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
- other issues:
  - revise 2008 MSE, Q1
Two-dimensional Arrays

- multi-dimensional arrays are the main data structure in sci./eng. applications
- in C, implemented via row-major ordering:
  - i.e. 2-D element $ws[i][j]$ is addressed as if 1-D element $ws[i*N+j]$, where $N$ is length of a row ($ws[i]$)
- e.g. an array of strings (of length up to $N-1$)

```c
N = 4; // define N 4
M = 2; // define M 2
MN = 8; /* MN = M*N */
ws: block MN; char ws[M][N];
i: data 1 ; int i = 1;
j: data 2 ; int j = 2;
load i; ws[i][j] = 'x';
mul #N ;
add j ;
storexr ; /* Xr = mem[i]*N+mem[j] */
load #'x';
store *ws ;
```
**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.)
**PeANUUt Macro Example: macro.ass**

- **definitions:**
  
  ```
  macro Get (x)  
  trap #2        ; read next char into x  
  store x       ; (read next char into AC)  
  ; (Memory[x] = AC)
  endmacro
  
  macro Set2 (x, e1, op, e2)  
  load e1       ; perform x = e1 + e2  
  op e2         ; (AC = <value of e1>)  
  store x       ; (AC = AC op <value of e2>)  
  ; (Memory[x] = AC)
  endmacro
  ```

- Get(x) and Set2(x,e1,op,e2) can be called as follows:
  
  ```
  Get  (ch)        ; scanf("%c", &ch);
  Set2 (n, ch, sub, #'0'); n = ch − '0';
  ```

- these are expanded to (see macro.lst, produced by assembler):
  
  ```
  trap #2
  store ch
  load ch
  sub #'0'
  store n
  ```
PeANUt Macros – Beware!

● beware of the following:
  \[ \text{Get('0')} \]
  \[ \text{Set2(n, sub, ch, '#0')} \]

● which is expanded to:
  \[
  \begin{align*}
  \text{trap} & \quad \#2 \\
  \text{store} & \quad '0'; \text{ run-time error?} \\
  \text{load} & \quad \text{sub}; \text{ assembly error} \\
  \text{ch} & \quad '#0'; \text{ assembly error} \\
  \text{store} & \quad \text{n}
  \end{align*}
  \]

● 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
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
Stack Addressing and Manipulation

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

<p>| | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>259</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>260</td>
<td>15</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>261</td>
<td>-1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SP (\rightarrow) 262</td>
<td>40</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- recall example:

```
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`
Macro Examples for the Stack: stackmacro.ass

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

- **Pop item from stack:**
  
  ```
  macro Pop (n)   ; pop n elements of the stack
  incsp #-n       ; (SP = SP – n)
  endmacro
  ```
Procedure Calls

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

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

1. return value (if any)
2. parameters
3. return address
4. local variables (if any)

→1 SP just before (explicitly) pushing parameters & just after popping them
→2 SP just before executing call P and just after executing P’s ret
→3 SP just before executing 1st instruction in P and just before executing P’s ret
→4 SP inside body of P, after (explicitly) allocating space for local variables (references to parameters, return value & local variables are relative to this position)
Procedure Call – Example without Local Variables

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

```c
ch = -1 ; void Write(char ch) {
  Write:
  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:

```c
; 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

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