Project 8:
Manual Translation of C code to Machine Code

In this project your assignment is to translate a provided C code file into equivalent low level machine code using the stack, operator, and procedure macros created in the previous project. You will be translating two sorting algorithms, the quicksort algorithm using the Lumuto partitioning scheme, and the insertion sort algorithm. A C++ implementation of these procedures is provided in the included `sort.cc` file. The following files are included in the initial distribution:

`macroAssembler.py`	This is an updated macroassembler with improved error checking and diagnostics. Use this version instead of the one provided with project 7.
`sort.cc`	An example C++ implementation of the project sort algoritms.

`Stack.h`
`Operators.h`
`Procedure.h`
	Provided macros to be used to implement stack operations and procedure call API's

`qsort.h`	Stub file where you will implement the PARTITION and QSORT functions
`isort.h`	Stub file where you will implement the ISORT function
`circfill.h`	Screen pattern filling code used by screen sorting demos
`qsortdemo.masm`	This demo uses the QSORT function defined in `qsort.h` to demonstrate the operation of the QSORT procedure visually by sorting screen memory.
`isortdemo.masm`	This demo uses the ISORT function defined in `isort.h` to demonstrate the operation of the ISORT procedure visually by sorting screen memory.
`partitionTest.tst`	Test file that will check the PARTITION function implementation that will be used with QSORT. After implementing PARTITION in `qsort.h` assemble `partitionTest.masm` and run the test using this script with CPUEmulator
`qsortTest.tst`	Test file that will check the QSORT function. After implementation of QSORT in `qsort.h`, assemble `qsortTest.masm` and run the test using this script with CPUEmulator
`isortTest.tst`	Test file that will check the ISORT function. After implementation of ISORT in `isort.h`, assemble `isortTest.masm` and run the test using this script with CPUEmulator

Important note on 16 bit comparison operator:

The sorting algorithms presented here use the comparison operators less-than or greater-than. The correct function of these algorithms requires that these operators satisfy the transitive property. That is if a<b and b<c, then this should imply that a<c. It is likely that the $lt and $gt operators you implemented in the previous project did not satisfy this property. If you implemented these operators just using twos complement subtraction followed by a sign check of the result, then the comparison operator could be invalid when overflow occurs in the subtraction step. For example, consider a=32767, b=32765, c=-2. Then in 16 bit two's complement a-b = 2 which is positive so lt(a,b) will be false, then b-c=32767 and so lt(b,c) is false. Therefore by the transitive property lt(a,c) should also be false, but in 16 bit two's complement a-c=**-32767** which will cause lt(a,c) to evaluate to **true** because there is not enough bits to represent the result with the correct sign. This can be solved by being a bit more sophisticated with the comparison. If the signs of the two inputs to the comparison operator are different then the subtraction is in danger of overflow. But in that case the comparison can be performed from the sign information alone. The $lt and $gt operators provided in `Operators.h` with this project do the comparison such that the transitive property of the operator is retained. Be sure to use the provided operators in your implementation of sort to ensure that the sort algorithms operate correctly for all input values.

The insertion sort is a $O(n^2)$ worse case sorting algorithm. Although its worse case performance is poor, it has several advantages that can be exploited in specific circumstances and therefore is still widely used in sorting. The first advantage is that it is an inplace sort which does not require any extra memory to perform (unlike merge sort). The second advantage is that it can be very quick when the input is nearly sorted (here we define nearly sorted as an element is at most k elements away from its sorted position). In this case the algorithm is just O(nk). This can be useful for hybridizing with a partial quick sort algorithm that only recurses until problem sizes are k or smaller. In that case, the partially sorted array can be quickly with an insertion sort. Combining these two algorithms provides a clever way to reduce the procedure call overhead for quick sort that occurs with small array sizes.

So how does an insertion sort algorithm work? It keeps a sorted list of the processed elements and then moves elements up to make room to insert the current element in the correct location. A pointer based version of this algorithm can be expressed in C code in the following code segment

```
// insertion sort
short isort (short *begin, short *end) {
  short *i = begin, *j, x ;
  while(i<end) {
    x = *i ;
    j = i-1 ;
    while(j>=begin && *j>x) {
      *(j+1) = *j ;
      j=j-1 ;
    }
    *(j+1) = x ;
    i=i+1 ;
  }
  return end-begin ;
}
```

The insertion sort code above has an interface similar to the C++ std::sort algorithm where the algorithm receives two pointers, one to the beginning of the list called `begin`, and one pointer to just past the last element of the array called `end`. The outer while loop, with iterator *i*, is keeping track of the current element to be inserted into the array (saved in x while space is made for its insertion). The inner loop (with iterator *j*) is moving items out of the way to make space for the current element to be inserted. Please note that the `&&` operator is doing more than a logical operation here. In C semantics once the logical operation can be determined, then the remaining elements are no longer executed. Thus the first test in the while loop protects the `*j` in the second part from executing with an invalid pointer. Your implementation should also implement this short circuit semantics. Finally, once the sort is completed, the sort returns the size of the array that was passed into the subroutine. Note that for the implementation this procedure has two arguments and three local variables. Your implementation should do the same.

The second part of this project is the implementation of the recursive quick sort algorithm. An important component of the quick sort algorithm is the partitioning step that partitions the array into two sub-arrays, one with elements less than the pivot, and the other with elements greater than the pivot. For this implementation of quick sort we use the simple Lumuto partitioning scheme. The partition function is passed in the begin and end pointers that define the sub-array to be partitioned, and returns a pointer that splits the array based on the pivot. The pivot is always selected from the last item in the array. This partitioning algorithm is expressed in C code as:

```
// Lomuto partitioning scheme
short *partition(short *begin, short *end) {
  short pivot = *(end-1) ;
  short *i=begin-1 ;
  for(short *j=begin;j<end;++j) {
    if(*j <= pivot) {
      i++ ;
      std::swap(*i,*j) ;
    }
  }
  return i ;
}
```

Note the swap function just transposes the contents of the *i* and *j* elements in the array. Note that in the implementation the swap operation will require temporarily storing a value which can be accomplished with the push operation. This function has two arguments and two local variables which should be reflected in your implementation. In your development process it is recommended that you develop and test this function first before starting on the quick sort algorithm.

Once the partition code is complete, then the quick sort algorithm can be expressed by a very elegant recursive algorithm. The C code for this algorithm is represented by

```
short qsort(short *begin, short *end) {
  short sz = end-begin ;
  short *split ;
  if(sz < 2)
    return sz ;
  split = partition(begin,end) ;
  qsort(begin,split) ;
  qsort(split,end) ;
  return sz;
}
```

The qsort algorithm describe above is a recursive divide an conquer algorithm that recursively splits the array until the size of the array is one element. The process of splitting the array will ensure that when all of the sub-problems are concatenated, the resulting array will be sorted. The algorithm requires one local variable to store the split location from the partition (the size could be recalculated on return). This will require the stack based procedure call facilities developed in Project 7 to effectively develop the reentrant procedure required to implement recursive calls to the qsort function.

Note that there are two additional visual tests that you can run to visualize the operation of the sorting algorithms. They run by first drawing a circle pattern on the screen and then run one of the sorting algorithms on the screen memory. These examples allow you to actually visualize the operation of the sorting algorithms. These programs are implemented in `qsortdemo.masm` and `isortdemo.masm` which will run the quick sort or insertion sort algorithm respectively.

The assignment: Convert the C implementations of quick sort and insertion sort as implemented in `sort.cc` into machine code using the stack and procedure macros. The sort subroutines should be implemented in the files `qsort.h` and `isort.h`. The quick sort implementation includes the functions `PARTITION` and `QSORT`, while the insertion sort implementation implements the function `ISORT`. Note that these procedures need to be implemented with the same name so that the test harnesses work. The test files are provided to test each part of these sorting algorithms. Once your sort procedures are working submit your implementations of `qsort.h` and `isort.h` to Canvas.