Name: Daniel Ems
Date: 05/19/2017
Current Module: x86 Assembly
Project Name: Fibonacci

Project Goals:

The goal of the project was to first, write a program than calculate the Fibonacci sequence. Second, is to accept a command line argument and print the corresponding value in the Fibonacci sequence.

Considerations:

- How can you write the Fibonacci sequence?
- How can you accept command line arguments?
- Will I have to call any functions, if so which ones?
- If I do call functions, will I have to align the stack appropriately?
- How many registers will I need?
- How can I organize my registers so they are easy to follow?
- How can I minimize any jumps I may have to do?
- What form of error checking will I need to do?
- What format will I print m values in?
- Will I need to account for the size of any numbers?

Initial Design:

The program consists of a Make file and a .s. The .s is comprised of three functions, numerous different registers and logic gates. The functions used are strtol, fprintf, and printf. The stack is used to store data, and roughly 10 different registers are used as well. The program implements a while loop, conditional jumps, and compares in order to control the flow of the program. Finally, there are two different methods to exit the program, and error, and a normal exit. The error is used to print error messages.

Data Flow:

The data flow of the program begins with _start and pushes main to the stack. Once main is pushed to the stack, and the instruction pointer begins executing. The program first compares the value of argc to ensure the user passed the appropriate number of command line arguments. If the user fails to do so, the program Errors. The Error function calls fprintf and will display an error message letter the user know of proper usage.

After the compare, room is created on the stack to store the name of the program for fprintf in the result of an error. Room is also created on the stack for a pointer which is passed to strtol. The strtol function takes the users command line argument, a pointer, and the number base you wish to convert. In the even the user should pass anything but a number as a command line argument, i.e. (30b), strtol will return a pointer to the “b. The pointer is placed on the stack in the location created earlier. That memory location is compared to a null byte, if the not equal flag is set, the program will error. This would mean an invalid character was passed to strtol.

If the user supplied the appropriate command line argument, the program will process the number and move on to the Fibonacci sequence. The program sets up for the Fibonacci sequence by xoring any registers that may be used for overflow. They are zeroed out so that in the even they are not used, there is no random data printed. The program also moves 0 into the counter and data register, and moves one

into the accumulator. These three registers will hold the Fibonacci results, and compare the current sequence number with the users command line argument.

The Fibonacci sequence is contained in a loop. The break condition is when rcx is equal to the users command line argument. The first thing to happen in the loop is the xadd rdx, rax. Xadd will switch the values in rdx and rax and then store their summation in the source. This is how you maintain the integrity of the two most recent Fibonacci numbers which are needed to identify the next number in the sequence.

Following xadd are the overflow registers. They are paired with another register, and then adc is called. Adc will take the overflow and add it to the register. Those registers are then flipped to mirror the ordering of the rdx and rax registers. This happens for three sets of overflow registers to handle numbers such as the 300^{th} number in the Fibonacci sequence.

Once all adds and exchanges have been made, the counter is compared to the users command line argument. If they are equal, the program will jump to print. If they are not the loop will increment counter and then continue through another iteration.

The print label contains all the necessary arguments for printf. The program is already properly aligned, so all that is needed to be done is move the arguments into the appropriate corresponding registers. The format string is placed into rdi, the overflow register that would be used last is placed in rsi, followed by the overflow register preceding it in rdx, and so on into rcx. Finally the original register is placed into r8. The function is then called and the stack is rewound by popping the value that was pushed at the beginning of the program, and exits normally.

Potential Pitfalls:

The potential pitfalls that would needed to be accounted for throughout this project were, stack alignment, volatile registers, register overflows, and error checking. Stack alignment was necessary in order to call functions that were utilized; fprintf, printf, strtol. It was not only necessary to make sure your stack was aligned before calling the function, but also that it was rewound appropriately afterwards.

In addition to ensuring the stack was aligned for function calls, you also had to ensure any data you needed to maintain was not stored in a volatile register. This was easily avoided by storing the data on the stack, however, this added to the attention needed to ensure your stack was aligned and rewound accordingly.

Register overflows were necessary for the minimum requirements from 94 -100, and even more so if you decided to calculate the sequence up to 300. This forced you to understand how adc was working in order to maintain the overflow bits, and print them in conjunction with the corresponding base register. Once this was accomplished with one set of overflow registers, adding more to achieve the flourish was easy.

Finally, error checking created a unique challenge in this project because it forced you to pass pointers to functions. Nothing new to us, however, creating pointers in assembly created a unique challenge of understanding pointers in a new way.

Test Plan:

User Test:

1. Test if you can check how many command line arguments were passed
2. Test if you can use strtol
   1. test if you can pass the appropriate arguments
   2. test if you can evaluate the users command line arguments
3. Test if you can implement the Fibonacci sequence
   1. Identify how you are going to implement the sequence
   2. see if you can calculate the Fibonacci sequence up to 10
   3. up to 50
   4. up to 100
4. Identify why 93 is accurate and 94 is not.
5. Test how you can exit the Fibonacci loop accurately.
6. Test if I can properly use printf
7. Test if I can create a pointer for strol
8. Test If I can error check the command line arguments for letters
9. Test If I can implement adc to print 94-100
10. Test if I can print all the way to 300
11. Implement additional error checking
12. Test If I can rewind my stack and exit appropriately.

Test Cases:

1. 1. Print the command line argument passed.
2. Implement a counter counts up to the command line argument, then prints
3. Implement strtol and retry test 2.
4. Try and print the first three sequences of the Fibonacci sequence.
5. Try and take a command line argument and print the corresponding Fibonacci value, between 1-10.
6. Try and extend test 5 to 50.
7. Try and extend test 6 to 100.
8. Try and print an overflow register after using adc.
9. Try and print the 94 Fibonacci number.
10. Try and break the program by passing a 30b
11. Try and Error out when the user passes 30b by evaluating the error pointer from strol.
12. Try and break the program by not passing a command line argument.
13. Try and break the program by passing multiple arguments.
14. Try breaking the program by passing a -1.
15. Try and error when a negative value is passed by comparing the value passed to 0.
16. Try and extend the number of overflow registers to calculate to 300.
17. Try and break the program by passing a value 500.
18. Try and error when a number > 300 is passed by comparing the value to 300.

Conclusion:

Assembly is a strange language. Very counter intuitive, and quite complex. However, once some basic understanding can be achieved it can be quite fun. I think the project has done a lot to further expand my knowledge of Assembly as well as memory, functions, gdb, and pointers. The necessity to understand the stack and registers in order to call functions had a very positive impact on my overall understanding of programming. Further more, the simplicity behind the code written makes it rather rewarding. To know that you are performing some highly complex tasks at almost the lowest level possible is a good feeling. Additionally, gdb is your best friend in assembly, as many have said, and they are correct.: I think it was a great project and I can not wait to see what is in store for us next week.