

Some announcements

- **Assignment 1** is due next **Wednesday 5th April at 12:00 (noon)**
- **Assignment 2** will be released towards the end of next week (draft version first) (due in week 11, Wednesday 17th May)
- Lab 4 next week (functions in PeANUt)

Two-dim arrays, macros and procedures in PeANUt assembly

- Two-dimensional arrays in assembly
- Macro definitions and usage
- The stack revisited
- Procedures
 - Arguments
 - Without local variables
- Reference:
 - Specification of the PeANUt computer (Section 4)

Two-dimensional arrays

- Multi-dimensional arrays are the main data structure in many scientific and engineering applications (e.g. matrices, grids)
- Two-dimensional array **ws[i][j]** is addressed as if it was a one-dimensional array with **ws[i*N+j]** with **N** the *stride* in the first dimension (matrix with **M** rows and **N** columns)
- Assuming row-major ordering, like in C

Two-dimensional array – Example

```

M = 4           ; #define M 4
N = 3           ; #define N 3
                ;
a:  block M*N   ; int a[M*N]; /* Matrix a */
b:  block M     ; int b[M];   /* Vector b */
i:  block 1     ; int i;
j:  block 1     ; int j;
                ;
                ; for (i=0; i<M; i++) {
                ;   for (j=0; j<N; j++) {
                ;     b[i] = b[i] + a[i*N+j]; /* a[i][j] */
                ;   }
                ; }
                ; }

```

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, not translated
- Exist only in the *assembly language level*, i.e. are *expanded* by the *assembler* (C macros exist only in the C language)
- Can be good programming style, especially if they correspond to meaningful (high level language) operations

5

COMP2300, 2006

PeANUt Macro examples continued

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

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

- Is expanded to:

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

- Macros are best for 'straight-line' code (don't use macros with branches etc.)

7

COMP2300, 2006

PeANUt macro examples

```
macro Get (x)
  trap #2
  store x
endmacro
```

```
macro Set2 (x, e1, op, e2) ; x = e1 op e2;
  load e1
  op e2
  store x
endmacro
```

- Macro definitions must be inserted at top of a program

6

COMP2300, 2006

PeANUt Macros – Beware!

- Beware of the following:

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

- Is expanded to:

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

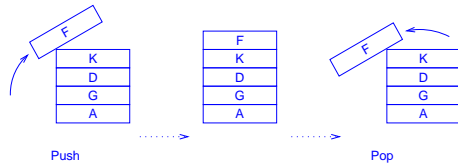
- Good use of macros can 'abstract' some low level details and it can clarify the program's structure
- Bad use of macros can obscure what is going on and is the origin of many errors

8

COMP2300, 2006

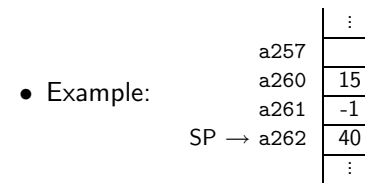
The stack and function calls

- A fundamental programming concept!! Hence *hardware* support needed (e.g. **SP**, 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



Stack addressing

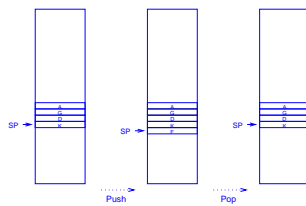
- Requires *stack addressing mode* (!): $AOP = \langle opspec \rangle + SP$
- Normally, $\langle opspec \rangle \leq 0$



```
load !0 ; AOP = 262 OP = 40
mul !-2 ; AOP = 260 OP = 15
store !-3 ; AOP = 257 OP = 600
```

Stack implementation

- Can be efficiently implemented as the memory pointed to by the **stack pointer (SP register)**



- **SP** is used to enable return of control to caller, as well as to pass *parameters* and *return values*

Stack manipulation

- The stack can be manipulated by software both implicitly and explicitly
- Implicitly: Via **call** and **ret** (**SP** is incremented and decremented automatically)
- Explicitly: Using commands like
 - **incsp #1**
 - **store !-1**

Macro examples for stack

- **Push** item onto stack

```
macro Push (i) ; /* Push value onto stack */
    load i ; /* AC = <value referred by i> */
    incsp #1 ; /* SP = SP + 1 */
    store !0 ; /* Memory[SP] = AC */
endmacro
```

- **Pop** item from stack

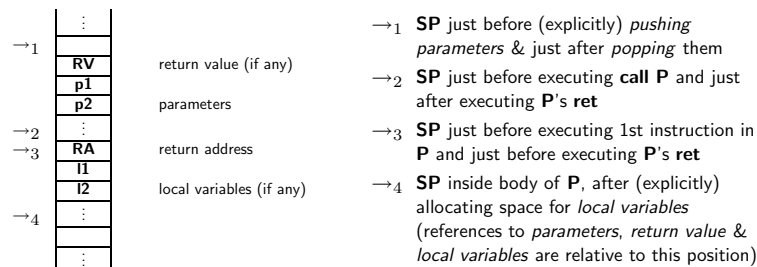
```
macro Pop (n) ; /* Pop n elements from stack */
    incsp #-n ; /* SP = SP - n */
endmacro
```

Procedure calls

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

Example C function declaration:

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



- 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 (1)

```
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 */

main:
    ...
    ; 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 *beginning* of each *instruction cycle*

Procedure call – Example without local variables (2)

