Derek Schatel
RUID: 032004123

<u>**Design Document – y86emul.c and y86dis.c**</u>

I. <u>y86emul.c</u>

This program initially reads the input file as one long string and pulls the ASCII input into a char array all in one go. I opted to do this for coding simplicity: it was easier to do this instead of, for example, using fgets to pull in each directive line-by-line from the input file, because it would have required stipulating a max line length and coding for edge cases when the line was longer than anticipated. After the entire file is stored in a char array, I tokenized the input and searched for each directive in order to appropriately store the various pieces of information. Since I had already read the file information into a file array, I ended up coding a small hex-to-decimal conversion function since I could not use fscanf for conversion. This allowed me to convert the hexadecimal numbers from the input file into decimal equivalents in order to create an array of the appropriate length to act as the memory and then to store all of the various bytes and instruction codes in the appropriate locations. I used a struct to simulate the CPU and memory, with a program counter, flags, the memory array, and an array representing the eight registers.

After properly loading the program into memory, I used a switch statement to evaluate the various machine instructions and perform the appropriate calculations. I found the most difficult part of this assignment to be properly coding the call, return, push and pop instructions. I wanted to code them in order to properly simulate a call stack in the memory with the stack growing toward lower memory addresses. I managed this by obtaining the call/return values in hex and then storing them byte-for-byte in a little-endian format in memory. This effectively ended up creating a stack using the ESP and EBP registers that kept track of where the program was at any given time.

II. <u>y86dis.c</u>

I ended up writing the disassembler after I finished the emulator, so it was trivial. I largely mirrored my code from the emulator with a few small exceptions. First, I only worried about the .text directive. I obtained both the machine instructions and the beginning memory address from the directive and then placed the instructions into an unsigned char array in a manner similar to how I arranged the instructions in the emulator. In order to determine the size of that array (since I did not use the .size directive for this), I simply took the length of the instructions string and divided it by 2 (since each 2 ASCII characters was one byte of instructions).

I then ran the array of instructions through a switch statement similar to the emulator's code, except I simply printed out comparable assembly instructions rather than manipulate the registers and memory locations. In order to make the code more readable, I displayed the lines of instructions with the memory address (in hexadecimal) that they started on, and I displayed the relevant register names and 32-bit immediate/displacement values. All of this information is printed out to the console, with one instruction per line.