Full Name:____

Andrew ID:____

Recitation Section:____

CS 15-213, Spring 2001

Exam 1

February 27, 2001

Instructions:

- Make sure that your exam is not missing any sheets, then write your full name and Andrew 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 70 points.
- This exam is OPEN BOOK. You may use any books or notes you like. You cannot, however, use any computers, calculators, palm pilots, ... Good luck!

1:	
2:	
3:	
4:	
5:	
6:	
TOTAL:	

Problem 1. (12 points):

For each of the following statements circle whether it is always true (True), never true (False), or sometimes true (Some). Assume the integer representation and implementation used by the IA32 architecture. Use the following definitions:

```
short sy = Some_arbitrary_short();
int x = Some_arbitrary_int();
int y = sy;
unsigned ux = x;
unsigned uy = y;
```

Also note that INT_MAX is the maximum positive integer and INT_MIN is the most negative integer.

`x & -1 == x`	True	False	Some
`INT_MAX + INT_MIN == 0`	True	False	Some
`x > 0` $\Rightarrow$ `x + INT_MAX < 0`	True	False	Some
`x + -x == 0`	True	False	Some
`(ux >> 1) == (x >> 1)`	True	False	Some
`(ux > uy)` $\Rightarrow$ `(x > y)`	True	False	Some
`ux > INT_MIN`	True	False	Some
`sy == y`	True	False	Some
`((unsigned) sy) == uy`	True	False	Some
`(~x + 1) == ~(x - 1)`	True	False	Some
`x >> 4 == x / 16`	True	False	Some
`ux & 255 == ux % 256`	True	False	Some

Problem 2. (12 points):

Consider the following 6-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 3.
- The last 2 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, NAN, and round-to-even).

Please fill in the table below. You do not have to fill in boxes with "——" in them. If a number is NAN, you may disregard the M, E, and V fields below. However, fill the Description, Hex, and Binary fields with valid data.

Here are some guidelines for each field:

- **Description** - A verbal description if the number has a special meaning
- **Hex** - The Hexadecimal equivalent of the Binary field
- **Binary** - Binary representation of the number
- M - Significand (same as the M in the formula above)
- E - Biased Exponent (same as the E in 2^E)
- V - Fractional Value represented

Please fill the M, E, and V fields below with rational numbers (fractions) rather than decimals

Description	Binary	Hex	M	E	V
Largest **Denormalized**					
Largest **Normalized** ($< \infty$)					
	`1 111 01`	`0x3D`			
——		`0x12`			
`2.0 + 0.375`					
`3.0 * 3.0`					

Problem 3. (12 points):

This problem tests your understanding of how while loops in C relate to IA32 assembly code. The following is the assembly code for function `foo`.

```
foo:
      pushl %ebp
      movl %esp,%ebp
      pushl %ebx
      movl 8(%ebp),%edx
      movl 12(%ebp),%ebx
      xorl %ecx,%ecx
      cmpl %ebx,%edx
      jg .L19
.L20:
      movl %edx,%eax
      imull %edx,%eax
      addl %eax,%ecx
      incl %edx
      cmpl %ebx,%edx
      jle .L20
.L19:
      movl %ecx,%eax
      popl %ebx
      movl %ebp,%esp
      popl %ebp
      ret
```

Fill in the blanks in the definition of `foo`. The only variables you need are `x`, `y`, and `result`.

```
int foo (int x, int y) {
  int result;

  ___________;
  while (________) {
    ________________;
    _________;
  }
  return result;
}
```

Problem 4. (12 points):

The following problem will test your understanding of stack frames. It is based on the following function:

```
int power(int *val, int n)
{
  int result = 1;

  if (n > 0) result = *val * power(val, n-1);

  return result;
}
```

A compiler on an IA-32 Linux machine produces the following object code for this function, which we have disassembled (using `objdump`) back into assembly code:

```
080483b4 <power>:
   80483b4: 55                      push   %ebp
-> 80483b5: 89 e5                   mov    %esp,%ebp
   80483b7: 83 ec 14                sub    $0x14,%esp
   80483ba: 53                      push   %ebx
   80483bb: 8b 5d 08                mov    0x8(%ebp),%ebx
   80483be: 8b 55 0c                mov    0xc(%ebp),%edx
   80483c1: b8 01 00 00 00          mov    $0x1,%eax
   80483c6: 85 d2                   test   %edx,%edx
   80483c8: 7e 10                   jle    80483da <power+0x26>
   80483ca: 83 c4 f8                add    $0xfffffff8,%esp
   80483cd: 8d 42 ff                lea    0xffffffff(%edx),%eax
   80483d0: 50                      push   %eax
   80483d1: 53                      push   %ebx
   80483d2: e8 dd ff ff ff          call   80483b4 <power>
   80483d7: 0f af 03                imul   (%ebx),%eax
   80483da: 8b 5d e8                mov    0xffffffe8(%ebp),%ebx
   80483dd: 89 ec                   mov    %ebp,%esp
   80483df: 5d                      pop    %ebp
   80483e0: c3                      ret
   80483e1: 8d 76 00                lea    0x0(%esi),%esi
```

A. On the next page, you have the diagram of the stack immediately after some function makes a call to `power()`. The value of register `%esp` is `0xbffff6d8`. The instruction to be executed next is denoted with an arrow (`->`) in the assembly code above. For each of the numeric values shown in the table, give a short description of the value. If the value has a corresponding variable in the original C source code, use the name of this variable as its description.

B. Assume that `power()` runs until it reaches the position denoted with an arrow (`->`) again. In the table on the next stage, fill in the updated stack. Use a numeric value (if possible, else write `n/a`) and provide a short description of the value. Cross out any stack space not used.

C. Which instruction (give its address) computes the value `n-1`?

Address	Numeric Value	Comments/Description
0xbffff6e4	2	
0xbffff6e0	0xbffff704	
0xbffff6dc	0x080483ff	
0xbffff6d8	0xbffff708	
0xbffff6d4		
0xbffff6d0		
0xbffff6cc		
0xbffff6c8		
0xbffff6c4		
0xbffff6c0		
0xbffff6bc		
0xbffff6b8		
0xbffff6b4		
0xbffff6b0		
0xbffff6ac		
0xbffff6a8		
0xbffff6a4		

Problem 5. (12 points):

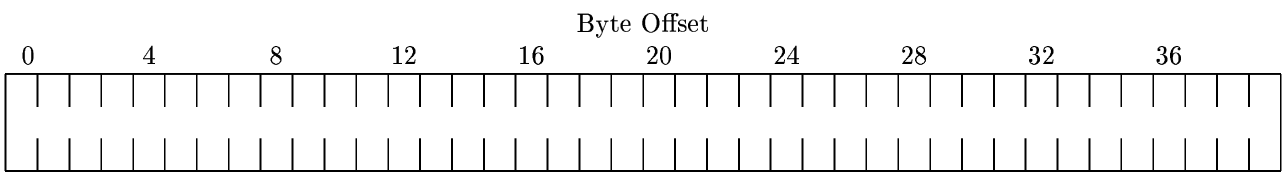
For the following problem assume the IA-32 Windows alignment convention—*i.e.*, values of type `double` must be 8-byte aligned (vs. the Linux convention where they are only 4-byte aligned). Consider the following definition:

```
typedef union {
 short s[3];
 int i;
 double d;
} union1;
```

```
typedef struct {
 short s;
 int i;
 double *d;
 union1 un;
 int j;
} struct1;

struct1 A[3][2];
```

The following template is provided as an aid to help you solve this problem. You do not have to use it and **anything written in this template will not be graded**.



What is the **byte** offset relative to the start of `A` for each of the following locations (assuming IA-32 conventions):

1. `&A[0][0]`
2. `&A[0][0].s`
3. `&A[0][0].i`
4. `&A[0][0].d`
5. `&A[0][0].un.s`
6. `&A[0][0].un.d`
7. `&A[1][0]`
8. `&A[0][1]`
9. `&A[0][0] + 1`
10. `&A[0][0].s + 1`
11. `&A[0][0].un.i + 1`
12. `&A[1]`

Problem 6. (10 points):

The following C file, `p.c`, contains a simple function called `process` as shown below.

```
extern int status;
void toggle(void);

void process(int n)
{
  if (((n < 0) && (status == 1)) || ((n > 0) && (status == -1))) toggle();
}
```

The function references an external global variable called `status` and a function called `toggle`. Both `status` and `toggle` are defined in the file `toggle.c`, which is shown below.

```
int status = 1;
int changes = 0;

void toggle(void)
{
  status = -status;
  changes++;
}
```

A relocatable object file `p.o` has been created and then disassembled using the commands

```
gcc -O2 -c -o p.o p.c
objdump -d p.o > p.bdis
```

The disassembled file `p.bdis` is shown on the next page. The relocation directives in the relocatable object file `p.o` are NOT displayed in `p.bdis`, because `objdump` was invoked without the `-r` flag. Not shown is the relocatable object file `toggle.o`, which was created as follows:

```
gcc -O2 -c -o toggle.o toggle.c
```

Your task is to circle all of the bytes in the disassembled object file `p.bdis` that the linker `ld` will modify when it creates an executable object file that includes the relocatable object files `p.o` and `toggle.o`.

```
process.o:     file format elf32-i386

Disassembly of section .text:

00000000 <process>:
   0: 55                   push   %ebp
   1: 89 e5                mov    %esp,%ebp
   3: 83 ec 08             sub    $0x8,%esp
   6: 8b 45 08             mov    0x8(%ebp),%eax
   9: 85 c0                test   %eax,%eax
   b: 7d 09                jge    16 <process+0x16>
   d: 83 3d 00 00 00 00 01 cmpl   $0x1,0x0
  14: 74 0d                je     23 <process+0x23>
  16: 85 c0                test   %eax,%eax
  18: 7e 0e                jle    28 <process+0x28>
  1a: 83 3d 00 00 00 00 ff cmpl   $0xffffffff,0x0
  21: 75 05                jne    28 <process+0x28>
  23: e8 fc ff ff ff       call   24 <process+0x24>
  28: 89 ec                mov    %ebp,%esp
  2a: 5d                   pop    %ebp
  2b: c3                   ret
```