Full Name:_____

Roll Number:_____

IT215: Systems Software, Winter 2012-13

First In-Sem Exam (2 hours)

February 11, 2013

Instructions:

- Make sure that your exam is not missing any sheets, then write your full name and roll number on the front.
- Clearly write your answer in the space indicated. For rough work, do not use any additional sheets. Rough work will not be graded.
- Assume IA32 machine running Linux/GAS unless stated otherwise.
- The exam has a maximum score of 50 points. The problems are of varying difficulty. The point value of each problem is indicated.
- This exam is CLOSED BOOK. Notes are NOT allowed.
- Anyone who copies of allows someone to copy will receive F grade.

  Good luck!

1 (10):
2 (3):
3 (3):
4 (2):
5 (8):
6 (12):
7 (12):
TOTAL (50):

Problem 1. (10 points):

Circle the correct answer.

1. Here is a small C program:

```
struct foo { int bar; int baz; };

int get_baz(struct foo *foot_ptr)
{
     return foot_ptr->baz;
}
```

After compiling the code, diassembling `get_baz`, and adding a few comments, we get:

```
get_baz:  push %ebp              ; save old frame base pointer
          mov  %esp, %ebp        ; set frame base pointer
          mov  0x8(%ebp), %eax   ; move foo_ptr to %eax
          -Missing Instruction Goes Here-
          leave                  ; prepare stack for return
          ret
```

What is the Missing Instruction?

(a) `mov $baz(%eax), %eax`

(b) `mov 0x4(%eax), %eax`

(c) `lea 0x4(%eax), %eax`

(d) `mov 0xc(%ebp), %eax`

2. If `%esp` has the value `0xBFFF0000` before a call instruction, the value immediately after the call instruction (before the first instruction of the called instruction) is:

(a) `0xBFFEFFFC`

(b) `0xBFFF0004`

(c) `0xBFFF0000`

(d) The address of the instruction after the call instruction.

3. Which of the following x86 instructions can be used to add two registers and store the result in a register without overwriting either of the original registers?

(a) `move`

(b) `lea`

(c) `add`

(d) None of the above

4. Consider the following two blocks of code, found in *separate files*:

```
/* main.c */
int i = 0;
int main()
{
  foo();
  return 0;
}
```

```
/* foo.c */
int i = 1;
void foo()
{
  printf(‘‘%d’’, i);
}
```

What will happen when you attempt to compile, link, and run this code?

(a) It will fail to compile.

(b) It will fail to link.

(c) It will raise a segmentation fault.

(d) It will print “0”.

(e) It will print “1”.

(f) It will sometimes print “0” and sometimes print “1”.

5. The function `bitsy` is declared in C as

```
int bitsy(int x);
```

and the (correctly) compiled IA32 code is:

```
bitsy:  push %ebp
        mov  %esp, %ebp
        sub  $0x8, %esp
        mov  0x8(%ebp), %eax
        not  %eax
        inc  %eax
        leave
        ret
```

What is the result (denoted here by a C expression) returned by `bitsy`?

(a) `!(x + 1)`

(b) `*(1 - x)`

(c) `-x`

(d) `(x > 0 ?  -x:  -x + 1)`

Problem 2. (3 points):

Consider the following code, being executed on a Little Endian Pentium machine where

- `sizeof(int) == 4`
- `sizeof(int *) == 4`
- `sizeof(char) == 1`

For each of the following assignment statements, fill in the blanks in the comments to indicate the result of the assignment. All answers must be in hex.

```
int main() {
    int array[2];
    int *ptr;
    int x;
    char c;

    array[0]= 0xaabbccdd;
    array[1] = 0x44556677;

    ptr = array;

    x = *((int *)ptr + 1);

    /* x = 0x_____________________________*/

    c = *((char *)ptr + 1);

    /* c = 0x_____________________________*/

    c = *((int *)ptr + 1);

    /* c = 0x_____________________________*/
}
```

Problem 3. (3 points):

Consider the executable object file `a.out`, which is compiled and linked using the command:

```
unix> gcc -o a.out main.c foo.c
```

and where the files `main.c` and `foo.c` consist of the following code:

```
/* main.c */
#include <stdio.h>

static int a = 1;
int b = 2;
int c;

int main()
{
  int c = 3;

  foo();
  printf(''a=%d, b=%d, c=%d'', a, b, c);
  return 0;
}
```

```
/* foo.c */
int a, b, c;

void foo()
{
   a = 4;
   b = 5;
   c = 6;
}
```

What is the output of `a.out`?

a=_____, b=_____, c= ____

Problem 4. (2 points):

The declaration of `myGlobal` in the following code is not executable, while the declaration of `myLocal` is executable, that is, the compiler does not emit any instructions to be executed at run time for the former, while it does for the latter. Briefly explain why (for both cases), and how `myGlobal` is initialized if there are no run time instructions for it.

```
int myGlobal = 10 + 7;

int mySub() {
    int myLocal = 0;
    ...
}
```

Answer:

Problem 5. (8 points):

A C function `looper` and the assembly code it compiles to is shown below:

```
looper:
  pushl %ebp
  movl %esp,%ebp
  pushl %esi
  pushl %ebx
  movl 8(%ebp),%ebx
  movl 12(%ebp),%esi
  xorl %edx,%edx
  xorl %ecx,%ecx
  cmpl %ebx,%edx
  jge .L25
.L27:
  movl (%esi,%ecx,4),%eax
  cmpl %edx,%eax
  jle .L28
  movl %eax,%edx
.L28:
  incl %edx
  incl %ecx
  cmpl %ebx,%ecx
  jl .L27
.L25:
  movl %edx,%eax
  popl %ebx
  popl %esi
  movl %ebp,%esp
  popl %ebp
  ret
```

```
int looper(int n, int *a) {
  int i;
  int x = ________________;

  for(i = ______________;

      ____________________;

      i++) {

    if (_____________________)

       x = ____________________;


    ________________________;
  }

  return x;
}
```

Based on the assembly code, fill in the blanks in the C source code.

Notes:

- You may only use the C variable names `n`, `a`, `i` and `x`, not register names.
- Use array notation in showing accesses or updates to elements of `a`.

In the following problem, you are given the task of reconstructing C code based on some declarations of C structures, and the IA32 assembly code generated when compiling the C code.

Below are the data structure declarations. (Note that this is a single declaration which includes several data structures; they are shown horizontally rather than vertically simply so that they fit on one page.) Assume the Linux conventions for data alignment discussed in class.

```
struct s1 {            struct s2 {
  char a[3];             struct s1 *f;
  struct s12 {           char g;
     int b;              int h[4];
     struct s2 *c;       struct s2 *i;
  } d;                 };
  int e;
};
```

You may find it helpful to diagram these data structures in the space below:

Problem 6. (12 points):

For each IA32 assembly code sequence below on the left, fill in the missing portion of corresponding C source line on the right.

A.
```
fun1:
    pushl %ebp
    movl %esp,%ebp
    movl 8(%ebp),%eax
    movl 12(%eax),%eax
    popl %ebp
    ret
```

```
int fun1(struct s2 *x)
{
  return x->_____________________ ;
}
```

B.
```
fun2:
    pushl %ebp
    movl %esp,%ebp
    movl 8(%ebp),%eax
    movl (%eax),%eax
    movl 12(%eax),%eax
    popl %ebp
    ret
```

```
int fun2(struct s2 *x)
{
  return x->_____________________ ;
}
```

C.
```
fun3:
    pushl %ebp
    movl %esp,%ebp
    movl 8(%ebp),%eax
    movl (%eax),%eax
    movl 8(%eax),%eax
    movl 8(%eax),%eax
    popl %ebp
    ret
```

```
int fun3(struct s2 *x)
{
  return x->_____________________ ;
}
```

D.
```
fun4:
    pushl %ebp
    movl %esp,%ebp
    movl 8(%ebp),%eax
    movl 24(%eax),%eax
    movl 24(%eax),%eax
    movl 16(%eax),%eax
    popl %ebp
    ret
```

```
int fun4(struct s2 *x)
{
  return x->_____________________ ;
}
```

Problem 7. (12 points):

Below is a segment of code that reads a string from standard input.

```
int getbuf() {
    char buf[8];
    Gets(buf);
    return 1;
}
```

The function `Gets` is similar to the library function `gets`. It reads a string from standard input (terminated by a newline or end-of-file character) and stores it (along with a null terminator) at the specified destination. `Gets` has no way of determining whether `buf` is large enough to store the whole input. It simply copies the entire input string, possibly overrunning the bounds of the storage allocated at the destination.

Below is the object dump of the `getbuf` function:

```
08048c4b <getbuf>:
 8048c4b:  55                    push   %ebp
 8048c4c:  89 e5                 mov    %esp,%ebp
 8048c4e:  83 ec 38              sub    $0x20,%esp
 8048c51:  8d 45 d8              lea    0xfffffff0(%ebp),%eax
 8048c54:  89 04 24              mov    %eax, (%esp)
 8048c57:  e8 f2 00 00 00        call   8048d4e <Gets>
 8048c5c:  b8 01 00 00 00        mov    $0x1,%eax
 8048c61:  c9                    leave
 8048c62:  c3                    ret
```

Suppose that we set a breakpoint in function `getbuf` and then use gdb to run the program with an input file redirected to standard input. The program stops at the breakpoint when it has completed the `sub` instruction at `0x08048c4e` and is poised to execute the `lea` instruction at `0x08048c51`. At this point we run the following gdb command that lists the 12 4-byte words on the stack starting at the address in `%esp`:

```
0x08048c51 in getbuf ()
(gdb) x/12w $esp
0x55683a58:  0x003164f8   0x00000001   0x55683a98   0x0030bab6
0x55683a68:  0x003166a4   0x555832e8   0x00000001   0x00000001
0x55683a78:  0x55683ab0   0x08048bf9   0x55683ab0   0x0035b690
```

A. What is the address of `buf`? `0x`_____

B. When the program reaches the breakpoint, what is the value of `%ebp`? `0x`_____

C. To which address will getbuf return after executing? `0x`_____

D. When the program reaches the breakpoint, what is the value of `%esp`? `0x`_____

E. Instead of having `getbuf` return to its calling function, suppose we want it to return to a function `smoke` that has the address `0x8048b20`.

   Below is an incomplete sequence of the hex values of each byte in the file that was input to the program (we have given you the first 8 padding values). Fill in the remaining blank hex values so that the call to `Gets` will return to `smoke`. Note that `smoke` does not depend on the value stored in `%ebp`.

0x01	0x02	0x03	0x04	0x05	0x06	0x07	0x08
0x____	0x____	0x___	0x___	0x___	0x___	0x___	0x___
0x____	0x____	0x___	0x___	0x___	0x___	0x___	0x___