

2-D Arrays, Macros and Procedures in PeANUt Assembler

- ref: [PeANUt Spec, sect 4]
- two-dimensional arrays
- macro definitions and usage
- the stack revisited
- procedures:
 - arguments
 - without local variables

PeANUt Macros

- important (yet simple) concept, widely used in the C language
- neither instructions nor procedures! Essentially, just a 'shorthand' or 'placeholder'
- macros are expanded by the assembler (as in C), not translated
 - i.e. exist only in the assembly language level
- definitions must be inserted at the top of a program
- can be good programming style, especially if they correspond to meaningful (high level language) operations
- are best for 'straight-line' code (don't use macros with branches etc.)

PeANUt Macro Example: macro.ass

- definitions:

```
macro Get (x)                ; read next char into x
    trap    #2                ; (read next char into AC)
    store   x                  ; (Memory[x] = AC)
endmacro

macro Set2 (x, e1, op, e2)    ; perform x = e1 + e2
    load    e1                 ; (AC = <value of e1>)
    op      e2                 ; (AC = AC op <value of e2>)
    store   x                  ; (Memory[x] = AC)
endmacro
```

- `Get(x)` and `Set2(x,e1,op,e2)` can be called as follows:

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

- these are expanded to:

```
trap    #2
store   ch
load    ch
sub     #'0'
store   dn
```

PeANUt Macros – Beware!

- beware of the following:

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

- which is expanded to:

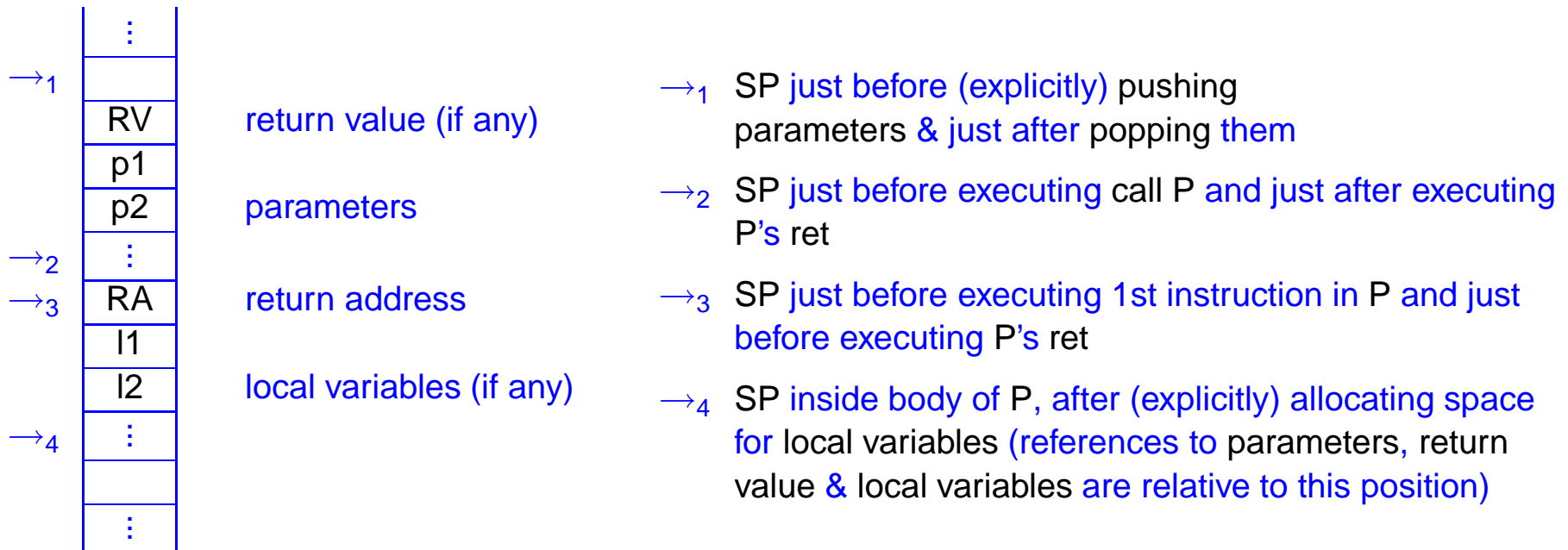
```
trap      #2  
store    ' 0 '      ; run-time error?  
load     sub       ; assembly error  
ch       # ' 0 '   ; assembly error  
store    n
```

- 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

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

- function definition (in InOut.asm; `Write(c)` 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.asm:

```
           ; 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

Procedure call – Example without Local Variables: Stack

