Procedures in PeANUt

- Number systems (bases) in `.mli` files
- Procedure / function calls
- Nested procedures
- The stack
  - Stack pointer
  - Stack addressing mode
  - Stack frame
- Reference: Specification of the PeANUt computer
- Additional reading: Section 3.7 in Bryant and O'Hallaron

Using other bases in `.mli` files

- Writing all instructions in binary can be tedious, although it often makes things clearer
- Notation
  - Octal **o** (1 digit → 3 bits)
    - o100 → 001 000 000
    - o123 → 001 010 011
    - o767 → 111 110 111
  - Hex **h** (1 digit → 4 bits)
    - h10 → 0001 0000
    - h79 → 0111 1001
    - h9D → 1001 1101

More on bases in `.mli` files

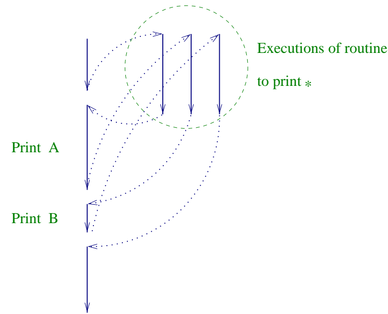
- Notation continued
  - Decimal **d** (n digits → 16 bits)
    - d5 → 0000 0000 0000 0101
    - d64 → 0000 0000 0100 0000
    - d79 → 0000 0000 0100 1111
  - Address **a** (octal) (n digits → 10 bits)
    - a5 → 0 000 000 101
    - a17 → 0 000 001 111
    - a167 → 0 001 110 111
- Examples:
  - 001 001 0 000 101 000 → o1 o1 a50 *; load* 50_8
  - 110101 0 000 000 011 → o6 o5 a3 *; trap 3*
  - 1110101 000 000 000 → hE o5 a0 *; compXR*

Simple procedure calls

- Motivation
  - Often, the instruction set does not include some operation that is regularly required
  - The user can effectively extend the instruction set by using procedures / functions
  - Procedures can be written in PeANUt (like functions in C)
- PeANUt Procedures
  - The instructions **CallProcedure** and **Return** are important
  - **CallProcedure** allows the PC to be *remembered*, and a new PC value given (so that execution can continue from a different place)
  - **Return** retrieves the *remembered* PC value and resets the PC to this value (so execution continues where it left off)

Procedure example

- Write a program that prints out $*A*B*$

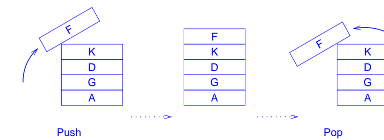


Procedure example code

```
START a10
AT a10
110100          a30    ;a10 CallProcedure a30
o0 o1   0 001 000 001  ;a11 Load 'A'
110101           a3    ;a12 Trap 3 (put)
110100          a30    ;a13 CallProcedure a30
o0 o1   0 001 000 010  ;a14 Load 'B'
110101           a3    ;a15 Trap 3 (put)
110100          a30    ;a16 CallProcedure a30
110101           a1    ;a17 Trap 1 (halt)


AT a30
o0 o1   0 000 101 010  ;a30 Load '*'
110101           a3    ;a31 Trap 3 (put)
1110000        o000    ;a32 Return
```

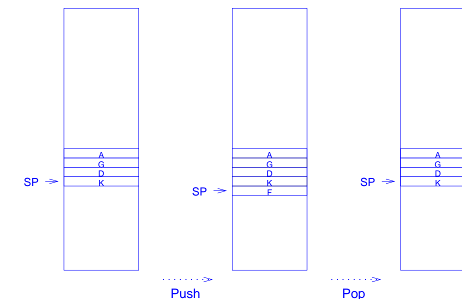

Nested procedures (procedures within procedures)

- Is it possible to nest procedures?
  - Can more than one PC value be saved?
  - If so, how are multiple PC values stored in an organised way?
- The Stack
  - What is a stack?
  - Consider a pile of books
  - LIFO (Last In, First Out) - compare with a queue (FIFO)

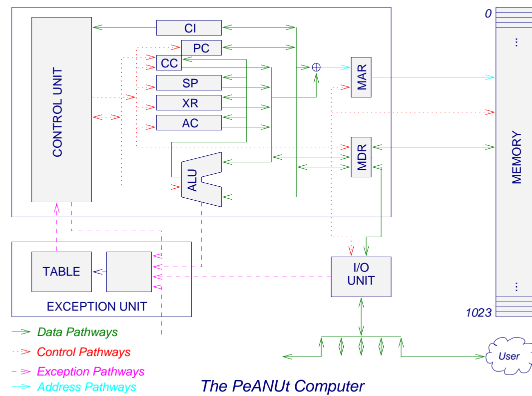


How can we make a stack?

- In special hardware (complex, finite size)
- In normal memory



The PeANUt architecture



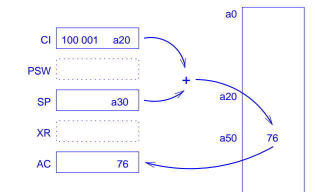
The PeANUt Computer

The stack pointer (SP register)

- Points to top of stack
- Automatically increases when a **CallProcedure** is made
- Automatically decreases when a **Return** is made
- Can also be manually incremented or decremented
- PC values are stored on the stack (to allow return from procedures)
- Procedure nesting limited only by potential stack size
- Stack size limited only by available memory
- Stack can be used to pass parameters

Stack addressing mode (mode bits 100)

- The address of the operand is given by adding the contents of the Stack Pointer (SP) and the opspec
- The operand is at **mem[SP + opspec]**



- Similar to indexed addressing mode

Procedure context

- What information is relevant for the duration of the execution of a procedure / function (and not at any other time)?
  - Return address
  - Parameters (sometimes)
  - Return value (sometimes)
  - Local variables (sometimes)
- The stack is used to store all the information associated with the execution of a procedure
- This information is known as a *stack frame*

Stack frame

- Conventions define the order of data within a stack frame
- Lifetime of stack frames
  - When the procedure is invoked, *a new stack frame is created, values are initialised*
  - When the procedure is terminated, *the stack frame is removed from the stack*
- The use of stack frames allows variables associated with all currently executing procedures to be accessible, without having them permanently allocated

Using the PeANUt's stack

- Creating the stack frame
- Consider the elements of a stack frame:
  - Return value (RV) *(later...)*
  - Parameters
  - Return address
  - Local variables (LV) *(later...)*
- On the PeANUt, stack frames will be constructed in the above order (return value first, local variables last) onto the stack
- For now, we will deal only with the parameters and the return address

A procedure call with no parameters

- To call the procedure, just use the **CallProcedure** instruction (the following steps are done automatically)
  - The SP is incremented by 1
  - The current PC value is placed on top of the stack
  - The procedure's start address is placed in the PC
- To return from a procedure, use the **Return** instruction (the following steps are done automatically)
  - The value on top of the stack is read and placed in the PC
  - The SP is decremented by 1
- Example on slide 6 (printing of *'*A*B*'*)

Dealing with parameters

- Unlike the return address, parameters must be *manually* placed on, and removed, from the stack
- Before calling a procedure, parameters must be placed on the stack
  1. Increment SP
  2. Store value to the top of the stack

  → Repeat for each parameter
- After returning from a procedure, the relevant stack frame must be removed
- Space on the stack for the parameters must be deallocated
  1. Decrement SP

  → Repeat for each parameter

Procedure with parameters example

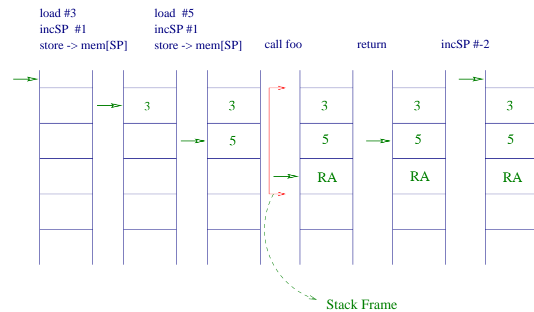
Main Program

```
int main() {
  ...
  foo(3,5);
  ...
}

START a10
AT a10
...
Load #3
IncSP #1
Store AC -> mem[SP]
Load #5
IncSP #1
Store AC -> mem[SP]
CallProcedure a30
IncSP #-2
Trap 1
```

Procedure

```
void foo(int a, int b) {
  printf("%d", a);
  printf("%d", b);
}

AT a30
Load mem[SP-2]
Add #48  ; convert into ASCII character
Trap 3   ; Put character
Load mem[SP-1]
Add #48  ; convert into ASCII character
Trap 3   ; Put character
Return
```


Example continued



- Something to think about: *value* vs *reference* parameters