MSG,A1          ;point to string
        TRAP    #15             ;system call

        MOVE.B  #4,D0           ;task 4 (read number from keyboard into D1)
        TRAP    #15             ;system call

        MOVE.B  D1,D2           ;copy from D1 to D2
        MOVE.W  D1,D3           ;copy from D1 to D3
        MOVE.L  D1,D4           ;copy from D1 to D4
        MOVE    D1,D5           ;copy from D1 to D5

        SIMHALT                 ;halt simulator


* Put variables and constants into memory here

        ORG     $9000                       ;data
MSG     DC.B    'Enter a decimal number: ',0    ;null terminated string

        END     START                       ;last line of source
```
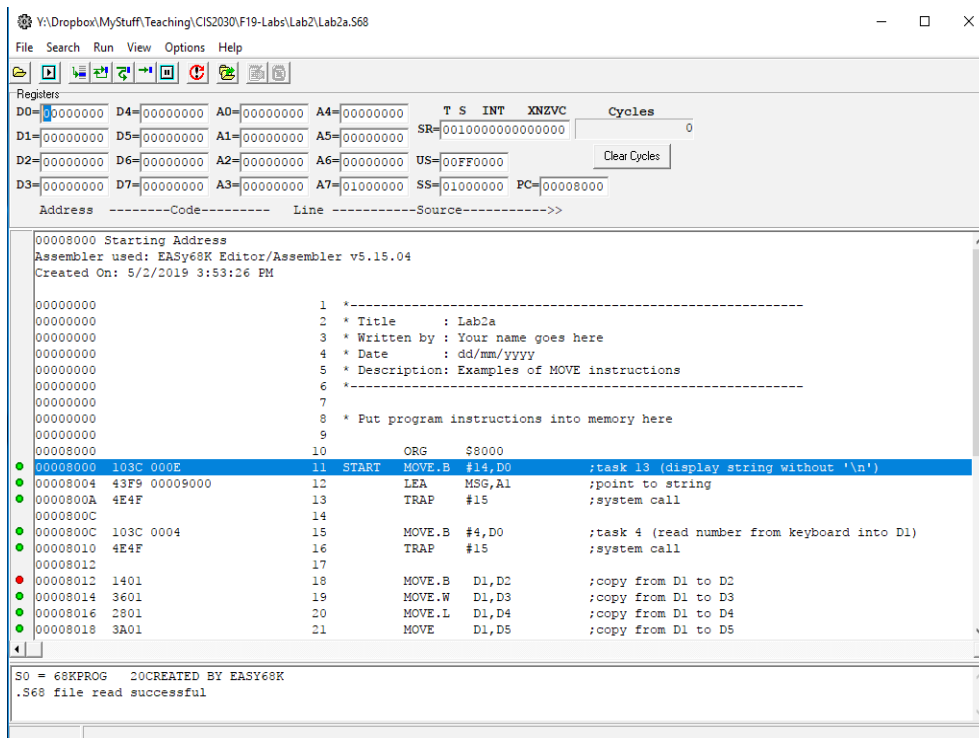
Step 3

Assemble the program, and then invoke the Easy68K simulator. Before running the program, set a breakpoint at the MOVE.B D1,D2 instruction on line 18. This can be done by using your mouse to click on the green dot on the left-hand-side of the window. The green dot will turn red, as illustrated below.

```
Y:\Dropbox\MyStuff\Teaching\CIS2030\F19-Labs\Lab2\Lab2a.S68                                    —    □    ×
File  Search  Run  View  Options  Help

Registers
D0=00000000  D4=00000000  A0=00000000  A4=00000000      T S  INT    XNZVC      Cycles
D1=00000000  D5=00000000  A1=00000000  A5=00000000   SR=0010000000000000              0
D2=00000000  D6=00000000  A2=00000000  A6=00000000   US=00FF0000              Clear Cycles
D3=00000000  D7=00000000  A3=00000000  A7=01000000   SS=01000000  PC=00008000

    Address  --------Code---------   Line ----------Source------------>>

00008000 Starting Address
Assembler used: EASy68K Editor/Assembler v5.15.04
Created On: 5/2/2019 3:53:26 PM

 00000000                            1  *---------------------------------------------------------------
 00000000                            2  * Title     : Lab2a
 00000000                            3  * Written by : Your name goes here
 00000000                            4  * Date       : dd/mm/yyyy
 00000000                            5  * Description: Examples of MOVE instructions
 00000000                            6  *---------------------------------------------------------------
 00000000                            7
 00000000                            8  * Put program instructions into memory here
 00000000                            9
 00008000                           10          ORG     $8000
 00008000 103C 000E                 11  START   MOVE.B  #14,D0        ;task 13 (display string without '\n')
 00008004 43F9 00009000             12          LEA     MSG,A1        ;point to string
 0000800A 4E4F                      13          TRAP    #15           ;system call
 0000800C                           14
 0000800C 103C 0004                 15          MOVE.B  #4,D0         ;task 4 (read number from keyboard into D1)
 00008010 4E4F                      16          TRAP    #15           ;system call
 00008012                           17
 00008012 1401                      18          MOVE.B  D1,D2         ;copy from D1 to D2
 00008014 3601                      19          MOVE.W  D1,D3         ;copy from D1 to D3
 00008016 2801                      20          MOVE.L  D1,D4         ;copy from D1 to D4
 00008018 3A01                      21          MOVE    D1,D5         ;copy from D1 to D5

S0 = 68KPROG   20CREATED BY EASY68K
.S68 file read successful
```

Now when you run the program, the simulator will execute all of the instructions *up to* the MOVE instruction on line 18.

Step 4

Run the program. When prompted in the console window to enter a decimal number, enter the following value: 305419896_{10}.

Step 5

The simulator should now be stopped at the breakpoint on line 18. What are the 32-bit hexadecimal values in data register D1 through D5? Record these values in the second column (labeled "Before") of the table below. **[2 points]**

Register	Before	After
D1		
D2		
D3		
D4		
D5		

Step 6

Now run the program to completion by pressing the run button. Once the program finishes executing, fill in the third column of the table above (labeled "After") with the new 32-bit hexadecimal values.

Questions

Now answer the following questions related to the previous program:

1. Why do data registers D2 and D3 contain different values even though according to the code they appear to be "copies" of D1? Be precise. **[1 point]**

2. Do data registers D3 and D5 contain the same or different values? If they contain the same values, explain why this is given that the assembly-language instructions on lines 19 and 21, respectively, of the listing file appear to be different from one another. Does the assembler make a size assumption? **[2 points]**

Part 2: Initializing registers using constant values

It is common practice to initialize registers prior to using them. For example, in the previous program, data register D1 was initialized with a 32-bit value entered by the user using the keyboard at runtime. Another common way to initialize registers prior to their first use is by using a MOVE instruction in combination with the *immediate* addressing to copy a constant value (specified at assemble time) into the register.
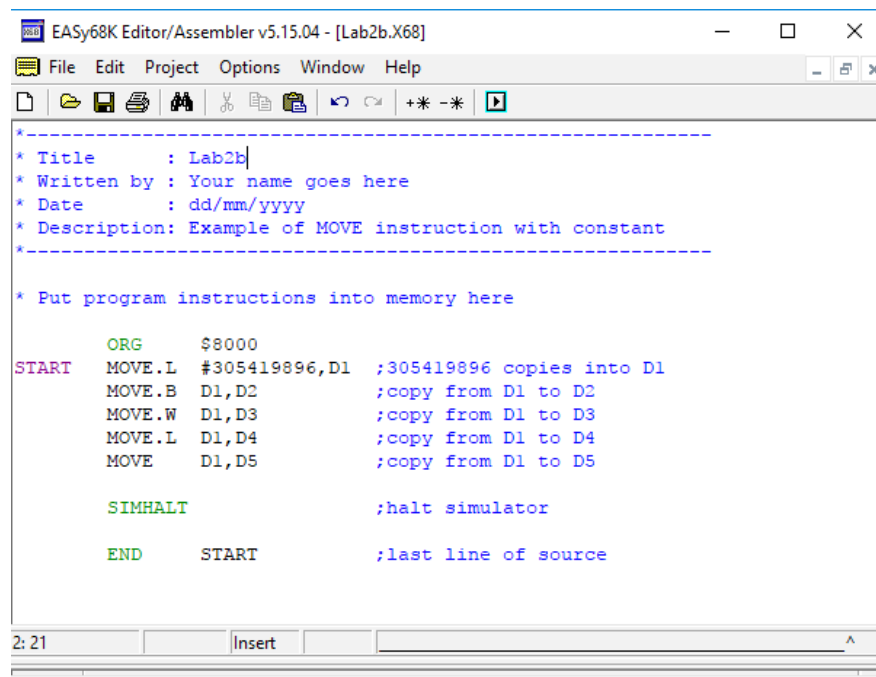
Recall from class, an immediate (or constant) value can be specified in an instruction that supports immediate addressing by placing a # before the numeric value. Moreover, numeric values can be specified either in hexadecimal, binary, or decimal by preceding the number with a $, %, or nothing, respectively. During the assembly process, the constant value is stored as part of the machine instruction using one (or two) extension word(s).

Step 1

Download the sample program called **Lab2b.X68** from the course website.

Step 2

Start Easy68K. Once running, load the file Lab2b.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 - [Lab2b.X68]               —    □    ×
File  Edit  Project  Options  Window  Help                        _  ꭍ  ×

*-----------------------------------------------------------
* Title      : Lab2b
* Written by : Your name goes here
* Date       : dd/mm/yyyy
* Description: Example of MOVE instruction with constant
*-----------------------------------------------------------

* Put program instructions into memory here

        ORG     $8000
START   MOVE.L  #305419896,D1   ;305419896 copies into D1
        MOVE.B  D1,D2           ;copy from D1 to D2
        MOVE.W  D1,D3           ;copy from D1 to D3
        MOVE.L  D1,D4           ;copy from D1 to D4
        MOVE    D1,D5           ;copy from D1 to D5

        SIMHALT                 ;halt simulator

        END     START           ;last line of source


2: 21                 Insert
```
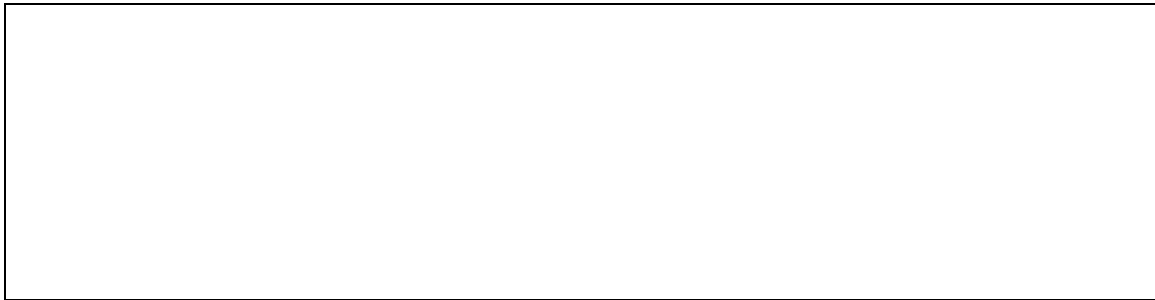
This program shows how the previous program (Lab2a.X68) can be adapted so that data register D1 is now initialized with the decimal value 305419896_{10} using a MOVE instruction. The # before the decimal value tells the assembler that the value is a constant specified by the programmer, and that the value is expressed in decimal versus binary (%) or hexadecimal ($). Notice that in this program the value to be loaded into D1 is known at assembly time, whereas the value loaded into D1 in the previous program was not known until runtime.

Step 3

Assemble the program. Line 11 of the listing file shows the machine code generated for the MOVE instruction used to initialize data register D1. What does the hexadecimal value 12345678 in the machine code represent? **[1 point]**
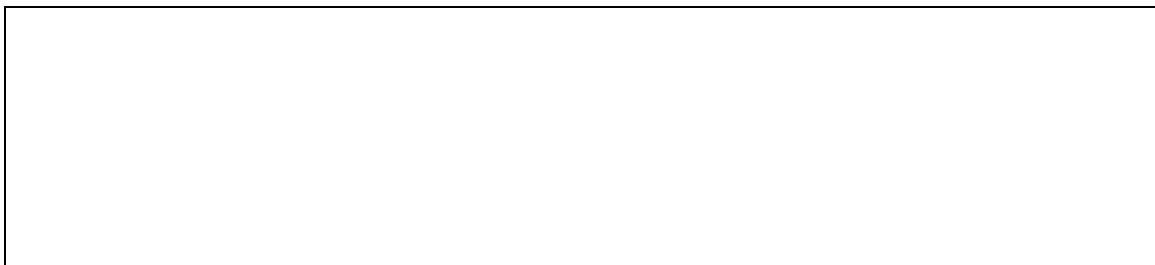
Step 4

Now run the program to completion by pressing the run button. Examine the results produced by the program and confirm to yourself that the program produces the same results as the previous program (Lab2a.X68).

Questions

Now answer the following questions related to the previous program:

3. Explain why the assembler instruction MOVE.L #305419896,D1 uses a .L suffix rather than a .W or .B suffix. **[1 point]**

4. Using the editor, change the previous instruction to `MOVE.W #305419896,D1`. Now try assembling the program. What happens? **[1 point]**

5. Re-write the assembler instruction `MOVE.L #305419896,D1`, but this time express the constant 305419896_{10} in hexadecimal. **[1 point]**

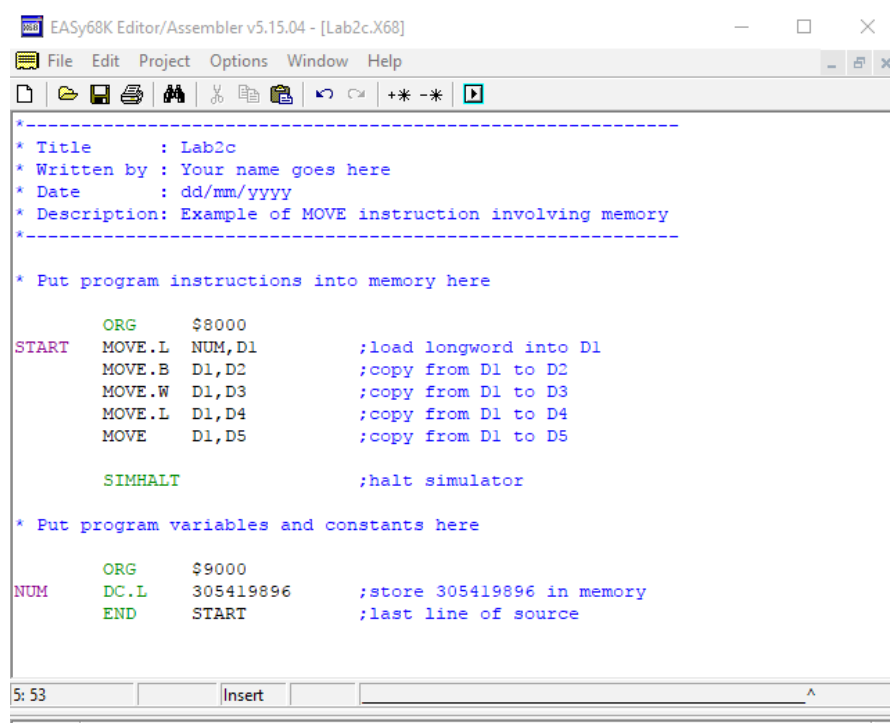
Part 3: Initializing registers using values in memory

Another common way to initialize a register is to load it with a value already present in memory. Using a MOVE instruction in conjunction with *absolute* addressing can do this. With absolute addressing the instruction uses the actual memory address of the value. At the assembly level, the programmer usually uses a label for the address rather than the actual numeric address. (This way, the programmer does not have to keep track of numeric addresses and can leave that to the assembler.) Once assembled, the address is stored in one (or two) extension word(s) following the instruction's operation word.

Step 1

Download the sample program called **Lab2c.X68** from the course website.

Step 2

Start Easy68K. Once running, load the file Lab2c.X68 using the File->Open File menu choice. You should see something similar to below. (Remember to properly comment your code.)

```
*------------------------------------------------------------
* Title      : Lab2c
* Written by : Your name goes here
* Date       : dd/mm/yyyy
* Description: Example of MOVE instruction involving memory
*------------------------------------------------------------

* Put program instructions into memory here

        ORG     $8000
START   MOVE.L  NUM,D1          ;load longword into D1
        MOVE.B  D1,D2           ;copy from D1 to D2
        MOVE.W  D1,D3           ;copy from D1 to D3
        MOVE.L  D1,D4           ;copy from D1 to D4
        MOVE    D1,D5           ;copy from D1 to D5

        SIMHALT                 ;halt simulator

* Put program variables and constants here

        ORG     $9000
NUM     DC.L    305419896       ;store 305419896 in memory
        END     START           ;last line of source
```

This program shows how the original program (Lab2a.X68) can be adapted so that data register D1 is now initialized with the decimal value 305419896_{10} that is present in memory at the time that the program starts running. The instruction MOVE.L NUM,D1 means copy the 32-bit contents of memory location NUM into data register D1. The absence of a # before the label in the instruction tells the assembler to treat the label as a memory address rather than a constant.

Step 3

Assemble the program. Line 11 of the listing file shows the machine code generated for the MOVE instruction used to initialize data register D1. What does the hexadecimal value 00009000 in the machine code represent? **[1 point]**

Step 4

Now run the program to completion by pressing the run button. Examine the output produced by the program and confirm to yourself that the program produces the same results as the original program (Lab2a.X68).

Step 5

On line 22 of the current program, you should see the following assembler statement:

```
NUM     DC.L    305419896
```

Replace the statement on line 22 with the following:

```
NUM     DC.W    305419896
```

Now try assembling your program. Do you get any assembly errors? Explain. **[1 point]**

Part 4: Addition instruction
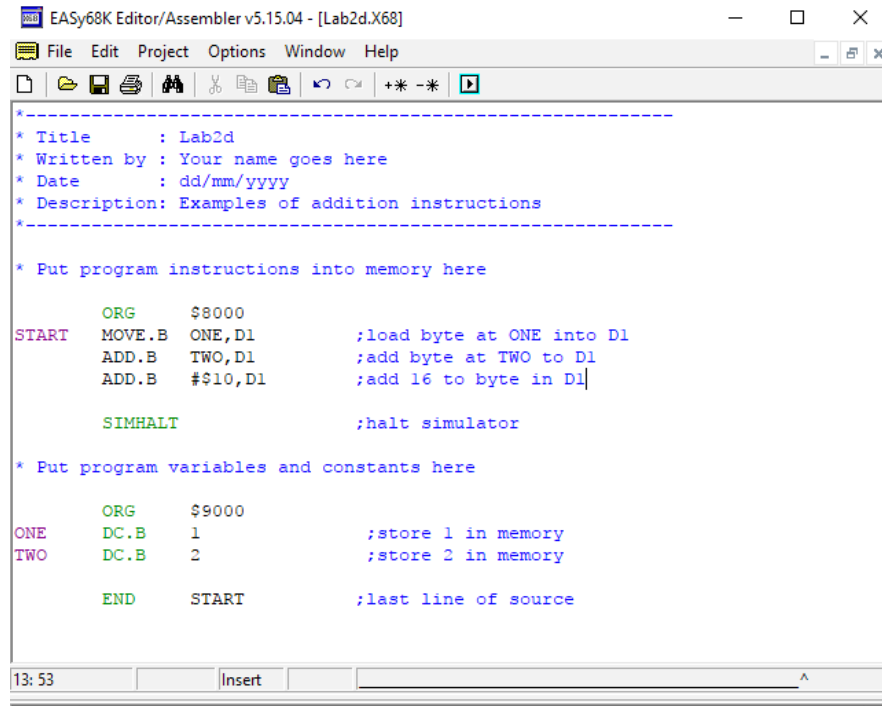
Next to the MOVE instruction, the most frequently used instruction is the basic ADD instruction. **Review the ADD instruction described on page 267 of your textbook**.

Step 1

Download the sample program called **Lab2d.X68** from the course website.

Step 2

Start Easy68K. Once running, load the file Lab2d.X68 using the File->Open File menu choice. You should see something similar to below. (Remember to properly comment your code.)
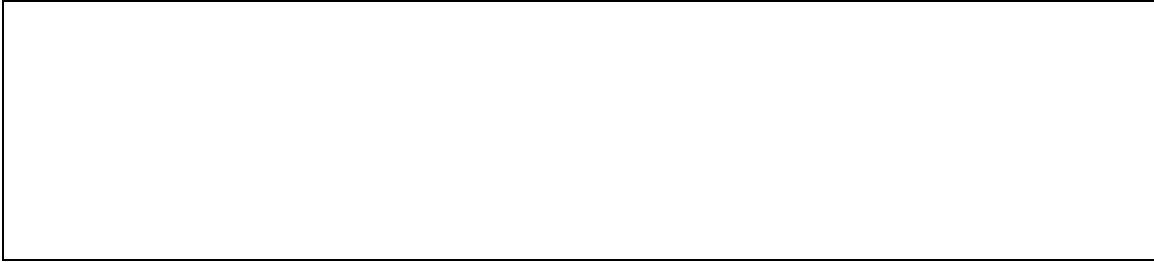
The program adds the 8-bit value at location ONE to the 8-bit value at location TWO, and then adds 16 to form a final sum which is saved in data register D1.

Step 3

Assemble the program and examine the listing file. What do the values 00009000, 00009001, 0010 on lines 11, 12, and 13, respectively, of the listing file refer to? **[2 points]**

Step 4

Run the program, and make sure you understand how it is functioning. Are the values being added considered to be signed or unsigned? Explain. **[1 point]**

```
┌─────────────────────────────────────────────────────────┐
│                                                           │
│                                                           │
│                                                           │
│                                                           │
│                                                           │
│                                                           │
│                                                           │
└─────────────────────────────────────────────────────────┘
```

Part 5: Subtract instruction

Unlike addition, subtraction is a non-commutative operation. Therefore, the order of the operands matters when performing a subtraction operation. **Review the SUB instruction described on page 352 of your textbook**.

Step 1

Download the sample program called **Lab2e.X68** from the course website.

Step 2

Start Easy68K. Once running, load the file Lab2e.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 - [Lab2e.X68]            —    □    ✕
File  Edit  Project  Options  Window  Help                    _ ⊕ ✕

*---------------------------------------------------------
* Title     : Lab2e
* Written by : Your name goes here
* Date      : dd/mm/yyyy
* Description: Example of subtraction
*---------------------------------------------------------

* Put program instructions into memory here

        ORG     $8000
START   MOVE.L  COUNT,D1        ;load longword at COUNT into D1
        MOVE.L  #COUNT,D2       ;load COUNT into D1
        SUB.L   D1,D2           ;subtract D1 from D2

        SIMHALT                 ;halt simulator

* Put program variables and constants here

        ORG     $9000
COUNT   DC.L    $1234           ;store 1234 in memory

        END     START           ;last line of source

1: 1              Insert
```

Trace through the program, and fill in the table below *after* each instruction executes. Show the full 32-bit hexadecimal contents of each register. **[3 points]**

Instruction	D1	D2
MOVE.L COUNT,D1		
MOVE.L #COUNT,D2		
SUB.L D1,D2		

Questions

Now answer the following questions related to the previous program:

6. Immediately after both MOVE instructions (lines 11 and 12 of the listing file) are executed, D1 and D2 contain different values despite both instructions referencing the same label COUNT. Explain why this is. **[1 point]**

7. Demonstrate to yourself that the *final* 32-bit hexadecimal value in D2 is correct by performing the subtraction operation by hand in the box below. Remember to use all 32-bits for both operands, and to do the calculation the same way the machine would – that is, using 2's complement arithmetic. Show your work. **[2 points]**

Part 6: EXT instruction

As explained in class, addition (and subtraction) instructions require the operands and result to be of the *same* data size. This begs the question, "What if I want to add a byte to a word, or a word to a long word?" This question is not as simple to answer as you may think! First, we must decide if we are treating the data as signed or unsigned. If signed, we can use the EXT instruction to sign extend a byte to a word or a word to a long word. Once the operands have the same size, we can proceed with the addition (or subtraction) operation. **Review the EXT instruction described on pages 85 and 308 of your textbook.**

Step 1

Download the sample program called **Lab2f.X68** from the course website.

Step 2

Start Easy68K. Once running, load the file Lab2f.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 - [Lab2f.X68]                    —  □  ×
File  Edit  Project  Options  Window  Help                        _  ⊡  ×
*-----------------------------------------------------------
* Title      : Lab2f
* Written by : Your name goes here
* Date       : dd/mm/yyyy
* Description: Adding numbers with different data sizes
*-----------------------------------------------------------

* Put program instructions into memory here

        ORG     $8000
START   MOVE.B  BYTE,D0        ;load byte at BYTE into D0
        EXT.W   D0             ;sign extend byte to word
        ADD.W   D0,WORD        ;add words and store sum at WORD

        SIMHALT                ;halt simulator

* Put program variables and constants here

        ORG     $9000
BYTE    DC.B    -1             ;store -1 in memory
WORD    DC.W    13             ;store 13 in memory
        END     START          ;last line of source

1: 1                  Insert
```

Step 3

Trace through the program, and fill in the table below with the value in data register D0 *before* and *after* each instruction executes. Show the full 32-bit hexadecimal contents of each register. **[2 points]**

Instruction	D0 (Before)	D0 (After)
MOVE.B BYTE,D0		
EXT.W D0		

Step 4

Now convert the hexadecimal values recorded in the last column of the previous table to decimal. (Assume the values are signed (i.e., in 2's-complement form), and perform the calculation by hand showing your work.) Do both values have the same decimal value? **[3 points]**

Explain why the label WORD has the hexadecimal address 00009002. What is the final decimal value stored at 00009002? Explain. **[3 points]**

```
```

Part 7: DIVU and DIVS instructions

As explained in class, division operations are complicated by the fact that there are two distinct division instructions – one for unsigned numbers and one for signed numbers – and by the fact that both the operands and result are mixed data sizes. With regards to the former, the unsigned division instruction (DIVU) must be used with unsigned data, while division operations performed on signed data must be performed using the signed (DIVS) division instruction. With regards to the latter, the dividend is a long word, the divisor is a word, and the result is two words: the upper word (bit positions 16 through 31) contains the remainder, while the lower word (bit positions 0 through 15) contains the quotient.

Due to these mixed data sizes, the previous EXT instruction is often used to prepare the operands prior to performing a division operation. Two other useful instructions are the CLR instruction and the SWAP instruction. The CLR instruction is used to set the contents of a register or memory location to zero, while the SWAP instruction is used to swap the upper and lower words in a data register.

Before proceeding, **review the signed and unsigned division on pages 84, 299 and 301** of your textbook, the **CLR instruction on pages 84 and 92**, and the **SWAP instruction on pages 86 and 358**.

Step 1

Download the sample program called **Lab2g.X68** from the course website.

Step 2

Start Easy68K. Once running, load the file Lab2g.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 - [Lab2g.X68]                    —    □    ×
File   Edit   Project   Options   Window   Help                         _ ᗺ ×

*----------------------------------------------------------
* Title      : Lab2g
* Written by : Your name goes here
* Date       : dd/mm/yyyy
* Description: Division, sign-extension, and swapping
*----------------------------------------------------------

* Put program instructions into memory here

        ORG     $8000
START   MOVE.B  DIVIDEND,D0    ;load byte at DIVIDEND into D0
        EXT.L   D0             ;sign extend word to longword
        DIVS    #15,D0         ;divide longword by 15
        CLR.L   D1             ;clear D1
        MOVE.W  D0,D1          ;move quotient to D1
        CLR.L   D2             ;clear D2
        SWAP    D0             ;swap quotient and remainder
        MOVE.W  D0,D2          ;move result to D2
        SIMHALT                ;halt simulator

* Put program variables and constants here

        ORG     $9000
DIVIDEND DC.B   -100           ;store -100 in memory
        END     START          ;last line of source

24: 15              Insert
```

Step 3

Trace through the program, and fill in the table below with the value in data registers D0, D1 and D2 *after* each instruction executes. Show the full 32-bit hexadecimal contents of each register. **[12 points]**

Instruction	D0	D1	D2
MOVE.B DIVIDEND,D0			
EXT.L D0			
DIVS #15,D0			
CLR.L D1			
MOVE.W D0,D1			
CLR.L D2			
SWAP D0			
MOVE.W D0,D2			

Make sure that you understand what each instruction is doing before proceeding.

Part 8: MULU and MULS instructions

As in the case of division, the methods for performing unsigned and signed multiplication are different, resulting in two distinct machine instructions: MULU and MULS. The former is used when multiplying unsigned values, while the latter is used to multiply signed values. In both cases, the operands must be words, and the result is a long word.
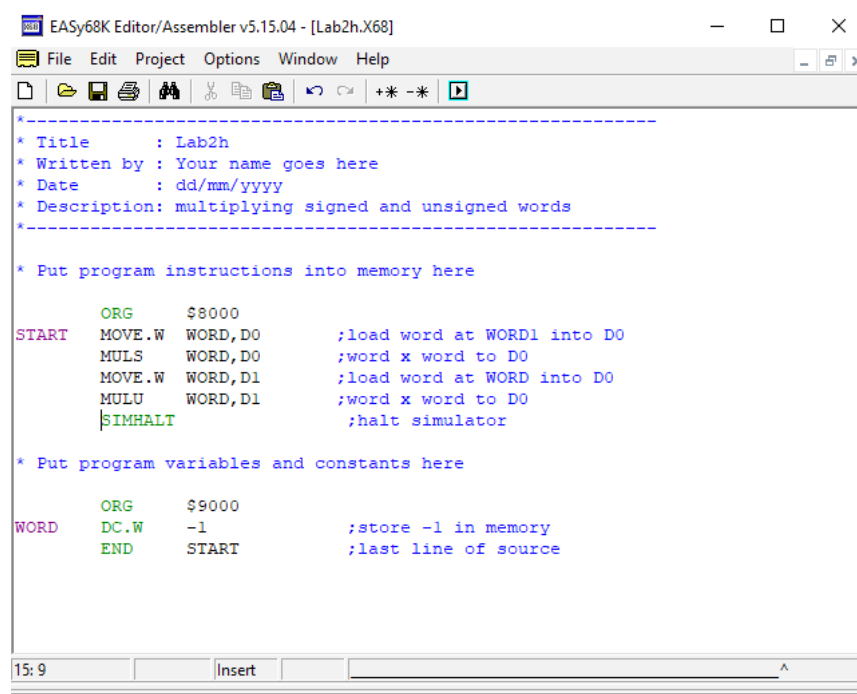
Before proceeding, **review the signed and unsigned multiplication instructions on pages 84, 327 and 328** of your textbook.

Step 1

Download the sample program called **Lab2h.X68** from the course website.

Step 2

Start Easy68K. Once running, load the file Lab2h.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 - [Lab2h.X68]          —    □    ×
File  Edit  Project  Options  Window  Help                      _ ⚓ ×

*---------------------------------------------------------
* Title      : Lab2h
* Written by : Your name goes here
* Date       : dd/mm/yyyy
* Description: multiplying signed and unsigned words
*---------------------------------------------------------

* Put program instructions into memory here

        ORG     $8000
START   MOVE.W  WORD,D0      ;load word at WORD1 into D0
        MULS    WORD,D0      ;word x word to D0
        MOVE.W  WORD,D1      ;load word at WORD into D0
        MULU    WORD,D1      ;word x word to D0
        SIMHALT              ;halt simulator

* Put program variables and constants here

        ORG     $9000
WORD    DC.W    -1           ;store -1 in memory
        END     START        ;last line of source


15: 9                 Insert                                    ^
```

Step 3

Assemble the program, then trace through the program, and fill in the table below with the value in data registers D0 and D1 *after* each instruction executes. Show the full 32-bit hexadecimal contents of each register. **[4 points]**

Instruction	D0	D1
MOVE.W WORD,D0		
MULS WORD,D0		
MOVE.W WORD,D1		
MULU WORD,D1		

Step 4

Once the program completes execution, do data registers D0 and D1 contain the same 32-bit values? Why not? **[2 points]**

Part 9: The Condition-Code Register (CCR)

As explained in class, after the processor executes a machine instruction the flags in the Condition Code Register (CCR) are updated to reflect the characteristics of the result. In general, different instructions will affect the CCR flags in different ways. Therefore, in practice, you must check the data sheet in Appendix B of your textbook to determine how a particular instruction affects the CCR flags.

The actual CCR flags can be found in the lower byte of the Status Register (SR). The figure below shows how the Easy68K displays the SR, which contains the CCR, during the simulation of a program.
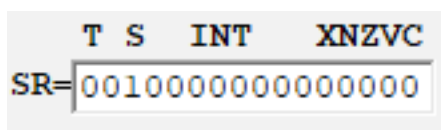
Once an instruction completes executing, the flags in the CCR are updated based on the result. If a particular flag (i.e., bit) in the CCR is set to one, the flag (and the corresponding condition associated with the flag) is considered *true*. Otherwise, if the flag is set to 0, the flag (and the condition associated with the flag) is considered to be *false*. For example, if the result is zero, the Zero (Z) flag in the CCR will be set to 1. However, if the result is non-zero, the Z-flag will be cleared to 0. Similarly if the most-significant bit of the result is one, the Negative (N) flag will be set to 1, otherwise, it will be cleared to 0.

You are encouraged to review your lecture notes to make sure that you understand what each flag in the CCR is telling you about the result of an operation. (Don't worry about the X-flag, as we will not be making use of this flag in our study of the 68000's ISA.)

Step 1

Download the sample program called **Lab2i.X68** from the course website.

Step 2

Start Easy68K. Once running, load the file Lab2i.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 - [Lab2i.X68]

File   Edit   Project   Options   Window   Help

*------------------------------------------------------------
* Title     : Lab2i
* Written by : Your name goes here
* Date       : dd/mm/yyyy
* Description: Add three 8-bit numbers
*------------------------------------------------------------

* Put program instructions into memory here

        ORG     $8000
START   MOVE.B  NUM1,D0     ;load byte at NUM1 into D0
        ADD.B   NUM2,D0     ;now add byte at NUM2
        ADD.B   NUM3,D0     ;now add byte at NUM3
        SIMHALT             ;halt simulator

* Put program variables and constants here

        ORG     $9000
NUM1    DC.B    100         ;store 100 in memory
NUM2    DC.B    20          ;store 20 in memory
NUM3    DC.B    150         ;store 150 in memory
        END     START       ;last line of source

1: 1                Insert
```

Step 3

Assemble the program, and then run the program. What is the final 32-bit result in data register D0? Express your answer in both hexadecimal and decimal. **[2 points]**

Step 4

Consider the decimal values of the three bytes that the program seeks to sum. If we assume that these bytes represented unsigned values, and if we perform the summation by hand, what should the sum be? **[1 point]**

So what went wrong? Was the algorithm faulty or have we failed to notice a subtlety? To better understand what is happening in the previous program we will re-run the program this time making use of the Easy68K's trace facility. We will also pay close attention to the flags in the CCR, and how these flags are affected by each program instruction.

Re-load the program, then trace through each program instruction. After each instruction executes, examine the flags in the CCR and record the value of each flag in the table below, along with the value contained in data register D0. **[7.5 points]**

Instruction	D0	Condition Code Register (CCR)			
		N	Z	V	C
`MOVE.B   NUM1,D0`					
`ADD.B    NUM2,D0`					
`ADD.B    NUM3,D0`					

Questions

Now answer the following questions related to the previous trace:

8. Why does the first addition instruction (see line 12 of the listing file) cause the Carry (C) flag in the CCR to be cleared and set to 0, but the second addition instruction (on line 13) cause the C-flag to be set to 1? Hint: Remember from class, if you are interpreting the numbers being summed as *unsigned* values, you will need to think about what range of values can be accurately represented with an unsigned (8-bit) representation. **[2 points]**

9. Explain why the final value in D0 does not match the expected value when the summation is performed directly by hand on the 3 decimal numbers. **[1 point]**

```



```

10. How would you fix the previous program so that it performs the computation correctly? **[1 point]**

```



```

Make the proposed changes to your program, and verify that the new program functions correctly. **[1 point]**

Part 10: Write your own program

Now it is your turn! Using the formula below, your task is to write a 68000 assembly-language program to convert Celsius (C) to Fahrenheit (F):

$$F = (C \times 9)/5 + 32$$

As an aid, we have provided the program Celsis-to-Farenheit.X68 available at the course website. The program uses two subroutines (i.e., functions) that are already written for you. The first, called READ_TEMP, displays a message on the screen prompting the user to enter a temperature in Celsius. The function then reads the value entered by the user, and stores the value as a byte at a location in memory, called CELSIUS. The second function, called

DISPLAY_TEMP, displays the original temperature value in Fahrenheit. This function assumes that on entry to the function the Fahrenheit temperature value is already computed, and is a 32-bit value contained in data register D1.

Your task is to complete the missing code; that is, to write the few lines of assembly code necessary to convert the temperature input by the user from Celsius to Fahrenheit. (In-line comments have been included as a guide.) As explained above, you can find the 8-bit Celsius value using the label CELSIUS. Just make sure that the final 32-bit Fahrenheit value is stored in data register D1. Also, keep in mind that Celsius temperatures can be negative, therefore, you will want to treat them as signed values. **[8.5 points]**

Evaluation

The deliverables for this lab are:

- This lab document, with the answers in the spaces provided.
- For all questions, the source files (.X86) with your name, student identification number, and the data appearing in the comment block at the start of the program.