ECE 468
Problem Set 5: Control statements, functions and calling conventions

1. Give the three address code (including labels!) for the following piece of code.

```
for (i = 0; i < N; i = i + 1) {
  for (j = i; j < N; j = j + 1) {
    k = k + 1;
  }
}
```

Assume that you have a three-address instruction `BLT A B L`, which branches to label `L` if `A < B`, and another instruction `JMP L`, which is an unconditional jump to label `L`. Otherwise, use the same three address instructions defined in problem set 4.

Answer:

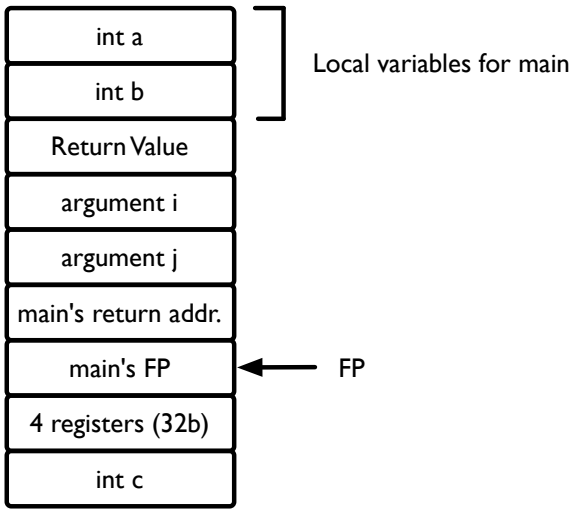
```
    ST(0) i           //i = 0

H0: LD(N) T1          //T1 = N
    LD(i) T2          //T2 = i
    BLT T2 T1 B0      //if i < N execute body B0
    JMP F0            //otherwise skip loop

B0: LD(i) T3          //T3 = i
    ST(T3) j          //j = T3

H1: LD(N) T4          //T4 = N
    LD(j) T5          //T5 = j
    BLT T5 T4 B1      //if i < N execute body B1
    JMP F1            //otherwise skip loop

B1: LD(k) T6          //k = k + 1
    ADD T6 1 T6
    ST(T6) k

I1: LD(j) T7          //j = j + 1
    ADD T7 1 T7
    ST(T7) j
    JMP H1            //go to top of loop
```

```
F1:
I0: LD(i) T8          //i = i + 1
    ADD T8 1 T8
    ST(T8) i
    JMP H0            //go to top of loop

F0: //end
```

2. Consider the following piece of code:

```
void main() {
  int i, j;
  i = foo(i, j);
}

int foo(int a, int b) {
  int c;
  ...
}
```

Assuming that, before executing `main`, the stack has nothing on it, show the stack *immediately after* calling `foo` (i.e., before `foo` returns). Assume there are 4 registers that need to be saved (not including the frame pointer and the stack pointer), and that we are using a callee-saves convention. On your stack, show each item on the stack, and give the size of each. Show what the frame pointer points to.
Answer: Everything is 4 bytes, except the register save area, which is 32.



3. Cam Piler, of problem set 4 fame, has another interesting idea. He thinks that if a program has no global variables, and all of its functions only take one argument, he can implement "pass by value-result" (*i.e.*, copy-in, copy-out) by treating all such arguments as pass-by-reference. Is he right? Why or why not? If he is right, why would this be a good optimization?

   Answer: Amazingly, Cam Piler is actually right this time. Copy-in, copy-out semantics are almost the same as pass-by-reference. The only times they will be different are: 1) if there is aliasing between arguments to a function (because within the function the arguments will look different, but they will be copied out to the same location) and 2) if a global variable is passed in as a function argument (because within the function the argument can change, but it doesn't affect the global variable until the function returns).

   In Cam's program, neither of these situations can happen, so pass-by-reference will behave exactly the same as copy-in, copy-out. His optimization is useful because pass-by-reference avoids any copying overhead.