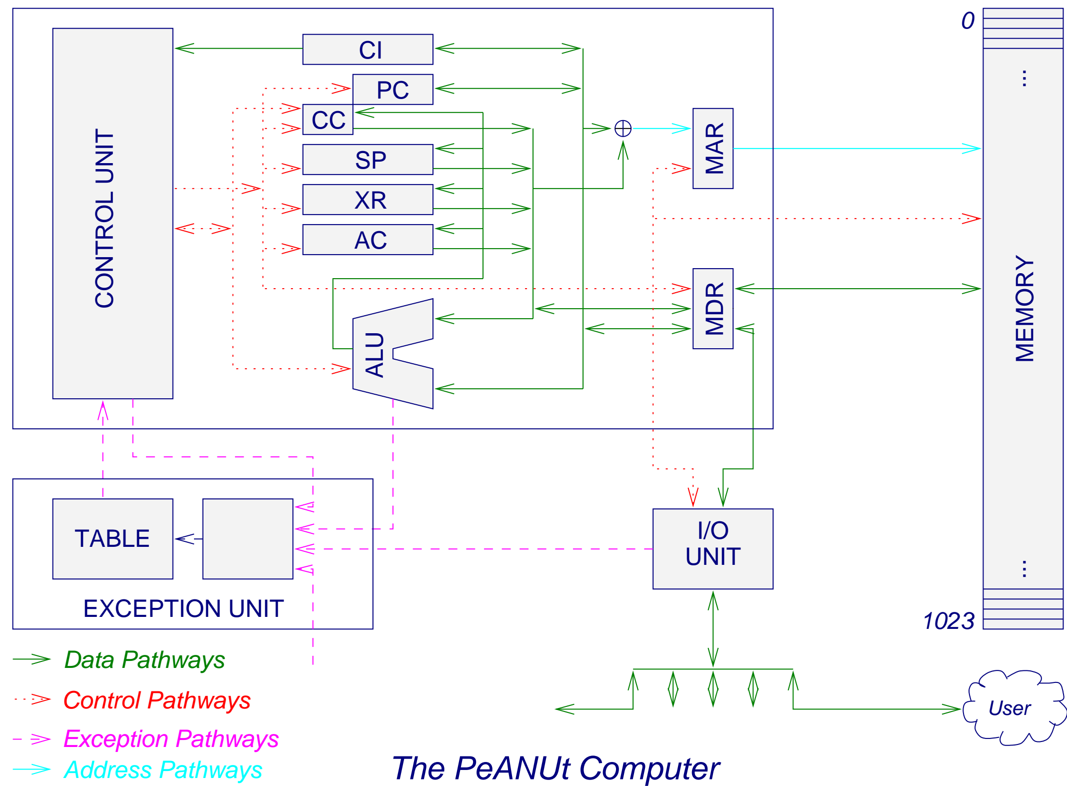
# Announcements

- NO lab session this week
  (lab 5 starts next week – PeANUt functions and bit operations, virtual memory, take reading brick along!)
- Assignment 2 is available... **start NOW!**
  (still **daft**, please contact `peter.strazdins@anu.edu.au` if there are ambiguities or mistakes...)
- Assignment 2 is due **Wednesday 17 May 12:00 (noon)**
  (responsible lecturer: Peter Strazdins)
- Assignment 1 marks next Tuesday, returned in lab 5
- My contact hours next week: only Monday 2-3 pm

# PeANUt repetition and virtual memory

- PeANUt repetition
  - Architecture
  - Instructions and addressing modes
  - The stack and procedures
  - Macros and traps
- Virtual memory
  - Introduction
  - Paging
- References:
  - Specification of the PeANUt computer (Section 3)
- Additional reading: 10.1 – 10.7 in Bryant and O'Hallaron

# PeANUt repetition – Architecture




The PeANUt Computer

# PeANUt repetition – Instructions

- All instructions are 16 bits long (one memory cell)
- Each instruction has up to three components:
  1. An opcode
  2. An operand specifier (**opspec**)
  3. A mode (addressing mode)
- An instruction's opcode identifies it
  - Instruction set: *PeANUt Specifications*, table on page 5
  - Assembly operations: *PeANUt Specifications*, table on page 15
- The operand is the data upon which the instruction operates
  - Some instructions do not need an operand

# PeANUt repetition – Addressing modes

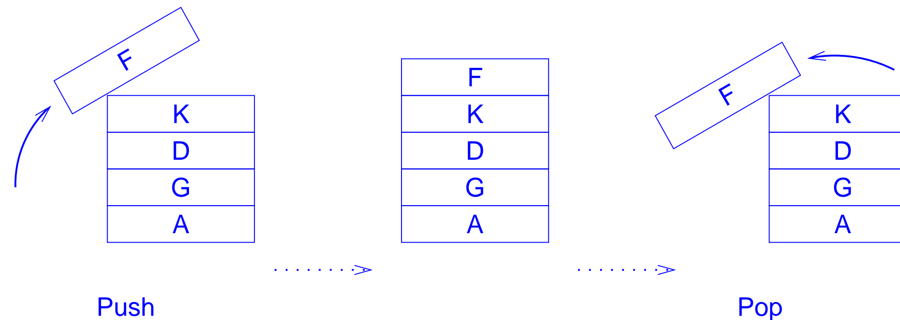
- Corresponding to most instructions is an operand (**OP**)
- **opspec** is the lowest 10 bits of the instruction
- Five addressing modes:

| | Mode | Symbol | Operation | Example |
|---|---|---|---|---|
| 1) | **Immediate**: | (#) | `OP = <opspec>` | **load #20** |
| 2) | **Direct**: | | `AOP = <opspec>`<br>`OP = Memory[AOP]` | **load 20** |
| 3) | **Stack**: | (!) | `AOP = <opspec> + SP`<br>`OP = Memory[AOP]` | **load !-1** |
| 4) | **Indexed**: | (*) | `AOP = <opspec> + XR`<br>`OP = Memory[AOP]` | **load *20** |
| 5) | **Indirect**: | (@) | `AOP = Memory[<opspec>]`<br>`OP = Memory[AOP]` | **load @20** |

# PeANUt repetition – The stack and function calls

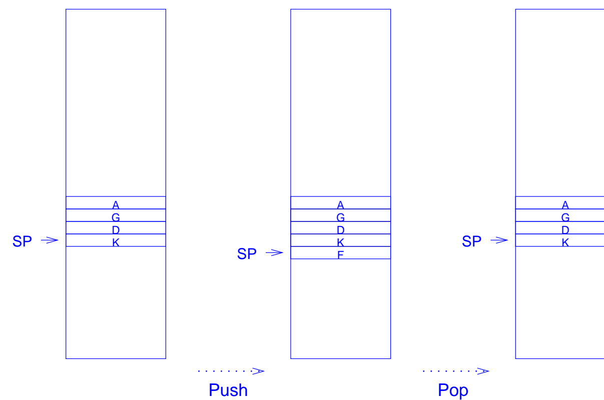
- A fundamental programming concept!! Hence *hardware* support needed (e.g. **SP**, stack addressing mode **!**)

- Uses a (reserved) part of (normal) memory called the **stack**
  - A **stack** is accessed *LIFO* (Last In, First Out), e.g. pile of books



# PeANUt repetition – Stack implementation

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

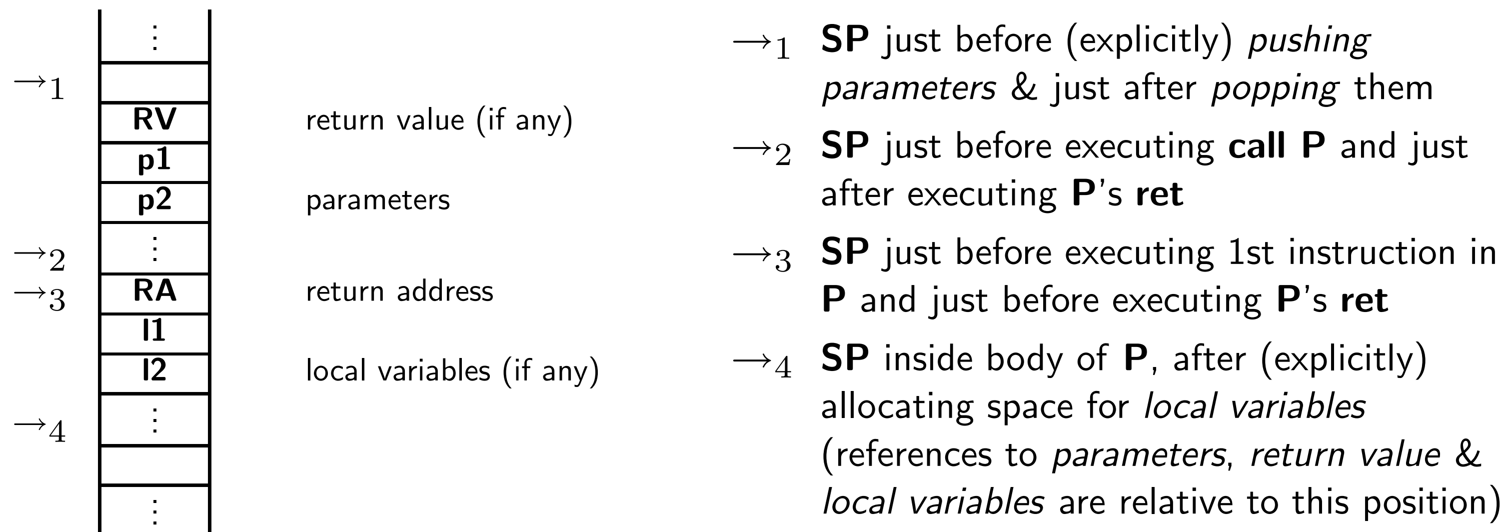



- **SP** is used to enable return of control to caller, as well as to pass *parameters* and *return values*

# PeANUt repetition – 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, ...;
  ...
}
```



# PeANUt repetition – Macros

- Context: Important (yet simple) computational concept
- Widely used in the **C** programming language
- Neither instructions nor procedures! Essentially, just a *shorthand*
- Exist only in the *assembly language level*, i.e. are *expanded* by the *assembler* (C macros exist only in the C language)
- Macros are best for *straight-line* code
- Can be good programming style, especially if they correspond to meaningful (high-level language) operations

# PeANUt repetition – Macro definition and usage

- Macro definitions must be inserted at top of a program

```
macro   Get     (x)                 ;  x = getchar();
        trap    #2
        store   x
endmacro
macro   Set2      (x, e1, op, e2)   ;  x = e1 op e2;
        load    e1
        op      e2
        store   x
endmacro
```

- Can be called as

```
Get     (ch)     ;  ch = getchar();
Set2    (n, ch, sub, #'0')
                 ;  n = ch - '0';
```

Are expanded into

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

# PeANUt repetition – Traps

- There is a special instruction called **trap**
- It is used to have PeANUt perform a service that is usually a call to the 'operating system'
- Depending on the operand of the trap, some particular operation will be performed, for example:
  - **trap #1** Halt: Tells the PeANUt to stop execution
  - **trap #2** Get: Allows you to read a character from keyboard
  - **trap #3** Put: Allows you to print out a character
- User definable traps (see lecture P9)

# Virtual memory

- Motivation: (Multiple) users regularly need to run jobs whose capacity exceeds that of main memory
- Virtual memory is a technique whereby main memory is made to appear to be larger than physical memory
- Needed because there is a memory hierarchy:
  - Many different mediums for the storage of data
  - Generally, there is a trade-off between speed and capacity (fast memories tend to be small; large memories tend to be slow)

| Medium | Access time (nsec) | Typical size |
|---|---|---|
| Registers | $\sim$10 | $<$ 1 KB |
| Cache memory | $\sim$25 | $<$ 2 MB |
| Main memory | 100 | $<$ 2 GB |
| Disk | 20,000,000 | $>$ 10 GB |

# Virtual memory – Nomenclature

- **Address space**: Range of addresses accessible
- **Logical/virtual addresses**: Addresses as seen by the programmer
- **Physical addresses**: Actual addresses in main memory

# Paging

- How do we share main memory between competing chunks of the virtual memory address space?
- One solution is called **paging**
  - Break all memory into equal sized chunks called pages
  - When accessing a virtual address, check if the corresponding page is in main memory (if not, move it into main memory and then access it)
- Exploits locality of (address) references
  - Memory accesses tend not to be random (in a program, they are often in a sequence)
  - Rule of thumb: A program spends about $90\%$ of its time in only $10\%$ of the code

# Paging issues

- How big should a page be? Influenced by disk technology
- What does main memory look like? It consists of a mixed group of pages, with each page occupying a slot, or page frame
- What does (disk) virtual memory look like?
  It consists of all of the pages
- Programmer's view of paging:
  - Is oblivious of it: All program addresses are virtual
  - Can only see performance degradation
    (when paging requires many disk accesses, called swapping)

# Virtual memory issues

- What pages should be resident in main memory (MM) at any time?
  - The **most used** pages
  - Different **paging policies** dictate how most used is defined

- Data consistency:
  - Upon a **page fault**, a page in main memory is removed
    - If simply thrown out, data may be lost
    - If always written back to disk, may be too slow
  - Solution: Write it back to disk, if it has been written to (made **dirty**) (hence the **D** bit in the **page table**)