6.823 Computer Architecture	DUE: 2/27/2002, EXTENSION: 3/4/2002
Timothy Danford, Killian Pohl, Bryan Russell	
Problem Set 1	*Instructors: Krste Asanovic, Srinivas Devadas*

Problem 5

(a)

We will assume that the following data is global:

```
opmask:    ADD 2047
stremsk:   STORE 0
bgemask:   BGE 0
bltmask:   BLT 0
accdata:   NOP
```

The accdata memory location is a global storage point for saving the value of the accumulator. We can use it, assuming that the macro substitution is one-pass (that is, macros cannot be called from macros). If this is an incorrect assumption, we can substitute the following code immediately before the endlbl label in ADDind and STOREind:

```
          BGE endlbl
          BLT endlbl
accdata:  NOP
```

In BGEind and BLTind, we don't need the branch instructions, so we can add just the following code before the endlbl label.

```
accdata: NOP
```

Here is the code for the macros:

```
.macro ADDind(n)
          STORE accdata
          CLEAR
          ADD n
          AND opmask
          STORE addinst
          CLEAR
          ADD accdata
addinst:  NOP
endlbl:
.end
```

```
.macro STOREind(n)
          STORE accdata
          CLEAR
          ADD n
          AND opmask
          ADD stremsk
          STORE strinst
          CLEAR
          ADD accdata
strinst:  NOP
endlbl:
.end

.macro BGEind(n)
          STORE accdata
          CLEAR
          ADD n
          AND opmask
          ADD bgemask
          STORE bgeinst
          CLEAR
          ADD accdata
bgeinst:  NOP
endlbl:
.end

.macro BLTind(n)
          STORE accdata
          CLEAR
          ADD n
          AND opmask
          ADD bltmask
          STORE bltinst
          CLEAR
          ADD accdata
bltinst:  NOP
endlbl:
.end
```

(b)

Description of the calling convention:

1. The arguments to the function (2 in this special case) are loaded, in reverse order, after the instruction that branches to the subroutine.

2. The return address is passed in the accumulator, and is assumed to be the word following the last word of the arguments in memory (i.e., after the word containing the first argument's data, since the args are listed in reverse order).

3. When the subroutine returns, the return value is placed in the accumulator and control is returned to the instruction at the original return address.

We will also define the following macros to help with the subsequent subroutine code. Again, we assume the global data from 5A. Furthermore, we assume the following additional value is defined:

```
submask:   SUB 0
```

Here is the code for the macros:

```
;; LOAD(n) == ACC <-- M[n]
.macro LOAD(n)
     CLEAR
     ADD N
.end

;; LOADind(n) == ACC <-- M[M[n]]
.macro LOADind(n)
          CLEAR
          ADD n
          AND opmask
          STORE loadinst
loadinst: NOP
endlbl:
.end

;; SUBind(n) == ACC <-- (ACC) - M[M[n]]
.macro SUBind(n)
          STORE accdata
          CLEAR
          ADD n
          AND opmask
          ADD submask
          STORE subinst
subinst:  NOP
endlbl:
.end

;; ADDoffset(m, n) == ACC <-- (ACC) + M[M[m] + M[n]]
.macro ADDoffset(m, n)
```

```
          STORE accdata
          CLEAR
          ADD m
          ADD n
          AND opmask
          STORE addinst
          CLEAR
          ADD accdata
addinst:  NOP
endlbl:
.end

;; SUBoffset(m, n) == ACC <-- (ACC) - M[M[m] + M[n]]
.macro SUBoffset(m, n)
          STORE accdata
          CLEAR
          ADD m
          ADD n
          AND opmask
          ADD submask
          STORE subinst
          CLEAR
          ADD accdata
subinst:  NOP
endlbl:
.end

;; LOADoffset(m, n) == ACC <-- M[M[m] + M[n]]
.macro LOADoffset(m, n)
          CLEAR
          ADD m
          ADD n
          AND opmask
          STORE loadinst
          CLEAR
loadinst: NOP
endlbl:
.end

;; STOREoffset(m, n) == M[M[m] + M[n]] <-- (ACC)
.macro STOREoffset(m, n)
          STORE accdata
          CLEAR
          ADD m
          ADD n
```

```
          AND opmask
          ADD stremsk
          STORE streinst
          CLEAR
          ADD accdata
streinst: NOP
endlbl:
.end
```

The code for the minimum subroutine:

```
minimum:      STORE ret_addr

              ;; min = array[0]
              LOADoffset ret_addr, array
              STORE temp
              LOADind temp
              STORE min

              ;; i = 1
              LOAD constant_1
              STORE i

              ;; loop constraint: ACC <-- i, when entering the loop
loop:         SUBoffset ret_addr, n  ;; loop if i < n
              BGE return

              ;; acc = array[i]
              LOADoffset ret_addr, array
              ADD i
              STORE temp
              LOADind temp

              ;; if(array[i] < min) ...
              SUB min
              BGE increment

              ;; min = array[i]
              ADD min
              STORE min

increment:    LOAD i
              ADD constant_1
              STORE i          ;; i++
              BGE loop
```

```
                ;; return the value of min in ACC
return:         LOAD min
                JUMPind ret_addr

n:              1 11111 11111 11111
array:          1 11111 11111 11110
min:            NOP
i:              NOP
ret_addr:       NOP
temp:           NOP
```

(c) Because we only have one spot to hold the return address, this calling convention allows only recursive function calls only if the return value of the recursive subroutine is itself immediately returned without further processing; in other words, if the deepest element of the recursive series of functions can return immediately to the return address of the first caller without changing the semantics of the recursion.

This means that only tail-recurive subroutines are allowed: lexically (as opposed to semantically) recursive functions. (It would probably also be possible for some subroutines to implement their own return address stack in "local" memory – in that case, if the subroutines were carefully designed not to touch previous elements of the stack and to keep a de-facto stack pointer as recursion progressed, a subroutine could recurse a finite number of times with this calling convention. But no infinite recursion is possible with this convention unless the function is tail-recursive, since the return address of the original caller would eventually be clobbered).

C-code for a tail-recursive minimum:

```
int minimum(int n, int *array) {
    return iterative_minimum(array[0], 1, n, array);
}

int iterative_minimum(int running_min, int i, int n, int *array) {
    if(i >= n) return running_min;
    if(array[i] < running_min) {
        return iterative_minimum(array[i], i+1, n, array);
    } else {
        return iterative_minimum(running_min, i+1, n, array);
    }
}
```

This is a lexically recursive function that, if translated into EDSACjr code in a straight-forward way, would run correctly with this calling convention (assuming we can handle more than 2 args).

The "minimum" function from 5.D would not work correctly with this calling convention.

(d)

Main points of the calling convention:

1. The original subroutine caller is presumed to have access to a block of memory somewhere to be used as a stack. Any time a subroutine is called, it is the caller's responsibility to set up a new activation record within that stack space. If the subroutine is recursive, it will assume it can place the next activation record immediately after its own activation record.

2. When the subroutine is called the accumulator holds the address of the top (lowest address) of the stack.

3. The following elements are assumed to have space in the stack, starting from the lowest address to the highest:

   (a) Offset 1: Return value

   (b) Offset 2: Return address

   (c) Offset 3: Address of Previous Activation Record

   (d) Offset 4, 5: Arguments

   (e) Offset 6...: Memory for local variables

4. When a subroutine returns, it has filled in its return value at the word at the beginning of its own activation record. It holds the address of the previous activation record in the accumulator (the contents of the 3rd word in its activation record) and returns to the given return address (the contents of the 2nd word in its activation record).

We'll use, again, the macros defined already. Here is the code for the minimum subroutine.

```
minimum:        STORE stack

                ;; min = array[0]
                LOADoffset stack, array
                STORE temp
                LOADind temp
                STOREoffset stack, min

                ;; if(n <= 1) goto return
                LOADoffset stack, n
                SUB constant_2
                BLT return_label

                ;; start setting up the next stack frame
                ;; first, let temp == addr of next frame
                LOAD stack
                ADDoffset stack, next_retval
```

```
              STORE temp

              ;; load all the values into next frame
              LOAD stack
              STOREoffset temp, prev_stack
              LOAD continue_addr
              STOREoffset temp, ret_addr
              LOADoffset stack, n
              SUB constant_1
              STOREoffset temp, n
              LOADoffset stack, array
              STOREoffset temp, array

              ;; put the address of next frame into ACC
              LOAD temp
              JUMP minimum  ;; jump to subroutine

continue:     STORE stack   ;; save the stack value on return
              LOADoffset stack, next_retval
              STOREoffset stack, min

              ;; acc = array[n - 1]
              LOADoffset stack, array
              ADDoffset stack, n
              SUB constant_1
              STORE temp
              LOADind temp
              SUBoffset stack, min
              BGE return_label

              ;; min = array[n - 1]
              LOADind temp
              STOREoffset stack, min

              ;; on return, load the correct return value
              ;; into the top of the stack, grab the address
              ;; of the previous stack into ACC, and then return
              ;; to the given return address
return_label: LOADoffset stack, min
              STOREoffset stack, retval
              LOADoffset stack, ret_addr
              STORE temp

              LOADoffset stack, prev_stack
              JUMPind temp
```

```
temp:          NOP
stack:         NOP
constant_1:    0 00000 00000 00001
constant_2:    0 00000 00000 00010

retval:        0 00000 00000 00000
ret_addr:      0 00000 00000 00001
prev_stack:    0 00000 00000 00010
n:             0 00000 00000 00011
array:         0 00000 00000 00100
min:           0 00000 00000 00101
next_retval:   0 00000 00000 00110
```