Andrew login ID:_________________________________
Full Name:_________________________________
Section:_________________________________

15-213/18-243, Fall 2010

Exam 1 - Version A

Tuesday, September 28, 2010

Instructions:

- Make sure that your exam is not missing any sheets, then write your Andrew login ID, full name, and section on the front.
- This exam is closed book, closed notes, although you may use a single 8 1/2 x 11 sheet of paper with your own notes. You may not use any electronic devices.
- 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 60 points.
- The problems are of varying difficulty. The point value of each problem is indicated. Good luck!

1 (10):	
2 (10):	
3 (6):	
4 (6):	
5 (4):	
6 (10):	
7 (14):	
TOTAL (60):	

Problem 1. (10 points):

General systems concepts. Write the correct answer for each question in the following table:

1	2	3	4	5	6	7	8	9	10
11	12	13	14	15	16	17	18	19	20

1. Consider the following code, what is the output of the printf?

```
int x = 0x15213F10 >> 4;
char y = (char) x;
unsigned char z = (unsigned char) x;
printf("%d, %u", y, z);
```

   (a) -241, 15
   (b) -15, 241
   (c) -241, 241
   (d) -15, 15

2. In two's compliment, what is $-TMin$?

   (a) $Tmin$
   (b) $Tmax$
   (c) 0
   (d) -1

3. Let $int\ x = -31/8$ and $int\ y = -31 >> 3$. What are the values of x and y?

   (a) $x = -3, y = -3$
   (b) $x = -4, y = -4$
   (c) $x = -3, y = -4$
   (d) $x = -4, y = -3$

4. In C, the expression "$15213U > -1$" evaluates to:

   (a) True (1)
   (b) False (0)

5. In two's compliment, what is the minimum number of bits needed to represent the numbers -1 and the number 1 respectively?

   (a) 1 and 2
   (b) 2 and 2
   (c) 2 and 1
   (d) 1 and 1

6. Consider the following program. Assuming the user correctly types an integer into stdin, what will the program output in the end?

```
#include <stdio.h>
int main(){
    int x = 0;
    printf("Please input an integer:");
    scanf("%d",x);
    printf("%d", (!!x)<<31);
}
```

   (a) 0
   (b) $TMin$
   (c) Depends on the integer read from stdin
   (d) Segmentation fault

7. By default, on Intel x86, the stack
   (a) Is located at the bottom of memory.
   (b) Grows down towards smaller addresses
   (c) Grows up towards larger addresses
   (d) Is located in the heap

8. Which of the following registers stores the return value of functions in Intel x86_64?
   (a) %rax
   (b) %rcx
   (c) %rdx
   (d) %rip
   (e) %cr3

9. The `leave` instruction is effectively the same as which of the following:
   (a) `mov %ebp, %esp`
   `pop %ebp`
   (b) `pop %eip`
   (c) `mov %esp, %ebp`
   `pop %esp`
   (d) `ret`

10. Arguments to a function, in Intel IA32 assembly, are passed via
   (a) The stack
   (b) Registers
   (c) Physical memory
   (d) The `.text` section
   (e) A combination of the stack and registers.

11. A buffer overflow attack can only be executed against programs that use the `gets` function.

    (a) True
    (b) False

12. Intel x86_64 systems are

    (a) Little endian
    (b) Big endian
    (c) Have no endianess
    (d) Depend on the operating system

13. Please fill in the return value for the following function calls on both an Intel IA32 and Intel x86_64 system:

Function	Intel IA32	Intel x86_64
`sizeof(char)`		
`sizeof(int)`		
`sizeof(void *)`		
`sizeof(int *)`		

14. Select the two's complement negation of the following binary value: `0000101101`:

    (a) `1111010011`
    (b) `1111010010`
    (c) `1000101101`
    (d) `1111011011`

15. Which line of C-code will perform the same operation as `leal 0x10(%rax,%rcx,4),%rax`?

    (a) `rax = 16 + rax + 4*rcx`
    (b) `rax = *(16 + rax + 4*rcx)`
    (c) `rax = 16 + *(rax + 4*rcx)`
    (d) `*(16 + rcx + 4*rax) = rax`
    (e) `rax = 16 + 4*rax + rcx`

16. Which line of Intel x86-64 assembly will perform the same operation as `rcx = ((int *)rax)[rcx]`?

    (a) `mov (%rax,%rcx,4),%rcx`
    (b) `lea (%rax,%rcx,4),%rcx`
    (c) `lea (%rax,4,%rcx),%rcx`
    (d) `mov (%rax,4,%rcx),%rcx`

17. If `a` is of type `(int)` and b is of type `(unsigned int)`, then `(a < b)` will perform

    (a) An unsigned comparison.
    (b) A signed comparison.
    (c) A segmentation fault.
    (d) A compiler error.

18. Denormalized floating point numbers are

    (a) Very close to zero (small magnitude)
    (b) Very far from zero (large magnitude)
    (c) Un-representable on a number line
    (d) Zero.

19. What is the difference between an arithmetic and logical right shift?

    (a) C uses arithmetic right shift; Java uses logical right shift.
    (b) Logical shift works on 32 bit data; arithmetic shift works on 64 bit data.
    (c) They fill in different bits on the left
    (d) They are the same.

20. Which of the following assembly instructions is invalid in Intel IA32 Assembly?

    (a) `pop %eip`
    (b) `pop %ebp`
    (c) `mov (%esp),%ebp`
    (d) `lea 0x10(%esp),%ebp`

Problem 2. (10 points):

Floating point encoding. Consider the following 5-bit floating point representation based on the IEEE floating point format. This format does not have a sign bit – it can only represent nonnegative numbers.

- There are $k = 3$ exponent bits. The exponent bias is 3.
- There are $n = 2$ fraction bits.

Recall that numeric values are encoded as a value of the form $V = M \times 2^E$, where E is the exponent after biasing, and M is the significand value. The fraction bits encode the significand value M using either a denormalized (exponent field 0) or a normalized representation (exponent field nonzero). The exponent E is given by $E = 1 - \mathit{Bias}$ for denormalized values and $E = e - \mathit{Bias}$ for normalized values, where e is the value of the exponent field `exp` interpreted as an unsigned number.

Below, you are given some decimal values, and your task it to encode them in floating point format. In addition, you should give the rounded value of the encoded floating point number. To get credit, you must give these as whole numbers (e.g., 17) or as fractions in reduced form (e.g., $3/4$). Any rounding of the significand is based on ***round-to-even***, which rounds an unrepresentable value that lies halfway between two representable values to the nearest even representable value.

Value	Floating Point Bits	Rounded value
$9/32$	`001 00`	1/4
1		
12		
11		
$1/8$		
$7/32$		

Problem 3. (6 points):

Accessing arrays. Consider the C code below, where H and J are constants declared with `#define`.

```
int array1[H][J];
int array2[J][H];

int copy_array(int x, int y) {
    array2[y][x] = array1[x][y];

    return 1;
}
```

Suppose the above C code generates the following x86-64 assembly code:

```
# On entry:
#    %edi = x
#    %esi = y
#
copy_array:
        movslq  %esi,%rsi
        movslq  %edi,%rdi
        movq    %rsi, %rax
        salq    $4, %rax
        subq    %rsi, %rax
        addq    %rdi, %rax
        leaq    (%rdi,%rdi,2), %rdi
        addq    %rsi, %rdi
        movl    array1(,%rdi,4), %edx
        movl    %edx, array2(,%rax,4)
        movl    $1, %eax
        ret
```

What are the values of `H` and `J`?

```
H =

J =
```

Problem 4. (6 points):

Structure alignment. Consider the following C struct:

```
struct {
    char a, b;
    short c;
    long d;
    int *e;
    char f;
    float g;
} foo;
```

1. Show how the struct above would appear on a 32 bit Windows machine (primitives of size k are k-byte aligned). Label the bytes that belong to the various fields with their names and clearly mark the end of the struct. Use hatch marks to indicate bytes that are allocated in the struct but are not used.

```
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+

+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+

+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
```

2. Rearrange the above fields in foo to conserve the most space in the memory below. Label the bytes that belong to the various fields with their names and clearly mark the end of the struct. Use hatch marks to indicate bytes that are allcoated in the struct that are not used.

```
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+

+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+

+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
```

Problem 5. (4 points):

Structure access. Consider the following data structure declaration:

```
struct ms_pacman{
  short wire;
  int resistor;
  union transistor{
    char bjt;
    int* mosfet;
    long vacuum_tube[2];
  }transistor;
  struct ms_pacman* connector;
};
```

Below are given four C functions and four x86-64 code blocks.

```
char* inky(struct ms_pacman *ptr){
  return &(ptr->transistor.bjt);
}

long blinky(struct ms_pacman *ptr){
  return ptr->connector->
         transistor.vacuum_tube[1];
}

int pinky(struct ms_pacman *ptr){
  return ptr->resistor;
}

int clyde(struct ms_pacman *ptr){
  return *(ptr->transistor.mosfet);
}
```

A	`mov  0x8(%rdi), %rax`
	`mov  (%rax),%eax`
	`retq`

B	`lea  0x8(%rdi), %rax` `retq`

C	`mov  0x4(%rdi), %eax` `retq`

D	`mov  0x18(%rdi),%rax`
	`mov  0x10(%rax),%rax`
	`retq`

In the following table, next to the name of each x86-64 code block, write the name of the C function that it implements.

Code Block	Function Name
A	
B	
C	
D	

Problem 6. (10 points):

Switch statement encoding. Consider the following C code and assembly code for a strange but simple function:

```
int lol(int a, int b)           40045c <lol>:
{                               40045c: lea    -0xd2(%rdi),%eax
    switch(a)                   400462: cmp    $0x9,%eax
    {                           400465: ja     40048a <lol+0x2e>
        case 210:               400467: mov    %eax,%eax
            b *= 13;            400469: jmpq   *0x400590(,%rax,8)
            ________            400470: lea    (%rsi,%rsi,2),%eax
        case 213:               400473: lea    (%rsi,%rax,4),%eax
            b = 18243;          400476: retq
            ________            400477: mov    $0x4743,%esi
        case 214:               40047c: mov    %esi,%eax
            b *= b;             40047e: imul   %esi,%eax
            ________            400481: retq
        case 216:               400482: mov    %esi,%eax
        case 218:               400484: sub    %edi,%eax
            b -= a;             400486: retq
            ________            400487: add    $0xd,%esi
        case 219:               40048a: lea    -0x9(%rsi),%eax
            b += 13;            40048d: retq
            ________
        default:
            b -= 9;
    }

    return b;
}
```

Using the available information, fill in the jump table below. (Feel free to omit leading zeros.) Also, for each case in the `switch` block which should have a `break`, write `break` on the corresponding blank line.

Hint: `0xd2 = 210` and `0x4743 = 18243`.

`0x400590:`	________	`0x400598:`	________
`0x4005a0:`	________	`0x4005a8:`	________
`0x4005b0:`	________	`0x4005b8:`	________
`0x4005c0:`	________	`0x4005c8:`	________
`0x4005d0:`	________	`0x4005d8:`	________

Problem 7. (14 points):

Stack discipline. This problem concerns the following C code, compiled on a 32-bit machine:

```
void foo(char * str, int a) {

  int buf[2];
  a = a; /* Keep GCC happy */
  strcpy((char *) buf, str);

}

/*
  The base pointer for the stack
  frame of caller() is: 0xffffd3e8
*/
void caller() {

  foo(‘‘0123456’’, 0xdeadbeef);

}
```

Here is the corresponding machine code on a 32-bit Linux/x86 machine:

```
080483c8 <foo>:
080483c8 <foo+0>:     push   %ebp
080483c9 <foo+1>:     mov    %esp,%ebp
080483cb <foo+3>:     sub    $0x18,%esp
080483ce <foo+6>:     lea    -0x8(%ebp),%edx
080483d1 <foo+9>:     mov    0x8(%ebp),%eax
080483d4 <foo+12>:    mov    %eax,0x4(%esp)
080483d8 <foo+16>:    mov    %edx,(%esp)
080483db <foo+19>:    call   0x80482c0 <strcpy@plt>
080483e0 <foo+24>:    leave
080483e1 <foo+25>:    ret

080483e2 <caller>:
080483e2 <caller+0>:  push   %ebp
080483e3 <caller+1>:  mov    %esp,%ebp
080483e5 <caller+3>:  sub    $0x8,%esp
080483e8 <caller+6>:  movl   $0xdeadbeef,0x4(%esp)
080483f0 <caller+14>: movl   $0x80484d0,(%esp)
080483f7 <caller+21>: call   0x80483c8 <foo>
080483fc <caller+26>: leave
080483fd <caller+27>: ret
```

This problem tests your understanding of the stack discipline and byte ordering. Here are some notes to help you work the problem:

- `strcpy(char *dst, char *src)` copies the string at address `src` (including the terminating '`\0`' character) to address `dst`.
- Keep endianness in mind.
- You will need to know the hex values of the following characters:

Character	Hex value	Character	Hex value
'0'	0x30	'4'	0x34
'1'	0x31	'5'	0x35
'2'	0x32	'6'	0x36
'3'	0x33	'`\0`'	0x00

Now consider what happens on a Linux/x86 machine when `caller` calls `foo`.

A. Stack Concepts:
a) Briefly describe the difference between the x86 instructions `call` and `jmp`.

b) Why doesn't `ret` take an address to return to, like `jmp` takes an address to jump to?

B. Just before `foo` calls `strcpy`, what integer `x`, if any, can you guarantee that `buf[x] == a` ?

C. At what memory address is the string "0123456" stored (before it is `strcpy`'d)?

We encourage you to use this space to draw pictures:

D. Just after `strcpy` returns to `foo`, fill in the following with hex values:

```
buf[0] = 0x______ ______ ______ ______

buf[1] = 0x______ ______ ______ ______

buf[2] = 0x______ ______ ______ ______

buf[3] = 0x______ ______ ______ ______

buf[4] = 0x______ ______ ______ ______
```

E. Immediately before the call to `strcpy`, what is the the value at `%ebp` (not what is `%ebp`)?

F. Immediately before `foo`'s `ret` call, what is the value at `%esp` (what's on the top of the stack)?

G. Will a function that calls `caller()` segfault or notice any stack corruption? Explain.