

Problem M1.1.B

Memory:

	...
A	A[0]
	A[1]
	...
	A[n-1]
	...
ONE	1
N	n
I	0
IDX	A

Program:

loop:	LD	I
	SUB	N
	BGE	done
	LDind	IDX
	BGE	cont
	CLEAR	
	SUBind	IDX
	STind	IDX
cont:	LD	IDX
	ADD	ONE
	ST	IDX
	LD	I
	ADD	ONE
	ST	I
	BGE	loop
done:	END	

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

Problem M1.2.A

```
.macro LISTPUSH
    STORE _TMP          ;; store accumulator (address of the new node)
    ADD _ONE            ;; accum <- address of the new node's next field
    STOREADR _STN      ;; address field of location _STN has the address
                        ;; of the new node's next field

    CLEAR
    ADD _HEAD          ;; accum <- M[_HEAD], current head pointer
_STN: STORE 0         ;; 0 will be replaced with the node's next field
                        ;; address. M[_TMP + 1] <- accum

    CLEAR
    ADD _TMP           ;; retrieve address of new node in accumulator
    STORE _HEAD       ;; M[_HEAD] <- accum; Update the head pointer
                        ;; to 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
    ADD _HEAD            ;; accum <- address of head node
    BLT _DONE           ;; if _HEAD < 0 (-1, ie *_INVALID), then return
    STORE _TMP          ;; save old value of head
    ADD _ONE            ;; accum <- address of head node's next field
    STOREADR _ADDN      ;; replace address field of _ADDN
                        ;; with address of head node's next field

    CLEAR
    _ADDN: ADD 0         ;; 0 will be replaced with the address of head
                        ;; node's next field. accum <- addr of 2nd node
    STORE _HEAD         ;; update head with the list's second node
    CLEAR
    ADD _TMP            ;; accum <- former head node pointer
    _DONE:

```

.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
_LOOP:    LISTPOP          ; accum <- address of removed node
          BLT _DONE       ; exit if an _INVALID node pointer is found
          FREE            ; de-allocates the removed node
          CLEAR
          BGE _LOOP
_DONE:
```

```
.end
```

Problem M1.3: Self Modifying Code on the EDSACjr

Problem M1.3.A

Writing Macros For Indirection

One way to implement `ADDind n` is as follows:

```
.macro ADDind(n)
    STORE    orig_accum ; Save original accum
    CLEAR    ; accum <- 0
    ADD      n          ; accum <- M[n]
    ADD      _add_op    ; accum <- ADD M[n]
    STORE    _L1       ; M[_L1] <- ADD M[n]
    CLEAR    ; accum <- 0
_L1: CLEAR    ; This will be replaced by
              ; ADD M[n] and will have
              ; the effect: accum <- M[M[n]]
    ADD      _orig_accum ; accum <- M[M[n]] + original accum
.end macro
```

The first thing we do is save the original accumulator value. This is necessary since the instructions we are going to use within the macro are going to destroy the value in the accumulator. Next, we load the contents of `M[n]` into the accumulator. We assume that `M[n]` is a legal address and fits in 11 bits.

After getting the value of `M[n]` into the accumulator, we add it to the `ADD` template at `_add_op`. Since the template has 0 for its operand, the resulting number will have the `ADD` opcode with the value of `M[n]` in the operand field, and thus will be equivalently an `ADD M[n]`. By storing the contents of the accumulator into the address `_L1`, we replace the `CLEAR` with what is equivalently an `ADD M[n]` instruction. Then we clear the accumulator so that when the instruction at `_L1` is executed, `accum` will get `M[M[n]]`. Finally, we add the original accumulator value to get the desired result, `M[M[n]]` plus the original content of the accumulator.

`STOREind n` can be implemented in a very similar manner.

```
.macro STOREind(n)
    STORE    _orig_accum ; Save original accum
    CLEAR    ; accum <- 0
    ADD      n          ; accum <- M[n]
    ADD      _store_op  ; accum <- STORE M[n]
    STORE    _L1       ; M[_L1] <- STORE M[n]
    CLEAR    ; accum <- 0
    ADD      _orig_accum ; accum <- original accum
_L1: CLEAR    ; This will be replaced by
              ; STORE M[n], and will have the
              ; effect: M[M[n]]<- orig. accum
.end macro
```

After getting the value of `M[n]` into the accumulator, we add it to the `STORE` template at `_store_op`. Since the template has 0 for its operand, the resulting number will have the `STORE` opcode with the value of `M[n]` in the operand field, and thus will be equivalently a `STORE M[n]` instruction. As before, we store this into `_L1` and then restore the accumulator value to its original value. When the PC reaches `_L1`, it then stores the original value of the accumulator into `M[M[n]]`.

`BGEind` and `BLTind` are very similar to `STOREind`. `BGEind` is shown below. `BLTind` is the same except that we use `_blt_op` instead of `_bge_op`.

```
.macro BGEind(n)
    STORE    _orig_accum ; Save original accum
    CLEAR    ; accum <- 0
    ADD      n          ; accum <- M[n]
    ADD      _bge_op    ; accum <- BGE M[n]
    STORE    _L1       ; M[_L1] <- BGE M[n]
    CLEAR    ; accum <- 0
    ADD      _orig_accum ; accum <- original accum
_L1: CLEAR    ; This is replaced by BGE M[n]
.end macro
```

Problem M1.3.B

Subroutine Calling Conventions

We implement the following contract between the caller and the callee:

1. The caller places the argument in the address slot between the function-calling jump instruction and the return address. Just before jumping to the subroutine, the caller loads the return address into the accumulator.
2. In the beginning of a subroutine, the callee receives the return address in the accumulator. The argument can be accessed by reading the memory location preceding the return address. The code below shows pass-by-value as we create a local copy of the argument. Since the subroutine receives the address of the argument, it's easy to eliminate the dereferencing and deal only with the address in a pass-by-reference manner.
3. When the computation is done, the callee puts the return value in the accumulator and then jumps to the return address.

A call looks like

```

                .....          ; preceding code sequence
clear
add      _THREE    ; accum <- 3
bge     _here     ; skip over pointer
_hereptr .fill    _here     ; hereptr = &here
_here   add      _hereptr  ; accum <- here+3 = return addr
        bge     _sub     ; jump to subroutine
                ; The following address location is
                ; reserved for argument passing and
                ; should never be executed as code:
_argument .fill 6      ; argument slot
                .....          ; rest of program
```

(note that without an explicit program counter, a little work is required to establish the return address).

The subroutine begins:

```
_sub   store    _return    ; save the return address
        sub     _ONE      ; accum <- &argument = return address-1
        store   _arg      ; M[_arg] <- &argument = return address-1
        clear
        ADDind  _arg      ; accum <- *(&arg0)
        store   _arg      ; M[_arg] <- arg
```

And ends (with the return value in the accumulator):

```
BGEind    _return
```

The subroutine uses some local storage:

```
_arg      clear          ; local copy of argument
_return   clear          ; reserved for return address
```

We need the following global constants:

```
_ONE      or             1          ; recall that OR's opcode is 00000
_THREE    or             3          ; so positive constants are easy to form
```

The following program uses this convention to compute fib(n) as specified in the problem set. It uses the indirection macros, templates, and storage from part M1.3.A.

```
;; The Caller Code Section
;;      .....          ; preceding code sequence
_caller  clear
         add          _THREE      ; accum <- 3
         bge          _here
_hereptr  .fill       _here
_here     add          _hereptr    ; accum <- here+3 = return addr
         bge          _fib        ; jump to subroutine

;; The following address location is reserved for
;; argument passing and should never be executed as code
arg0     .fill       4          ; arg 0 slot. N=4 in this example

_rtpnt   end

;; The fib Subroutine Code Section

; function call prelude
_fib     store        _return     ; save the return address
         sub          _ONE
         store        _n         ; M[_n] <- &arg0 = return address-1
         clear
         ADDind       _n         ; accum <- *(&arg0)
         store        _n         ; M[_n] <- arg0

; fib body
         clear
         store        _x         ; x=0
         add          _ONE
         store        _y         ; y=1

         clear              ; if(n<2)
         add          _n
         sub          _TWO
         blt         _retn

         clear
         store        _i         ; for (i = 0;

_forloop clear          ; i < n-1;
         add          _n
         sub          _ONE
         sub          _i
         sub          _ONE
         blt         _done
_compute clear
```



```

        add      _x
        add      _y
        store   _z      ; z = x+y
        clear
        add      _y
        store   _x      ; x = y
        clear
        add      _z
        store   _y      ; y = z

_next   clear
        add      _i      ; i++)
        add      _ONE
        store   _i
        bge     _forloop

_retn   clear
        add      _n
        BGEind  _return  ; return n

_done   clear
        add      _z
        BGEind  _return  ; return z

;; Global constants (remember that OR's opcode is 00000)

_ONE    or 1
_TWO    or 2
_THREE  or 3
_FOUR   or 4
```

These memory locations are private to the subroutine

```
_return  clear      ; return address
_n       clear      ; n
_x       clear
_y       clear
_z       clear
_i       clear      ; index
_result  clear      ; fib
```

Now we can see how powerful this indirection addressing mode is! It makes programming much simpler.

The 1 argument-1 result convention could be extended to variable number of arguments and results by

1. Leaving as many argument slots in the caller code between the subroutine call instruction and the return address. This works as long as both the caller and callee agree on how many arguments are being passed.
2. Multiple results can be returned as a pointer to a vector (or a list) of the results. This implies an indirection, and so, yet another chance for self-modifying code.

Problem M1.3.C

Subroutine Calling Other Subroutines

The subroutine calling convention implemented in Problem M1.3.B stores the return address in a fixed memory location (`_return`). When `fib_recursive` is first called, the return address is stored there. However, this original return address will be overwritten when `fib_recursive` makes its first recursive call. Therefore, your program can never return to the original caller!