Problem M1.1: Self-Modifying Code (Spring 2015 Quiz 1, Part A)

In this problem we will use and extend the EDSACjr instruction set from Handout 1, shown in Table A-1.

Opcode	Description
ADD *n*	Accum ← Accum + M[*n*]
SUB *n*	Accum ← Accum - M[*n*]
LD *n*	Accum ← M[*n*]
ST *n*	M[*n*] ← Accum
CLEAR	Accum ← 0
OR *n*	Accum ← Accum | M[*n*]
AND *n*	Accum ← Accum & M[*n*]
SHIFTR *n*	Accum ← Accum shiftr *n*
SHIFTL *n*	Accum ← Accum shiftl *n*
BGE *n*	If Accum ≥ 0 then PC ← *n*
BLT *n*	If Accum < 0 then PC ← *n*
END	Halt machine

Table A-1. EDSACjr Instruction Set

Problem M1.1.A

Write a program that loops over an n-item array and replaces each item with its absolute value, as shown in the following pseudo-code:

```
for (i = 0; i < n; i++)
    A[i] = |A[i]|
```

Part of the program is already written for you, and to simplify your job you can assume the loop will be executed only once. The memory map on the next page shows the memory contents before the program starts. Array A is stored in memory in a contiguous manner, starting from location A. Memory locations N, I, and ONE hold the values of n, i, and 1, respectively. If you need to, you can use additional memory locations for your own variables. You should label each variable and define its initial value.

Memory:

	…
A	A[0]
	A[1]
	…
	A[n-1]
	…
ONE	1
N	n
I	0

Program:

```
loop:   LD      I
        SUB     N
        BGE     done




        LD      I
        ADD     ONE
        ST      I
        BGE     loop
done:   END
```

Problem M1.1.B

Tired of writing self-modifying code, Ben Bitdiddle decides to extend EDSACjr to support indirect addressing. However, because registers are expensive, Ben does not want to add an index register. Instead, he implements the indirect addressing instructions shown in Table A-2. To execute an indirect addressing instruction, the new architecture first reads the target address from memory and then loads/stores the data from/to memory.

Opcode	**Description**
ADDind *n*	Accum ← Accum + M[M[*n*]]
SUBind *n*	Accum ← Accum – M[M[*n*]]
LDind *n*	Accum ← M[M[*n*]]
STind *n*	M[M[*n*]]← Accum

Table A-2. Additional Indirect Addressing Instructions

Using the instructions in Table A-1 and Table A-2, rewrite the program from Question 1 without using self-modifying code. As before, you can use additional memory locations for your own variables. You should label each variable and define its initial value.

Memory:

	…
A	A[0]
	A[1]
	…
	A[n-1]
	…
ONE	1
N	n
I	0

Program:

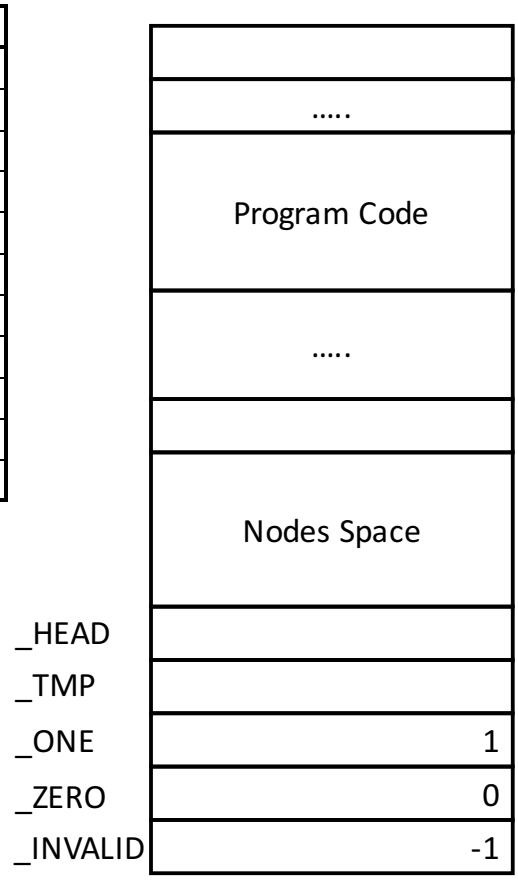
```
loop:   LD      I
        SUB     N
        BGE     done




        LD      I
        ADD     ONE
        ST      I
        BGE     loop
done:   END
```

Problem M1.2: Self-modifying Code (Spring 2017 Quiz 1, Part A)

In this question, you will implement linked-list operations using self-modifying code on an EDSACjr machine. The memory layout is shown in the figure on the right. You have access to the named memory locations as indicated. Linked-list nodes consist of two words: the first is an integer `value`, the second is an address pointing to the `next` node. `_HEAD` contains the address of the first node of the list (or `_INVALID` if it is empty). The `next` field of the last node is `_INVALID`. All valid addresses are positive. You may create new local and global labels as explained in the EDSACjr handout.

Table A-1 shows the EDSACjr instruction set.

Opcode	Description	Bit Representation
ADD *n*	Accum ← Accum + M[*n*]	00001 *n*
SUB *n*	Accum ← Accum - M[*n*]	10000 *n*
STORE *n*	M[n] ← Accum	00010 *n*
CLEAR	Accum ← 0	00011 00000000000
OR *n*	Accum ← Accum | M[*n*]	00000 *n*
AND *n*	Accum ← Accum & M[*n*]	00100 *n*
SHIFTR *n*	Accum ← Accum shiftr *n*	00101 *n*
SHIFTL *n*	Accum ← Accum shiftl *n*	00110 *n*
BGE *n*	If Accum ≥ 0 then PC ← *n*	00111 *n*
BLT *n*	If Accum < 0 then PC ← *n*	01000 *n*
END	Halt machine	01010 00000000000

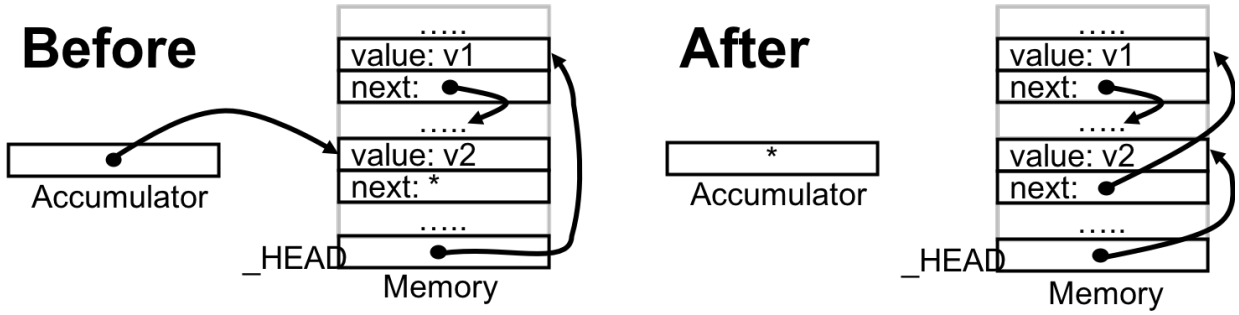


You may also use the following macros if required.

Macro	Description
STOREADR *n*	Replace the address field of location *n* with the contents of the accumulator
LOADADR *n*	Load the address field of location *n* into the accumulator

Problem M1.2.A

Write a macro for **LISTPUSH**, which pushes the node pointed to by the accumulator to the head of the list. **LISTPUSH** takes one argument, the memory address of the new node, which is available in the accumulator. As shown in the figure below, **LISTPUSH** stores the current `_HEAD` pointer in the new node's `next` field, and updates the `_HEAD` pointer to point to the new node. Implement the macro using the EDSACjr instruction set and macros provided above. Do not refer to "`value`" or "`next`"; they are for illustration only. You need not worry about memory allocation; the new node's address is provided in the accumulator.




```
.macro LISTPUSH
     STORE _TMP      ;; store accumulator (address of the new node)
```

```
.end
```

Problem M1.2.B

Write a macro for **LISTPOP**, which removes the node at the head of the list and stores its address in the accumulator, or stores `_INVALID` (-1) in the accumulator if the list is empty. Implement the macro using the EDSACjr instruction set and macros provided above.

```
.macro LISTPOP
     CLEAR           ;; accumulator is not an input









.end
```

Problem M1.2.C

Assume there exists a macro called **FREE** that takes an address as input in the accumulator and deallocates it (just like `free(void* ptr)` in C). Write a macro for **LISTCLEAR**, which uses the **FREE** macro and your **LISTPOP** macro to remove and deallocate all nodes in the list. Assume all valid node addresses are positive, or else a pointer is `_INVALID` (-1). Implement the macro using the EDSACjr instruction set and macros provided above.

```
.macro LISTCLEAR




.end
```

Problem M1.3: Self Modifying Code on the EDSACjr

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

Problem M1.3.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 M1.3.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 M1.3.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 M1.3.B is used for `fib_recursive`.