Stack Frames and Recursion

CIS*2030
Lab Number 6

Name: Zachery Abbas

Mark: __________/94

Overview

In addition to parameters, a subroutine may require its own *local* variables. Such variables are created upon entry to the subroutine, accessible only to code within the subroutine, and destroyed upon exit from the subroutine. There are three important questions that must be addressed when seeking to implement local variables: (1) *when* is the memory for local variables allocated/de-allocated, (2) *how* is the memory allocated/deallocated, and (3) *where* are the local variables located in memory?

The answers to all of these questions involves a special data-structure called a *stack frame*. A stack frame is a data structure created on the stack at the beginning of the subroutine to provide temporary storage for local variables. At the end of the subroutine, the stack frame is deallocated, destroying the local variables and returning the stack to its former state. Most ISAs, including the 68000, provide a pair of complementary instructions for allocating and deallocating stack frames.

The main advantage of dynamically allocating memory for local variables on the stack versus statically allocating memory using assembler directives, like DC and DS, is that memory space for local variables is allocated only when needed. Moreover, each time that a particular subroutine is called, a new stack frame is allocated to it (with a new set of local variables). This allows the function to be recursive, as it ensures that the local variables in the subroutine are not bound to the subroutine, but to each instance (or call) to the subroutine.

Objectives

Upon completion of this lab you will:

- Understand activation records,
- Understand how local variables are implemented,
- Understand what is meant by a stack frame and a frame pointer,
- Understand how to link and unlink stack frames,
- Understand how stack frames can be used for temporary values and local variables,
- Understand the role of stack frames in recursion,
- Understand how to draw a memory map showing the contents of the stack when using stack frames.

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 3.2.1 (LINK and UNLK)

Introduction

As explained in class, the 68000 provides two complementary instructions for allocating and deallocating a stack frame: LINK and UNLK. Figure 1a shows how the former instruction can be used to allocate a stack frame with 6 bytes of memory on the stack.

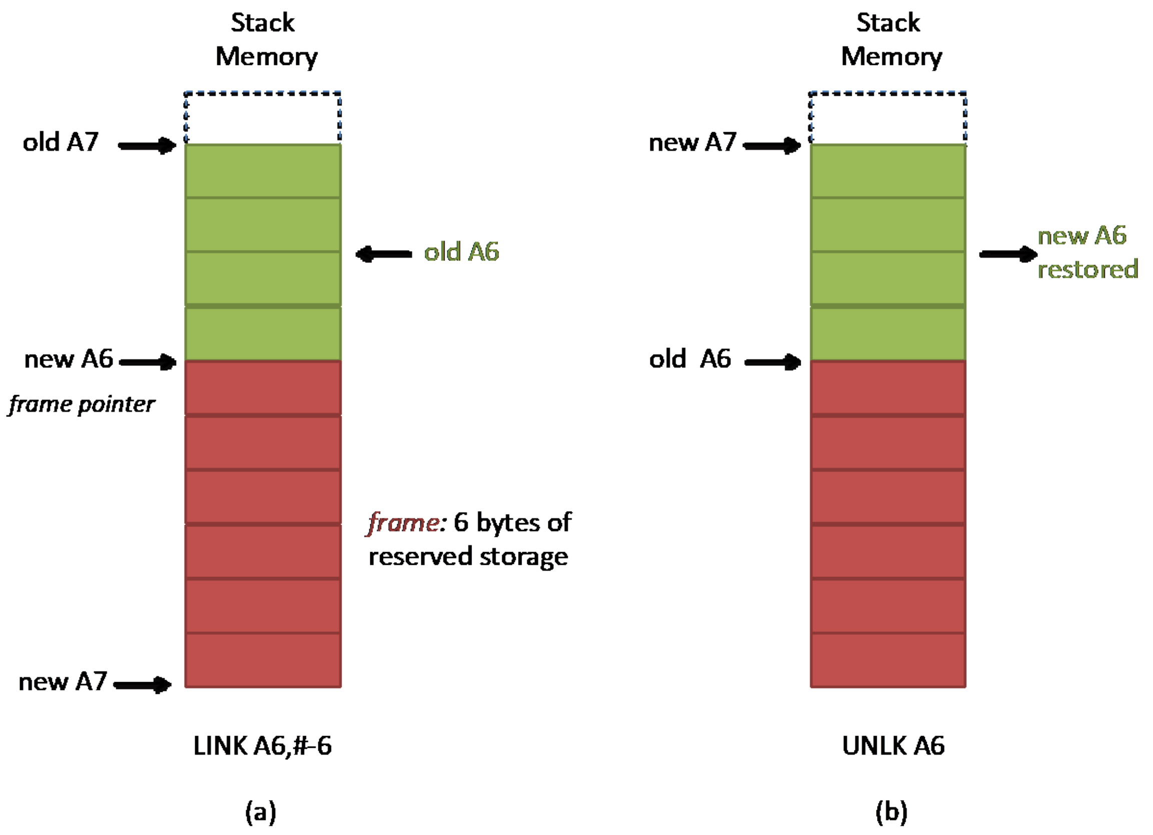


Figure 1: Allocation and de-allocation of stack frames.

The instruction `LINK A6,#-6` first saves the original contents of `A6` by *pushing* `A6` onto the stack. It then saves the original contents of the stack pointer by copying `A7` into `A6`. Finally, it adds the immediate operand (`-6`) to `A7` moving the stack pointer to the position shown. The resulting six bytes of allocated space is referred to as the *frame*, and the new pointer (`A6`) is referred to as the *frame pointer*. The frame can now be used to store local or temporary values, and can be accessed using the frame pointer (not the stack pointer). The frame pointer is used when accessing the frame because the frame pointer remains fixed, while the stack pointer will probably change as the size of the stack grows (e.g., as a result of saving working registers).

As illustrated in Fig. 1b, The `UNLK A6` instruction can be used to deallocate the stack frame. The deallocation occurs by copying the current value in `A6` into the `A7`, and then popping the old value of `A6` from the stack and putting it back into `A6`. This has the effect of restoring both the system stack and the old frame pointer (`A6`) back to their original states.

When used together in the context of a *subroutine*, the previous instructions provide the following functionality:

- A way to *dynamically* allocate memory for local or temporary variables at the *beginning* of a subroutine.
- Through the frame pointer, a way to *access* local values or subroutine parameters that is independent of the stack pointer. Note: since the stack grows from higher memory addresses towards lower memory addresses, local variables will always be at fixed *negative* displacements from the frame pointer, while any formal parameters passed to the subroutine will be at fixed *positive* displacements. The previous displacements are fixed because the location of the frame pointer is fixed.
- A way to *free* memory used for local or temporary variables at the *end* of a subroutine.

The best way to learn about the mechanics of the LINK and UNLK instructions is to work through an example, as will be done next.

Part 1: Stack Frames and Local Variables

Consider the C function shown in Fig. 2. The function has three parameters: `a`, `b`, and `c`. The first two parameters are *in* parameters, while the third parameter is an *in-out* parameter. The function also makes use of two *local* variables: `x` and `y`. These local variables are used solely within the function, and only exist while the function is executing. Therefore, at the assembly language level, they are implemented using a stack frame. The function computes the expression $c = a^2 + b^2$.

```
void sumSquares(int a, b, *c) {

    int x,y;

    x  = a * a;
    y  = b * b;
    *c = x + y;
}
```

Figure 2: A C function that computes $c = a^2 + b^2$.

Step 1

Download the sample program called **Lab6a.X68** from the course website.

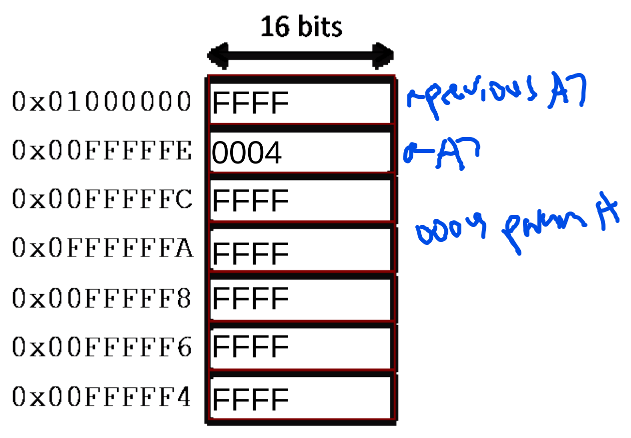
Step 2

Start Easy68K. Once running, load the file Lab6a.X68 using the File->Open File menu choice. Read through the program carefully.

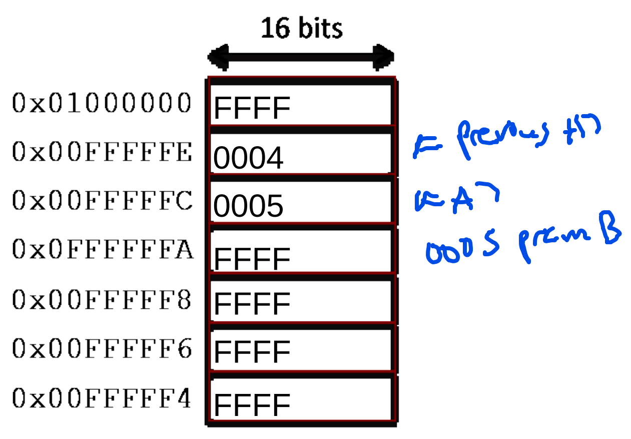
As shown below, the program starts with a prologue called `main`. This prologue is responsible for (1) passing the parameters (`a`, `b`, and `*c`) to the subroutine `sumSquares` using the stack, (2) calling `sumSquares`, which then uses the previous parameters to compute c = $a^2 + b^2$, and (3) removing the previous parameters from the stack upon return.

```
main     MOVE.W   #4,-(A7)          ;push a = 4
         MOVE.W   #5,-(A7)          ;push b = 5
         PEA      c                 ;push address of c
         BSR      sumSquares        ;call subroutine
         LEA      8(A7),A7          ;remove parameters from stack
```

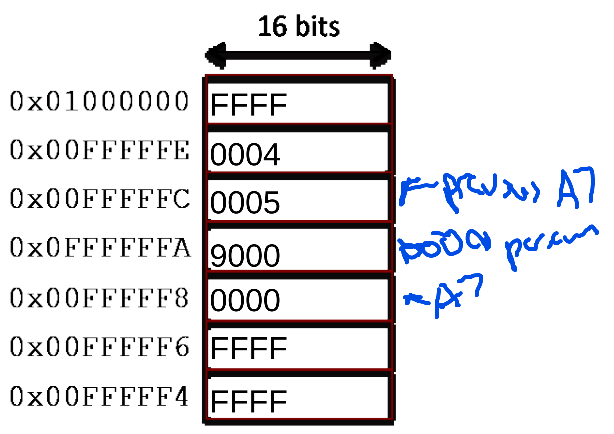
Assemble the program, then use the Easy68K's *trace* facility to execute the first four instructions in `main` located on lines 18 through 21. After each instruction executes, show the contents of the stack in the memory-maps on the next page. (Notice that the memory maps are based on a *word* view of memory, so each memory location contains a 2-byte value.) Make sure to draw the *initial* location of the stack pointer (`A7`), as well as the location of the stack pointer *after* each instruction executes. Also, add labels to right-hand side of the memory map that identify what each item on the stack refers to (e.g, "parameter a", "return address", etc.) Of course, the actual contents of each memory location should be expressed as a 16-bit (hexadecimal) value. **[8 points]**



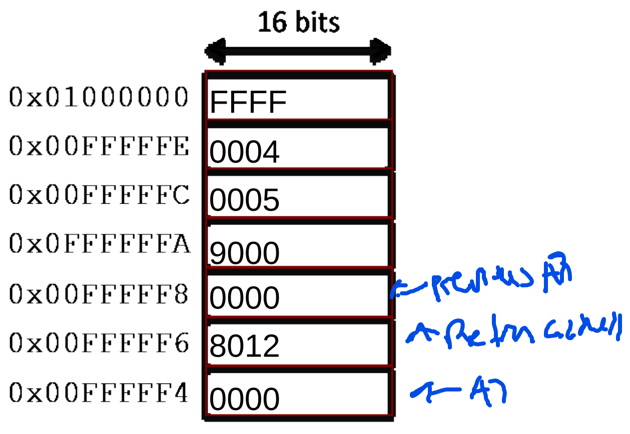
After `MOVE.W #4,-(A7)`



After `MOVE.W #5,-(A7)`



After `PEA C`



After `BSR sumSquares`

Step 3

The subroutine `sumSquares` computes the expression c = $a^2 + b^2$, and is based on the high-level code in Fig. 2.

```
sumSquares
        LINK     A6,#-8              ;allocate storage for two longwords
        MOVEM.L  A0/D0,-(A7)         ;save working registers
        MOVE.W   aOffset(A6),D0      ;get a
        MULS     D0,D0               ;compute a * a
        MOVE.L   D0,xOffset(A6)      ;x = a * a
        MOVE.W   bOffset(A6),D0      ;get b
        MULS     D0,D0               ;compute b * b
        MOVE.L   D0,yOffset(A6)      ;y = b * b
        MOVEA.L  cOffset(A6),A0      ;get address of c
        MOVE.L   xOffset(A6),D0      ;get x
        ADD.L    yOffset(A6),D0      ;add x + y
        MOVE.L   D0,(A0)             ;save in c
        MOVEM.L  (A7)+,A0/D0         ;restore working registers
        UNLK     A6                  ;deallocate stack frame
        RTS                          ;return to caller
```

The subroutine begins by allocating a stack frame of size 8 bytes for the two local variables `x` and `y`. Each variable is represented as a longword, as the subsequent multiplication operations generate a 32-bit product. Address register `A6` is used as the frame pointer, and provides access to both the formal parameters on the stack and the local variables in the frame.

Using the Easy68K trace facility, execute the first instruction in subroutine `sumSquare` located on line 28. After the instruction executes, show the contents of the stack in the memory-map below. Make sure to draw the location of the stack pointer (`A7`) and the location of the frame pointer (A6). Also, add labels to the side of the memory map that identify what each item on the stack refers to (e.g., "parameter a", "return address", "caller's frame pointer", "local variable x", etc.) The actual contents of each memory location should be expressed as a 16-bit (hexadecimal) value. **[9 points]**

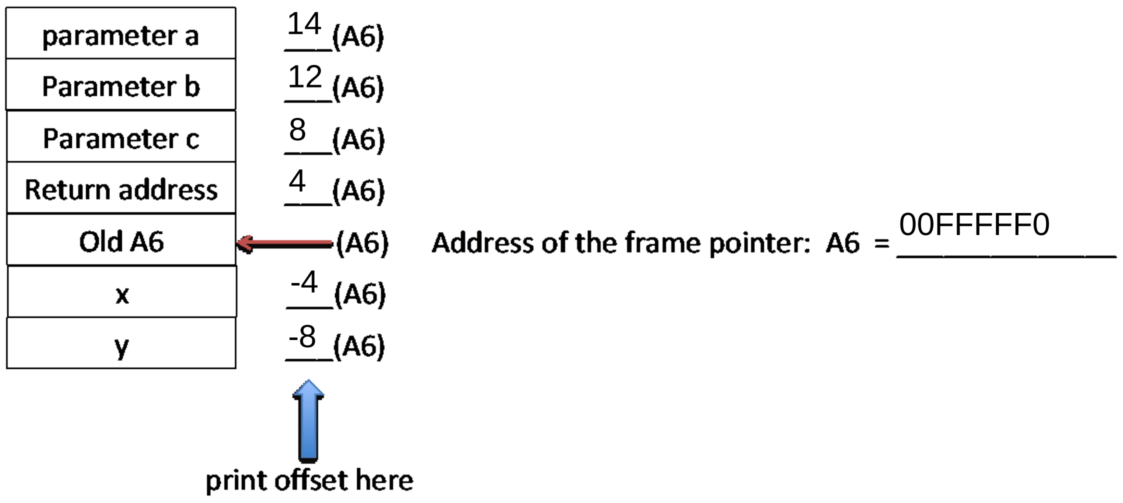
16 bits

0x01000000	FFFF	
0x00FFFFFE	0004	← param a
0x00FFFFFC	0005	← param b
0x0FFFFFFA	9000	} address of ptr c
0x00FFFFF8	0000	
0x00FFFFF6	8012	} return address
0x00FFFFF4	0000	
0x00FFFFF2	0000	
0x00FFFFF0	0000	← A6
0x00FFFFEE	FFFF	} local var x (all 8 bytes allocated
0x00FFFFEC	FFFF	
0x00FFFFEA	FFFF	
0x00FFFFE8	FFFF	← A7 local var y
0x00FFFFE6	FFFF	
0x00FFFFE4	FFFF	
0x00FFFFE2	FFFF	
0x00FFFFE0	FFFF	
0x00FFFFEF	FFFF	

Step 4

As the location of the frame pointer is fixed (i.e., it always points at the caller's frame pointer saved on the stack) it is not necessary to know the actual address of the formal parameters or the local variables. All that is required is to know how far away from the frame pointer the formal parameters and local variables are located on the stack. This way, a simple offset value can be added to the frame pointer to access them. In practice, all formal parameters passed to the subroutine on the stack have positive offsets from the frame pointer (e.g., parameter a has an offset of +14). Local variables have negative displacements from the frame pointer (e.g., x has an offset of −4). Knowing the offset of each item on the stack means that instructions can use indirect addressing with offset when accessing both formal parameters and local variables, as shown in the assembler code.

Complete the figure below, by first writing the actual memory address of the frame pointer (A6). Then write the (positive or negative) offsets to be added to the frame pointer (A6) to form the address of each item on the stack. Write these offsets in the spaces "___" provided below. **[4 points]**

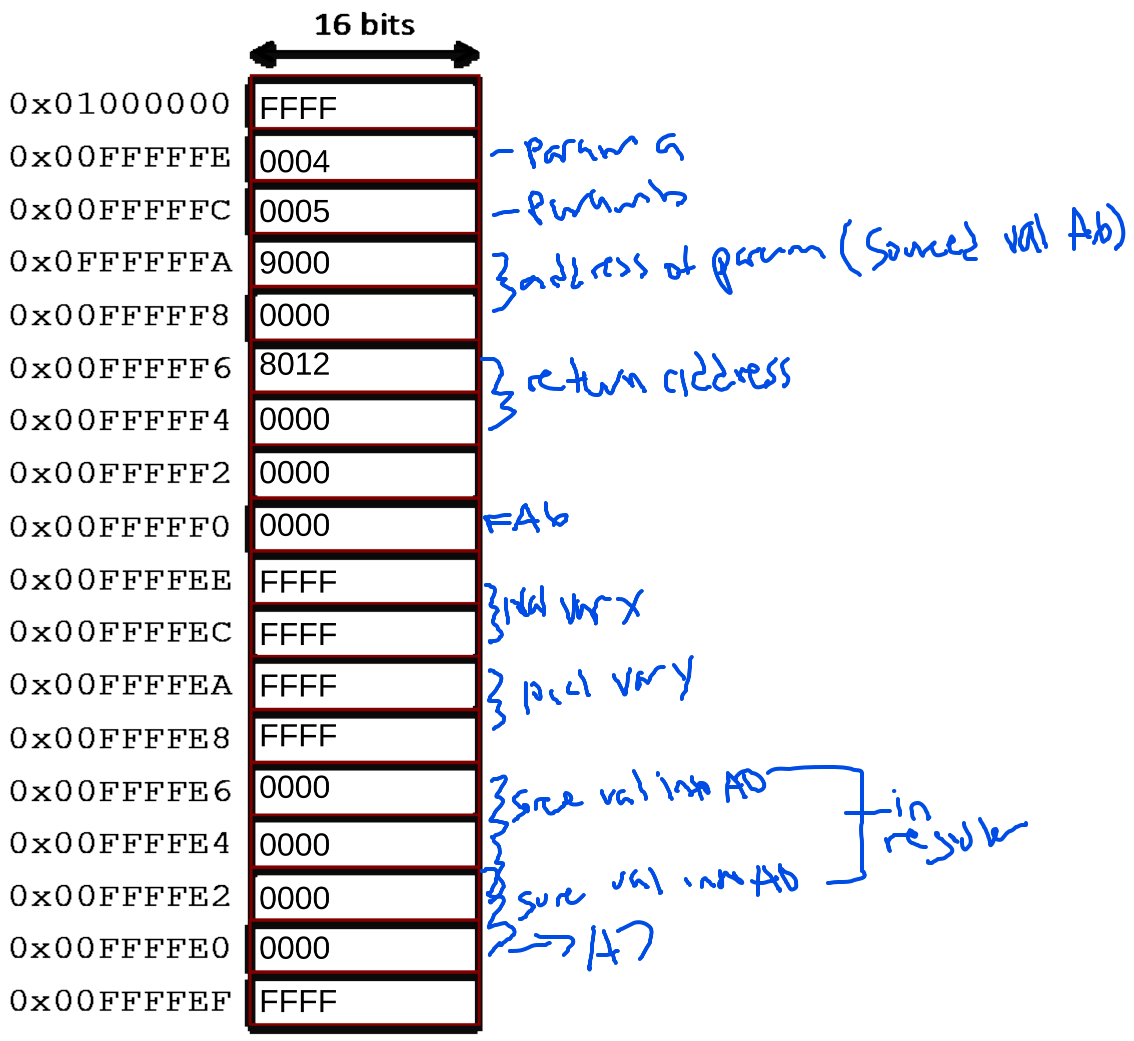


Notice that the subroutine's three formal parameters are at positive offsets from the (fixed) frame pointer, while subroutine's the two local variables are at negative offsets!

Step 5

As previously stated, we use the frame pointer and not the stack pointer when accessing formal parameters and local variables. This is because once the subroutine saves the working registers on the stack, the stack pointer no longer points to the bottom of the frame, but to the top of the stack. Moreover, any subsequent push or pop operations involving the stack will cause the stack pointer to keep changing locations. The frame pointer, however, is fixed, and does not change value. Therefore, all addresses can be specified relative to the frame pointer at assemble time.

Use the Easy68K trace facility to execute the second instruction in subroutine `sumSquare` located on line 29. This instruction saves the working registers on top of the stack. After the instruction executes, show the (final) contents of the stack in the memory-map below. Make sure to draw the location of the stack pointer (A7) and the location of the frame pointer (A6). Also, add labels to the side of the memory map that identify what each item on the stack refers to (e.g., "parameter a", "return address", "caller's frame pointer", "local variable x", "Saved value in D0", etc.) The actual contents of each memory location should be expressed as a 16-bit (hexadecimal) value. **[11 points]**



You now have a complete memory map showing the state of the stack during the execution of `main` and `sumSquares`.

Step 5

Using the memory map from step 5 as a guide, trace through the subroutine instructions on lines 30 through 39. Make sure that you understand how the program accesses the formal parameters and local variables when performing the operations required to compute c = $a^2 + b^2$.

Step 6

Now, use the trace facility to execute the last three instructions in the subroutine located on lines 40 through 42. The purpose of these instructions is to "undo" the earlier stack operations in reverse order. This includes restoring the working registers, deallocating the stack frame and restoring the caller's frame pointer, and popping the return address off of the stack to return control to the `main` code.

Show the contents of the stack in the memory-maps bellow after each instruction executes. Make sure to clearly show the location of the stack pointer (A7). **[3 points]**

	16 bits	
0x01000000	FFFF	
0x00FFFFFE	0004	
0x00FFFFFC	0005	
0x0FFFFFFA	9000	
0x00FFFFF8	0000	
0x00FFFFF6	8012	
0x00FFFFF4	0000	
0x00FFFFF2	0000	
0x00FFFFF0	0000	
0x00FFFFEE	0010	
0x00FFFFEC	0000	
0x00FFFFEA	0019	
0x00FFFFE8	0000	←A7
0x00FFFFE6	0000	
0x00FFFFE4	0000	
0x00FFFFE2	0000	
0x00FFFFE0	0000	
0x00FFFFEF	FFFF	

After `MOVEM.L (A7)+,A0/D0`

	16 bits	
0x01000000	FFFF	
0x00FFFFFE	0004	
0x00FFFFFC	0005	
0x0FFFFFFA	9000	
0x00FFFFF8	0000	
0x00FFFFF6	8012	
0x00FFFFF4	0000	←A7
0x00FFFFF2	0000	
0x00FFFFF0	0000	
0x00FFFFEE	0000	
0x00FFFFEC	0000	
0x00FFFFEA	0019	
0x00FFFFE8	0000	
0x00FFFFE6	0000	
0x00FFFFE4	0000	
0x00FFFFE2	0000	
0x00FFFFE0	0000	
0x00FFFFEF	FFFF	

After `UNLK A6`

	16 bits	
0x01000000	FFFF	
0x00FFFFFE	0004	
0x00FFFFFC	0005	
0x0FFFFFFA	9000	
0x00FFFFF8	0000	←A7
0x00FFFFF6	8012	
0x00FFFFF4	0000	
0x00FFFFF2	0000	
0x00FFFFF0	0000	
0x00FFFFEE	0010	
0x00FFFFEC	0000	
0x00FFFFEA	0019	
0x00FFFFE8	0000	
0x00FFFFE6	0000	
0x00FFFFE4	0000	
0x00FFFFE2	0000	
0x00FFFFE0	0000	
0x00FFFFEF	FFFF	

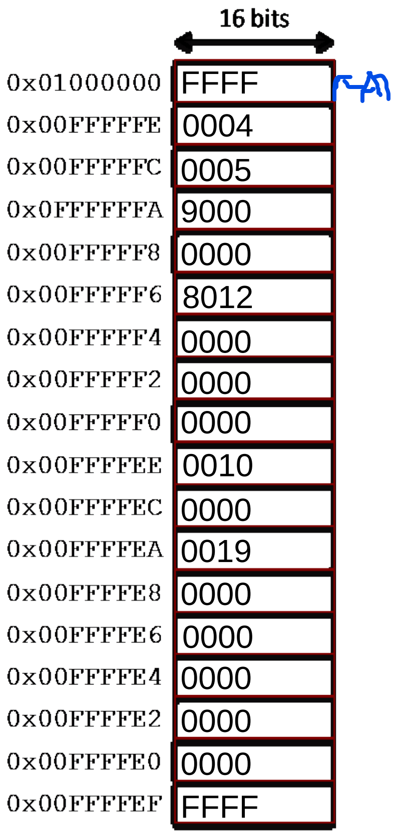
After `RTS`

What 32-bit (hexadecimal) value is contained in the frame pointer (A6) *before* and *after* execution of the `UNLK A6` instruction? **[1 point]**

Before: 00FFFFF0 After: 00000000

Step 7

Upon return to `main`, the caller is responsible for removing all parameters from the stack. This is accomplished using the `LEA 8(A7),A7` instruction. Show the final state of the stack after this instruction executes. **[2 points]**



0x01000000	FFFF
0x00FFFFFE	0004
0x00FFFFFC	0005
0x0FFFFFFA	9000
0x00FFFFF8	0000
0x00FFFFF6	8012
0x00FFFFF4	0000
0x00FFFFF2	0000
0x00FFFFF0	0000
0x00FFFFEE	0010
0x00FFFFEC	0000
0x00FFFFEA	0019
0x00FFFFE8	0000
0x00FFFFE6	0000
0x00FFFFE4	0000
0x00FFFFE2	0000
0x00FFFFE0	0000
0x00FFFFEF	FFFF

Observe that once the parameters are removed from the stack, the contents of the memory, except for the variable `c`, and all of the registers are returned to their original states prior to calling the subroutine!

Part 2: Stack Frames and Recursion

Perhaps the most common mathematical function to be computed in a recursive way is *factorial n*. Written as *n!*, factorial n is defined as follows:

$$n! = n \times (n-1) \times (n-2) \times (n-3) \times \ldots \times 3 \times 2 \times 1$$

Factorial n is said to be recursive, because it can be defined in terms of another factorial:

$$n! = \begin{cases} 1 & if\ n = 0 \\ n \times (n-1)! & if\ n \geq 1 \end{cases}$$

The previous definition can be easily implemented as a recursive C function, as illustrated in Fig. 3.

```
int factorial(int n) {

    int fact;

    if(n == 0) return 1;
    fact = n x factorial (n-1);
   return fact;
}
```

Figure 3: C function for computing *n!*.

The function is said to be recursive, because it computes *n!* by calling itself multiple times. For example, if the function is first called with `n=3`, the if-condition is false, and the function begins to calculate `3 x factorial(2)`. This computation results in a recursive call to compute `factorial(2)`. Once again, the if-condition is false, and the function begins to calculate `2 x factorial(1)`. This computation results in a final recursive call to compute `factorial(1)`. This time, the if-condition is true, and 1 is returned from the call to `factorial(1)` to `factorial(2)`. This, in turn, results in `2 x factorial(1)` being calculated as 2 x 1 = 2. This value from `factorial(2)` is then returned to `factorial(3)`, and `3 x factorial(2)` is calculated as 3 x 2 = 6. Finally, 6 is returned as the answer to 3! to the calling code.

What would happen if the previous code was called with a negative argument? **[4 points]**

it will loop till biggest value then roll over until 0 is returned

As explained in class, in order to support recursion, it is necessary that the function's formal parameters and local variables be generated and associated with each call to the function. This can be achieved by creating a stack frame upon each entry to the function. Each stack frame will contain the formal parameters and local variables associated witch each function call. However, at any given time, only the frame associated with the active frame pointer will be directly accessible. For example, Fig. 4a shows the stack frame for the first call (Call 1) to a subroutine called A. Notice that the formal parameters and local variables are accessible via the frame pointer during Call 1. Now, let's assume that subroutine A calls itself. Figure 4b shows the stack frame for the second call (Call 2). Notice that the old frame pointer associated with Call 1 is no longer active, but saved on the stack frame associated with Call 2. Moreover, the new frame pointer now points to the stack frame for Call 2. This frame pointer can now be used to access the formal parameters and local variables unique to the second call, but not the first call. The latter will only be accessible once a return is made to the first call, and the old frame pointer restored.

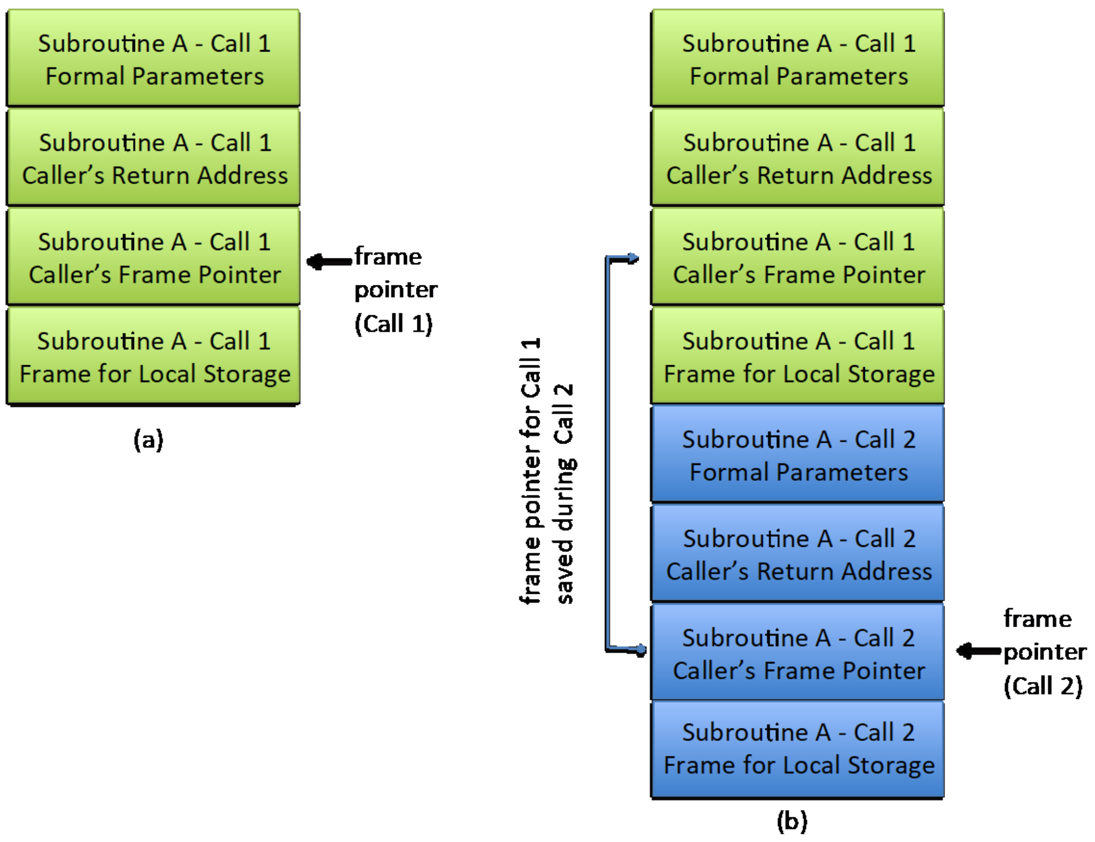

Subroutine A - Call 1 Formal Parameters
Subroutine A - Call 1 Caller's Return Address
Subroutine A - Call 1 Caller's Frame Pointer
Subroutine A - Call 1 Frame for Local Storage
frame pointer (Call 1)
(a)
frame pointer for Call 1 saved during Call 2
Subroutine A - Call 2 Formal Parameters
Subroutine A - Call 2 Caller's Return Address
Subroutine A - Call 2 Caller's Frame Pointer
Subroutine A - Call 2 Frame for Local Storage
frame pointer (Call 2)
(b)

Figure 4: Only the current call's frame is active during recursion.

Step 1

Download the sample program called **Lab6b.X68** from the course website.

Step 2

Start Easy68K. Once running, load the file Lab6b.X68 using the File->Open File menu choice.

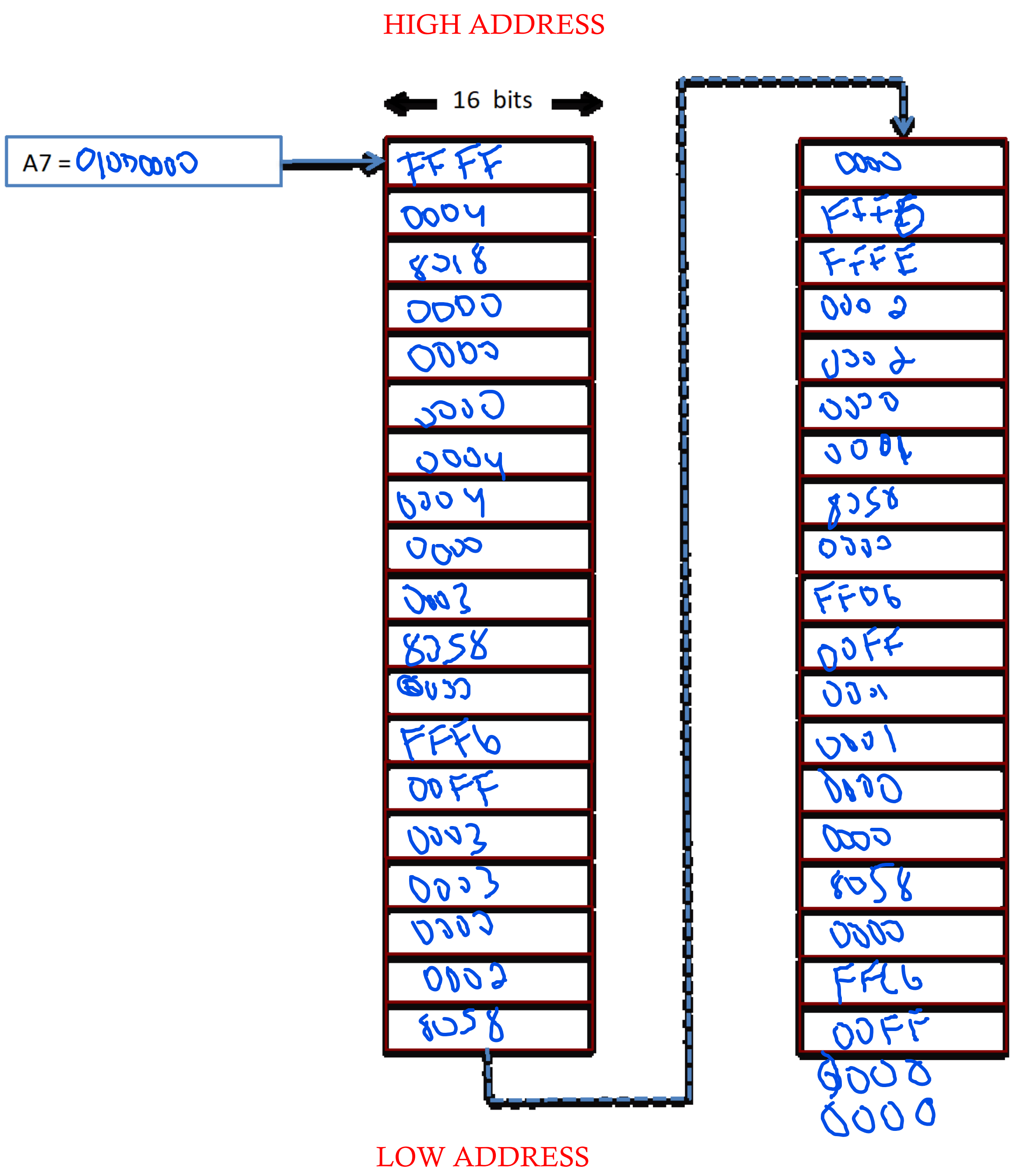
The program computes *n!* recursively, where *n* is assumed to be a non-negative integer. The parameter *n* is passed to the subroutine on the stack, while the result (a 16-bit integer) is returned in data register D1. The subroutine uses one working register, D0, which is saved and restored upon entry and exit, respectively.

Run the program. When prompted to enter the value of *n*, enter the value 5. This will cause the program to compute factorial 5. Run the program to make sure it correctly outputs the value 120.

Step 3

Reload the program. Set a breakpoint at on line 24, where the instruction is labeled `callFactorial`. Run the program. When prompted to input the value of *n*, enter the value 4. When the simulator stops at the breakpoint, fill in the value of the stack pointer in the figure on the following page.

Now, trace through the program one instruction at a time until the program terminates. Every time that an instruction affects the contents of the stack, the stack pointer, or the frame pointer report the changes by filling in the blanks in the memory map on the following page.


HIGH ADDRESS
16 bits
A7 = 01070000
FFFF
0004
8318
0000
0000
0000
0004
0004
0000
0003
8358
0000
FFF6
00FF
0003
0003
0000
0002
8058
0000
FFF6
FFFE
0002
0002
0000
0001
8358
0000
FF06
00FF
0001
0001
0000
0000
8058
0000
FFF6
00FF
0000
0000
LOW ADDRESS

Questions:

1. How many recursive call are made in the program? **[2 points]**

4 recursive calls

2. Every time a recursive call is made, the size of the runtime stack grows. By how many bytes does it grow per call? **[2 points]**

16 bytes per call

3. What is the maximum integer you may enter such that the subroutine still returns the correct value? **[2 points]**

any value greter then 9 will be wrong so max is 9

4. Since the stack grows every recursive call, it is possible that the maximum size of the stack may become exhausted. Assuming that the size of the stack is 512K bytes, what is the minimum input value that would exhaust the stack? **[3 points]**

32 000

5. What do you think limits the factorial subroutine from calculating the factorial for very large numbers? Explain. **[3 points]**

n>24^16, factor of n+1 cant be completed, multiplication takes words as operands

In the next portion of the lab, you will gain experience writing non-recursive and recursive subroutines that employ stack frames for dynamic memory allocation for local variables.

Part 3: Non-Recursive Subroutine

Figure 5 shows a C function for computing $c = (a + b)^2$ based on the formula $a^2 + 2ab + b^2$.

```
void sumSquared(int a, b, *c) {

     int x,y,z;

     x = a * a;
     y = b * b;
     z = 2 * a * b;
     *c = x + y + z;
}
```

Figure 5: A C function to compute $c = (a + b)^2$.

Your task is to write a 68000 program to implement and test the previous function. The responsibilities of the caller (`main`) and callee (`sumSquared`) are listed below:

Caller:

- Pass the three parameters `a`, `b`, and `*c` on the stack. Assume that `a` and `b` are 16-bit values, and that the address of `c` is 32 bits.
- Call subroutine `sumSquared`.
- Remove parameters from the stack upon return

Callee Setup:

- Create a stack frame to dynamically allocate memory for the three local variables `x`, `y`, and `z`.
- Save all working registers on the stack, as needed.

Callee Running:

- Use the frame pointer to access all formal parameters and local variables.

Callee Return:

- Update variable `c` from within the subroutine.
- Restore all working registers, as needed.
- De-allocate stack frame, restoring both memory space and caller's frame pointer

- Return to caller

Make sure to fully comment and document your code. **[20 points]**

Part 4: Recursive Subroutine

The pseudo-code given in Fig. 6 recursively displays the digits that constitute an integer *n*. For example, if *n=1234*, the function will print the values 1, 2, 3, and 4 on the screen one character (or digit) at a time.

```
void display (int n){

     int remainder, quotient;

    if(!n)
       return n;

    remainder = n % 10;
    quotient = n / 10;
    display(quotient);
    printf("%d",remainder);
}
```

Figure 6: A recursive function for displaying an integer *n* on the screen.

Your task is to write a 68000 program to implement and test the recursive in Fig. 6. The responsibilities of the caller (`main`) and callee (`display`) are listed below:

Caller:

- Pass the parameter `n` on the stack. Assume that `n` is a 16-bit value.
- Call subroutine `display`
- Remove parameters from the stack upon return

Callee Setup:

- Create a stack frame to dynamically allocate memory for two local variables `quotient` and `remainder`.
- Save all working registers on the stack, as needed.

Callee Running:

- Use the frame pointer to access all formal parameters and local variables.

- Make a recursive call to itself (i.e., the callee) as required.

Callee Return:
- Restore all working registers, as needed.
- De-allocate stack frame, restoring both memory space and caller's frame pointer
- Return to caller

Make sure to fully comment and document your code. **[20 points]**