Due: Thursday, Feb 10, 2011. 11:59pm (no extensions).

What to submit: For parts 1, 2, and 4: an ASCII file with answers, submitted under id ‘ex3’. For part 3: a C file with your code, submitted under id ‘ex3part3’. Use the submit.pl script in cs3214/bin/submit.pl to submit from the command line, or use the submit website.

This exercise is intended to reinforce the content of lectures 3-5. As before, you’re asked to use gcc 4.1.2 for an IA32 target. Specify the -O switch unless the question asks otherwise.

1. Switch Statements. Lecture 3/4 points out that gcc chooses a strategy for emitting switch statements based on the values that occur in the case arms. For a small number of cases, it’ll generate if-else chains. For a larger number of cases, it’ll switch to using a jump table.

(a) Assume the case values are 0, 1, 2, ... N for a chosen N. How large does N need to be for gcc to emit a jump table?

(b) Let’s assume you’ve chosen N to be large enough such that gcc uses a jump table for a switch statement with cases 0, 1, ... N. Now assume the case values are M, M+1, M+2, ... M+N for a given constant M. Does gcc’s code strategy change? Does your answer depend on how M is chosen?

(c) Now assume the case values are M, M+4, M+2*4, M+3*4, ... M+N*4. What strategy does gcc use in this case?

2. Variations on Passing Arguments. A discussed in lecture, there are different ways in which a function might prepare the arguments for the function(s) it calls, without violating the assumptions made by the ABI specification. It is useful to gain an intuition regarding these variations.

Consider the following variation of the call_swap() function shown in Lecture.

```c
void swap(int *xp, int *yp)
{
    int t0 = *xp;
    int t1 = *yp;
    *yp = t0;
    *xp = t1;
}

int zip1 = 15213;
int zip2 = 91125;

void call_swap(void)
{
    swap(&zip1, &zip2);
    swap(&zip1, &zip2);
}
```

Compile this code with the following flags (in this order):
For each variant, trace the value of the stack pointer relative to the position it had when the function was entered. Include only stack pointer-changing instructions. Do not include the effect of the ‘ret’ instruction at the end of the function. For instance, if the function

```c
void empty()
{
    char a[20];
}
```

were compiled to

```c
empty:
    pushl %ebp  # sp -= 4
    movl %esp, %ebp  # bp = sp
    subl $32, %esp  # sp -= 32
    leave  # sp = bp; sp += 4
    ret
```

then your answer would be $-4 -32 +36$.

The sum of all stack pointer movements must be zero so that when the function returns it finds the return address at the correct location. (Hint: ‘leave’ is short for ‘mov $ebp, $esp; pop $ebp’ It restores the stack pointer from the base pointer, then restores the caller’s base pointer from the stack.)

3. **Variable Arguments.** When programming in C, it is important to understand how functions such as `printf()` work, which take a variable number of arguments. Read the man pages for stdarg(3) to learn how these work, then implement the function `miniprintf` which acts like `printf()`, except that it understands only the `%d` format control. No other cases need to be handled. Implement your function portably. With your implementation, the following program should output ‘CS3214’:

```c
#include <stdio.h>
#include <stdarg.h>

/*
 * miniprintf works like printf, except that it only
 * understands the '%d' identifier.
 * No other cases need to be handled.
 */
void miniprintf(char *format, ...)
```
{    // Your code goes here. 
}

int main(int ac, char *av[]) 
{
    minipprintf("CS%d%d%d\n", 3, 2, 1, 4);
    return 0;
}

Your function should be no longer than 15 lines of code.
4. **Nested vs. Multi-Arrays.** Consider the following example code:

```c
#include <stdio.h>

int nestedarray[4][5] = {
    { 1, 5, 2, 0, 6},
    { 1, 5, 2, 1, 3},
    { 1, 5, 2, 1, 7},
    { 1, 5, 2, 2, 1},
};

int * multiarray[4] = {
    nestedarray[0],
    nestedarray[1],
    nestedarray[2],
    nestedarray[3]
};

int get_nestedarray_elem(int nestedarray[4][5], int row, int col)
{
    return nestedarray[row][col];
}

int main(int ac, char *av[])
{
    printf("%d\n", get_nestedarray_elem(multiarray, 0, 3));
}
```

The compiler reports the following error message when compiling the program:

```
array.c: In function ‘main’:  
array.c:24: warning: passing argument 1 of ‘get_nestedarray_elem’ from incompatible pointer type
```

Ignoring this warning and running the program, we obtain the following output:

```
$ ./array
134518300
```

The value 134518300 appears to be an address. What is it the address of? Give an equivalent C expression `& (....).`