

Subroutines: Stacks and Functions

CIS*2030
Lab Number 5

Name: _____

Mark: _____/149

Overview

One of the keys to managing code complexity is to break a large program down into smaller, more manageable parts, where each part performs a single, well-defined task. In the C programming language, we refer to these “parts” as *functions*. At the assembly-language level, we refer to functions as *subroutines*. Just like functions, subroutines are individual pieces of code that typically perform a specific task, and which can be called multiple times from any location within a program. To support the implementation of subroutines ISAs provide a runtime *stack*. As explained in class, a stack is a Last-In-First-Out (LIFO) data structure that is used to implement the basic function call-return mechanism, parameter passing, nested function calls, recursive function calls, and memory allocation for local variables defined within a function.

Objectives

Upon completion of this lab you will:

- Understand how the runtime stack on the 68000 is implemented,
- Understand how the various instructions that are used to implement subroutines (i.e., functions) affect the stack,
- Understand how word boundaries are maintained on the stack when performing byte operations,
- Understand how to call and return from a subroutine,
- Understand how to pass arguments to a subroutine using the stack,
- Understand how to save register values on the stack, and,
- Understand how to draw a memory map showing the contents of the stack.

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 2.2.4 (stack pointer)
- Section 2.2.5 (supervisor mode bit)
- Section 3.2.1 (MOVEM, PEA)
- Section 3.2.7 (BSR, RTS, JSR)

Introduction

As explained in class, most ISAs provide hardware support for the implementation of a *runtime* stack. In the case of the 68000, there are two runtime stacks: one for the operating system, and another for user programs executed by the operating system (see Fig. 1). Only one of these stacks is visible to the program that is running. In particular, if the bit 13 in the Status Register (SR) is set to 1, the processor is in supervisor state, and the operating system's stack is visible to the OS. Otherwise, if bit 13 in the SR has been set to 0 by the operating system, the processor is now in user state, and the user program's stack is visible to the user program. The provision of separate stacks for the operating system and programs that the operating system runs increases the likelihood that the operating system can continue to run safely, even if a user program corrupts its own stack.

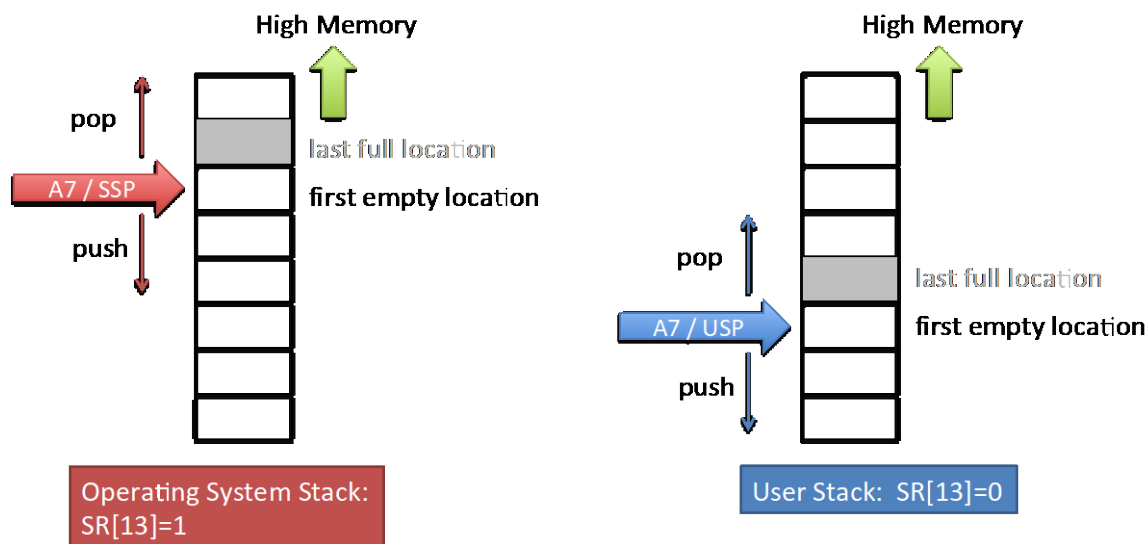


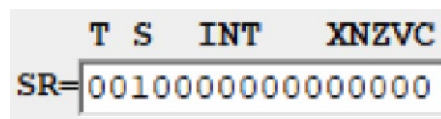
Figure 1: The two runtime stacks.

As illustrated in Fig. 1, the ISA of the 68000 describes a stack structure where stack is located in main memory and grows downward. A stack pointer (SP) is used to point to the top of the stack. In practice, there are two stack pointers: one for the operating system's stack, and

one for the user program's stack. When the processor is in supervisor state and running the operating system, the System Stack Pointer (SSP or A7) acts as the system stack pointer. When the processor is in user state and running a user program, the User Stack Pointer (USP or A7) acts as the user stack pointer. Notice that there are two A7 registers that act as the SSP and USP, respectively. However, only one of these two registers is visible to the program that is running, so there is no confusion as to which stack is being used. If the processor is in supervisor state, the A7 associated with the SSP is visible, and stack operations involving A7 will affect the operating system stack. Conversely, if the processor is in user state, the other A7 is visible, and stack operations involving A7 will affect the user stack. Also, as explained in class, register A7 can also be referenced using the symbolic name, SP. Use of the latter is quite common, as it aids in program clarity.

Part 1: The Runtime Stack and Easy68K

Start Easy68K, and then invoke the simulator. (It is not necessary to load or write a program at this stage.) When the Easy68K simulator is first invoked, the contents of the status register (SR) appear as follows:



Notice that S, bit 13 of the status register, is equal to 1. This means that the processor is in supervisor state and the system stack pointer SSP (or A7) is the active stack pointer. It also means that by default, when you write and then run a program, your program will run in supervisor state. Therefore, any stack operations that you perform in your program will affect the system stack and not the user stack!

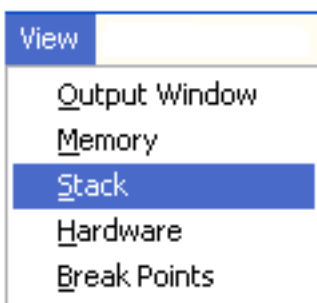
Examine the simulator register display, and then fill in the 32-bit (hexadecimal) values associated with the A7, SSP, and USP in the table below. Note: When displaying the name of the system stack pointer and the user stack pointer registers, Easy68K shortens SSP to SS and USP to US to stay in step with the other registers, which are all 2-characters in length.

[1.5 points]

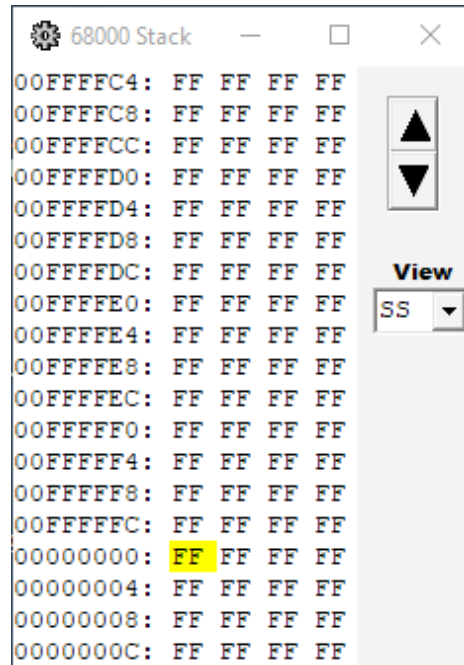
Stack Pointer	Value
A7	
SS	
US	

Notice that SS and A7 contain the same 32-bit address. This address refers to the top of the operating system's stack. The address in US refers to the top of the user stack. In an actual computer running an operating system, the operating system is responsible for initializing the previous registers; that is, the operating system decides where its own stack starts in main memory, but it also decides where the user program's stack is located, and initializes the US with this address prior to running the user program.

It is possible to examine the contents of the stack directly. To do this, select Stack from the view menu as illustrated below.



A new window, similar to the one below, should appear.



The system stack (SS) is selected by default as the stack pointer. The memory addresses on the left-hand-side, and the 32-bit values on the right-hand side, describe the current state of the system stack. In practice, the pull-down view menu can be used to select any register as a stack pointer, including the user stack pointer (US). The location of the current stack pointer (in this case, SS) is highlighted in yellow. To scroll through the stack memory, you can use the **Up/Down** buttons.

Part 2: Push and Pop Stack Operations

As explained in class, unlike some other ISAs, the 68000 does not include specific push and pop stack instructions. Rather, the push operation is implemented by combining a move instruction with pre-decrement addressing, and the pop instruction is implemented by combining a move instruction with post-increment addressing. The previous address modes are based on the fact that the stack grows from a higher memory address towards a lower memory address.

To save the 32-bit contents of register D0 on the runtime stack we implement the *push* operation as follows:

```
MOVE.L D0, -(A7)
```

To restore the 32-bit contents of register D0 from the runtime stack we implement the *pop* operation as follows:

```
MOVE.L (A7)+, D0
```

It is also possible to push and pop *byte* and *word* values. However, as explained in class, to avoid potential address errors arising from memory alignment issues, the stack pointer (A7) is incremented or decremented by a value of 2, rather than 1, in the case of byte operations involving the stack.

In situations where it is necessary to save the contents of several items on the stack, it is not necessary to use a *sequence* of push and pop instructions. Rather, the MOVEM instruction can be used to push and pop multiple items. For example, to save the 32-bit contents of registers A1, D0, and D3 on the stack we can do the following:

```
MOVEM.L A1/D0/D3, -(A7)
```

To restore the contents of the previous registers we can do the following:

```
MOVEM.L (A7)+, A1/D0/D3
```

As we will see later in this lab, the previous instruction is very useful when seeking to make function calls transparent to the calling code.

Step 1

Download the sample program called **Lab5a.X68** from the course website.

Step 2

Start Easy68K. Once running, load the file Lab5a.X68 using the [File->Open File](#) menu choice. You should see something similar to below. (Remember to properly comment your code.)

```

EASy68K Editor/Assembler v5.16.01 - [L5a.X68]
File Edit Project Options Window Help
-----
* Title      : Lab5a
* Written by : Your name goes here
* Date       : dd/mm/yyyy
* Description: Push and Pop Operations involving the
*             runtime stack (A7)
*-----
                ORG      $8000

* Initialize registers with values that are easy to recognize
* when viewed on the stack

START:  MOVE.L  #$D0D0D0D0,D0    ;initialize D0
        MOVE.W  #$D1D1,D1       ;initialize D1
        MOVE.B  #$D2,D2         ;initialize D2
        LEA     $A1A1A1A1,A1     ;initialize A1

* Save previous values on the (runtime) stack in sequence

        MOVE.L  D0,-(A7)         ;push long word in D0
        MOVE.W  D1,-(A7)         ;push word in D1
        MOVE.B  D2,-(A7)         ;push byte in D2

* Restore previous values from stack in reverse order

        MOVE.B  (A7)+,D2         ;pop 8-bit value into D2
        MOVE.W  (A7)+,D1         ;pop 8-bit value in D1
        MOVE.L  (A7)+,D0         ;pop 32-bit value in D0

* Save/restore D1 and A1 using a single push/pop instruction

        MOVEM.L A1/D1,-(A7)      ;push 32-bit values in A1 and D1
        LEA     0,A1             ;clear A1
        CLR.L   D1              ;clear D1
        MOVEM.L (A7)+,A1/D1      ;pop 32-bit values into D1 and A1

                END      START
38: 22      Insert

```

Step 3

Assemble the program. Then, set a breakpoint at the move instruction on line 21. Run the program up to the breakpoint, and then complete the table below by showing the 32-bit (hexadecimal) values contained in each of the four registers: D0-D2 and A1. **[2 points]**

D0	D1	D2	A1

Step 4

Now select **Stack** from the **view** menu. What are the initial values in A7 and SS? [1 point]

A7	SS

Step 5

Using the trace facility, execute the three move (push) instructions on lines 21-23. Notice how the stack view of memory is automatically updated as values are pushed onto the stack. In the table below, record the 32-bit (hexadecimal) value contained in the stack pointer before and after each instruction executes. [3 points]

Instructions	A7/SS (Before)	A7/SS (After)
MOVE.L D0, -(A7)		
MOVE.W D1, -(A7)		
MOVE.B D2, -(A7)		

Why does the MOVE.B instruction result in the stack pointer being decremented by two (bytes) rather than one (byte)? Explain. [2 points]

--

Use the **Stack** view to see what values get pushed onto the stack, and then use that information to complete the memory map below. (Notice: the memory map is organized to provide a *byte-view* of memory.) When completing the memory map, show the initial and the final positions of the stack pointer before and after each push instruction. Also, show the contents of each memory location as an 8-bit (hexadecimal) value. [7 points]

0x01000000		← A7/SS
0x00FFFFFF		
0x00FFFFFFE		
0x00FFFFFFD		
0x00FFFFFFC		← A7 = after Move.l -(A7)
0x00FFFFFFB		
0x00FFFFFFA		← A7 = after move.w A, -(A7)
0x00FFFFFF9		
0x00FFFFFF8		← A7 = after move.b D2, -(A7)
0x00FFFFFF7		
0x00FFFFFF6		
0x00FFFFFF5		
0x00FFFFFF4		
0x00FFFFFF3		
0x00FFFFFF2		
0x00FFFFFF1		

Step 6

Using the trace facility, execute the three pop (move) instructions on lines 27-29. In the table below, record the 32-bit (hexadecimal) value contained in the stack pointer before and after each instruction executes, along with the contents of D0-D2 after each instruction executes.

[4.5 points]

Instructions	A7/SS (Before)	A7/SS (After)	D2/D1/D0 (After)
MOVE.B (A7)+, D2			
MOVE.W (A7)+, D1			
MOVE.L (A7)+, D0			

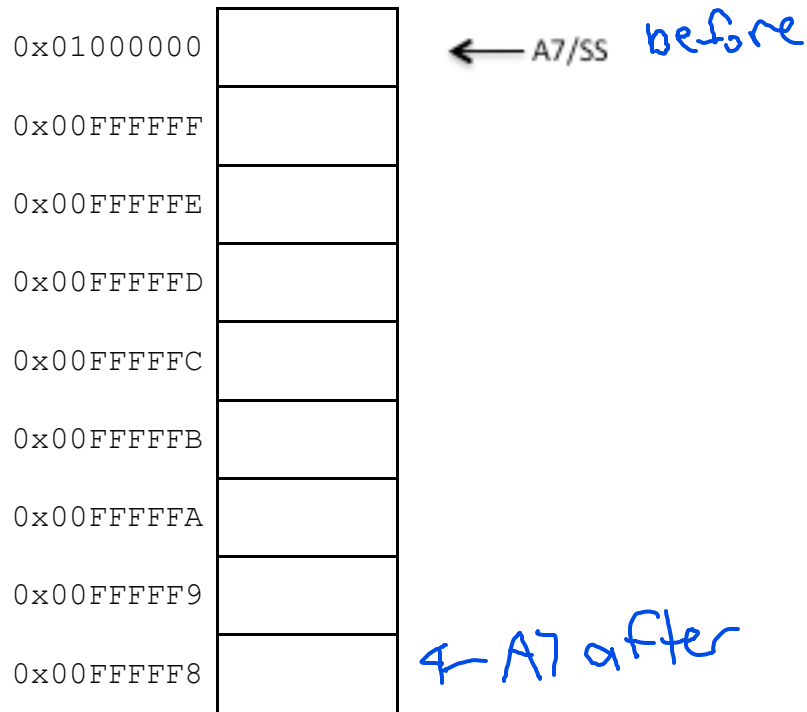
Use the stack view to see what values get popped from the stack. Are the original values pushed onto the stack still on the stack? Explain. **[2 points]**

Step 7

Using the trace facility, execute the four instructions on lines 33-36. In the table below, record the 32-bit (hexadecimal) value contained in the stack pointer before and after each instruction executes, as well as the values of D1 and A1 after each instruction executes. **[4 points]**

Instructions	A7/SS (Before)	A7/SS (After)	A1 (After)	D1 (After)
MOVEM.L A1/D1, -(A7)				
LEA 0, A1				
CLR.L D1				
MOVEM.L (A7)+, A1/D1				

Use the **Stack** view to see what values get pushed onto the stack, and then use that information to complete the memory map below. When completing the memory map, show the initial and the final positions of the stack pointer before and after each push instruction. Also, show the contents of each memory location as an 8-bit (hexadecimal) value. **[6 points]**

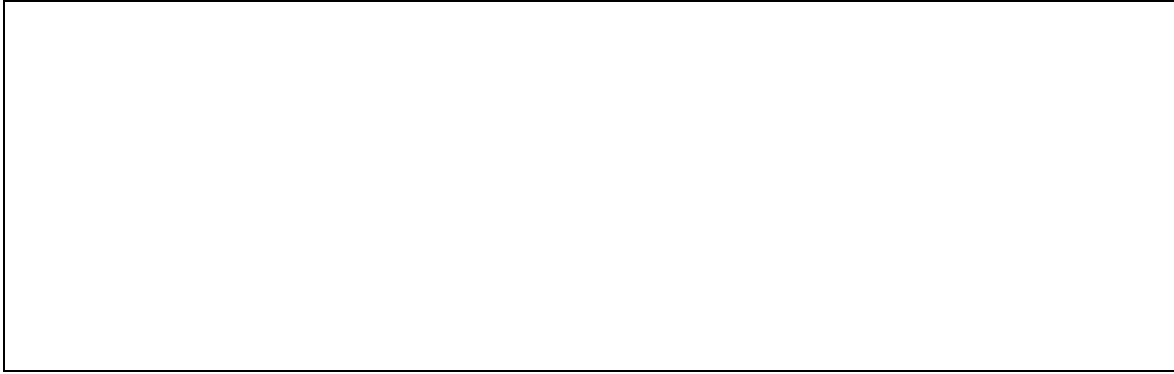


Step 8

Based on Lab5a.X68 create a new program **Lab5b.X68** by adding the following statement to the start of the original program (remember to delete the original **START:** label):

```
START:  ANDI.W  #$DFFF, SR
```

What affect does the previous instruction have on the status register SR and the operation of the new program? Explain. Does the program operate correctly? Your explanation should make reference to which stack is being used. **[2 points]**



Assemble the program, and then trace through the execution of the new program to ensure that your previous answer is correct.

Part 3: Using the Stack to Call and Return from Subroutines

Just like C functions, subroutines are individual pieces of code that can be located anywhere in memory, and which can be called by other pieces of code by *name* (Fig. 2). At the assembly-language level, the name of a subroutine is the *label* assigned to it in the source code. As described in class, the 68000 ISA describes two instructions that can be used to call a subroutine: BSR and JSR. The primary difference between these two instructions is the address modes that they employ for specifying the starting address of the subroutine. Similar to the branch instructions we looked at earlier, the BSR instruction uses PC-relative addressing, where the address of the subroutine is specified using a label. However, just like the earlier branch instructions, using PC-relative addressing means that the location of the subroutine in memory is encoded as a 16-bit displacement that is added to the contents of the PC to form the starting address of the subroutine at runtime. Since the displacement is limited to 16 bits, the subroutine must be located within a range of +/- 32K bytes from the BSR instruction that is used to call the subroutine. To call a subroutine located *anywhere* in memory the JSR instruction can be used. Unlike BSR, the JSR instruction supports a variety of address modes that can be used to specify the location of the subroutine. In each case, the starting address of the subroutine is treated as a full 32-bit address.

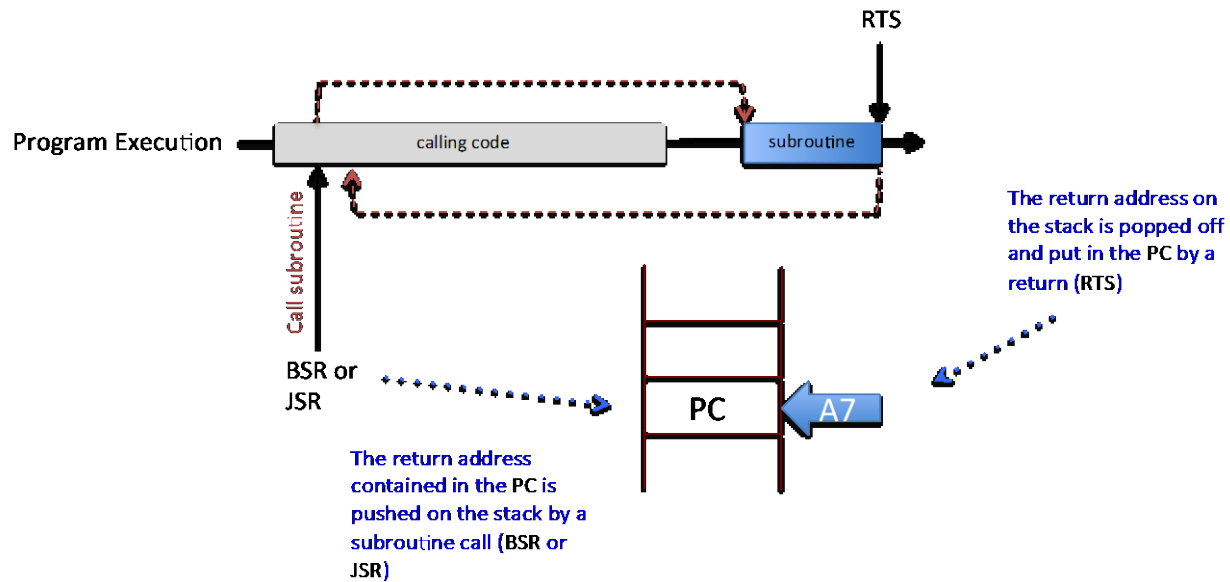


Figure 2: Subroutine Call-Return mechanism.

As explained in class, both BSR and JSR instructions do two things: First, they save the return address (i.e., the address of the following instruction in memory) by pushing the 32-bit value contained in the PC onto the runtime stack. Second, they pass control to the subroutine, by updating the PC with the starting address of the subroutine. Once the subroutine completes executing, it can return to the calling code by executing the RTS instruction. This instruction restores the saved PC (i.e., return address) by popping the long word off of the top of the stack and putting the value into the PC. Of course, prior to executing the RTS instruction it is up to the programmer to ensure that the stack pointer is pointing to the return address on the stack! This can be done by following a strict convention for calling and returning from subroutines, which will be described later in this lab.

Step 1

Download the sample program called **Lab5c.X68** from the course website.

Step 2

Start Easy68K. Once running, load the file Lab5c.X68 using the [File->Open File](#) menu choice. You should see something similar to below. (Remember to properly comment your code.)

```

EASy68K Editor/Assembler v5.16.01 - [L5c.X68]
File Edit Project Options Window Help

*-----*
* Title      : Lab5c
* Written by  : Your name goes here
* Date       : dd/mm/yyyy
* Description: Demonstration of subroutine call-return using
*              runtime stack (A7) to hold return address
*-----*

        ORG      $8000

**** Main (calling) Code ****

START:  MOVE.L   #13,D0           ;task number for displaying string
        LEA      MSG0,A1         ;point to MSG0
        TRAP     #15             ;system call

        BSR      FOO            ;call subroutine foo()

        LEA      MSG2,A1         ;point to MSG2
        TRAP     #15             ;system call

        SIMHALT

**** Subroutine foo() ****
FOO      LEA      MSG1,A1         ;point to MSG1
        TRAP     #15             ;system call
        RTS                      ;return to calling code

MSG0     DC.B     '1. Running calling code',0
MSG1     DC.B     '2. Control passed from calling code to subroutine foo()', 0
MSG2     DC.B     '3. Control passed from subroutne foo() back to calling code',0

        END      START

```

Step 3


Assemble the program. Now, examine the listing file and identify the memory addresses for the three instructions listed in the table below; that is, identify where each instruction is located in memory. When completing the table below, show a full 32-bit (hexadecimal) address. **[1.5 points]**

Instruction	Memory Address
BSR FOO	
LEA MSG2,A1	
LEA MSG1,A1	

Step 4

Set a breakpoint at the `BSR FOO` instruction, and then run the program. What is the current 32-bit (hexadecimal) value contained in the PC, and what does this value represent? [2 points]

Now, click on the view menu and select the Stack option. Prior to using `BSR FOO` to call the subroutine `foo` the runtime stack should look similar to the one shown below:

0x01000000	FF	 <div style="border: 1px solid black; padding: 2px; display: inline-block;">A7 = 01000000</div>
0x00FFFFFF	FF	
0x00FFFFFFE	FF	
0x00FFFFFFD	FF	
0x00FFFFFFC	FF	
0x00FFFFFFB	FF	
0x00FFFFFFA	FF	
0x00FFFFFF0	FF	
0x00FFFFFF9	FF	

Step 5

Now, use the trace facility to execute the `BSR FOO` instruction. What is the current 32-bit (hexadecimal) value contained in the PC, and what does this value represent? [2 points]

Notice that a long word has been pushed onto the runtime stack as a result of executing the `BSR FOO` instruction. Show the value of this long word, and draw the new location of the stack pointer (A7) in the memory map below. **[3 points]**

0x01000000	
0x00FFFFFF	
0x00FFFFFE	
0x00FFFFFD	
0x00FFFFFC	
0x00FFFFFB	FF
0x00FFFFFA	FF
0x00FFFFF0	FF
0x00FFFFF9	FF

What does the previous long word value represent? **[1 point]**

Step 6

Now that control has been passed to subroutine *foo*, continue to trace through the execution of the subroutine one instruction at a time until you reach the `RTS` instruction. Before executing the `RTS` instruction, what are the 32-bit (hexadecimal) values contained in the PC and A7? [1 point]

Before Executing the RTS Instruction	
PC	A7

What are the values contained in the PC and A7 after executing the `RTS` instruction? [1 point]

After Executing the RTS Instruction	
PC	A7

Now that program control has returned to the `LEA MSG2,A1` instruction immediately following the original `BSR FOO` instruction, continue with the execution of the program. Notice that in the **I/O display window**, a sequence of statements is printed during the execution of the program showing how program control starts in the calling code, is passed to the subroutine, and finally passed back to the calling code.

Sim68K I/O

```
1. Running calling code
2. Control passed from calling code to subroutine foo()
3. Control passed from subroutine foo() back to calling code
```

Part 4: Using the Stack to Preserve Register Values Between Calls

As discussed in class, a subroutine should be written in such a way that it is *transparent* to the calling code. In other words, since both the calling code and the subroutine must share the same data and address registers, the subroutine should not assume that it can overwrite the original contents of registers, as these values may be required by the calling code once

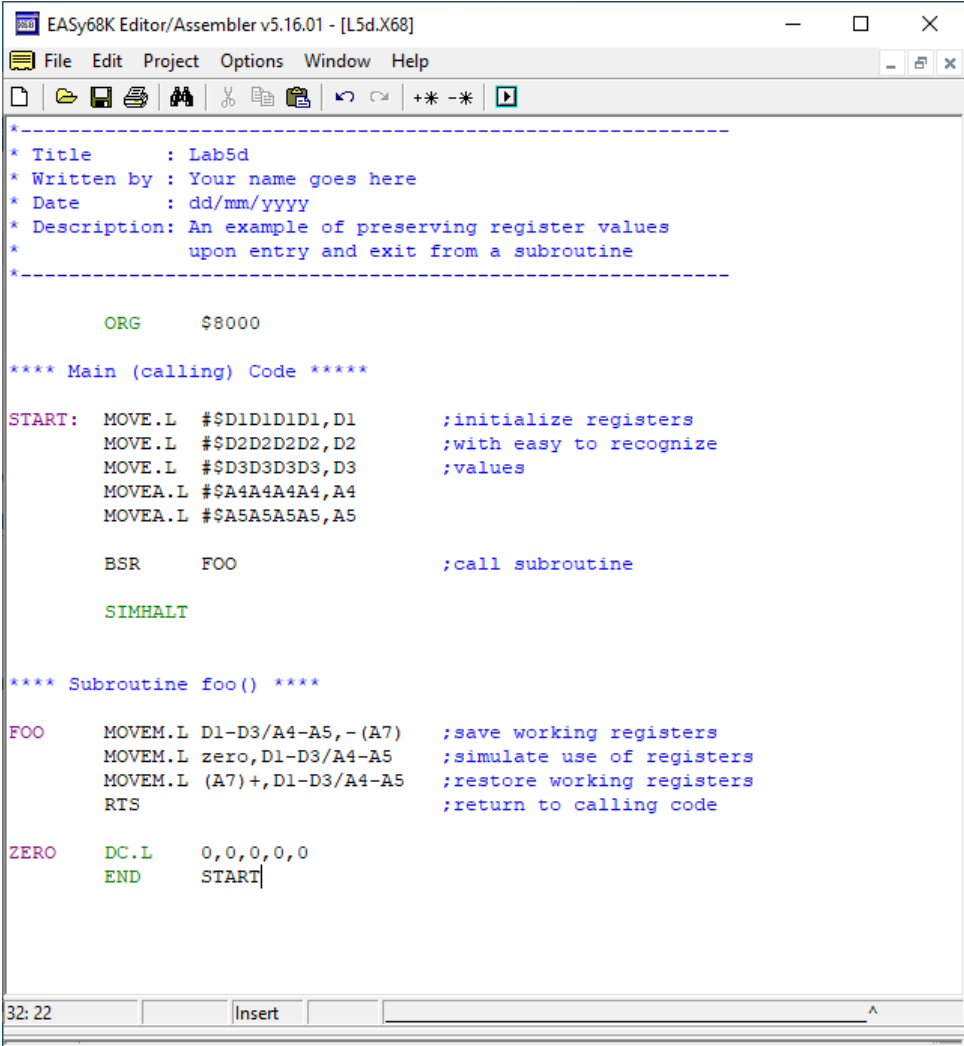
the subroutine returns. Transparency ensures that the original contents of registers remains unchanged during the execution of the subroutine. Transparency is achieved by saving any registers used by the subroutine on entry to the subroutine, and by restoring the contents of these registers at the end of the subroutine just prior to returning to the calling code. Such registers are commonly referred to as *working registers*, and are (temporarily) saved on the runtime stack.

Step 1

Download the sample program called **Lab5d.X68** from the course website.

Step 2

Start Easy68K. Once running, load the file Lab5d.X68 using the [File->Open File](#) menu choice. You should see something similar to below. (Remember to properly comment your code.)



```
EASy68K Editor/Assembler v5.16.01 - [L5d.X68]
File Edit Project Options Window Help

*-----*
* Title      : Lab5d
* Written by  : Your name goes here
* Date       : dd/mm/yyyy
* Description: An example of preserving register values
*             upon entry and exit from a subroutine
*-----*

        ORG      $8000

**** Main (calling) Code ****

START:  MOVE.L   #$D1D1D1D1,D1      ;initialize registers
        MOVE.L   #$D2D2D2D2,D2      ;with easy to recognize
        MOVE.L   #$D3D3D3D3,D3      ;values
        MOVEA.L  #$A4A4A4A4,A4
        MOVEA.L  #$A5A5A5A5,A5

        BSR      FOO                ;call subroutine

        SIMHALT

**** Subroutine foo() ****

FOO      MOVEM.L  D1-D3/A4-A5,-(A7)   ;save working registers
        MOVEM.L  zero,D1-D3/A4-A5   ;simulate use of registers
        MOVEM.L  (A7)+,D1-D3/A4-A5   ;restore working registers
        RTS                          ;return to calling code

ZERO     DC.L     0,0,0,0,0
        END      START

32:22      Insert
```

Step 3

Assemble the program. Next, set a breakpoint at the first instruction in subroutine *foo*:
`MOVEM.L D1-D3/A4-A5, -(A7)`. Now, run the program until it stops executing at the breakpoint. Go to the view menu and select the **Stack** option. Show the value of the return address pushed onto the runtime stack, and draw the location of the stack pointer (A7) in the (long word) memory map below. **[2 points]**

0x01000000	
0x00FFFFFFC	
0x00FFFFFF8	
0x00FFFFFF4	
0x00FFFFFF0	
0x00FFFFFEC	
0x00FFFFFE8	
0x00FFFFFE4	
0x00FFFFFE0	

→ A7

Step 4

Using trace mode, execute the first instruction at the start of the subroutine. Show the values of all of the working registers that are pushed onto the runtime stack, and draw the final location of the stack pointer (A7) in the (long word) memory map below. **[4 points]**

0x01000000	
0x00FFFFFFC	
0x00FFFFFF8	
0x00FFFFFF4	
0x00FFFFFF0	
0x00FFFFEC	
0x00FFFFE8	
0x00FFFFE4	
0x00FFFFE0	

Step 5

Using trace mode, execute the next instruction: `MOVEM.L zero,D1-D3/A4-A5`. This instruction simply seeks to simulate the use of working registers in the subroutine, and is for illustrative purposes only. What are the 32-bit (hexadecimal) values contained in the working registers after the instruction executes? **[2.5 points]**

D1	D2	D3	A4	A5

Step 6

Using trace mode, execute the next instruction: `MOVEM.L (A7)+,D1-D3/A4-A5`. This instruction restores the original values contained in the working registers from the stack before the subroutine returns. What are the 32-bit (hexadecimal) values in working registers after the instruction executes? **[2.5 points]**

D1	D2	D3	A4	A5

In the memory map below, show the values on the stack once the working registers have been restored. Draw the current location of the stack pointer (A7). **[1.5 points]**

0x01000000	
0x00FFFFFFC	
0x00FFFFFF8	
0x00FFFFFF4	
0x00FFFFFF0	
0x00FFFFEC	
0x00FFFFE8	
0x00FFFFE4	
0x00FFFFE0	

← A7

What is the stack pointer (A7) currently pointing at? Explain. **[2 points]**

Run the remainder of the program. Notice that the contents of the working registers are protected within the subroutine and, therefore, preserved. As a general rule when writing subroutines, you should always save the working registers upon entering a subroutine and restore them just before returning from the subroutine.

Part 5: Passing arguments to Subroutines

As discussed in class, there are two common ways to pass arguments to subroutines: in *registers* or on the *stack*. The former method is efficient, but lacks flexibility due to the limited number of available data and address registers. The latter method accommodates any number of arguments, but is less efficient due to the time required to push, access, and remove arguments residing in memory. In practice, both methods are employed, and sometimes even together.

We will focus mainly on passing arguments to subroutines using the stack, as passing values via registers is clear enough. Both the *caller* (main code) and the *callee* (subroutine) have specific tasks to perform, as defined by the C language and described below.

The *caller* performs the following tasks:

- Pushes the arguments that the subroutine is expecting onto the runtime stack. These arguments must be pushed in the right order; that is, in the order in which the subroutine is expecting them to appear on the stack.
- Calls the subroutine.
- After the call, the caller is responsible for removing the original arguments from the runtime stack. This freeing of memory can be done in a single step by adding a value to the stack pointer equal to the total size of the arguments pushed onto the stack in bytes.

The *callee* performs the following tasks:

- Saves any working registers by pushing them onto the runtime stack.
- Performs some task. To do this, it may be necessary to access the arguments on the runtime stack using indirect addressing with displacement. The callee never pops the arguments off of the stack!
- Restore any working registers by popping them off of the runtime stack.
- Return to the calling code.

In general, arguments can either be *passed-by-value* or *passed-by-reference*. In the latter case, the `PEA` instruction can be used to push the address of the argument onto the runtime stack. Also, in situations where it is necessary to pull information from the stack or put information on the stack without changing the stack pointer, the indirect addressing with offset addressing mode can be used: `d16(A7)`. Table 1 provides a summary of the most common operations that can be performed on the system stack, either by the caller or the callee as appropriate.

Table 1: Summary of Stack Operations.

Stack Operation	Instruction
Push	MOVE <ea>, - (A7)
Pop	MOVE (A7) +, <ea>
Push Address	PEA <ea>
Pull (without changing stack pointer)	MOVE <ea>, d16 (A7)
Put (without changing stack pointer)	MOVE <ea>, d16 (A7)

An example of the previous caller/callee requirements is given next. This example illustrates several of the stack operations listed in Table 1.

Step 1

Download the sample program called **Lab5e.X68** from the course website.

Step 2

Start Easy68K. Once running, load the file Lab5e.X68 using the [File->Open File](#) menu choice. You should see something similar to below. (Remember to properly comment your code.)

```

EASy68K Editor/Assembler v5.16.01 - [L5e.X68]
File Edit Project Options Window Help
-----
* Title      : Lab5e
* Written by : Your name goes here
* Date       : dd/mm/yyyy
* Description: An example of a subroutine to compute the
*              average of a list of 16-bit numbers
*-----
      ORG      $8000

**** Main Code ****

MAIN    PEA     LIST           ;push pointer to list
        MOVE.W  #LENGTH,-(A7) ;push length of list
        BSR     AVERAGE       ;calculate average
        LEA     6(A7),A7        ;remove arguments

        SIMHALT

**** Subroutine: AVERAGE(LENGTH,*LIST) ****
*      ARG1: LENGTH at 18(A7)      *
*      ARG2: *LIST at 16(A7)      *
*      Returns average in data register D0 *
*****
AVERAGE MOVEM.L  D1/D2/A1,-(A7) ;save working registers
        CLR.L    D2             ;set sum to zero
        MOVEA.L  18(A7),A1       ;initilize pointer to list
        MOVE.W   16(A7),D1       ;initiaize loop counter
LOOP     BEQ     EXIT           ;counter = 0? yes, then exit
        ADD.W    (A1)+,D2        ;no, add list item to sum
        SUB.W    #1,D1          ;decrement loop counter
        BRA     LOOP           ;do it again!
EXIT     EXT.L    D2             ;extend sum to form dividend
        DIVS     16(A7),D2       ;compute quotient and remainder
        MOVE.W   D2,D0          ;move quotient to D0
        MOVEM.L  (A7)+,D1/D2/A1 ;restore working registers
        RTS

LIST     DC.W    147,23,-56,432,79 ;list of numbers to average
LENGTH   EQU     (*-LIST)/2        ;size of list in words

        END      MAIN

```

The program above uses a subroutine to compute the average of a list of 16-bit signed numbers. The subroutine has two arguments: the length of the list and a pointer to the start of the list in memory. Prior to calling the subroutine, the caller pushes both of these items onto the runtime stack. The subroutine is then called, causing the return address to be pushed onto the stack. On entry, the subroutine first saves all working registers on the stack, before copying (pulling) its arguments from the stack and putting them into registers. The subroutine then proceeds to compute the average of the list of numbers. Although both the

remainder and the quotient are computed, only the quotient is returned to the calling code in data register D0. Upon return, the calling code removes the original arguments from the stack, putting the stack pointer back to its original location prior to the call. Before proceeding, simulate the program and make sure that you understand the mechanics of this program.

Step 3

Below, you will find a series of memory maps corresponding to the part of memory used to implement the runtime stack. At the bottom of each map is a particular instruction from the program above that affects the runtime stack in some specific way. Complete each map showing the effect that the instruction has on the stack after it executes. In each case, show what (hexadecimal) values get pushed onto the stack or removed from the stack. Also, draw an arrow showing where the stack pointer is pointing in each case. [28 points]

0x01000000	
0x00FFFFFF	
0x00FFFFFFE	
0x00FFFFFFD	
0x00FFFFFFC	
0x00FFFFFFB	
0x00FFFFFFA	
0x00FFFFFF9	
0x00FFFFFF8	
0x00FFFFFF7	
0x00FFFFFF6	
0x00FFFFFF5	
0x00FFFFFF4	
0x00FFFFFF3	
0x00FFFFFF2	
0x00FFFFFF1	
0x00FFFFFF0	
0x00FFFFFFF	
0x00FFFFFFE	
0x00FFFFFFD	
0x00FFFFFFC	
0x00FFFFFFB	
0x00FFFFFFA	

After `PEA LIST`

| 0000 832

0x01000000	
0x00FFFFFF	
0x00FFFFFFE	
0x00FFFFFFD	
0x00FFFFFFC	
0x00FFFFFFB	
0x00FFFFFFA	
0x00FFFFFF9	
0x00FFFFFF8	
0x00FFFFFF7	
0x00FFFFFF6	
0x00FFFFFF5	
0x00FFFFFF4	
0x00FFFFFF3	
0x00FFFFFF2	
0x00FFFFFF1	
0x00FFFFFF0	
0x00FFFFFFF	
0x00FFFFFFE	
0x00FFFFFFD	
0x00FFFFFFC	
0x00FFFFFFB	
0x00FFFFFFA	

After `MOVE.W #LENGTH, -(A7)`

222 0055

0x01000000	
0x00FFFFFF	
0x00FFFFFFE	
0x00FFFFFFD	
0x00FFFFFFC	
0x00FFFFFFB	
0x00FFFFFFA	
0x00FFFFFF9	
0x00FFFFFF8	
0x00FFFFFF7	
0x00FFFFFF6	
0x00FFFFFF5	
0x00FFFFFF4	
0x00FFFFFF3	
0x00FFFFFF2	
0x00FFFFFF1	
0x00FFFFFF0	
0x00FFFFFFF	
0x00FFFFFFE	
0x00FFFFFFD	
0x00FFFFFFC	
0x00FFFFFFB	
0x00FFFFFFA	

After `BSR AVERAGE`

22 0000 8327

0x01000000	
0x00FFFFFF	
0x00FFFFFFE	
0x00FFFFFFD	
0x00FFFFFFC	
0x00FFFFFFB	
0x00FFFFFFA	
0x00FFFFFF9	
0x00FFFFFF8	
0x00FFFFFF7	
0x00FFFFFF6	
0x00FFFFFF5	
0x00FFFFFF4	
0x00FFFFFF3	
0x00FFFFFF2	
0x00FFFFFF1	
0x00FFFFFF0	
0x00FFFFFEF	
0x00FFFFFEE	
0x00FFFFFED	
0x00FFFFFEC	
0x00FFFFFEB	
0x00FFFFFEA	

After **MOVEM.L D1/D2/A1, -(A7)**

000' 0112
011 0112
0011 0112

0x01000000	
0x00FFFFFF	
0x00FFFFFFE	
0x00FFFFFFD	
0x00FFFFFFC	
0x00FFFFFFB	
0x00FFFFFFA	
0x00FFFFFF9	
0x00FFFFFF8	
0x00FFFFFF7	
0x00FFFFFF6	
0x00FFFFFF5	
0x00FFFFFF4	
0x00FFFFFF3	
0x00FFFFFF2	
0x00FFFFFF1	
0x00FFFFFF0	
0x00FFFFFEF	

After **LEA 6(A7), A7**

0x01000000	
0x00FFFFFF	
0x00FFFFFFE	
0x00FFFFFFD	
0x00FFFFFFC	
0x00FFFFFFB	
0x00FFFFFFA	
0x00FFFFFF9	
0x00FFFFFF8	
0x00FFFFFF7	
0x00FFFFFF6	
0x00FFFFFF5	
0x00FFFFFF4	
0x00FFFFFF3	
0x00FFFFFF2	
0x00FFFFFF1	
0x00FFFFFF0	
0x00FFFFFEF	
0x00FFFFFEE	
0x00FFFFFED	
0x00FFFFFEC	
0x00FFFFFEB	
0x00FFFFFEA	

After **MOVE.L (A7)+, D1/D2/A1**

0x01000000	
0x00FFFFFF	
0x00FFFFFFE	
0x00FFFFFFD	
0x00FFFFFFC	
0x00FFFFFFB	
0x00FFFFFFA	
0x00FFFFFF9	
0x00FFFFFF8	
0x00FFFFFF7	
0x00FFFFFF6	
0x00FFFFFF5	
0x00FFFFFF4	
0x00FFFFFF3	
0x00FFFFFF2	
0x00FFFFFF1	
0x00FFFFFF0	
0x00FFFFFEF	
0x00FFFFFEE	
0x00FFFFFED	
0x00FFFFFEC	
0x00FFFFFEB	
0x00FFFFFEA	

After **RTS**

In this portion of the lab, you will gain experience writing subroutines, making multiple subroutine calls, and passing arguments to subroutines using both registers and the stack, and returning values using multiple registers.

Part 6: Programming Problem Description

Your task is to write a program to input a list of numbers from the keyboard, and then output the minimum, maximum, and the average of the numbers in the list. To do this, you will need to write three subroutines, and a single main code that calls these subroutines. The functionality of each subroutine is described below. **[20 points each]**

Function 1: *int get_data(*list)*

This subroutine is responsible for reading the list of numbers entered by the user, and is based on the following steps:

- An ASCII text message should be sent to the screen prompting the user to enter the number of 32-bit values in the list. The user must supply a value in the range of 1 to 10, and error checking should be performed to ensure that this is what happens.
- Once the user has entered a valid list size, a series of ASCII text messages should be sent to the screen prompting the user to enter each value.
- As each number is entered, it should be stored in the list. The address of the list should be passed in an address register to the subroutine from the main code.
- All input-output that occurs within the subroutine should be handled using the simulator's I/O TRAP #15. In particular, task 14 can be used to display a null-terminated ASCII string on the screen, while task 4 can be used to read a signed number entered at the keyboard. The Easy68K help menu can be used to learn more about simulator I/O.

Function 2: *int Statistics(int number, *list, *struct)*

This second subroutine is responsible for scanning the list of numbers entered by the user, and then determining the maximum value in the list, the minimum value, and the average of the values in the list. The main steps to be performed are as follows:

- The subroutine has three arguments, all of which are passed to the subroutine from the main function on the stack. These arguments include the number of values in the list, the address of the list, and the address of a structure used to hold the results of the calculation. This structure has four members: the maximum value in the list, the minimum value in the list, the quotient, and the remainder for the average. The maximum and minimum are longwords, while the quotient and remainder are words.

- The subroutine should first compute the maximum, and then use the address of the structure to store this result in the appropriate location within the structure.
- Next, the subroutine should compute the minimum, and then store the result in the appropriate location in the structure.
- Finally, the subroutine should compute the average, and store the quotient and remainder in the appropriate locations within the structure.
- Altogether, there are four return values, all of which are returned (or updated) from within the subroutine.

Function 3: `display(int max, int min, int quotient, int remainder)`

This subroutine is responsible for displaying the minimum, maximum and average values on the screen, and is based on the following steps:

- The maximum, minimum, quotient, and remainder are to be passed into this subroutine as individual values on the stack.
- The subroutine will then display the minimum value on the screen, followed by the maximum value, followed by the average value. The latter should be expressed as a quotient followed by a remainder.
- All I/O should be performed using TRAP #15. Task 3 can be used to display a signed number on the screen, while task 6 can be used to display a single ASCII character on the screen. Consult the Easy68K help menu for more details.

The *main* code (or calling code) should setup all of the required memory storage for up to 10 values in the list, as well as the memory required by the structure used to hold the minimum, maximum, quotient and remainder. The main code will then call the previous functions in order, providing them with the appropriate input parameters. The calling code is responsible for removing all items pushed onto the stack once finished. Also, each individual subroutine should be transparent to the calling code.

Make sure to fully document your code, clearly stating the input parameters and return values for each subroutine, along with an explanation of how they are used.

Below, you can see examples of interacting with the program. In this first case, the user first indicates that they wish to enter 5 values. The user then enters the values 1, 2, 2, 3, and 4 in sequence. The program then calculates the statistics, before outputting the results.

```
Sim68K I/O
Enter number of values in list (1-10): 5
Enter number: 1
Enter number: 2
Enter number: 2
Enter number: 3
Enter number: 4
Minimum = 1
Maximum = 4
Average = 2 2/5
```

In this second example, the user initially fails to enter a value number of items in the range of 1 to 10, and is repeatedly prompted to do so.

```
Sim68K I/O
Enter number of values in list (1-10): -2
Enter number of values in list (1-10): 13
Enter number of values in list (1-10): 5
Enter number: 1
Enter number: 2
Enter number: 2
Enter number: 3
Enter number: 4
Minimum = 1
Maximum = 4
Average = 2 2/5
```