

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 2006 exam Q1 (cont.)

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`) (Q: does a C compiler do it *exactly* this way?)

		<code>N</code>	<code>= 4</code>	<code>;</code>	<code>#define N 4</code>		
		<code>M</code>	<code>= 2</code>	<code>;</code>	<code>#define M 2</code>		
		<code>MN</code>	<code>= 8</code>	<code>;</code>	<code>/* MN = M*N */</code>		
<code>ws:</code>	<code>block</code>	<code>MN</code>	<code>;</code>	<code>char ws[M][N];</code>			<code>ws:</code>
<code>i:</code>	<code>data</code>	<code>1</code>	<code>;</code>	<code>int i = 1;</code>			
<code>j:</code>	<code>data</code>	<code>2</code>	<code>;</code>	<code>int j = 2;</code>			
			<code>;</code>				
	<code>load</code>	<code>i</code>	<code>;</code>	<code>ws[i][j] = 'x';</code>			
	<code>mul</code>	<code>#N</code>	<code>;</code>				
	<code>add</code>	<code>j</code>	<code>;</code>				
	<code>storexr</code>		<code>;</code>	<code>/* XR=mem[i]*N+mem[j] */</code>			<code>i:</code>
	<code>load</code>	<code># 'x'</code>	<code>;</code>				<code>1</code>
	<code>store</code>	<code>*ws</code>	<code>;</code>				<code>j:</code>
							<code>2</code>

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) ; scanf ("%c", &ch);
Set2 (n, ch, sub, #'0'); n = ch - '0';
```

- these are expanded to:

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

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, $opspec \leq 0$

- 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`

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

