Andrew login ID:______________________________________

Full Name:______________________________________

CS 15-213, Spring 2002

Exam 1

Febrary 26, 2002

Instructions:

- Make sure that your exam is not missing any sheets, then write your full name and Andrew login ID on the front.
- Write your answers in the space provided below the problem. If you make a mess, clearly indicate your final answer.
- The exam has a maximum score of 84 points.
- The problems are of varying difficulty. The point value of each problem is indicated. Pile up the easy points quickly and then come back to the harder problems.
- This exam is OPEN BOOK. You may use any books or notes you like. You may use a calculator, but no laptops or other wireless devices. Good luck!

1 (11):	
2 (19):	
3 (9):	
4 (10):	
5 (11):	
6 (12):	
7 (12):	
TOTAL (84):	

Problem 1. (11 points):

Assume we are running code on a 6-bit machine using two's complement arithmetic for signed integers. A "short" integer is encoded using 3 bits. Fill in the empty boxes in the table below. The following definitions are used in the table:

```
short sy = -3;
int y = sy;
int x = -13;
unsigned ux = x;
```

Note: You need not fill in entries marked with "–".

Expression	Decimal Representation	Binary Representation
Zero	0	
–	-3	
–		11 0010
ux		
y		
$x >> 2$		
TMax		

Problem 2. (19 points):

Consider the following 8-bit floating point representation based on the IEEE floating point format:

- There is a sign bit in the most significant bit.
- The next 3 bits are the exponent. The exponent bias is $2^{3-1} - 1 = 3$.
- The last 4 bits are the fraction.
- The representation encodes numbers of the form: $V = (-1)^s \times M \times 2^E$, where M is the significand and E is the biased exponent.

The rules are like those in the IEEE standard(normalized, denormalized, representation of 0, infinity, and NAN). FILL in the table below. Here are the instructions for each field:

- **Binary:** The 8 bit binary representation.
- **M:** The value of the significand. This should be a number of the form x or $\frac{x}{y}$, where x is an integer, and y is an integral power of 2. Examples include 0, $\frac{3}{4}$.
- **E:** The integer value of the exponent.
- **Value:**The numeric value represented.

  Note: you need not fill in entries marked with ”—”.
- **Smallest/Largest:** These refer to the absolute value. The number's relationship with 0 is indicated with positive/negative.

Description	Binary	M	E	Value
Positive zero				$+0.0$
—	`0 000 0101`			
Largest denormalized (positive)				
Smallest normalized (negative)				
Negative Two				-2.0
—				-14.5
Negative infinity		—	—	$-\infty$

Problem 3. (9 points):

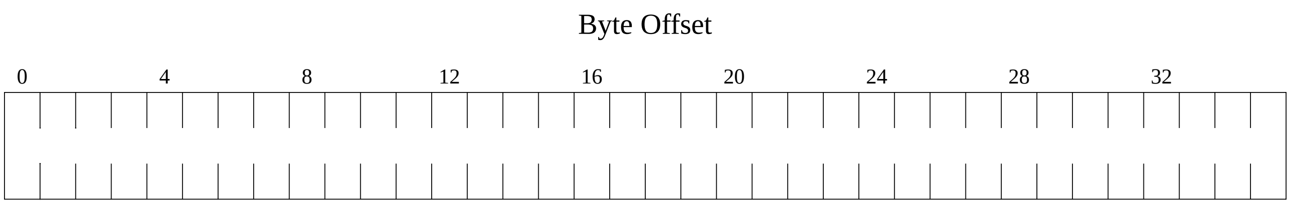
Consider the following C type definitions:

```
typedef struct {            typdef union {
     char c[2];                  int i;
     int *intp;                  char buf[6];
     union1 u;                   float f;
     double d;              } union1
     short s;
} struct1
```

Note: Assume the IA-32 **Windows** alignment convention for this problem – in particular, values of the type **double** must be 8-byte aligned. (vs. Linux where they are only 4-byte aligned).

A. What byte alignment is necessary for an object of type union1?

B. On the below template, draw out the allocation of data for an object of type struct1. Allowing for a maximum of 36 bytes, mark off and label each element in the struct as well as cross-hatch the unused, but allocated space. Clearly indicate the right hand boundary with a vertical line.



C. How many bytes need to be allocated for an object of type struct1?

D. Think about redefining the fields of struct1 in a different order so as to minimize the amount of unused, but allocated space for each variable of type struct1. Now how many bytes are required for an struct1 object?

Problem 4. (10 points):

Fill in the blanks of the C code. The assembly code and the memory status for the C code are given as follows.

Assembly code:

```
08048430 <foo>:
 8048430:   push   %ebp
 8048431:   mov    %esp,%ebp
 8048433:   push   %ebx
 8048434:   mov    0x8(%ebp),%ebx
 8048437:   xor    %eax,%eax
 8048439:   cmp    $0x4,%ebx
 804843c:   mov    0xc(%ebp),%ecx
 804843f:   mov    0x10(%ebp),%edx
 8048442:   ja     804846f <foo+0x3f>
 8048444:   jmp    *0x8048500(,%ebx,4)
 804844b:   nop
 804844c:   mov    %ecx,%eax
 804844e:   and    %edx,%eax
 8048450:   jmp    804846f <foo+0x3f>
 8048452:   mov    %esi,%esi
 8048454:   mov    %ecx,%eax
 8048456:   or     %edx,%eax
 8048458:   jmp    804846f <foo+0x3f>
 804845a:   mov    %esi,%esi
 804845c:   mov    %ecx,%eax
 804845e:   xor    %edx,%eax
 8048460:   jmp    804846f <foo+0x3f>
 8048462:   mov    %esi,%esi
 8048464:   mov    %ecx,%eax
 8048466:   not    %eax
 8048468:   jmp    804846f <foo+0x3f>
 804846a:   mov    %esi,%esi
 804846c:   lea    (%edx,%ecx,1),%eax
 804846f:   mov    (%esp,1),%ebx
 8048472:   leave
 8048473:   ret
```

Memory information given by gdb:

```
>gdb foo
(gdb) x /8w 0x8048500
0x8048500 : 0x0804844c 0x08048454 0x0804845c 0x08048464
0x8048510 : 0x0804846c 0x00000000 0x00000000 0x00000000
```

C code:

```
int foo(int op, int a, int b)
{
    int result = 0;
    switch (op)
    {
        case 0:_________________________;

        case 1:_________________________;

        case 2:_________________________;

        case 3:_________________________;

        case 4:_________________________;
    }
    return result;
}
```

Problem 5. (11 points):

The memory stack can become a large performance bottleneck for programs. For instance, each function must at least save the old base pointer as well as restore it once it is done executing. This leaves out other stack interactions, such as function calls.

One proposed way of increasing the performance of stack operations is to emulate a stack using registers on the processor itself. The Sun SPARC and Intel Itanium architectures both use register stacks in this fashion.

Now suppose you work for a processor company that has created a very specific processor for the scientific community. Hence, it only needs to deal with 4 byte quantities.

The company is the final stages of completing its processor, and is finishing up the documentation for their product. However, the technical writers are having problems understanding the register stack and they want you to explain it to them. Basically, they want to explain to C programmers how the processor emulates stack operations.

The following are the relevant operations the processor can perform:

- alloc: similar to operations performed at the beginning of a function
- push: a stack push operation
- pop: a stack pop operation
- call: used to call a function, similar to call from IA-32

The following are the registers the processor can use:

- RET*n*: return address registers
- BP*n*: stack frame pointer registers
- STK*n*: general purpose stack registers

Note that, although there is no reserved base pointer register (such as ebp), there is a notion of a base pointer.

A. Using the information above, fill in what each of the processor's stack operations perform.

- alloc
- push
- pop
- call

B. What is the problem with using registers in this fashion?

C. Why is the notion of a base pointer still required?

Problem 6. (12 points):

Dr. Evil was very impressed by your performance with the binary bomb and he has decided to hire you as a computer security consultant.

You are presented with the following code written by his minions. The provided code is responsible for handling incoming command requests from foreign hosts. The network connections have been tied to standard input (stdin), so input is handled exactly as it would have been had a user entered the input from the keyboard.

A. Which lines of code are weak? Please fix them.

B. Given the code as shown below, please show how you could position an exploit string in memory to ensure that it is executed.

Specifically, assume that the box below represents your input. The beginning of the buffer is located at memory address 0xbadbeef0. You have determined that your exploit code requires 32 bytes. Please draw lines to separate this box into sections, with one section for each component of the input. The label each box to indicate what it stores and its offset within the input string. For example, one section should be the text of the exploit. In other words, the hex code representing the assembly, Please don't forget to show the padding between the sections.

0xBADBEEF0 ⟶ High addresses

```
#define NUM_CMDS 10
extern char *cmds[NUM_CMDS];

1. int readcmd(void)
2.
3.          char *cmdname;
4.          char *result_fname;
5.          int index, fd;
6.          char buffer[512];
7.
8.          gets(buffer);
9.
10.         sscanf(buffer, "%d", &index);
11.
12.         /* Get command name */
13.         cmdname = cmds[index];
14.
15.         result_fname = process_cmd(cmdname,arg);
16.
17.         /* Log command under result for later analysis */
18.
19.         fd = open(result_fname, O_CREAT | O_WRONLY | O_APPEND);
20.         write(fd, buffer, sizeof(buffer));
21.
22.         return 0;
23.
```

Problem 7. (GG points):

Consider the following C function:

```
int FindMin(int *A, int size)
{
    int i;
    int min = A[0];

    for(i = 0; i < size; i++)
    {
       if(A[i] < min) {
          min = A[i];
       }
    }

    return min;
}
```

Fill in the assembly code for the body of the function.

```
FindMin:
   push %ebp
   mov %esp, %ebp
   mov 0x8(%ebp), %ecx
   mov 0xc(%ebp), %edx
```

```
   mov %ebp, %esp
   pop %ebp
   ret
```