Problem M1.1: Self Modifying Code on the EDSACjr [1.5 Hours]

This problem gives us a flavor of EDSAC-style programming and its limitations. Please read Handout #1 (EDSACjr) and Lecture 2 before answering the following questions (You may find local labels in Handout #1 useful for writing self-modifying code.)

Problem M1.1.A — Writing Macros For Indirection

With only absolute addressing instructions provided by the EDSACjr, writing self-modifying code becomes unavoidable for almost all non-trivial applications. It would be a disaster, for both you and us, if you put everything in a single program. As a starting point, therefore, you are expected to write *macros* using the EDSACjr instructions given in Table H1-1 (in Handout #1) to *emulate* indirect addressing instructions described in Table M1.1-1. Using macros may increase the total number of instructions that need to be executed because certain instruction level optimizations cannot be fully exploited. However, the code size *on paper* can be reduced dramatically when macros are appropriately used. This makes programming and debugging much easier.

Please use following global variables in your macros.

```
_orig_accum:   CLEAR          ; temp. storage for accum
_store_op:     STORE 0        ; STORE template
_bge_op:       BGE 0          ; BGE template
_blt_op:       BLT 0          ; BLT template
_add_op:       ADD 0          ; ADD template
```

These global variables are located somewhere in main memory and can be accessed using their labels. The `_orig_accum` location will be used to temporarily store the accumulator's value. The other locations will be used as "templates" for generating instructions.

Opcode	**Description**
ADDind *n*	Accum ← Accum + M[M[*n*]]
STOREind *n*	M[M[n]] ← Accum
BGEind *n*	If Accum ≥ 0 then PC ← M[n]
BLTind n	If Accum < 0 then PC ← M[n]

Table M1.1-1: Indirection Instructions

Problem M1.1.B **Subroutine Calling Conventions**

A possible subroutine calling convention for the EDSACjr is to place the arguments right after the subroutine call and pass the return address in the accumulator. The subroutine can then get its arguments by offset to the return address.

Describe how you would implement this calling convention for the special case of one argument and one return value using the EDSACjr instruction set. What do you need to do to the subroutine for your convention to work? What do you have to do around the calling point? How is your result returned? You may assume that your subroutines are in set places in memory and that subroutines cannot call other subroutines. You are allowed to use the original EDSACjr instruction set shown in Handout #1 (Table H1-1), as well as the indirection instructions listed in Table M1.1-1.

To illustrate your implementation of this convention, write a program for the EDSACjr to iteratively compute `fib(n)`, where n is a non-negative integer. `fib(n)` returns the nth Fibonacci number (`fib(0)=0, fib(1)=1, fib(2)=1, fib(3)=2`…). Make `fib` a subroutine. (The C code is given below.) In few sentences, explain how could your convention be generalized for subroutines with an arbitrary number of arguments and return values?

The following program defines the iterative subroutine `fib` in C.

```
int fib(int n) {
  int i, x, y, z;
  x=0, y=1;
  if(n<2)
    return n;
  else{
    for(i=0; i<n-1; i++){
      z=x+y;
      x=y;
      y=z;
    }
    return z;
  }
}
```

Problem M1.1.C **Subroutine Calling Other Subroutines**

The following program defines a ***recursive*** version of the subroutine `fib` in C.

```
int fib_recursive (int n){
  if(n<2)
    return n;
  else{
    return(fib(n-1) + fib(n-2));
  }
}
```

In a few sentences, explain what happens if the subroutine calling convention you implemented in Problem M1.1.B is used for `fib_recursive`.