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. The lab pages have links to a quick GDB summary and to a printable GDB reference card; you can also find other information on Google.

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.)

Finally, 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. The general rule of thumb is that any instructions that add to, subtract from, or `and` with the stack pointer are definitely part of the prologue, and (for this assignment) can safely be ignored.

Problem 1

You will be submitting the C code of your program, with comments giving the answers to the questions below.

1. Write a small program, `problem1.c`, that calls the following shift function (downloadable from the lab Web page) with parameters -14 and -3:

   ```
   int shift_left2_rightn(int x, int n)
   {
     x <<= 2;
     x >>= n;
     return x;
   }
   ```

   Then compile it without optimization and create an executable. Investigate and comment on the assembly language as to the use of registers. In particular, comment on which registers are used for the shift amounts, which are used for variables, etc.

   Note: the first four 32-bit registers (`%eax`, `%edx`, `%ecx`, and `%ebx` all have "mini" registers inside them. Dropping the "e" gives you a 16-bit register, so `%ax` is the low-order (least significant) 16 bits of `%eax`. Changing the "x" to "h" or "l" gives you an 8-bit registers, so "`%bl`" refers to the lower-order 8 bits of `%ebx`. Just to make matters fun, the "h" version is the *high-order* 8 bits of the `low-order` 16, so `%bh` is bits 2^8 through 2^{15} of `%ebx`. Got that? Fortunately, the "h" registers aren't used by the compiler. There's a diagram of all this on page 136 of the textbook.

2. How did the shift amounts get set up? In other words, how did the compiler get the shift amounts into a position where they could be used?

3. Set source-level breakpoints at the line that calls the shift function (in main) and at the line that returns from it (i.e., at the `return` statement in the shift function). Run the program, and see where it stops. Does the first breakpoint correspond to the assembly-level `call` instruction? Why or why not? Does the second breakpoint correspond to the `ret`? Again, why or why not?

4. Run the program again, letting it stop at the first breakpoint. Use GDB to step your way, one instruction at a time, through the execution of the shift function. Figure out how to quickly look at what happens in the registers after each instruction is executed. When the debugger says `%eip` is pointing to a particular instruction, has that instruction had an effect on the registers yet? (I.e., has it executed?)

5. Run the program a third time. What were the registers before and after the C-level `call` statement? (For "after" in this and subsequent questions, report only the registers that changed.) For all registers, report them in the number base that makes the most sense. Generally, this means that small numbers should be given in decimal and large ones in hex.

6. What were the registers before and after the C-level `return` statement in `shift_left2_rightn`?

7. What were the registers before and after the `assembly`-level `call` instruction?

8. What were the registers before and after the `assembly-level ret` instruction?

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

Submission. Use `cs105submit` to submit ONLY the C source code, `problem1.c`, with comments indicating the numbered answers to all the questions given above.

Problem 2

Figure 3.8, 15 Points

1. Convert the following function (downloadable from the lab Web page) into a `main` function, with the values of `x`, `y`, and `z` hardwired to `8`, `19`, and `35`, respectively:

```
int arith(int x,
          int y,
          int z)
{
    int t1 = x+y;
    int t2 = z*48;
    int t3 = t1 & 0xFFFF;
    int t4 = t2 * t3;

    return t4;
}
```

2. Compile this program into an executable without optimization, saving the assembly-language version in a file.

3. Use GDB to set a breakpoint at the first executable statement of `main`. Step through the instructions, checking out the registers at each instruction. Add comments to each instruction, explaining (a) the specific purpose of that instruction, and (b) what change it produces in the registers. If you don't know the purpose of the instruction, say so.

4. Disassemble the executable using either gdb or objdump, and compare the result to the assembly-language version generated by "`gcc -S`". How does the disassembled version differ from the one generated by the compiler? Comment on notational differences and anything else you observe.

Submission. Use `cs105submit` to submit ONLY the assembly-language version, `problem2.s`, with comments indicating the numbered answers to the questions above.

Problem 3

Consider the following function (downloadable from the assignment Web page):

```
int loop_while(int a, int b)
{
  int i = 0;
  int result = a;
  while (i < 256) {
    result += a;
    a -= b;
    i += b;
  }
  return result;
}
```

1. Write a main program that calls `loop_while` with parameters of your choosing.

2. Compile the C program unoptimized, using the `-g` switch, and use GDB to step through it enough to understand how it works. You may wish to use `-S` to look at the generated assembly code as well.

3. Recompile the C program using the `-O` switch, and and again use GDB or `-S` to help figure out the assembly code.

4. In the comments of your C program, discuss the differences between the optimized and unoptimized versions of the program.

5. In the C code, add comments identifying `test-expr` and `body-statement`. Where are the corresponding lines in the assembly code? Are those lines contiguous or separated?

6. In C, write a goto version of `loop_while` by using the standard transformation given in class, calling it `goto_version_1`, and recompile the program (with `-O`).

7. In the program comments, describe the differences (if any) in the generated assembly code between `loop_while` and `goto_version_1`.

8. Translate the assembly code for `loop_while` back into C, using goto as necessary for the jumps, and call it `goto_version_2`. Recompile this program with `-O`, and in the comments describe the differences (if any) in the assembly code generated from `goto_version_2`.

Submission. Use `cs105submit` to submit the commented C program, `problem3.c` containing `main`, the two functions, `loop_while` and `goto_version`, and comments indicating the numbered answers to the questions above.