CPSC 313
06W Term 2
Problem Set #2

Due: Sunday, January 28, 2006 at 11:59 PM (thirteen-hour grace period)
All of your solutions should be turned in on paper.

1. Variables declared in a C program can be stored in either registers or memory, and if in memory accessed in a variety of ways:

   - with positive offsets from the frame pointer register `%ebp` (as for arguments to a function)
   - with negative offsets from the frame pointer register `%ebp` (as for local variables in a function)
   - using absolute symbolic addressing (as for global or static variables)

   The following table details the storage decisions made by the compiler for 4 variables declared in a (hypothetical) C program.

Variable name	Variable storage class	Storage location
`i`	Local in register	`%ecx`
`a`	Argument in memory	`8(%ebp)`
`l`	Local in memory	`-4(%ebp)`
`g`	Global in memory	`g`

   Assume that there is an additional variable named `t` which is held in register `%esi`.

   Fill in the following table with the assembly language instructions necessary to perform the computation at the top of each column, with the `x` in the expression replaced by the C variable whose name is given in the first column of each row (so the top left blank entry is `t = &i` and the bottom right one is `*g = t`). You must ignore the fact that these assignments are inconsistent in their types; if `t = &x` is legal for some assignment of types to `t` and `x`, then `t = x` is certainly not legal for the same assignment of types. Assume that the types are modified as necessary to make each assignment statement legal — each will be legal for some assignment of types.

   Some of these assignments may be illegal for reasons other than typing. If you suspect one of them to be illegal, then explain why rather than giving me assembly instructions.

   If you need a temporary register, then use `%ebx`.

`x`	`t = &x`	`t = x`	`t = *x`	`x = t`	`*x = t`
`i`					
`a`					
`l`					
`g`					

2. Textbook problem 3.32

3. Textbook problem 3.34

4. Implement in IA32 assembly language a procedure that returns the maximum element of a list of integers. A prototype for your max procedure would be:

```
int max( int list[], int itemCount );
```

   Use the following file (generated by gcc -O2 -S on a skeleton procedure) as a starting point.

```
.file "foo.c"
.text
.p2align 4,,15
.globl max
.type max, @function
max:
pushl %ebp
movl %esp, %ebp

# use pushl to save any registers you use here

# body here

# use popl to restore saved registers here

movl %ebp, %esp
popl %ebp
ret
.size max, .-max
.ident "GCC: (GNU) 4.1.0 (SUSE Linux)"
.section .note.GNU-stack,"",@progbits
```

   Remember that the arguments list[] (which is a pointer to the base of the array) and itemCount are stored on the stack at 8(%ebp) and 12(%ebp), respectively. As a first step, save any registers you use to the stack where the comment indicates and restore them where the comment indicates. Now write your max procedure where the comment indicates. Be sure to comment your code.

   Write a test program in C and test your procedure thoroughly. Turn in the code and sample test output. Those of you who don't know C well might find the following C program helpful as a starting point. Note that it doesn't call max in an interesting or meaningful way, but your program should. In fact, it should probably call max in every interesting and meaningful way and check that the result is correct. Note that your test program need not be interactive;

the sample program is just in case you want to, but don't know how to, read integers from stdin in C.

```
#include <stdio.h>

extern void max (int list[], int itemCount);

int main()
{
    char buf[256];
    int i, m;
    int aList[100];
    int anItemCount;
    printf("enter an integer: ");
    fgets(buf, sizeof(buf), stdin);
    i = atoi(buf);
    printf("you typed %d\n",i);
    m = max(aList, anItemCount);
}
```