

# Introduction to Easy68K

---

CIS\*2030  
Lab Number 1

Name: Conor Roberts

Mark:           /34

## Overview

This lab introduces **Easy68K** – the simulation tool that we will be using throughout this semester to explore the Instruction-Set Architecture (ISA) of the Motorola 68000 CPU.

## Objectives

Upon completion of this lab you will be able to:

- Enter assembly-language programs into Easy68K and execute them on the simulator,
- Examine the contents of registers,
- Examine the contents of memory,
- Execute instructions step by step for the purpose of debugging.

## Activities

This lab is broken into three sections, as described in the table below:

<b>Preparation</b>	Write and run a simple program using Easy68K; introduce coding standards for this course
<b>Practice</b>	Inspecting registers and memory, tracing a program
<b>Play</b>	Play a fun game written in 68000 assembler

## Evaluation

No marks are allocated to section 1 of this lab, only for sections 2 and 3. However, you are expected to complete all three sections of this lab.

## Preparation

You must prepare for this lab by reading through the **Section 1 – Preparation**. Also, you must complete the following reading assignments from your textbook.

- Sections 0.1, 0.2, and 0.3 (number systems, un/signed numbers, ASCII, arithmetic)
- Section 2.1, 2.2, and 2.3 (registers and memory)
- Section 3.3 (instruction formats)
- Sections 4.1, 4.3 and 4.4 (assembler directives)

## Introduction

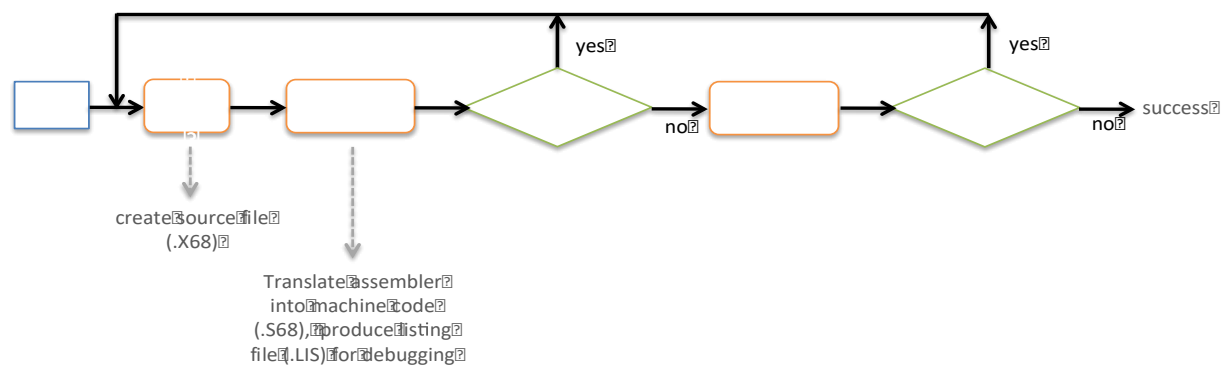
Easy68K is a simulation tool that can be used to create and run assembly-language programs for the Motorola 68000 architecture. Easy68K bundles with an editor, cross-assembler, and simulator, all consolidated in an easy-to-use Integrated Development Environment (IDE).

The process through which programs are created and run involves three main steps:

- First, the user creates an assembly-language program, referred to as a *source* file, using the built-in editor (EDIT.68K). The editor stores the source file in ASCII format, and appends the suffix (extension) .X68 to the source filename.
- Second, the user assembles the source file (.X68) using the built-in *cross-assembler*. The goal of the cross-assembler is to translate the assembler code contained in the source file into machine code for the Motorola 68000 architecture, as well as to produce a listing file. With regards to the former, the cross-assembler creates a file with the same name as the source file, but with an .S68 extension. This file contains the machine code generated by the cross-assembler (in a text format known as Motorola S-Records) that will be run by the simulator. The simulator also uses the listing file to provide source-level debugging. The listing file has the same name as the original source file, but with a .LIS extension.
- Third, the user uses the simulator (SIM68K) to run the machine code generated by the cross-assembler, as if the code were running on an actual Motorola 68000 processor.

Before proceeding, it is important to emphasize a few things. First, if your program contains *syntax* errors, these errors will arise during the assembly step and will appear in the listing

(.LIS) file. If one or more errors are found, you will then need to return to the source file (.X68) and edit it to correct the error(s). Once the program assembles and is free of syntax errors, you can simulate the program to ensure that it runs correctly. If your program does not run correctly, facilities in the simulator (e.g., the ability to inspect the contents of registers or memory) can be used to identify the *semantic* (or *logical*) errors in your program. Of course, once you have identified these errors, you will need to edit the source file, correct the errors, assemble the program again, and simulate the program to ensure that it runs correctly. The entire process is illustrated in Fig. 1.



**Figure 1:** Using Easy68K to develop and run an assembly-language program.

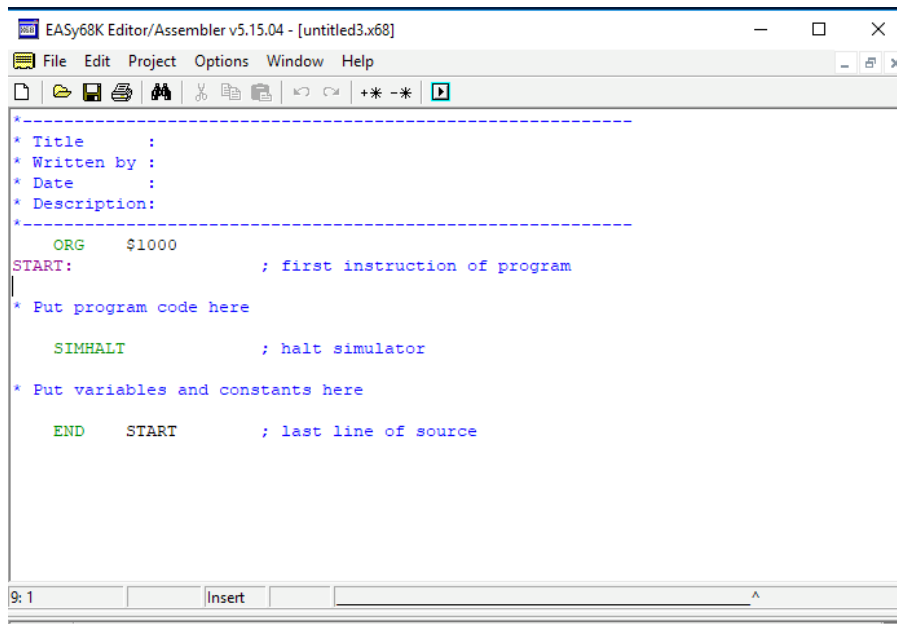
It is also important to recognize that there are a few subtle differences between an actual hardware implementation based on the 68000 processor, like the one described in your textbook and used in previous course offerings, and the synthetic 68000 processor simulated by Easy68K. In particular, where in memory programs can be loaded, memory-mapped addresses for I/O devices, and the set of systems calls that can be made to the operating system are unique to each platform. These differences will be highlighted, when appropriate, in this and remaining labs.

## Preparation – Introduction to Easy68K

In this section, you will prepare for your lab exercise by writing and executing your first 68000 assembly-language program. In so doing, you will become familiar with the Easy68K and the basic tasks of writing, assembling, running, and debugging a program.

### Step 1

Begin by running Easy68K. This program is installed on the SOCS Windows Servers, and can also be freely installed on your own computer running Windows. The first thing that you should see is a template, similar to the one shown below.



```
EASy68K Editor/Assembler v5.15.04 - [untitled3.x68]
File Edit Project Options Window Help
-----
* Title      :
* Written by :
* Date       :
* Description:
*-----
      ORG      $1000
START:                ; first instruction of program
|
* Put program code here

      SIMHALT        ; halt simulator

* Put variables and constants here

      END      START    ; last line of source

9: 1      Insert
```

### Step 2

Using the template as a starting point, enter the program shown below. Each line of an assembly-language program contains some combination of labels, executable instructions, assembler directives and comments. Both labels and full-line comments must start in column 1. Also, each line is separated into fields separated by one or more spaces or tabs.

```

EASy68K Editor/Assembler v5.15.04 - [Lab1a.X68]
File Edit Project Options Window Help
-----
* Title      : My first 68000 assembly-language program!
* Written by : Your name goes here
* Date       : dd/mm/yyyy
* Description: Say hello!
*-----
* Put program instructions into memory here

      ORG      $1000
START  MOVE.B   #13,d0          ;task 13 (display string)
      LEA      MSG,A1          ;point to string
      TRAP     15              ;system call

      SIMHALT                   ;halt simulator

* Put variables and constants into memory here

MSG     DC.B    'Hello World!',0 ;null-terminated string
      END      START          ;last line of source
1: 1      Insert

```

Below, is a summary of the program features:

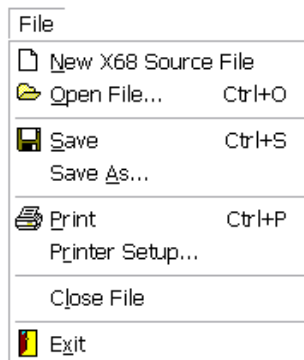
- Lines that begin with an `*` are *full-line* comments. As stated above, these must begin in column 1 to avoid generating a syntax error during the assembly process. *In-line* comments, like `system call`, appear in the right-most column.
- The *origin* assembler directive specifies where to start loading program instructions (or data) into the target processor's memory space. In this case, `ORG $1000` specifies that the program instructions that follow in the source file are to be loaded into memory starting at memory address 0x00001000. (The prefix `$` indicates that the address is specified as a hexadecimal number.)
- `START` is a *label*. Programmers use labels to reference a particular line of code or a particular memory address. In this case, `START` is referenced by the `END START` assembler directive, and is used to indicate where (in memory) the program should start execution.
- `MOVE.B #13,D0` is the first actual executable *instruction* to be assembled. `MOVE.B` is the opcode and `#13,D0` are the operands. Once executed, the instruction will move the (decimal) value 13 into the 68000's internal *data* register, D0.
- The second program instruction is `LEA MSG,A1`. This instruction loads the value associated with the label `MSG` into *address* register A1. The value of `MSG` is the starting (memory) address of the ASCII string defined by the label `MSG`. Therefore, once executed, address register A1 will be initialized to point to the starting address of the ASCII string in memory.

- The final instruction is `TRAP 15`. In general, TRAP instructions provide a way for a program to *request services* from the operating system. In the case of Easy68K, `TRAP 15` is used to perform *input-output tasks* that are built into the Easy68K simulator. By placing different *task numbers* in data register D0, different input-output tasks can be performed. In this case, task 13 tells the simulator to display the null-terminated string pointed to by address register A1 on the console window.
- The assembler directive `define constant (DC)` is used to load an ASCII string into memory at the current location. The line `MSG DC.B 'Hello World!',0` loads the ASCII string (enclosed in single quotes) followed by 0 (the null-terminator for the string) into 12 consecutive bytes of memory starting from the current memory location.
- `SIMHALT` is an assembler directive that causes the simulator to halt execution.

### Step 3

Save your program (i.e., source file). You can do this through the [File->Save](#) menu choice, as illustrated below. By default, the Easy68K editor appends the extension `.X68`. Save your program with the filename **Lab1a.X68**.

**File menu:**



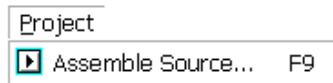
### Step 4

Once your program is saved, take a few moments to read through the code to determine what the program might do.

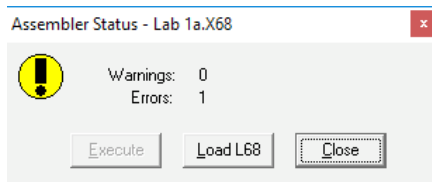
### Step 5

Before we can simulate the program to verify its behaviour, we first need to assemble the program. This can be done through the [Project->Assemble Source](#) menu choice, as illustrated below.

### Project menu:

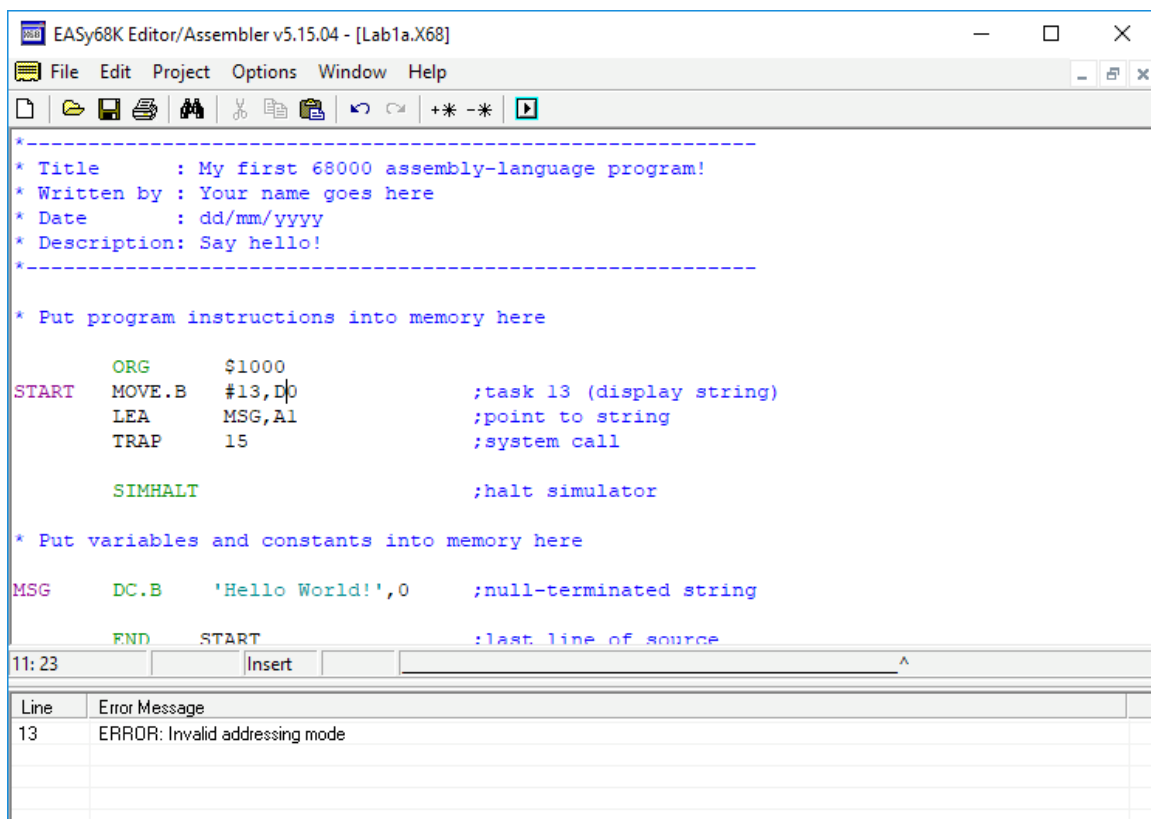


Notice that the assembler has flagged a syntax error in the source file.



The presence of the syntax error in the source file means that the cross-assembler was unable to successfully translate your program into machine code. As a result, an executable file (.S68) is still output by the cross-assembler, but a listing file (.LIS) for debugging purposes is also generated, which you should use for identifying errors. Click on the [Close](#) button.

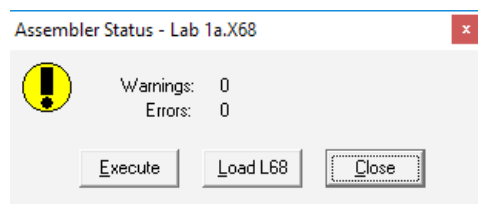
In practice, syntax errors detected during the assembler process are displayed at the bottom of the editor window, as illustrated below.



## Step 6

Using your mouse, double click on the error message (at the bottom of the window) to highlight the line of code in the source file where the error is located. In this case, the offending instruction is located on line 14 and is missing a # character in front of the decimal value 15. The # character indicates that an *immediate* addressing mode is to be used for locating the operand. (Without the # the instruction would try to get the trap number from memory location 15, which is not allowed by the 68000's ISA for this type of instruction.)

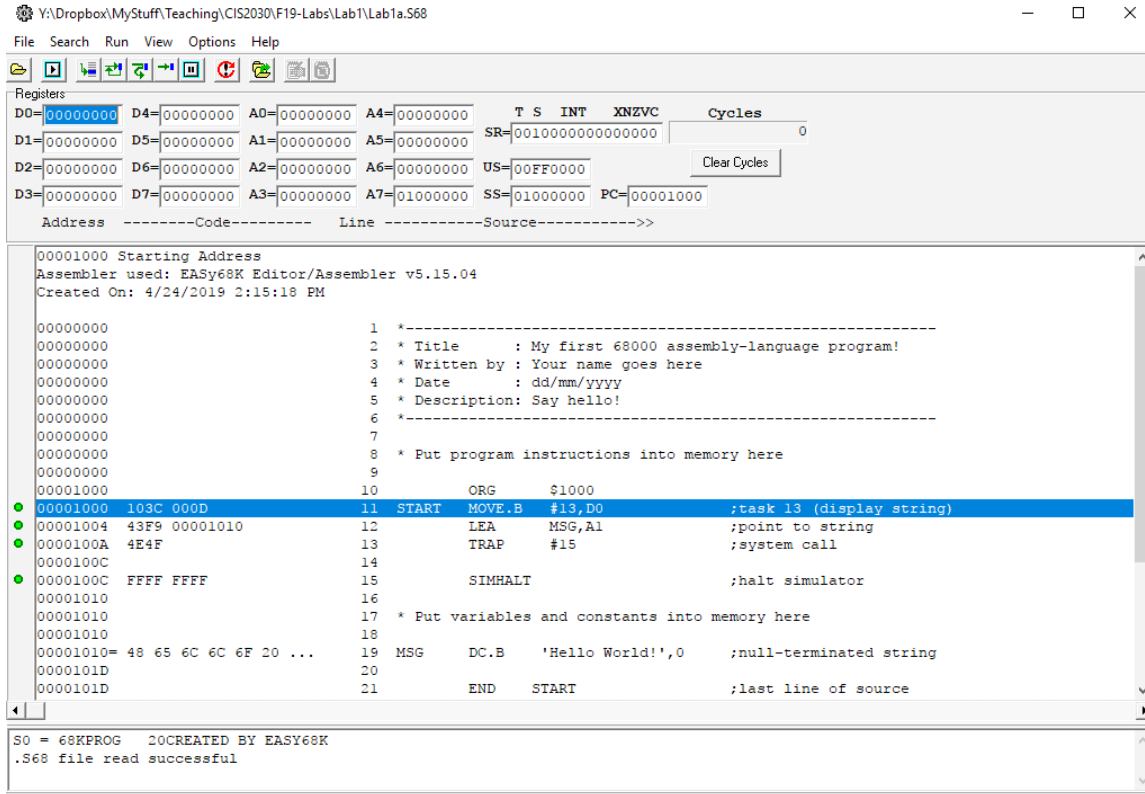
Fix the error on line 14 by inserting a # character directly in front of the 15, then use the [Project->Assemble Source](#) menu choice to re-assemble the code. If no further syntax errors are detected, you should see a dialog box that provides you with the option of executing the assembled program on the simulator, as illustrated below.



## Step 7

Click on the [Execute](#) button to load the machine code into the 68000 simulator. Two new windows should open: A *simulator* window and a *console* for I/O. The simulator window is shown below.





The simulator window shows the source program (.X68) before and after assembly, along with some other very useful information. On the right-hand side of the window, the original assembly language code appears with line numbers. These line numbers are simply for reference purposes. On the left-hand side, the first column of hexadecimal numbers represents the memory address of each instruction or data. (Notice that the first non-zero memory hexadecimal address (0x00001000) matches the operand associated with the first ORG assembler directive on the right-hand side. The second column of hexadecimal numbers on the left-hand side is the machine code generated for each instruction, intermixed with various data. The machine instructions are represented as hexadecimal numbers, and have different sizes (e.g., 2 bytes, 8 bytes, etc.).

## Step 8

Using your mouse, scroll down to the bottom of the window where the cross-assembler's symbol table is conveniently displayed (see below).

```

SYMBOL TABLE INFORMATION
Symbol-name      Value
-----
MSG              1010
START           1000

```

The left column (symbol-name) in the symbol table identifies each label used in the source program. (In practice, programmers use labels to identify a particular location in a program or a particular memory location by name.) The right column (value) identifies the (hexadecimal) memory address associated with each label, which were determined by the cross-assembler during the assembly process. Notice that the label `START` has the hexadecimal value `0x1000`, the same value specified in the first `ORG` directive in the source file. This address corresponds to the memory location of the first program instruction. Also, notice that the label `MSG` has the hexadecimal value `0x1010`. This address corresponds to the starting memory address of the ASCII string.

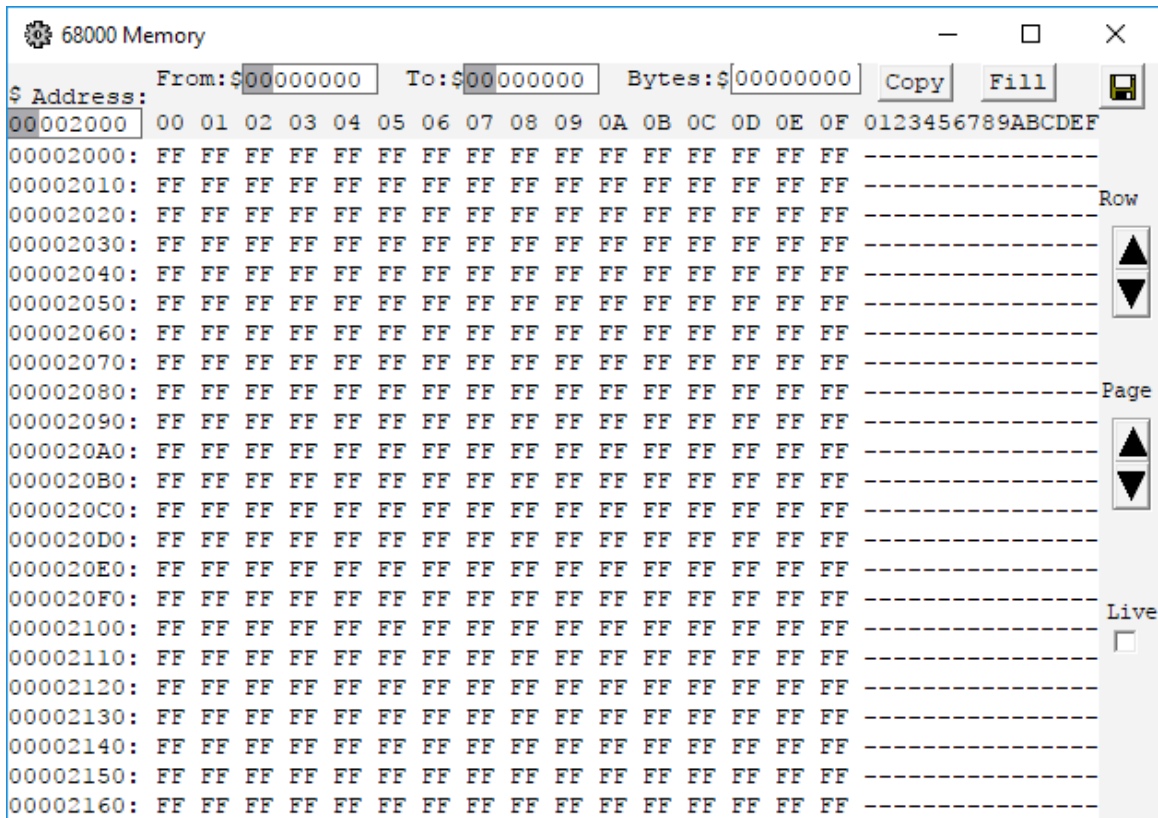
## Step 9

Now, look at the top of the simulator window. As illustrated below, the simulator conveniently shows the current values contained inside the simulator's synthetic registers. These registers include 68000's data registers (D0-D7), address registers (A0-A7), status register (SR), user stack pointer (US), system stack pointer (SS), and program counter (PC). The simulator updates each register after each program instruction executes. The ability to see the contents of the registers as a program executes provides the user with useful debugging information.

Registers									
D0=	00000000	D4=	00000000	A0=	00000000	A4=	00000000	T S INT XNZVC	Cycles
D1=	00000000	D5=	00000000	A1=	00000000	A5=	00000000	SR=	0010000000000000
D2=	00000000	D6=	00000000	A2=	00000000	A6=	00000000	US=	00FF0000
D3=	00000000	D7=	00000000	A3=	00000000	A7=	01000000	SS=	01000000
								PC=	00001000

## Step 10

Another important window (that does not open automatically when the simulator is invoked) is the *memory* window. You can find the memory window by going to [View](#) on the menu bar and selecting [Memory](#) from the drop-down menu. You should see something similar to the figure below.



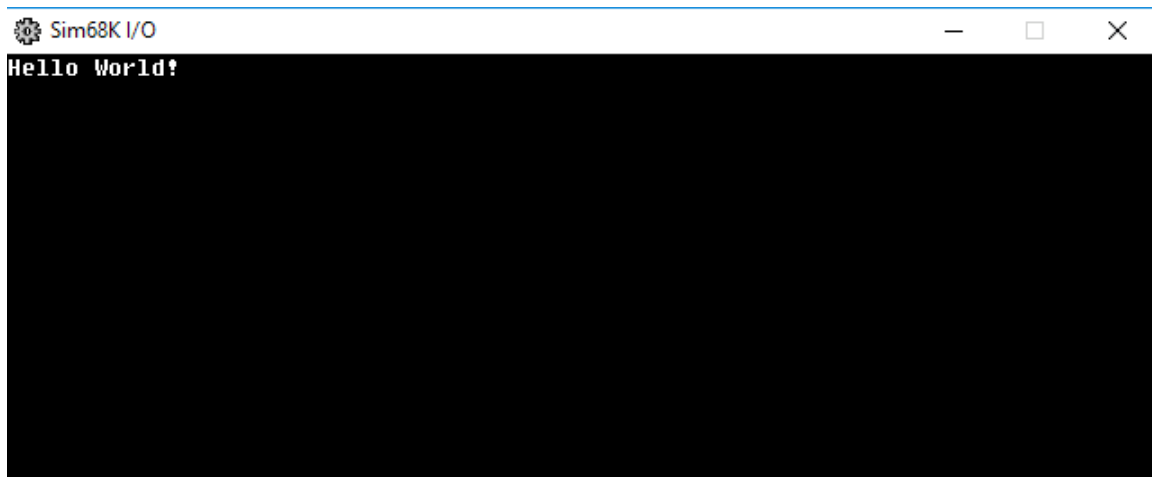
The memory window allows the user to look at the contents of memory. In particular, it allows the user to inspect *any* location in the entire 16 Megabyte address space of the (simulated) 68000 processor. Each row within the window displays a (32-bit hexadecimal) memory address followed by the contents of 16 consecutive bytes of memory. Like the memory address at the start of a row, the 16 bytes are shown in hexadecimal format. An ASCII interpretation of each of the 16 consecutive bytes is shown at the end of each row. (If the byte does not map to a graphical ASCII character, a dash '-' is displayed.) The user can use the row/page buttons or the mouse to scroll through memory. It is also possible to jump to a specific address by typing the address into the Address field located at the top of the first column on the left-hand side of the memory window. Note: the memory window can also be used to modify the contents of a particular memory location. This can be done by directly typing the new value into the appropriate location in hexadecimal format. However, this operation is not frequently used in practice.

## Step 11

Let's run the program. This can be done by clicking the [Run](#) button on the tool bar (shown below).



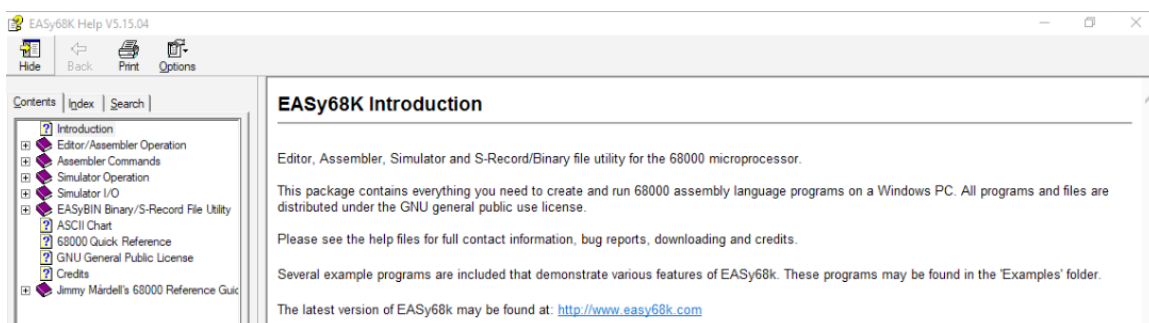
The output generated by the program appears on the final window opened by the simulator – the console – as shown below.



Did the program do what you thought that it would?

In practice, the console can be used in conjunction with various system calls (implemented as trap instructions in the 68000 in the source file) to display characters, strings, numbers, etc. on the screen, as well as to read characters, strings, numbers, etc., from the keyboard.

More information about Easy68K and the editor, cross-assembler and simulator can be found by selecting the [Help](#) option in the tool bar. You are encouraged to read through the help menus frequently (outside of lab) until you are acquainted with the basic functionality of Easy68K.



## Practice – Learning about the 68000's ISA

Now it is your turn! Using Easy68K, you will now begin to explore the 68000's ISA. You will examine the contents of the processor's internal registers and memory. Also, you will learn how to debug a program by executing a single instruction at a time. When completing this part of the lab, make sure to answer all of the questions in the spaces provided.

### PART 1: Programming in 68000 Assembly

#### Step 1

Download the sample program called **Lab1b.X68** from the course website.

#### Step 2

Start Easy68K. Once running, load the file Lab1b.X68 using the [File->Open File](#) menu choice. Look at the code in the source file, and then briefly explain what the program does in the box below. **[1.5 points]**

Loads bytes into memory, adds them together and displays all the data in a table

#### Step 3

Once loaded, document the program with your personal information, as illustrated below.

```
*-----
* Title       : Lab1b
* Written by  : Your name goes here (Student ID)
* Date       : dd/mm/yyyy
* Description: Load, Stores, and Arithmetic
*-----
```

#### Step 4

The program makes use of *four* labels. These labels and their corresponding values are tabulated in the cross-assembler's symbol table. In the table below, print the name of each label and the (32-bit hexadecimal) memory address associated with each label. **[2 points]**

Label	Address
NUM1	0x00009000
NUM2	0x00009001
START	0x00008000
SUM	0x00009002

Take a moment to think about what these labels are referring to.

### Step 5

Prior to running the program, what are the initial values in data registers D0 and D1? Print these 32-bit hexadecimal values in the table below. **[1 point]**

Register Name	Initial Contents
D0	00000000
D1	00000000

### Step 6

If the memory window is not already open, use the [View->Memory](#) menu option to open it. Now modify the [Address](#) field in the memory window to jump directly to the hexadecimal address 0x00009000. What 8-bit (byte) values are contained in locations 0x00009000 through 0x00009002? Print these (8-bit hexadecimal) values in the table below. **[1.5 points]**

Memory Address	Value in Memory
0x00009000	0A
0x00009001	34
0x00009002	80

### Step 7

Run the program by clicking on the [Run](#) button in the simulator's tool bar.

### Step 8

What are the final values in data registers D0 and D1? Print these 32-bit hexadecimal values in the table below. **[1 point]**

Register Name	Final Contents
D0	0000000A
D1	0000003E

### Step 9

Use the [Address](#) field in the memory window to jump directly to the hexadecimal address 0x00009000. What values are now contained in locations 0x00009000 through 0x00009002? Print these (8-bit hexadecimal) values in the table below. **[1.5 points]**

Memory Address	Final contents in Memory
0x00009000	0A
0x00009001	34
0x00009002	3E

Are the values the same as those recorded in step 6? Explain. **[1 point]**

The first two values are the same but not the third. The third value now represents the sum of the first two values.

### Step 10

Click the [Rewind Program](#) button in the tool bar.



Write the 32-bit hexadecimal values in D0 and D1 in the table below. Write the 8-bit hexadecimal value located at memory location 0x00009002 in the table below. What happened to the value of D0 and D1? What happened to the value in 0x00009002? **[1.5 points]**

Register Name	Contents
D0	00000000
D1	00000000

Memory Address	Value in Memory
0x00009002	3E

### Step 11

Now click the [Reload Program](#) button on the tool bar.



Now view and write the 8-bit hexadecimal value at memory location 0x00009002. What happened to the value in 0x00009002? Write the value in the table below. **[1 point]**

Memory Address	Value in Memory
0x00009002	80

Notice that Reload Program resets both the contents of the registers and the contents of memory, whereas Rewind Program resets only the contents of the registers.

### Step 12

So far when running a program, we have executed the program from start-to-finish in a continuous flow of execution. Sometimes, however, we may wish to *trace* through a program one instruction at a time. This is especially true when seeking to debug a program.

[Reload](#) the program. What are the initial values in data register D0 and the program counter (PC)? Print the 32-bit hexadecimal values in the table below. **[1 point]**

Register Name	Contents
D0	00000000
PC	00008000

What does the *value* in the PC represent? **[1 point]**



The value that the program has been instructed to load instructions starting from.

Now, click the [Trace Into](#) button (located in the tool bar) once.



### Step 13

What are the current values contained in registers D0 and PC? Write the 32-bit hexadecimal values in the table below. **[1 point]**

Register Name	Contents
D0	0000000A
PC	00008006

Explain why the PC increased by the amount that it did. *Hint: Look at the size (in bytes) of the machine code generated by the assembler for the first assembly-language instruction on line 11.* **[1 point]**

The move instruction uses 6 bytes

### Step 14

Next, observe that the instruction on line 12 is highlighted [blue](#).

Has the highlighted instruction executed yet? *Hint: What is the value in the left-most column of the simulator window on line 12? Is it the same as the value of the program counter? What is the program counter telling you?* **[1 point]**

Left most column value: 00008006  
PC: 00008006

They are the same. The program is telling me that the next instruction to be run is at that address

### Step 15

Using the [Trace Into](#) button, keep stepping through the program until you reach the addition instruction on line 13. What is the current value of the PC upon reaching line 13? Print the 32-bit hexadecimal value in the table below. **[1 point]**

Register Name	Contents
PC	0000800C

Now click the [Trace Into](#) button to execute the instruction on line 13. What is the current value of the PC upon reaching line 14? **[1 point]**

Register Name	Contents
PC	0000800E

Why has the PC only increased by a value of 2, when before the PC increased by a value of 6? What does this tell you about 68000 instruction formats in general? **[1 point]**

The add instruction uses less memory than the move instruction. All instructions do not use the same amount of memory.

## PART 2: A Closer Look

You will now take a closer look at the effects of the assembly process on the original source code. In so doing, you will revisit some of the key concepts discussed in class related to the interpretation of binary information, translation of assembly instructions to binary (machine) instructions, and assembler directives.

### Step 1

Reload the program (i.e., **Lab1b.X68**).

### Step 2

Consider the origin assembler directive (**ORG** \$8000) located on line 10 of the simulator window. To learn more about the **ORG** directive read **Section 4.6.1 on page 112 your textbook**. What is the memory address of the first executable instruction? Express your answer as a 32-bit hexadecimal number. **[1 point]**

Answer: 00008000

In general, assembler directives, like **ORG**, provide programmers with a way of instructing the assembler to perform the assembly process in a certain preferred way. However, unlike executable instructions, assembler directives themselves are not translated into machine code. For example, notice that in the case of the **ORG** directive on line 10, the assembler has not generated any machine code; that is, column 2 (relative to the left-hand side of the window) is empty. The purpose of the **ORG** directive on line 10 is to simply instruct the assembler to place the instructions that follow into memory starting at address 0x00008000.

### Step 3

Next, consider the four executable instructions located on lines 11 through 14. In contrast to the previous **ORG** directive, all four executable instructions have been translated into (binary) machine code. The machine code for each instruction is displayed in column 2 of the window, while column 1 shows the address of each instruction. Use this information to complete the table below: **[4 points]**

Assembly Instruction	Machine Code	Address of Instruction
MOVE.B NUM1,D0	1039	00009000
MOVE.B NUM2,D1	1239	00009001
ADD.B D0,D1	D200	0000800C
MOVE.B D1,SUM	13C1	00009002

Compare the machine code instructions to each other. Are there any similarities? Any differences? Write your answer in the box below. **[2 points]**

The add instruction does not use an address around 00009000.

All of the move instructions reference an address around 00009000 because that's where those variables are defined to be.

They all come with 16-bit machine code and 32-bit addresses to reference to.

You can find more information about the binary format of the previous instructions on pages 316 and 267 of your textbook.

#### Step 4

Now open the memory window, and use the [Address](#) field to go to memory location 0x00008000. Compare the values that you see displayed in the memory window to the values located in column 2 of the previous table in [Step 3](#). Are they the same? Explain what you are seeing and write your answer in the box below: **[1 point]**

All of the information is the exact same it is just separated differently. The instructions are divided into those memory locations.

#### Step 5

Consider the *define constant* assembler directives ([DC.B](#)) located on lines 20 through 22 of the simulator window. To learn more about the [DC.B](#) directive read Section 4.6.4 on page 113 your textbook. To learn about labels read Section 4.3.1 on page 107 of your textbook. Identify the value of all three labels in the table below. **[1.5 points]**

Assembler Directive			Value of Label
NUM1	<a href="#">DC.B</a>	10	NUM1 =0A
NUM2	<a href="#">DC.B</a>	52	NUM2 =34
SUM	<a href="#">DC.B</a>	128	SUM =80

Now open the memory window, and use the [Address](#) field to go to memory location 0x00009000. Compare the values that you see displayed in the memory window to the three

constants defined on lines 20 through 22 of the program code. Is there a correspondence? [1 point]

There is a direct correspondence. All of the values are there. Each constant has its own memory location.

### Step 6

As discussed in class, at the machine level, a binary number can be interpreted any way the programmer wishes. With this in mind, prior to running the program, interpret the byte at memory location 0x00009001 as an ASCII character, then as an 8-bit unsigned number. (As ASCII characters are 7-bits, not 8-bits, use only the 7 least-significant bits of the original byte. See your textbook for the ASCII table.) Also, interpret the byte at memory location 0x00009002 as an unsigned number, then as a signed number (in 2's complement form). Print your answers in the table below. Unsigned and Signed numbers should be written in decimal. [2 points]

Address	Interpretation of Byte	Value
0x00009001	ASCII character	4
0x00009001	Unsigned number	52
0x00009002	Unsigned number	128
0x00009002	Signed number	128

### Step 7

Run the program. Interpret the byte at memory location 0x00009002 as an ASCII character, an unsigned number, and a signed number. Print your answers in the table below. Unsigned and Signed numbers should be written in decimal. [1.5 points]

Address	Interpretation of Byte	Value
0x00009002	ASCII character	>
0x00009002	Unsigned number	62
0x00009002	Signed number	62

## Play – AirStrike

A fun way to learn more about assembly-language programming is to write a small 2D game. Although you may not be able to do so at this point, you can get a feel for more advanced assembly language programming by playing such games written by others, and then examining the assembler code.

As an example of this, download the folder **AirStrike.zip** from the course website. Unzip the program. Start Easy68K, then once running, load the file Air.X68 using the [File->Open File](#) menu choice. Take a few minutes to look over the code in the source file. Notice that the code mainly consists of data, in the form of 8-bit and 32-bit values, and instructions in the form of loads and stores from and to memory, system calls, data movement instructions, basic arithmetic instructions, like addition, and branches. In practice, once a high-level program is compiled, and all of the high-level statements, data structures, and library routines are stripped away, this is what you are typically left with!

Airstrike is a basic “shoot’em up” arcade style game written by Yassine Loudad and adapted to run on the Easy68K by Chuck Kelly. Start the game running in the Easy68K simulator. When prompted, press “n” to commence with the first game. Once running, press “e” to move your plane up, and “d” to move it down. To fire, press “l”.

Earn points by achieving a score greater than zero! Have fun!