mystery.s deasassembly

Method

In order to begin discovering what mystery.s I first ran the program a few times with integer arguments with increasing value starting at 1. I quickly recognized the fibonacci sequence and realized that the program was calculating and returning the nth fibonacci number, n being the integer argument. To determine the exact method this calculation was being done I made a copy of the code and wrote notes about each line to remind myself about what the instructions were accomplishing. Then I started tracing the programs using different arguments. I started getting a general idea about how it worked but the going was slow so I then began trying to replicate the assembly code by writing C code and compiling to see what the C code generated. I repeated this process until the assembly code my C code generated was extremely close to the original assembly code in mystery.s, the main difference being label names and address offsets for different variables in some functions.

Algorithm

After creating C code that generated the same assembly as mystery.c I discovered the mechanism for which the fibonacci numbers were being calculated. The steps involved are as follows

- First the string argument representing is converted by main into an integer.
- This integer is passed to the dothething method as its single argument
- dothething performs the bulk of the calculation. It is based on the recursive algorithm for calculating fibonacci numbers

```
int fibonacci(int n)
{
	if(n == 0) return 0;
	else if(n == 1) return 1;
	else return fibonacci(n - 1) + fibonacci(n - 2);
}
```

- However mystery.c contains an optimization that utilizes an integer array called num. num contains 200 elements and is initialized to -1 for all its elements in the main method.
- When a number, for example, 4 is passed as the nth fibonacci number to find, recursive calls to dothething are made similar to the code above until dothething(0) and dothething(1) are called. When either 1 or 0 are reached for the first time they are stored in num at index n, n being the current argument passed to dothething. Now, at the beginning of dothething there is the conditional

```
if(num[n] != -1){
        result =  num[n];
}
```

- Since the 0th fibonnaci number and 1st fibonacci number have already been calculated, further calls to dothething(0) and dothething(1) will skip the rest of the method and just look up the values in the num array.
- So, once an nth number is calculated, it is stored in the num array for easy lookup, removing the necessity for the recursive calculation of a number once it has already been calculated.
- As an example, if n = 4 and the line

```
X = add(dothething(3), dothething(2))
```

- is reached during the first execution of the method. First dothething(3) is recursively found. Then dothething(2) is calculated. At this point the 0th, 1st, and 2nd fibonacci numbers would have been found and stored in num. So when dothething(2) is called all it does is look up the value of num[2] and returns the value since the 2nd fibonacci number has already been found.
- This optimization removes the need for an unnecessary amount of recursive calls to dothething

Optimization by the compiler

When I compiled my mystery.c program with the -O flag and without the flag I noticed two main differences in the assembly that was generated. First, the optimized assembly used registers to store local variables often instead of storing them on the stack. This is probably useful because operating with values inside registers is faster than having to access values on the memory of the stack which is much slower than the registers. The second thing I noticed was that conditional statements I made were consolidated a bit. This leads to less lines of code overall, and will result in a faster execution.