Project 7: Implementing Stack Operations and Procedure Calls

For this project you will be using the provided macroAssembler.py program that provides facilities for developing macros. The provided code includes test files that can be run with the CPUEmulator provided by nand2tetris to evaluate the correctness of your macro implementations. Once completed the three .h files listed below will be submitted to Canvas which will implement the stack-based programming environment that supports re-entrant procedure calls.

The macros will be placed in the following files:

1. `Stack.h`
   This file contains the definitions of stack operation macros `$pushD`, `$pushA`, `$popAD`, `$setPTR`, `$getPTR`, and the argument and local variable accessors `$getLocal`, `$setLocal`, `$getArgument`, `$setArgument`. The tests for these macros are provided in the files `StackBasicTest.tst` which tests the basic stack operations and `StackTest.tst` which includes tests of the variable accessor macros.
2. Operators.h
   This file contains the definition of the arithmetic and logical stack-based operators including `$add`, `$sub`, `$neg`, `$eq`, `$lt`, `$gt`, `$and`, `$or`, and `$not`. The file `OperatorTest.tst` provides tests for these operators. It is probably best to implement and test these after the `Stack.h` operations have been defined.
3. Procedure.h
   This file contains the definition of procedure and local variable memory management operators. These include `$procedureCall`, `$return`, `$pushFrame`, and `$popFrame`. The file `ProcedureBasicTest.tst` tests `$procedureCall` and `$return`, while the file `ProcedureTest.tst` does a complete test that invokes features of all the macros assigned in the project.

Stack Operator Macros

1. `$pushD`
   This macro will push the contents of the D register onto the stack. The stack is specified to build down in memory, so the outcome of this operation is that the stack pointer (SP) is decremented by 1. The value that was pushed on the stack is in the D register after this macro finishes.
2. `$pushA`
   This is a convenience macro. It should be equivalent to D=A followed by a `$pushD`
3. `$popAD`
   This macro will pop the value on the top of the stack into registers A and D. SP will be incremented by one in this process.
4. `$getPTR`
   This macro will pop an address off the stack and get the contents of memory located at that address. The contents of that memory address Is then pushed on the stack. This performs the operation of reading a dereferenced pointer. The D register will contain the value that was pushed on the stack.
5. `$setPTR`
   This macro will pop an address off the stack and then pop a value off the stack. It will then store the value popped off from the stack at the address popped from the stack. When completed, the D register will contain the value that was assigned to the pointer. The effect of this operation is like writing to a dereferenced pointer.

Variable Accessors

1. $setLocal id
   This macro will write D to the local variable numbered by the integer id where id is a value between 0 and nlocals-1. This macro essentially implements *(LCL-id)=D. Here LCL is the pointer to the current local variable space. The value that the variable was assigned to the local variable remains in D after the macro completes.
2. $getLocal id
   This macro will read the value of the local variable numbered by the integer id into the A and D registers. This macro essentially implements AD=*(LCL-id).
3. $setArgument id
   This works the same as `$setLocal` except operating with the ARG pointer instead of LCL
4. $getArgument id
   This works the same as `$getLocal` except operating with the ARG pointer instead of LCL

Stack-Based Operators

1. `$add`
   This implements x=pop(),y=pop();push(x+y).
2. `$sub`
   This implements x=pop(),y=pop();push(x-y).
3. `$neg`
   This implements x=pop();push(-x).
4. `$not`
   This implements x=pop();push(!x). (where ! is the bitwise complement)
5. `$eq`
   This implements x=pop(),y=pop(); if x==y push(-1) else push(0).
6. `$gt`
   This implements x=pop(),y=pop(); if(x>y push(-1) else push(0).
7. `$lt`
   This implements x=pop(),y=pop(); if(x<y push(-1) else push(0).
8. `$and`
   This implements x=pop(),y=pop(); push(x&y).
9. `$or`
   This implements x=pop(),y=pop(),push(x|y).
10. `$halt`
    This is a provided macro that just implements an infinite loop to halt further progress of the processor.

Procedure Call Support

1. `$procedureCall nargs procedure`

   This macro is used to call a procedure and manage the argument stack on return from the procedure. The macro takes two arguments, the first is the number of procedure arguments pushed on the stack, and the second is the starting address of the procedure program. This macro pushes the return address onto the stack, then unconditionally jumps to the procedure. Upon return from the procedure excess arguments that remain on the stack will be removed such that the first argument. This can be achieved by adding nargs-1 to SP.

2. `$return`

   This will return to the callee. This is accomplished by popping the return address off the stack and then performing an unconditional jump to the return address.

3. `$pushFrame nargs nlocal`

   This operation will be important for saving state of the callee and setting up the pointers for accessing the procedure arguments and local variables. The macro takes to arguments which are the number of procedure arguments and number of local variables to allocate. The `pushFrame` operation will first save the variables LCL,ARG,THIS, and THAT to the stack. Then it will setup the ARG and LCL pointers to point to the base address of the arguments and local variables respectively. The stack pointer will be adjusted to reserve room for the local variables to make sure that subsequent push operations will not overlap with the local variables. See the slides for the `macroAssembler` to get more detailed implementation details.

4. `$popFrame nargs nlocal`

   This macro will undo all the operations that were performed in the `pushFrame` step. It will deallocate the local variables and restore the callee pointers LCL, ARG, THIS, and THAT from the stack. After `popFrame` is finished the stack should be structured the same as it was prior to the `pushFrame` call. Details discussion of the implementation of this call can be found in the `macroAssembler` slides.