NAME:

Login name:

Computer Science 217
Final Exam
Saturday May 24, 2008
9am-12:00pm

This test has six (6) questions, with points ranging from 10 to 20. Put your name on ***every page***, and write out and sign the Honor Code pledge before turning in the test.

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

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

QUESTION 1: Good Bug Hunting *(20 POINTS)*

1a**)** This C code should print **`"Equal"`** when the two integers are equal, and "**`Not equal`**" otherwise. Indicate *all* cases where the wrong answer is printed, and then fix the bug. Assume variables **`i`** and **`j`** are integers that have already been set to some value. *(5 points)*

```
if (i = j)
  printf("Equal\n");
else
  printf("Not equal\n");
```

1b**)** To support generic implementations of functions like **`sort()`**, we need comparison functions for different data types. The function should return a *positive integer* if the first argument is larger than the second, a *zero* when the two numbers are equal, and a *negative integer* otherwise. The function **`CompareFloats()`**, which compares two float numbers, has a bug. When does the function produce the wrong answer? Fix the bug by providing a correct implementation. You need only rewrite the part of the code that is in error. *(5 points)*

```
int CompareFloats(void *p1, void *p2) {
   float *fp1 = (float *) p1;
   float *fp2 = (float *) p2;
   return (int) (*fp1 - *fp2);
}
```

1c**)** The function **`oddeven()`** counts the number of times two adjacent bits (odd and even bits) have different values. For example, the number "00 11 01 00" should produce a 1 because of the single occurrence of "01", whereas the number "01 10 01 10" should produce a 4 because every pair of bits has the odd and even bit disagree. The function is nearly correct but has a bug. What is it? Fix the bug by making a small modification to the existing code. Do not introduce any "if" statements – everything should be implemented using bit-wise operations. *(5 points)*

```
int oddeven(unsigned int x) {
  int b;

  for (b=0; x!=0; x>>=2)
    b += ((x & 1) ^ (x & 2));

  return b;
}
```

1d**)** This C code should remove the last character of a string **`str[]`**. Assume the string is already allocated enough space in memory (and ends properly with a '\0'). Identify the bug and fix both the specification of what the code does and the code itself accordingly. *(5 points)*

```
int i=0;

while (str[i] != '\0')
  i = i + 1;

str[--i] = '\0';
```

QUESTION 2: Short and Sweet *(20 POINTS)*

2a) The **`getchar()`** function returns an **`int`** instead of a **`char`** to have enough space to encode the **`End Of File`** (**`EOF`**). Why can't this be handled more elegantly by reserving one of the encodings of a character (e.g., an ASCII code, in the case of ASCII) for **`EOF`**? *(3 points)*

2b) Virtual memory divides the address space into pages of a certain size (e.g., 4 KB). What is the advantage of a *large* page size over a *small* page size? What is the disadvantage? *(2 points)*

2c) Why are the page tables kept in the *operating system's* virtual address space? Why are the pages that store the page tables non-swappable (i.e., "pinned" in physical memory)? *(4 points)*

2d) In machine language, a jump instruction (such as **`jle`** or **`jge`** in the IA32 language) often expresses the new address as a *displacement* from the current Instruction Pointer, rather than an *absolute address*. What is an advantage of representing the address as a displacement? *(2 points)*

2e) Function calls and system calls, while conceptually similar, are implemented somewhat differently. List one similarity and one difference in how they are executed, and why. *(4 points)*

2f) What are two kinds of references in a **`".o"`** file (generated by the assembler) that may remain unresolved, and hence need to be resolved by the linker. *(2 points)*

2g) In implementing **`malloc()`** why is a "good fit" strategy better than a "first fit" strategy, even though it may require sequencing through a larger collection of free blocks before selecting the block to use? Why is a "good fit" strategy preferable over a "best fit" strategy that finds the free block that wastes the least amount of space? *(3 points)*

QUESTION 3: Put a Fork in It *(10 POINTS)*

This question concerns the following program named **`cos217.c`**.

```
#include <stdio.h>
#include <unistd.h>

void cos217(void) {
  pid_t pid;

  fflush(NULL);
  if (!(pid = fork())) {
    fflush(NULL);
    fork();
    printf("cos217\n");
  }
}

int main(void) {
  cos217();
  printf("cos217\n");
  return 0;
}
```

3a) How many times does the program print "cos217"? Explain your answer. *(4 points)*

3b) The UNIX "**`cat`**" command concatenates files and prints to standard out. The UNIX "**`wc`**" command prints the number of lines, words, and bytes in a file; the "**`-l`**" option prints only the number of lines. What is the output of "**`cat cos217.c cos217.c | wc -l`**"? Note that the argument to "cat" is the cos217.c *file*, not the *output* produced in question 3a. *(3 points)*

3c) What is the output of "**`cat cos217.c cos217.c | wc -l | wc -l`**"? *(3 points)*

QUESTION 4: Can't Stop This Program We Started *(20 POINTS)*

Write a program that, every time the user enters "control-C", prints a line of `*' characters, where the number of asterisks correspond to the number of seconds in wallclock time since the last time "control-C" was entered. (For the first "control-C", print asterisks corresponding to the number of seconds since the program began executing.) The program should terminate, *without* printing a line of asterisks, when a "control-C" is entered after ***80*** or more seconds have elapsed without the user entering a "control-C". Round all time measurements ***up*** *to the nearest integer* number of seconds (e.g., 79.2 to 80, and 36.8 to 37). The following function and data structure are provided as reference:

```
#include <sys/time.h>

int gettimeofday(struct timeval *tp);
```

The **`gettimeofday()`** function gets the system's notion of the current time, expressed in elapsed seconds and microseconds since 00:00 Universal Coordinated Time, January 1, 1970. The **`tp`** argument points to a **`timeval`** structure, which includes the following members:

```
long tv_sec; /* seconds since January 1, 1970 */
long tv_usec; /* and microseconds */
```

which are assigned by the function. In practice, the **`gettimeofday`**() function has a second argument that is typically ignored; we have omitted that argument in the interest of simplicity. Upon successful completion, **`0`** is returned. Otherwise, **`-1`** is returned.

4a) Write a function "**`int timediff(struct timeval t1, struct timeval t2)`**" that returns the time difference between **`t2`** and **`t1`** in seconds, rounded up. You may assume that **`t2`** corresponds to a time larger than **`t1`**. Please implement the "rounding up" yourself, rather than calling the **`ceil()`** function from the math library. *(6 points)*

4b) Write the rest of the program, assuming the **`timediff()`** function is available. To keep the code simpler, feel free to use global variables and to assume that a "control-C" does not occur while the signal handler is handling an earlier control-C. *(14 points)*

QUESTION 5: RISCy Business *(15 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 Light Emitting Diodes (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. You may use labels such as "**FOO:**" to refer to the address of a particular instruction in memory.

```
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, putting a 0 in the top 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 A
```

(Note that the two shift instructions differ from a related question on the fall 2005 final exam.) 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.

5a) Write a program to display (on the output LEDs) the *inverse* of the bits entered at the input switches. For example, the input at the switches were “10101100”, the output should be “01010011” on the LEDs. *(5 points)*

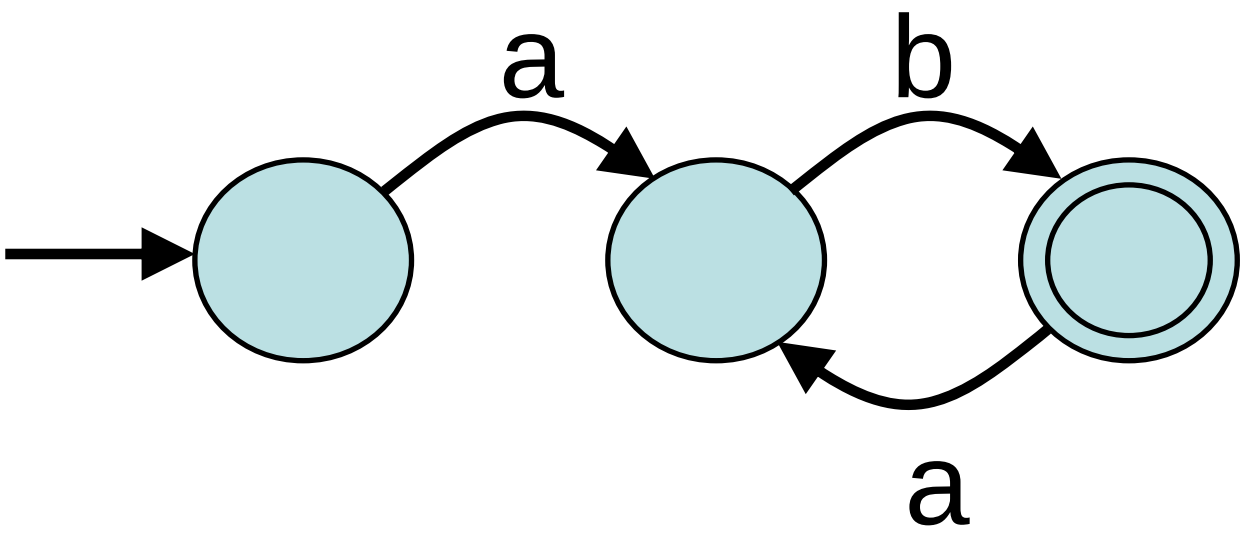
5b) Write a program that reads an 8-bit input from the switches and displays one bit at a time in the last bit position of the output LEDs, starting from the rightmost input bit and stopping once all remaining bits are zeroes. For example, an input of “00010101” would be displayed as six consecutive outputs “00000001”, “00000000”, “00000001”, “00000000”, “00000001”, and ending with “00000000” since the leftmost three bits are zeroes. The input “00000000” would be displayed simply as “00000000” since all eight bits are zeroes. Please explain/document your code. Use a loop (and a jump instruction to return to the beginning of the loop) to sequence through the input bits, rather than writing separate code to handle each bit. *(10 points)*

Continue your answer to question #5b.

QUESTION 6: Even Steven *(15 POINTS)*

A DFA cannot be designed to accept any string. For example, you *cannot* design a DFA to recognize a string with an equal number of zeroes and ones. However, you *can* design a DFA that accepts strings with an equal number of occurrences of 01 and 10. Draw a DFA that accepts such strings, assuming each character is either a 0 or a 1. For example, the DFA should report success for: the empty string (which has no occurrence of either 01 or 10), 010 (which has one instance of 01 and one of 10), 0111111110 (which has one of each), 1000000001 (which has one of each), and 10101 (which has two of each). The DFA should report failure for 01111, 10000, 01111101, and 11111111000000. Please draw neatly.

Draw neatly and use as few states as possible. Your DFA diagram should include a start state (marked with an incoming arrow) and success states (circled twice). As an example, consider a DFA that reports "success" for an input that repeats the characters "ab" one or more times and reports "failure" otherwise. That DFA would be drawn as:



The inputs "ab", "abab", and "abababab" would lead to success, whereas "a", "aba", "abc", "789", or "cabab" would not. On the **next page**, draw your DFA to report success for a string with an equal number of occurrences of 01 and 10, where each character is either a 1 or a 0.

Put your answer to question #6 here: