

## Procedure calls and address parameters in PeANUt

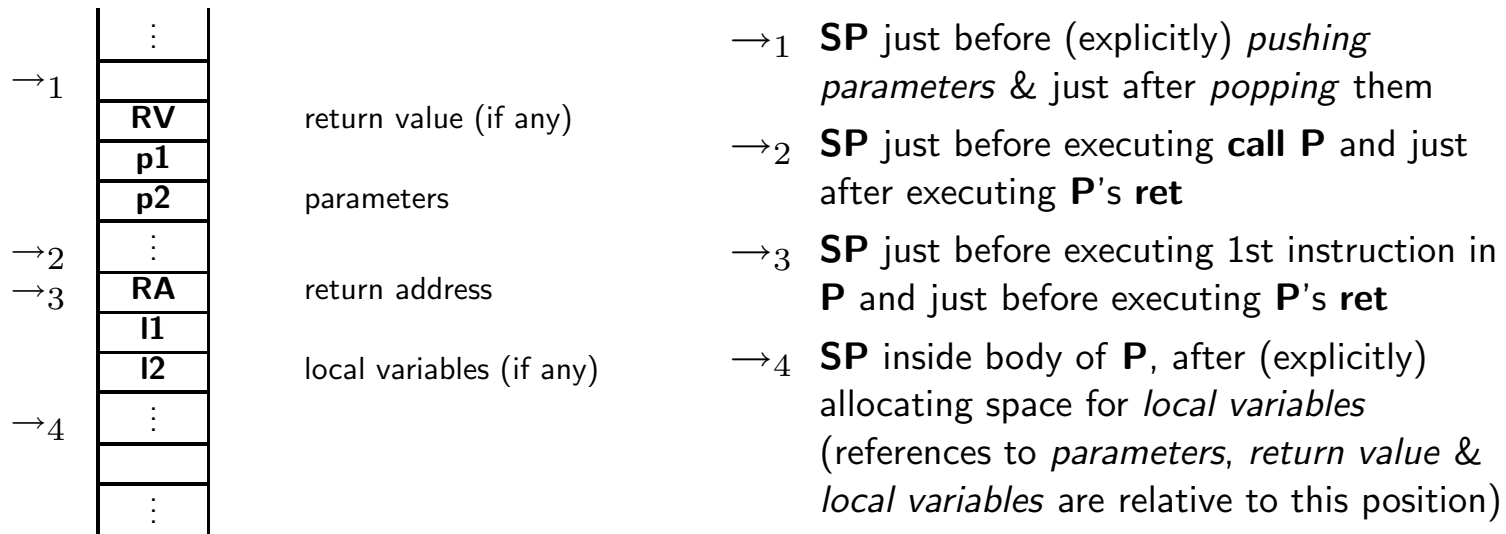
- Procedure calls
  - With local variables
  - With return values (non-void functions)
- Address parameters
  - Ability to modify data
  - Indirect memory reference via pointers
- Assembly coding style
- Reference:
  - Specification of the PeANUt computer (Section 4)

# Procedure calls (1)

- The *procedure call convention* is given by the **stack frame**

Example C function declaration:

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



## Procedure calls (2)

- *Call convention* needed so that *caller* and procedure can *agree* (on *where* to exchange data)
- Role of the *stack* for **RV**, *parameters* & **RA**
- *Symmetric* use of **SP**
- Common errors:
  - Using the wrong *mode* when *pushing* parameters
  - Forgetting to *allocate* **RV** *slot* for non-**void** function
  - Having **SP** 1 beyond the last *parameter* before executing **call**
  - Not *popping* same number of *slots* after the **call** instruction
- For procedures, machine needs **SP**, **!**, **call** and **ret**

## Procedure call – Example with two local variables (1)

```

                                ; void WriteInt(
                                ;   int x,
                                ;   unsigned int n) {
                                ;
                                ;           /* what is at !-2? */
                                ;           int NSp;           /* # of ' 's to print */
                                ;           unsigned int aX;    /* = |x| */
                                ;           Nlocs = 2;
WriteInt:
                                ;
                                ;           incsp   #Nlocs   ;           /* SP=SP+2 */
                                ;           load    !n       ;           Nsp = n - 1;   /* AC=Mem[SP-3] */
                                ;           sub     #1       ;           /* AC=AC-1 */
                                ;           store   !Nsp     ;           /* Mem[SP-1]=AC */
                                ;           ...
                                ;           incsp   #-Nlocs   ; } /* WriteInt() */ /* SP=SP-2 */
                                ;           ret      ;           /* PC=Mem[SP]; SP=SP-1 */
                                ;           ...

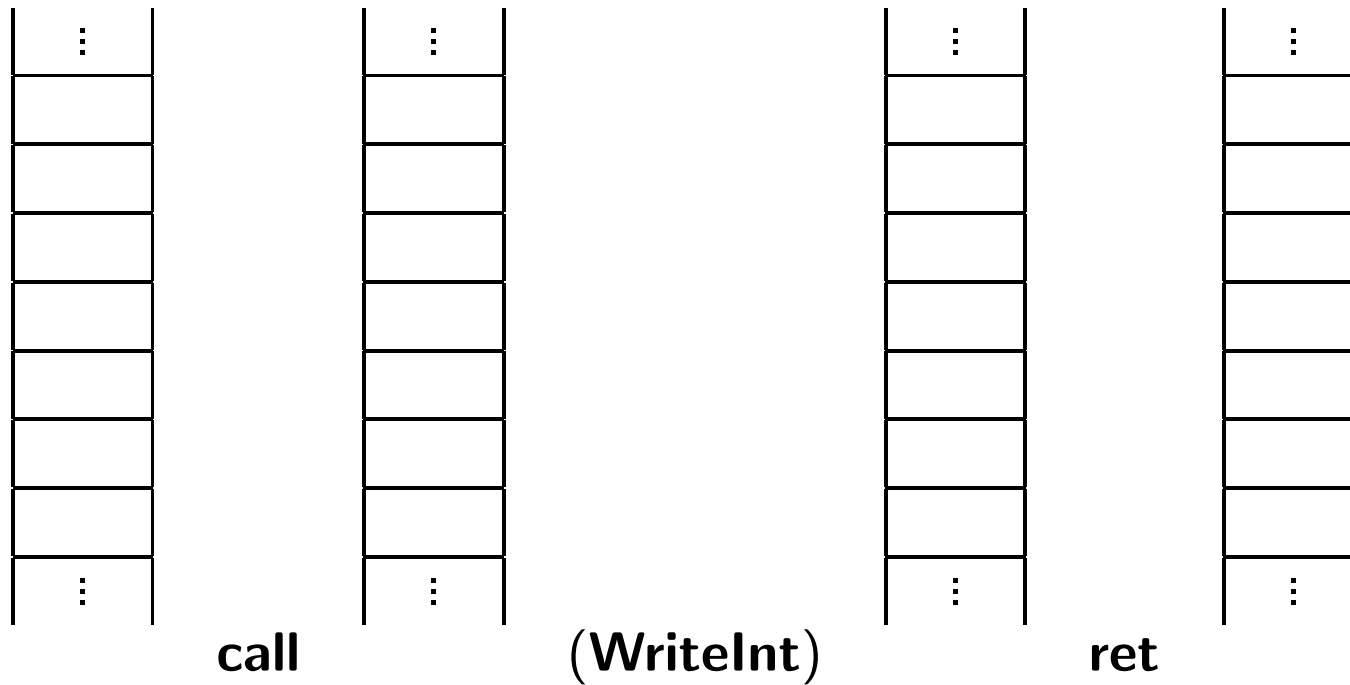
```

Procedure **WriteInt** is available in module **InOut.ass**

## Procedure call – Example with two local variables (2)

```
...
                                ; WriteInt(n, 6);
load    n                       ; /* Push(n) */ /* AC=Mem[n] */
incsp   #1                       ; /* SP=SP+1 */
store   !0                       ; /* Mem[SP]=AC */
load    #6                       ; /* Push(#6) */ /* AC=6 */
incsp   #1                       ; /* SP=SP+1 */
store   !0                       ; /* Mem[SP]=AC */
call    WriteInt ; /* SP=SP+1; Mem[SP]=PC;
                                ; PC=WriteInt */
incsp   #-2                       ; /* Pop(2) */ /* SP=SP-2 */
```

## Procedure call – Example with two local variables (3)



## Non-void functions in PeANUt

- We implement local variables via the stack
  - Good: Economy, privacy, recursion and sharable
  - Bad: We don't have stack-indirect or stack-indexed addressing modes in PeANUt
- Inside procedures
  - We have to increment the **SP** by number of local variables just after entry (first instruction within procedure)
  - Decrement **SP** by number of local variables just before (each) **ret**
  - Parameters, local variables and return values (**RVs**) have stack offsets
- Non-**void** function call is similar, but it must first make (empty) slot for the return value, which gets accessed after **call** before it is *popped*

## Non-void function – Example with one return value (1)

```

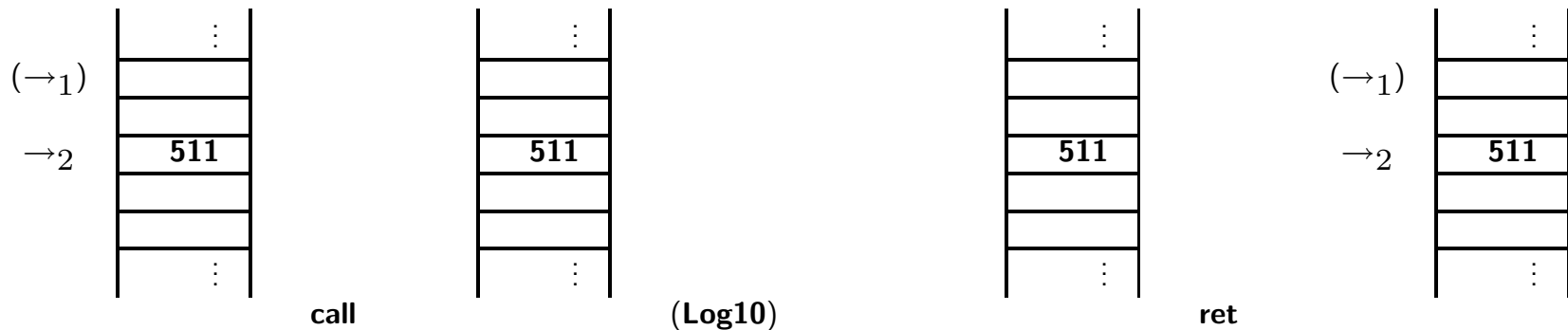
RV      = -3    ; int Log10(
x       = -2    ;   unsigned int x) {
        ;
Logx    = 0     ;   unsigned int Logx;
NLocs   = 1     ;
Log10:  ;
incsp   #NLocs ;   /* SP=SP+1 */
load    !x     ;   if (x != 0) { /* AC=Mem[SP-2] */
cmp     #0     ;   /* compare AC,0 */
beq     Lendif ;
...
Lendif: ;   } /* if */
load    !Logx  ;   return Logx; /* AC=Mem[SP] */
store   !RV    ;   /* Mem[SP-3]=AC */
incsp   #-NLocs ; } /*Log10()*/ /* SP=SP-1 */
ret     ;   /* PC=Mem[SP]; SP=SP-1 */
...

```

## Non-void function – Example with one return value (2)

```

log:   block 1      ; int log;
      ; log = Log10( 511 );
      incsp  #1     ; /* SP=SP+1 */ /* make RV slot */
      Push  (#511) ; /* AC=511; SP=SP+1; Mem[SP]=AC */
      call  Log10  ; /* SP=SP+1; Mem[SP]=PC; PC=Log10 */
      Pop   (1)     ; /* SP=SP-1 */
      load  !0     ; /* AC=Mem[SP] */ /*store RV*/
      store log    ; /* Mem[Log]=AC */
      incsp #-1    ; /* SP=SP-1 */ /*pop RV slot*/
    
```



## Address parameters in PeANUt

- In general there are two different types of parameters: Value (local copy) and reference (pointers)
- Full power of the procedure concept requires the ability to modify data (parameters)
- Do via passing address (not value) on stack when calling
- Procedures thus reference the *real* memory location indirectly via this *pointer*

## Address parameters – Example (1)

```

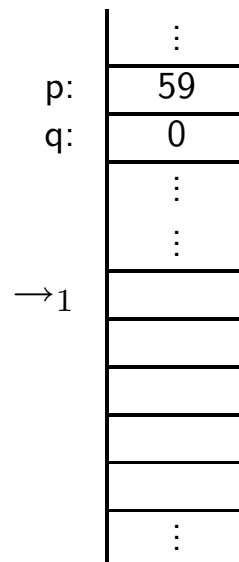
                                ; void Sum(
a      = -3                    ;   int a,
b      = -2                    ;   int b,
c      = -1                    ;   int *c) {
Sum:                                     ;
load   !c                      ;   *c = a + b;
storexr                               ;           /* XR=Mem[SP+c] */
load   !a                      ;           /* AC=Mem[SP-3] */
add    !b                      ;           /* AC=AC+Mem[SP-2] */
store  *0                      ;           /* Mem[XR]=AC */
ret                               ; } /* Sum() */
...

```

## Address parameters – Example (2)

```
p:      block    1      ; int
q:      block    1      ; p,
        ...          ; q;
        ...          ;
        ...          ; Sum( 5, p, &q);
        load     #5     ; /* Push(5); */
        incsp    #1     ;
        store    !0     ;
        load     p      ; /* Push(p) */
        incsp    #1     ;
        store    !0     ;
        loada   q       ; /* Push(address q) */
        incsp    #1     ;
        store    !0     ;
        call     Sum    ;
        incsp    #-3    ; /* Pop(3) */
```

## Address parameters – Example (3)



- Copy address parameter stack slot to **XR**, then access memory location via **\*0**

## Address parameters – Arrays

- For *array parameters*, put address plus index value in **XR**
- An address parameter is a *pointer* to a memory location that the procedure may modify (unfortunately, we cannot use **@** indirect mode)
- Example:

```

                                ; void WriteString(
                                ;   char s[]) { /* same as char *s */
                                ;
                                ;   int i;
WriteString:
                                ;
                                incsp #1 ;
                                ...
                                load  !s ;   printf("%c", s[i]);
                                add   !i ;
                                storexr ;   /* XR = Mem[SP+s]+Mem[SP+i] */
                                load  *0 ;
                                trap  #3 ;   /* Put */
                                ...

```

## Coding style

- Use C code to make things clearer
  - Often line-by-line correspondence with assembly code
  - Short comment for each important variable declaration
  - Properly indented (2 spaces)
  - Syntactically correct
  - Use symbolic constants
  
- Macros
  - Sensibly chosen
  - Well commented when define

## More on coding style

- Algorithm
  - Simple is beautiful
  - No unnecessary output (like prompts)
  - Don't bother checking input
  - Own procedures (if any) must not use global variables
  
- Assembly code
  - Properly indented
  - Keep identifiers as meaningful as possible (in 6 letters)
  - Beware of optimisations
  - No spaghetti code