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 `ws[i]`)

```plaintext
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 ; i: 1
storexr ; /*XR=mem[i]*N+mem[j]*/ j: 1
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.)

PeANUt Macro Example: `macro.ass`

```plaintext
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:

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

- these are expanded to (see macro.lst, produced by assembler):

```plaintext
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 ;

The Stack and Function Calls

● a fundamental programming concept!!
  ■ hence hardware support is needed (e.g. SP register, stack addressing mode)

● 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

<table>
<thead>
<tr>
<th>SP</th>
<th>262</th>
<th>260</th>
<th>15</th>
<th>261</th>
<th>40</th>
</tr>
</thead>
<tbody>
<tr>
<td>SP →</td>
<td></td>
<td></td>
<td>259</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

recall example:

load 10 ; 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
  ; (use before call proc. with value parameter)
  load e
  ; (AC = <value referred by e>)
  incsp #1
  ; (SP = SP + 1; done last in case e!=num)
  store 10
  ; (Memory[SP] = AC)
  
  endmacro

● Pop item from stack:
  
  macro Pop (n)
  ; pop n elements of the stack
  ; (use at end of call)
  incsp #-n
  ; (SP = SP – n)
  
  endmacro

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.ass
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 ii, i2, ...;
    ...
}
```

→ 1
- RV return value (if any)
- p1 p2 parameters

→ 2
- RA return address
- l1 l2 local variables (if any)

→ 3
- SP just before (explicitly) pushing parameters & just after popping them
- SP just before executing call P and just after executing P's ret
- SP just before executing 1st instruction in P and just before executing P's ret
- SP inside body of P, after (explicitly) allocating space for local variables (references to parameters, return value & local variables are relative to this position)

→ 4
- SP just before (explicitly) pushing parameters & just after popping them
- SP just before executing call P and just after executing P's ret
- SP just before executing 1st instruction in P and just before executing P's ret
- 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: Stack

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

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

```assembly
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:

```assembly
; Write('a');
load #a ; /* Push('#a') */ /* AC='a' */
incsp #1 ; /* SP=SP+1 */
store 10 ; /* 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