NAME:

Login name:

Computer Science 217
Final Exam
January 25, 2006
1:30-4:30pm

This test has six (6) questions. Put your name on ***every page***, and write out and sign the Honor Code pledge before turning in the test.

Please look through all of the questions at the beginning to help in pacing yourself for the exam. The exam has 100 points and lasts for 180 minutes, so the time spent per question should be *less* than twice its point value.

``I pledge my honor that I have not violated the Honor Code during this examination."

Question	Score
1 (25 pts)	
2 (20 pts)	
3 (20 pts)	
4 (10 pts)	
5 (10 pts)	
6 (15 pts)	
Total	

QUESTION 1: Short Answer *(25 POINTS)*

1a) Identify two *syntax* errors and one *semantic* error in the function **`foo()`** below, which tries to print an integer, along with its square and cube. Clearly identify which errors are syntax errors, and which one is the semantic error, and show the corrections that should be made. ***(2 points)***

```
int foo(int n) {
   int n2, int n3;
     /* This function is busted
   n2 = n * n;
   n3 = n2 * n2;
   printf("n=%d, n^2 = %d, n^3 = %d\n", n, n2, n3);
   return 0;
}
```

1b) What output does this fragment of code produce? ***(2 points)***

```
#define FORMAT "%s is a string"
printf(FORMAT, FORMAT);
```

1c) Suppose that you call **`scanf("%f%d%f", &x, &i, &y)`** where **`x`** and **`y`** are float variables and **`i`** is an **`int`**. If the user enters **`12.3 45.6 789`**, what will be the values of **`x`**, **`i`**, and **`y`** after the call? ***(2 points)***

1d) What does the following code print to stdout? *(3 points)*

```
printf("%d\n", ((~3) ^ (~1)) + (4 >> 1));
```

1e) What does **`mystery(6)`** print to the stdout? *(3 points)*

```
void mystery(int i) {
    if (i) {
        mystery(i/2);
        putchar('0' + (i % 2));
    }
}
```

1f) Give a short, high-level description of what the **`mystery()`** function does. *(2 points)*

1g) Compute the result of "65-14" in eight-bit binary arithmetic, using 2's complement arithmetic, and translate the result back into decimal form. Show your work. (*3 points*)

1h) Explain how a virtual address is mapped to a physical address. Assume the page has already been brought into main memory. (*2 points*)

1i) What is the difference between a *function* and a *system call*? *(2 points)*

1j) Why do the functions in **`<stdio.h>`** do buffering? When should a programmer call **`fflush()`**? *(2 points)*

1k) What is the difference between the **`SIGALRM`** and **`SIGPROF`** signals? *(2 points)*

QUESTION 2: Signals *(20 POINTS)*

Write a program that enthusiastically prints the following output:

```
10… 9… 8… 7… 6… 5… 4… 3… 2… 1… Happy New Year!
10… 9… 8… 7… 6… 5… 4… 3… 2… 1… Happy New Year!
10… 9… 8… 7… 6… 5… 4… 3… 2… 1… Happy New Year!
…
```

over and over again, ***even if the user types Cntl-C*** to try to make it stop. After printing "10… ", the program should print each element (a number with three dots, or the phrase "Happy new year!") ***one second*** (in wall-clock time) after printing the previous item, and then immediately print "10…" and so on. The user should be able to stop the program only if Cntl-C is typed ***twice*** within the same countdown, i.e., twice between the print of "10…" and "Happy New Year!" on the same line. Please modularize your code and use #defines and comments where relevant for more a more readable and extensible; comments make it easier to achieve partial credit. Rather than using the *sleep()* system call, implement the one-second countdowns using a signal handler. You may want to write pseudocode on a separate sheet of paper before writing your final answer.

QUESTION 3: ***Assembly Language*** *(20 POINTS)*

The Intel Architecture has hundreds of assembly-language instructions and numerous registers and addressing modes. Consider instead a simple computer with 8-bit addresses, an 8-bit-wide memory, ten two-byte instructions, and two registers (the program counter PC and an accumulator A). Rather having a keyboard and monitor, input comes from eight binary switches and output goes to eight LEDs. For example, the **`IN`** command loads the register **`A`** with the value of the input switches, and the **`OUT`** command displays the contents of register A on the LEDs (e.g., if the rightmost bit of A is a one, the rightmost LED is lit; otherwise, the LED is not lit). The ten instructions are listed below. Some of the instructions have an operand (referred to as "**`n`**" below) stored in the second byte of the instruction. For more readable code, the "jump not zero" instruction **`JNZ`** and the "load instruction" **`LDI`** can refer to a label (e.g., "**`JNZ FOO`**" or "**`LDI FOO`**") instead of a hexadecimal number.

```
IN: Load A with the value of the input switches
OUT: Display the contents of A on the output LEDs
LDI n: Load A with the value n
STORE (n): Store the contents of A in Memory[n]
READ (n): Read the contents of Memory[n] into A
JMP: Load the contents of A into the program counter PC
JNZ n: If A is not 0, then load PC with the value n
SHR: Right shift A by one bit, replicating the sign bit
SHL: Left shift A by one bit, putting a 0 in the bottom bit
NANDM (n): Compute the bit-wise AND of A and Memory[n], and
      store the one's complement of the result back in
      Memory[n]  [this is equivalent to
      "Memory[n] = ~(A & Memory[n])" in C]
```

For example, the short program

```
FOO: IN
     STORE (0x85)
     IN
     OUT
     READ (0x85)
     OUT
     LDI FOO
     JMP
```

reads two values from the switches and displays them on the LEDs in reverse order (temporarily storing the first value in Memory[0x85], where 0x85 is in hexadecimal notation), and repeats forever. The questions below relate to the assembly-language instructions defined for this ten-instruction machine.

3a) What addressing mode does **`LDI`** and **`JNZ`** use? What addressing mode does **`STORE,`** **`READ,`** and **`NANDM`** use? *(3 points)*

3b) The **`SHR`** instruction does an ***arithmetic*** shift, in that it replicates the sign bit. For example, the **`SHR`** instruction would convert the binary number "0000 1110" (0x0E) to "0000 0111" (0x07), but would convert the binary number "1111 0000" (0xF0) to "1111 1000" (0xF8) to implement signed arithmetic. In contrast, a ***logical*** shift would convert "1111 0000" (0xF0) to "0111 1000" (0x78).   Since this machine does not have a logical-right-shift instruction, an assembly language programmer (or a compiler) would need to implement it using multiple assembly-language instructions. Write the assembly-language code for a logical right-shift operation on the contents of Memory[0x80] and storing the result back in Memory[0x80]. You may assume that any memory locations (other than 0x80) may be used for reading and writing temporary values. Hint: your program should not need more than ten lines of assembly-language code and may need to use **`NANDM`** more than once to manipulate the bits. *(7 points)*

3c) Write a program that lights a single LED in each step, starting with the rightmost LED, moving to the left one LED at a time, and then returning back to the right, repeating indefinitely. That is, the program should first illuminate only the rightmost LED, and then illuminate only the second-to-last LED, and so on, continuing all the way to the left and then returning all the way to the right, and so on. For example, the lights should cycles as follows:

0000000**1**
000000**1**0
00000**1**00
0000**1**000
000**1**0000
00**1**00000
0**1**000000
10000000
00000000
10000000
0**1**000000
00**1**00000
000**1**0000
0000**1**000
00000**1**00
000000**1**0
0000000**1**
00000000
0000000**1**
000000**1**0
….

You may use labels such as "`FOO:`" to refer to the address of a particular instruction in memory, and refer to that label in other instructions (e.g., "`LDI FOO`"). For this question, assume the language ***does*** have a ***logical*** shift right command (called `LSHR`) that operates on the register `A`. The program can be written on the next page. *(10 points)*

Write your answer to question #3c here:

QUESTION 4: ***Deterministic Finite Automata*** *(10 points + 2 extra credit)*

You have been asked to write a program that inspects an ASCII string (input via stdin) for the occurrence of the string "nano" anywhere in the string. For example, the inputs "banano" and "banananonano" have the string "nano" at least once, but the string "banananananashanana" does not. Draw a deterministic finite automaton that recognizes the substring "nano", showing and labeling all transitions (using the word "other" to label the transitions for any characters not otherwise specified, and "all" to label transitions taken for all characters) and clearly indicating the initial state with S (start) and accepting state(s) as F (finish). Please draw neatly. *(Extra credit of two points: Draw the DFA with no arcs intersecting each other.)*

QUESTION 5: ***Make*** *(10 points)*

Suppose you are given this makefile:

```
myprog: myprog.o one.o two.o three.o four.o
      gcc -o myprog myprog.o one.o two.o three.o four.o
myprog.o: myprog.c
      gcc -c myprog.c
one.o: one.c
      gcc -c one.c
two.o: two.c two.h four.h
      gcc -c two.c
three.o: three.c four.h
      gcc -c three.c
four.o: four.c four.h
      gcc -c four.c
```

Show the commands executed by **`make`** each time it is invoked in this sequence. Assume that the working directory contains all **`.h`** and **`.c`** files, the makefile, and no other files. Note that the **`touch`** command changes the date/time stamp of the given file to the current date/time, just as if the given file had been edited. (*five parts, 2 points each*)

```
$ make
```

```
$ touch myprog.c
$ make
```

```
$ make
```

```
$ touch two.h
$ make
$ touch four.h
$ make
```

QUESTION 6: ***Memory Management*** *(15 points)*

UNIX has a command named `tail`. In its simplest form, `tail` reads lines from stdin until end-of-file, and writes the last ten of the lines to stdout. Thus it prints the "tail" of stdin to stdout.

The program shown below is an attempt to implement that simple form of `tail`. It contains several *logic* and *stylistic* errors. On the next page, please rewrite the program to eliminate those errors. You may assume that no line contains more than 255 characters, including the '\n'. You need not write comments.

Note: The program could be made more efficient by using a circular queue to avoid moving elements within the array. Your rewritten program should not use a circular queue. Instead it should move elements within the array as the given program does

```
#include <stdio.h>

int main(void)
{
   char *pcLine;
   char *ppcLines[10];
   int i;

   for (i = 0; i < 10; i++)
      ppcLines[i] = NULL;

   while (fgets(pcLine, 256, stdin) != NULL)
   {
      for (i = 0; i < 9; i++)
         ppcLines[i] = ppcLines[i+1];
      ppcLines[9] = pcLine;
   }

   for (i = 0; i < 10; i++)
      if (ppcLines[i] != NULL)
         fputs(ppcLines[i], stdout);

   return 0;
}
```

Write your answer to question #6 here: