Stack Frames

CIS*2030
Fall 2024

Neatly print your name, student identification number, and lab information below:

Name:

ID:

Section:

Overview

In addition to parameters, a subroutine often requires a *local workspace* either for local variables or temporary values. Memory for such variables/values is created upon entry to the subroutine, accessible only to code within the subroutine, and destroyed upon exit from the subroutine. As explained in class, a *stack frame* is used to dynamically allocate the local workspace used by a subroutine. 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 of (or call to) the subroutine.

Objectives

Upon completion of this lab you will:

- Learned how to create a local workspace for a subroutine using a stack frame.
- Understand how to use a frame pointer to access local/temporary variables and formal parameters.

Preparation

Prior to attending your lab section, you must complete the following reading assignments from your textbook, and understand how each instruction works to the degree that you can use them correctly and confidently in an actual program:

- Section 3.2.1: Data-Movement Instructions (`LINK` and `UNLK`)

You should also review all lecture notes related to the above topics. You will not have time to perform these reading assignments during your lab session and complete the lab questions. Therefore, it is your responsibility to come prepared!

Evaluation

This lab has 2 questions. The first *with* an asterisk '*' beside it can be completed prior to the lab. However, the second question must be performed during your scheduled lab session.

Question	1*	2	Total
Max points	17	30	47
Your Points			

All marking is performed in real time *during* the lab. Therefore, when you complete a question raise your hand to let the lab instructor know that you are prepared to be marked. Marks will be given based on the written answer you provide and/or your ability to orally answer related questions asked by the lab instructor. The key to obtaining full marks is to demonstrate comprehension.

Before proceeding, download all .BIN and .X68 files from CourseLink. Make sure to place these files in the *same* directory.

Question 1: Stack Frames and Local Variables

Consider the C function shown in Fig. 1. 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 1: A C function that computes c = $a^2 + b^2$.

Step 1

Open the sample program called **sumSqr.X68** in the Easy68K environment.

The program starts with a prologue called `main` that 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.

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 below.

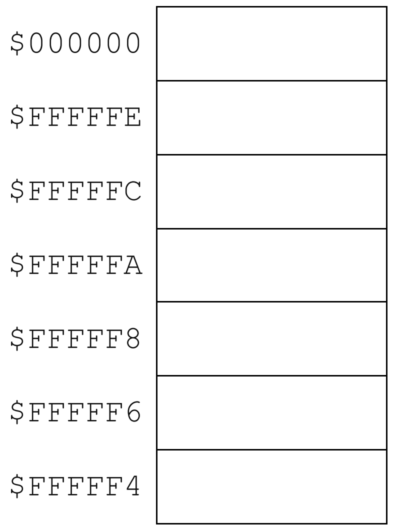
$000000	
$FFFFFE	
$FFFFFC	
$FFFFFA	
$FFFFF8	
$FFFFF6	
$FFFFF4	

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

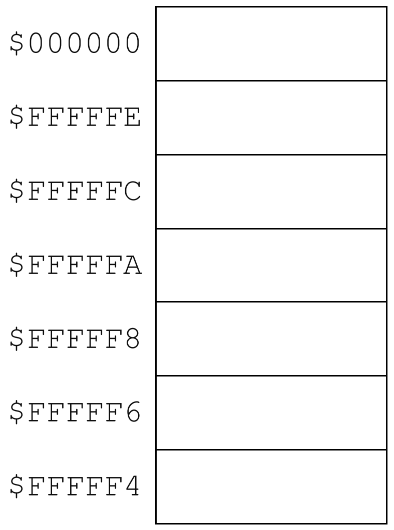
$000000	
$FFFFFE	
$FFFFFC	
$FFFFFA	
$FFFFF8	
$FFFFF6	
$FFFFF4	

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

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.



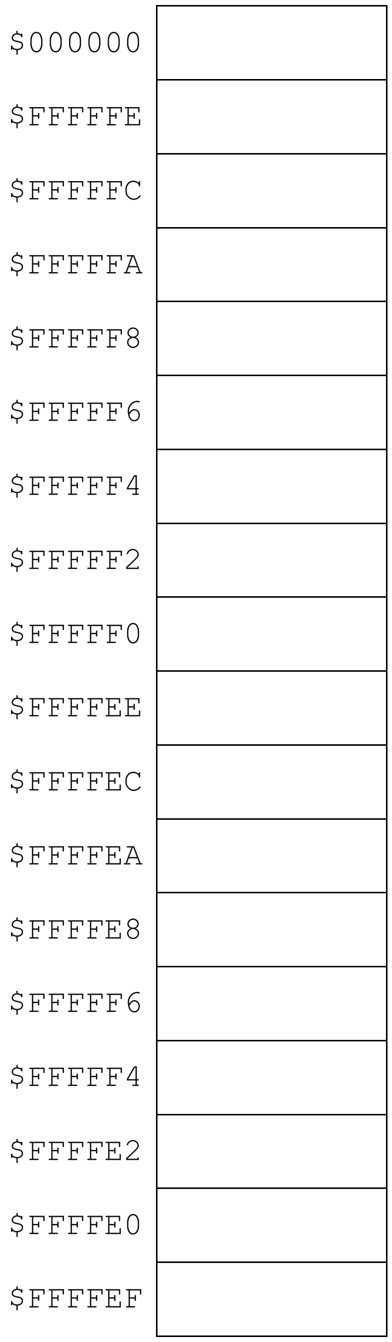
After `PEA C`



After `BSR sumSquares`

The subroutine `sumSquares` computes the expression $c = a^2 + b^2$ and is based on the high-level code in Fig. 1. 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.

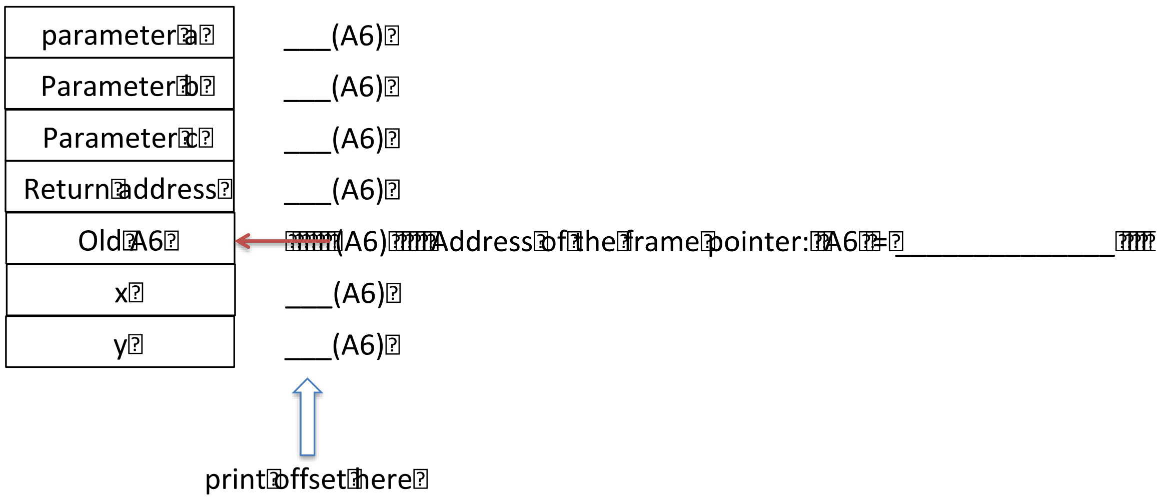
Using the Easy68K trace facility, execute the first instruction in subroutine `sumSquare` located on line 28. After the instruction executes, show the contents of the stack in the memory-map that follows. Make sure to draw the location of the stack pointer (`A7`) and the location of the frame pointer (`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.



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.



Notice that the subroutine's three formal parameters are at positive offsets from the (fixed) frame pointer, while the subroutine's 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. 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.

`$000000`	
`$FFFFFE`	
`$FFFFFC`	
`$FFFFFA`	
`$FFFFF8`	
`$FFFFF6`	
`$FFFFF4`	
`$FFFFF2`	
`$FFFFF0`	
`$FFFFEE`	
`$FFFFEC`	
`$FFFFEA`	
`$FFFFE8`	
`$FFFFF6`	
`$FFFFF4`	
`$FFFFE2`	
`$FFFFE0`	
`$FFFFEF`	

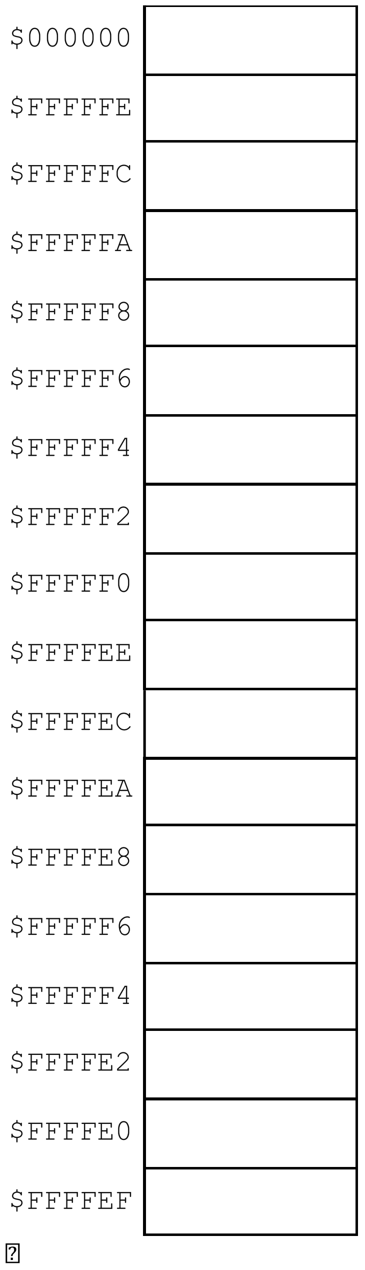
You now have a complete memory map showing the state of the stack during the execution of `main` and `sumSquares`.

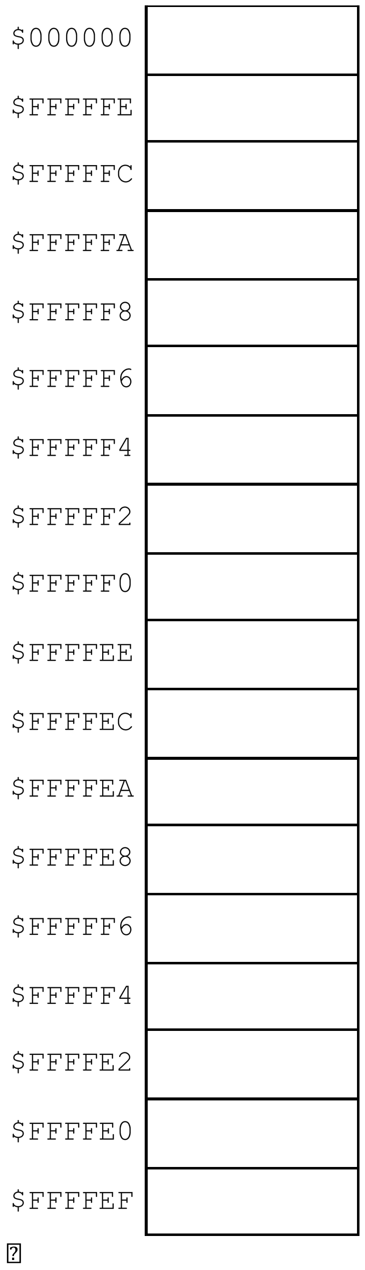
Step 5

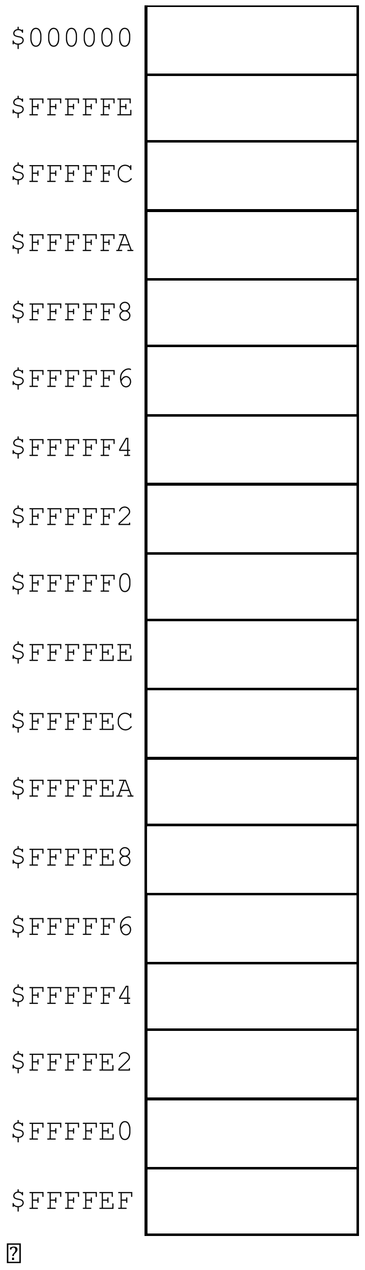
Using the memory map from step 5 as a guide, trace through the subroutine instructions on lines 30 through 39. Make sure that you understand how the program accesses the formal parameters and local variables when performing the operations required to compute $c = a^2 + b^2$.

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 that follow after each instruction executes. Make sure to clearly show the location of the stack pointer (`A7`).







After `MOVEM.L (A7)+,A0/D0` After `UNLK A6` After `RTS`

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

Before: ______________________ After: _________________________

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.

$000000	
$FFFFFE	
$FFFFFC	
$FFFFFA	
$FFFFF8	
$FFFFF6	
$FFFFF4	
$FFFFF2	
$FFFFF0	
$FFFFEE	
$FFFFEC	
$FFFFEA	
$FFFFE8	
$FFFFF6	
$FFFFF4	
$FFFFE2	
$FFFFE0	
$FFFFEF	

⍰

Question 2: The Mandelbrot Set

Benoit Mandelbrot was a French mathematician who discovered the Mandelbrot set while working at IBM. The Mandelbrot set is a *fractal* that is generated with the recursive expression $Z_n = Z_{n-1}^2 + C$, where Z_n and C are complex numbers. The term fractal simply means that when the previous expression is iterated over n a never-ending pattern of points emerges. When these points are plotted in the complex plane, zooming in to different scales reveals the emergence of self-similar patterns.

Figure 1a shows an example. Black colors represent points inside the Mandelbrot set. All other points are colored according to their proximity to the set. Zooming in reveals similar patterns at different scales, though the colors may vary. Before proceeding, click on the link here and watch the brief video to gain a better understanding.

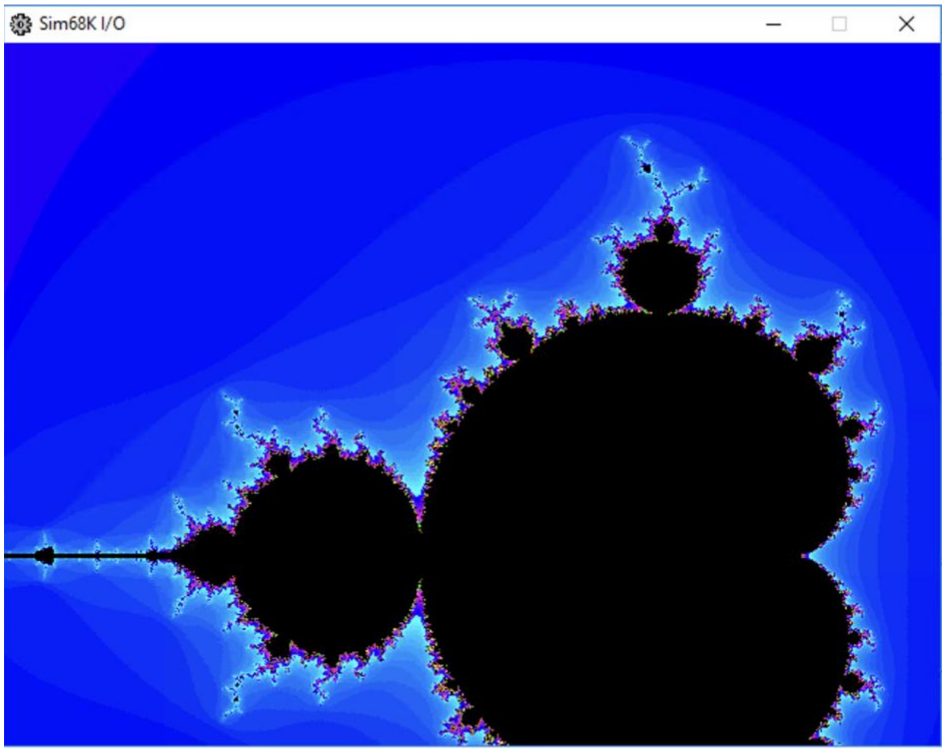


Figure 2: A Mandelbrot Set visualization

Now that you have a better understanding of the Mandelbrot Set, we will work together to develop a program that computes and displays the Mandelbrot Set on the Easy68K console. *Your* task is to implement a subroutine called `Iterate`. *Our* code will then call this subroutine, providing it with the *x* and *y* coordinates of a point in the plane and a pointer to a variable named `iteration`. As explained in the video, this iteration value is used to determine the color of the point. Therefore, our code will use the value that your code provides to set the color and render the point on the Easy68K console.

The implementation of the `Iterate` subroutine is based on the C code provided below:

```
void Iterate(int x, int y, int* iteration) {

     float a=0.0;
     float b=0.0;

     *iteration = 0;
     while(*iteration < 100 && a * a + b * b < 1024){
          t = a * a – b * b + x     /* real */
          b = (a * b) << 1 + y      /* imaginary */
          a = t;
          *iteration++;
     }
}
```

Figure 3: C-like pseudo code of the algorithm.

You need to be aware of two things:

- The *a* and *b* variables are implemented using *fixed-point* arithmetic. Each variable is 32 bits. The upper 24 bits encode the integer portion of the number, while the lower 8 bits encode the fraction. The left-most bit acts as a sign bit. For example, +1.0 is encoded as 0x0000000100, while -1.0 is encoded as 0xFFFFFF00.
- Addition or subtraction involving variables *a*, *b*, *x*, and *y* can be done directly using the integer `add` and `sub` instructions that are part of the 68000's ISA. However, the same thing is not true for multiplication. Therefore, we have provided you with a subroutine, called `FP_mul`, that you can call, as needed, to multiply two 32-bit fixed-point numbers with previous format. (This subroutine is required because the multiplication of two 32-bit fixed-point numbers, each with an 8-bit fraction, results in a 64-bit product with a 16-bit fraction. Normalizing the 16-bit fraction back to an 8-bit fraction requires dividing the 16-bit fraction by 256 since $2^8 = 256$. This effectively shifts the fraction 8 bits to the right. Additional steps are required to handle the remaining 24-bits, but we will not go into these details here.) The two numbers should be passed to the subroutine by *value* using the stack. The subroutine will return their product back to you in data register `D0`.

Start by opening `Mandelbrot.X68` in Easy68K. You will notice that a binary file, `N4.BIN`, is loaded into your program via the directive `INCBIN`. This binary file contains the calling code. You are also provided with a label `FP_mul` which you will use to call the 32-bit fixed-point multiplication function when needed.

The **caller**, which you will not see but is nevertheless present in `Mandelbrot.X68`, operates according to the following specifications:

- It will pass `X`, `Y`, and `&iteration` on the stack using the usual C convention of passing parameters right to left. Each parameter is 32-bits.
- It will call `Iterate`, pushing the return address on to the stack,
- Finally, it will expect the iteration number, computed by your subroutine, to be stored in memory at address `iteration`.

Your task is to implement the subroutine `Iterate` so that it operates according to the following specifications:

Callee Setup:

- Your subroutine is allowed to use one data register (`D1`) and one address register (`A1`). This means that your subroutine will require its own *local workspace* to hold temporary values when performing the computations illustrated in Fig. 3, as there will not be enough available registers.
- You will need to first determine the total number of bytes required to hold all temporary values and then create a suitably large stack frame. Use address register `A6` as the frame pointer.
- Your subroutine must be transparent to the calling code. Therefore, all working registers should be saved and restored using the stack.

Callee Running:

- Your subroutine should implement the pseudo-code in Fig. 3. The main loop iterates until one of the two loop conditions evaluates to false. Once the loop terminates, our program will have access to the iteration number that was updated on each iteration of the loop.
- Use the frame pointer (`A6`) to access all subroutine parameters and temporary values that reside on the stack.
- To compute X^2, Y^2 or `XY` use the subroutine `FP_mul`, which is part of our code that you do not see. You can call this subroutine using `JSR FP_mul`. This subroutine requires the 32-bit multiplier and multiplicand to be passed by *value* on the stack. Both of these parameters should be removed from the stack by your subroutine after the call returns. The 32-bit product will be returned to your subroutine in data register `D0`.

Callee Return:

- Restore all working registers, as needed.
- De-allocate the stack frame and then return.
- Our code will handle the removal of the original parameters passed to your subroutine on the stack.

Submit

You are required to submit your files online. Each file will be uploaded separately; that is, do not zip your files. In the Week 9 folder on CourseLink you will find a Dropbox into which you can submit the following:

- The source (.X68) files,
- the executable files,
- the listing (.L68) files, and
- the log (.txt) files.