CS 105
Lab 2: Debugger
Playing with X86 Assembly

See class calendar for lab and due dates

Introduction and Goals

The goals of this assignment are to do some basic investigation of the X86 architecture and assembly language, and to begin learning how to use GDB. There is a link on the lab page to an information sheet on using GDB. If you have suggestions for changes, please email them to me; if you find a better web page, please email me the url.

It will be useful to know that you can get the compiler to generate the assembler source for a program by using "`gcc -S foo.c`". You should also know that, to use the debugger effectively, you will need to compile with the "`-g`" switch. In fact, you should just get in the habit of always compiling with "`-g`"; the situations where it's undesirable are extremely unusual. (Along the same lines, it's usually wise to compile *without* "`-O`" because that will make debugging more difficult, and debugging is nearly always more important than optimization.)

Also, note that the prologue and epilogue generated by the compiler may be different from what we saw in class. In particular, the prologue often contains other instructions that manipulate the stack pointer, and the epilogue may use the `leave` instruction as a substitute for the `mov/pop` pair.

Problem 1

Practice Problem 3.5, 15 Points

- Write a small program, `problem1.c`, that calls the shift routine from Practice Problem 3.5 with parameters -14 and -3. Then compile it and create an executable. Investigate and comment on the assembly language as to the use of registers, e.g., what registers are used for the shift amounts, which are used for variables, etc.
- Use GDB to step your way through the execution of the function.
- Set breakpoints at the call to the shift function and at the return from it (i.e., at the `return` statement).
- Check out the values of all registers before the call instruction, immediately after the call instruction has been executed, and before and after the (C-level) return statement.

Submit ONLY the C source code, `problem1.c`, with comments indicating:

1. How the shift amounts were set up (in other words, how the compiler got the shift amounts into a position where they could be used),
2. Which registers were used for which purposes,
3. What the registers were before and after the call instruction,
4. What the registers were before and after the return statement.

5. Why the function returns the value it does. (Hint: only 5 bits are needed to represent values between 0 and 31.)

Problem 2

Figure 3.8, 15 Points

- Convert the `arith` routine from Figure 3.8, p. 146 into a `main` function, with the values of `x`, `y`, and `z` hardwired to `8`, `19`, and `35`, respectively.
- Compile this program into an executable, saving the assembly-language version in a file.
- Use GDB to step through the execution of the program, again checking out the registers at each instruction, before and after the call.
- Disassemble the executable and compare the result to the saved assembly-language version.

Submit ONLY the assembly language version, `problem2.s`, with comments about (a) any differences with the disassembled version and (b) changes in the registers at each step.

Problem 3

Practice Problem 3.11, 15 Points

- Write a main program that calls `loop_while`.
- Compile and execute the C program using the `-O` switch, and use GDB to step through its execution.
- Answer Question A in the comments of your C progam.
- Answer Question B in the comments of your C progam.
- Answer Question D by rewriting `loop_while` as a function named `goto_version`.
- In the program comments describe the differences in the assembly code between `loop_while` and `goto_version`.

Submit Submit the commented C program, `problem3.c` containing `main` and the two functions, `loop_while` and `goto_version`.