Overview and Introduction

One topic is re-visited and a new topic is introduced in this lab. Lab 3 introduced the idea of using a stack frame to pass parameters to procedure calls, and you were told that this is the mechanism that is used in C for parameter passing. Actually register based passing is also used, but since the stack frame is memory based it allows for much larger parameter lists. Additionally, the relatively small number of general purpose registers available in the Intel Architecture makes register based parameter passing impractical.

C compilers, in addition to compiling to machine language, can also compile to assembly. In this lab you will use Borland's Turbo C to compile C language programs into assembly. The Borland product has an environment available from the DOS command line using the command "TC." You can also call the compiler directly from the command line by using TCC. To compile to assembly you use a command line parameter. The correct command line to compile the file srcfile.c to assembly is:

```
TCC -S srcfile.c
```

The output will then be srcfile.asm. You will need to assemble this file use PWB/MASTM. However, the file will NOT compile as-is because of symbols and macros and functions defined in the Turbo C environment that are unknown to PWB/MASTM. This means you cannot use library functions such as printf when using the procedure to produce an executable. ALL PROGRAMS COMPILED IN THIS MANNER WILL BE VERIFIED USING CODEVIEW: NO I/O IS NECESSARY.

While assembling files from compiler generated .ASM is a useful pedantic exercise, it is not a productive procedure for developing projects: a more powerful tool is to create programs in C or C++ that use in-line assembly. After completing the first part of this lab, you will see how C statements produce assembly language code. In-line assembly is a way to embed assembly statement into a C program. WARNING: in-line assembly is implementation and compiler dependent. The particulars will vary from platform to platform.

In Turbo C++ in-line assembly is accomplish with the key-word "asm." The program of Figure A

Figure A

```c
void main(void) {
    char* String = "Hello World\n$";

    asm {
        mov ah, 0x09
        mov dx, offset String
        int 0x21
    }
}
```

demonstrates a way to accomplish the familiar "Hello World" program using in-line assembly. You will note that this implementation is completely machine and platform dependent, and is far more limited in functionality than the printf library function. It is true that the asm executes much much faster than the
compiler generated code (when done correctly), however using C has so many other advantages that it is the overwhelming first choice. This is a key point in using in-line assembly: ONLY USE IT WHEN NECESSARY. You can be most productive by augmenting the power of a high level language with in-line assembly when required.

A convenient feature of the Turbo C++ compiler is the use of pseudo-variables. You can directly access the general purpose registers by using these pre-defined "variables." Note that you will not find this functionality in most compilers. The program of Figure A is re-written below in Figure B using these pseudo variables:

---

**Figure B**

```c
void main(void) {
    char* String = "Hello World\n$";
    _AX = 0x0900;
    _DX = (unsigned short) String;
    asm int 0x21;
}
```

---

One final warning: labels are interpreted by the compiler, not the assembler. Therefore any labels you wish to use in your in-line assembly must be defined in a block of C code. A label cannot be defined in a block of assembly.

---

**Incorrect (label undefined)**

```c
void main(void) {
    char* String = "Hello World\n$";
    _CX = 3;
    _AX = 0x0900;
    _DX = (unsigned short) String;

    asm {
        AGAIN:
        int 0x21;
        loop AGAIN
    }
}
```

---

**Correct**

```c
void main(void) {
    char* String = "Hello World\n$";
    _CX = 3;
    _AX = 0x0900;
    _DX = (unsigned short) String;

    AGAIN:
    asm {
        int 0x21;
        loop AGAIN
    }
}
```
Objectives

Items which you should understand by the time you complete this lab are:

- Compiling to Assembly language
- Using inline assembly statements in C
- Passing arguments to/from functions
- Accessing registers via pseudovariables
- Mixed language programming
- Memory Models
- Addressing modes commonly used (pointers, pass-by-value, pass-by-reference).

Pre-Lab

Given the C program in Figure 1:

```
#define SIZE 10

void main ( void) {
    int a[SIZE] = {2,6,4,8,10,31,89,67,54,37};
    int hold = 0, i = 0, j = 0;
    /*Look at the array here -- verify it's order*/
    /* sort the array
    for(j=1; j <= SIZE - 1; j++) {
        for( i = 0; i <= SIZE -2;i++) {
            if ( a[i] > a[i+1]) {
                hold = a[i];
                a[i] = a[i+1];
                a[i+1] = hold;
            }
        }
    }
```

1. Using an editor, enter this program into a file named srcfile.c and compile it to assembly language with the DOS command line:
   `TCC -S srcfile.c`
   Print the generated assembly program, srcfile.asm. Mark the various addressing modes used.
2. Recall from lab 4 the procedure for using a stack frame. You were told in lab 4 that a stack frame is used in C for passing parameters and for reserving space for local variables. To verify this, mark the section at the beginning of "main" that completes setting up the stack frame. Additionally, mark all places within main where space on the stack frame is used for addressing local variables. As a final step, change the C program of Figure 1 to the program of Figure 2 (the only change is to make a[] a global variable). Compile this program to assembly and note the differences in addressing used. Additionally, how many bytes were used for local variables on the stack frame in the program of Figure 1? How many bytes were used for local variables on the stack frame in the program of Figure 2? From what you know of how stack frames work: what do you think the macro N_SCOPY@ does in the compiled assembly?

3. Modify the program of Figure 1 to create two different C programs:
   a) Put the sort routine in a function in which the array is passed call by reference. (Look at your C book if this is confusing.
   b) Put the sort routine in a function in which the array is passed by value (this will require some trickery). You will have to put a struct wrapper around the array to get C to do this:

```
typedef struct tag_call_by_value_array {
    int a[SIZE];
} call_by_value_array;
```
Lab Procedure

1. Create an executable assembly language program for the program in Figure 1 using MASM/PWB. To do so you will need to compile the file to assembly and then fix any problems that arise when you try to assemble with MASM/PWB. There are two main sources of error: there are assembler directives inserted by TCC that are not understood by MASM and some PROCS are missing that result in the stack frame being only partially set up. To fix the program, comment out the offending compiler directives, and replace the PROC N_SCOPY@ with a PROC of your own design (Or you can eliminate the PROC call and replace it with the functionality in-line).

   **Do not** edit the program line-by-line changing the addressing mode every time a variable is used. You should only have to change a few lines of code and write a few new lines of code (or 1 PROC). If you don't fix the use of a stack frame, you will not receive credit for this portion of the lab.

2. Compile the programs from Pre-lab section 3 into two assembly files using TCC -S. Point out the addressing modes used. In particular, show where the data was passed to/from the functions and where the stack was used for these purposes. Which was more efficient for passing the array argument: call by value or call by reference? Then recompile both program using the large memory model (TCC -ml -S srcfile.c). Show where the addressing modes were changed and explain why.

3. Write, compile, and execute a C program of your own design that computes the sum of an array and outputs whether the sum is odd or even. You must use in-line assembly to complete this task. However, you do not need to compile to assembly as an intermediate step. Simply use the TC programming environment, and compile directly to executable. In addition:

   A. You cannot use any arithmetic operators ("+", "-", etc.), you must perform all these operations with in-line assembly (such as "asm add ax, var1" or "asm div al, dl", etc.)

   B. The program must contain at least one function.

Lab report

You will have to write a lab report for this lab summarizing the results of all parts of the lab.