Assignment 3: Memory layout and allocation

15-411: Compiler Design
David McWherter (cache@cs) and Noam Zeilberger (noam@cs)

Due: Thursday, October 11, 2007 (1:30 pm)

Problem 1 — Structure representation

[20 points]
Consider the following C struct definitions:

```
struct foo {
        char     x1;
        int      x2;
        long     x3;
        char     x4;
        int      x5;
        long     x6;
        char     x7;
        int      x8;
        long     x9;
};

struct bar {
        int      y1;
        char     y2;
};

struct baz {
        struct bar z1;
        struct foo z2;
        struct bar z3;
};

struct qux {
        struct bar z1;
        struct bar z2;
        struct foo z3;
};
```

(a) Conforming to the x86-64 ABI, describe the layout of `struct foo` and `struct bar` by giving the offset of each field, the total size of the structure, and its alignment requirement.

(b) If we violate the ABI by reordering fields, we can give a more space-efficient layout of `struct foo`. Describe a minimum-space layout that still obeys the x86-64 alignment restrictions.

(c) Conforming to the x86-64 ABI, describe the layout of `struct baz` and `struct qux`.

Problem 2 — Alloca

[20 points]

Assume again that you are working with C. Consider the function `alloca()`, which allocates dynamic memory off the stack.

The implementation of `alloca()` on x86-32 is easy, since it maintains both a stack pointer and base pointer in dedicated registers. Whenever a function returns to its caller, it automatically frees the memory allocated by `alloca()` with the standard function epilogue.

On other architectures, such as the PowerPC, there is no dedicated base pointer, and only a stack pointer. On function entry, the stack pointer is decremented by a fixed amount (determined at compile-time), to hold local variables. Variables are referenced by offsets relative to the stack, and the function returns by incrementing the stack pointer the same amount it was decremented.

Clearly, this makes implementing `alloca()` more difficult. Changing the stack pointer means that the code can no longer access local variables correctly.

Despite these problems, you want to still support `alloca()` on PowerPC, mostly because your favorite editor, Emacs, depends on it.

There are two ways to implement `alloca()` in this situation. (A) You can change your compiler to generate special code for functions in which `alloca()` is called. (B) You can implement `alloca()` as a library function which approximates `alloca()` using `malloc()` and `free()`.

Your job is to describe in detail how to implement both options.

Hint for (B): You must not rely on garbage collection. "Approximating" `alloca()` means that you must ensure that allocated space does not live "long" after the stack frame of the function calling `alloca()`. "Not long" means, "not longer than the next call to `alloca()`."

Problem 3 — Reverse Engineering

[20 points]

Consider the following x86-64 assembly code:

```
MYSTERYFUNC:
        movl    %edx, %r8d
        movl    $0, %ecx
        cmpl    %edx, %ecx
        jge     .L7
.L5:
        movslq  %ecx,%rax
        movl    %r8d, %edx
        subl    %ecx, %edx
        movslq  %edx,%rdx
        movl    -4(%rdi,%rdx,4), %edx
        movl    %edx, (%rsi,%rax,4)
        incl    %ecx
        cmpl    %r8d, %ecx
        jl      .L5
.L7:
        ret
```

(a) This function takes two `int` arrays as its first two arguments (in addition to other argument(s)). Describe in English what this function does.

(b) Does this function *always* do what you said in (a)? If it does not, specify what invariants the arguments must satisfy to ensure that it is does?

(c) Suppose we change the function to take two `long int` arrays. How exactly must the original assembly code be changed?

(d) You plan to use this code on the next Mars Rover. Your hardware engineer tells you that cosmic rays will likely destroy your CPU's ability to read and write `int`-sized data. You can, however, be guaranteed that you can read and write `long int`-sized data.

How exactly must the original assembly code be changed to work on the extreme Martian landscape? (Hint: Meaning that you cannot use 32-bit reads and writes) You can assume that the length of the arrays are even.