

PeANUt Macros – Beware!

- beware of the following:

```
Get('0')
Set2(n, sub, ch, #'0')
```

- which is expanded to:

```
trap    #2
store  '0'    ; run-time error?
load   sub    ; assembly error
ch     #'0'   ; assembly error
store  n
```

- good use of macros can 'abstract' some low level details and can clarify the program's structure
- bad use of macros can obscure what is going on and be the origin of many errors

Stack Addressing and Manipulation

- requires stack addressing mode (!): AOP = opspec + SP
- normally, opspec ≤ 0

- recall example:

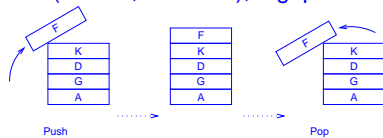
259	:
260	15
261	-1
SP → 262	40
	:

```
load    !0      ; AOP = 262  OP = 40
mul     !-2     ; AOP = 260  OP = 15
store   !-3     ; AOP = 259  OP = 600
```

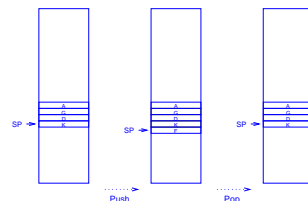
- the stack can be manipulated by software both implicitly:
 - via **call** and **ret** (SP is inc/decremented automatically)
 and explicitly, using instructions like:
 - **incsp #1,**
 - **store !-1**

The Stack and Function Calls

- a fundamental programming concept!!
 - hence hardware support is needed (e.g. SP register, stack addressing mode)
- uses a (reserved) part of (normal) memory called the **stack**
 - a stack is accessed **LIFO** (Last In, First Out), e.g. pile of books



- recall the **stack** can be efficiently implemented as the memory pointed to by SP:



- enables return of control to caller, as well as to pass parameters and return values

Macro Examples for the Stack: stackmacro.asm

- Push item onto stack:

```
macro Push (e)                ; push value of e onto stack
                                ; (use before call proc. with value param)
                                ; (AC = <value referred by e>)
    load    e                    ; (SP = SP + 1; done last in case e=!num)
    incsp   #1                   ; (Memory[SP] = AC)
    store   !0
endmacro
```

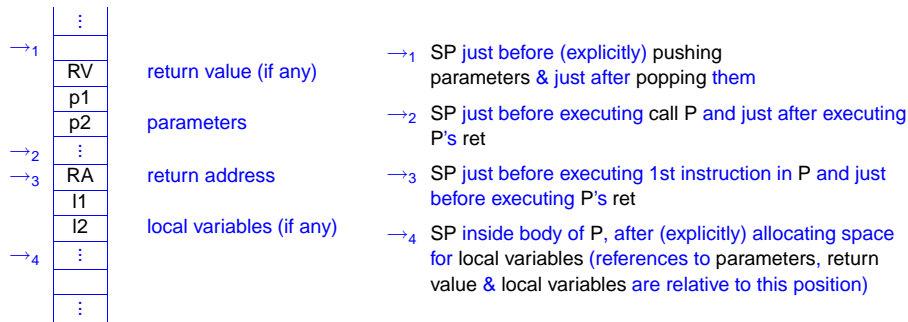
- Pop item from stack:

```
macro Pop (n)                 ; pop n elements of the stack
                                ; (use at end of call)
                                ; (SP = SP - n)
    incsp   #-n
endmacro
```

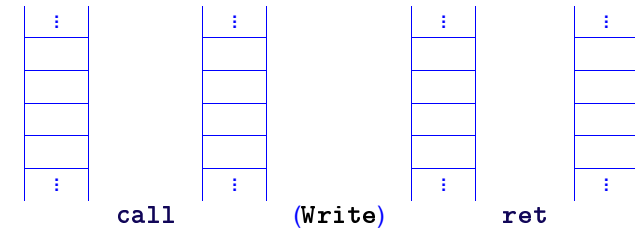
Procedure Calls

- the procedure call convention determines the order of contents of the stack frame
- example C function declaration:

```
int P(int p1, int p2, ...) {
    int l1, l2, ...;
    ...
}
```



Procedure call – Example without Local Variables: Stack



- for local variables, function must allocate/deallocate upon entry/exit *(later...)*

Procedure Call – Example without Local Variables

- function definition (in InOut.ass; `Write(c)` is equivalent to `printf("%c", c)`)

```
Write:      ch      = -1      ; void Write(char ch) {
           ;
           load    !ch    ; printf("%c",ch); /* AC=Mem[SP-1] */
           trap   #3      ; /* write AC to stdout */
           ret     ; }      /* PC=Mem[SP]; SP=SP-1 */
```

- call from procedure-example1.ass:

```
           ; Write('a');
load     #'a'    ; /* Push('a') */ /* AC='a' */
incsp   #1      ; /* SP=SP+1 */
store   !0      ; /* Mem[SP]=AC */
call    Write   ; /* SP=SP+1;
                 ; Mem[SP]=PC;
                 ; PC=Write */
incsp   #-1     ; /* Pop(1) */ /* SP=SP-1 */
```

- remember: PC is incremented at the *beginning* of each instruction cycle