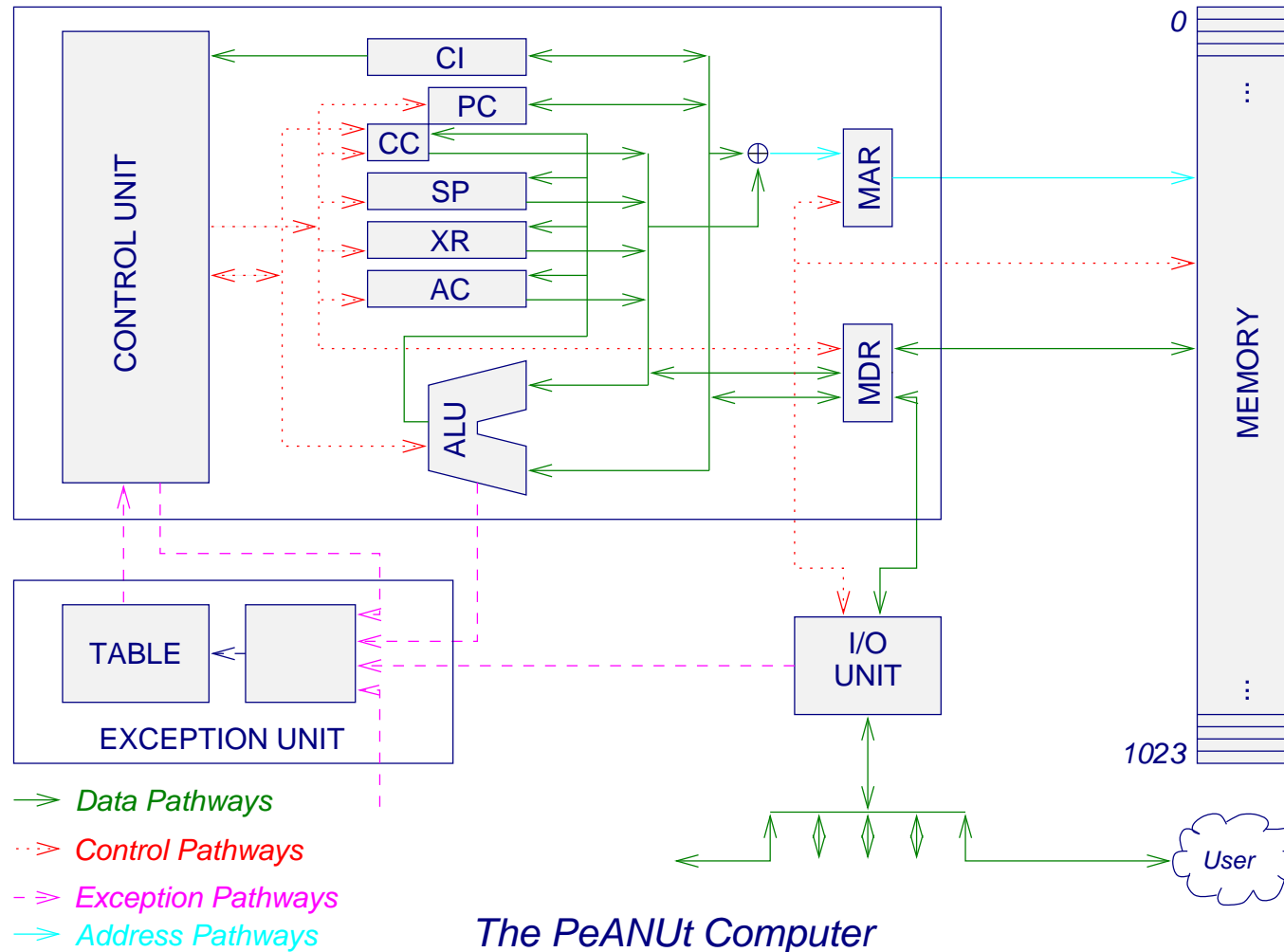


PeANUt Repetition and Virtual Memory

- ref: [PeANUt Spec, sect 3]; additionally [O'H&Bryant, sect 10.1–10.7] or [Null&Lobur, sect 6.5]
- PeANUt repetition
 - architecture
 - instructions and addressing modes
 - the stack and procedures
 - macros and traps
- virtual memory concepts
 - introduction
 - paging
- other issues:
 - MSE feedback

PeANUt Repetition – Architecture



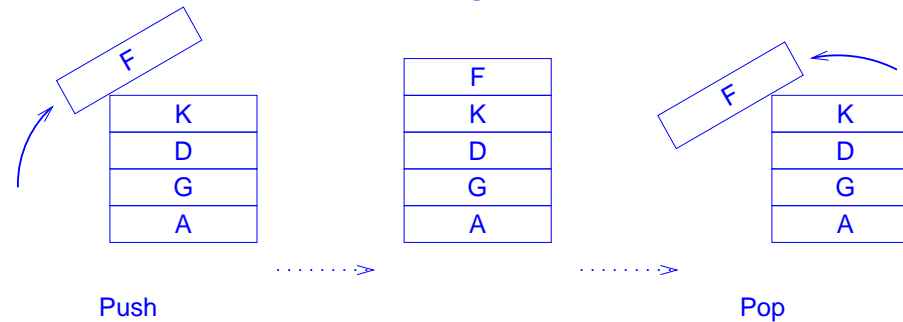
PeANUt Repetition – Instructions and Addressing Modes

- all instructions are 16 bits long (one memory cell)
- each instruction has *up to* three components:
 1. an opcode
 2. an (operand specifier (opspec) (in lowermost 10 bits)
 3. a mode (addressing mode) (in top 3 bits)
- the opcode identifies its operation [PeANUt Spec, table 1]
- the operand is the data upon which the instruction operates
 - some instructions do not need an operand, some have an implicit one
- there are 5 addressing modes:

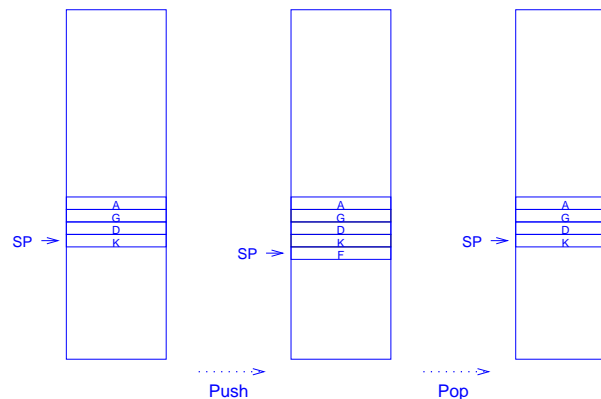
| | | | |
|---------------|-----|--|----------|
| 1) immediate: | (#) | OP = opspec | load #20 |
| 2) direct: | | AOP = opspec OP = Memory[AOP] | load 20 |
| 3) stack: | (!) | AOP = opspec + SP OP = Memory[AOP] | load !-1 |
| 4) indexed: | (*) | AOP = opspec + XR OP = Memory[AOP] | load *20 |
| 5) indirect: | (@) | AOP = Memory[opspec] OP = Memory[AOP] | load @20 |

PeANUt Repetition – The Stack and Function Calls

- a fundamental programming concept! Hence *hardware* support needed (e.g. SP, !)
- uses a (reserved) part of (normal) memory called the **stack**
 - is accessed *LIFO* (Last In, First Out), e.g. pile of books



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

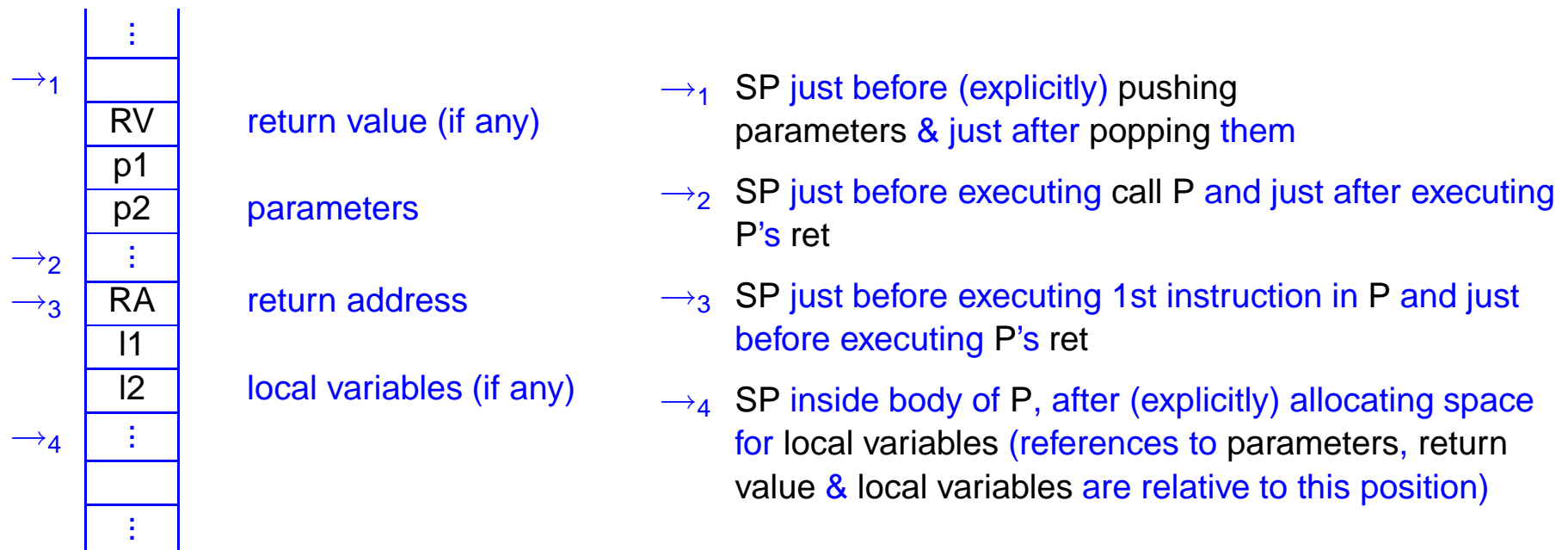


- enables return of control to caller & passing parameters and return values

PeANUt repetition – Procedure Call Conventions

- determines order of data in 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)
- are best for ‘straight-line’ code
- can be good programming style, especially if they correspond to meaningful
(high-level language) operations (
- e.g. cachesim.ass: printing strings; others?
- must be defined at the top of the program, used later (e.g. macro.ass)

PeANUt Repetition – Traps

- there is a special instruction called **trap**
- used to have PeANUt perform a 'operating system' service
- depending on the trap's operand, some particular operation will be performed, e.g.:
 - **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
- some are user-definable/modifiable (lecture P9)
- some relate to virtual memory

Reflection for COMP2300: to PeANUt or not to PeANUt? Possible alternatives:

- MARIE [Null&Lobur, Ch 4] – no indexed or stack mode :(
- Pep/8 [Computer Systems, J.S. Warford] – much like PeANUt
- 8088 Assembler/Simulator [Tanenbaum, Appendix C] – subset of the x86 assembly language
- a simplified RISC machine?

Virtual Memory

- motivation: (multiple) users regularly need to run jobs whose capacity exceeds that of physical memory (main memory)
 - one of the first (and most important) instances of virtualization
 - also, in a multiprocessing environment, each program 'sees' memory in the same way, regardless of where it is executing in physical memory
- virtual memory is a technique whereby program-addressable memory is made to appear to be larger than physical memory
- needed because there is a memory hierarchy:

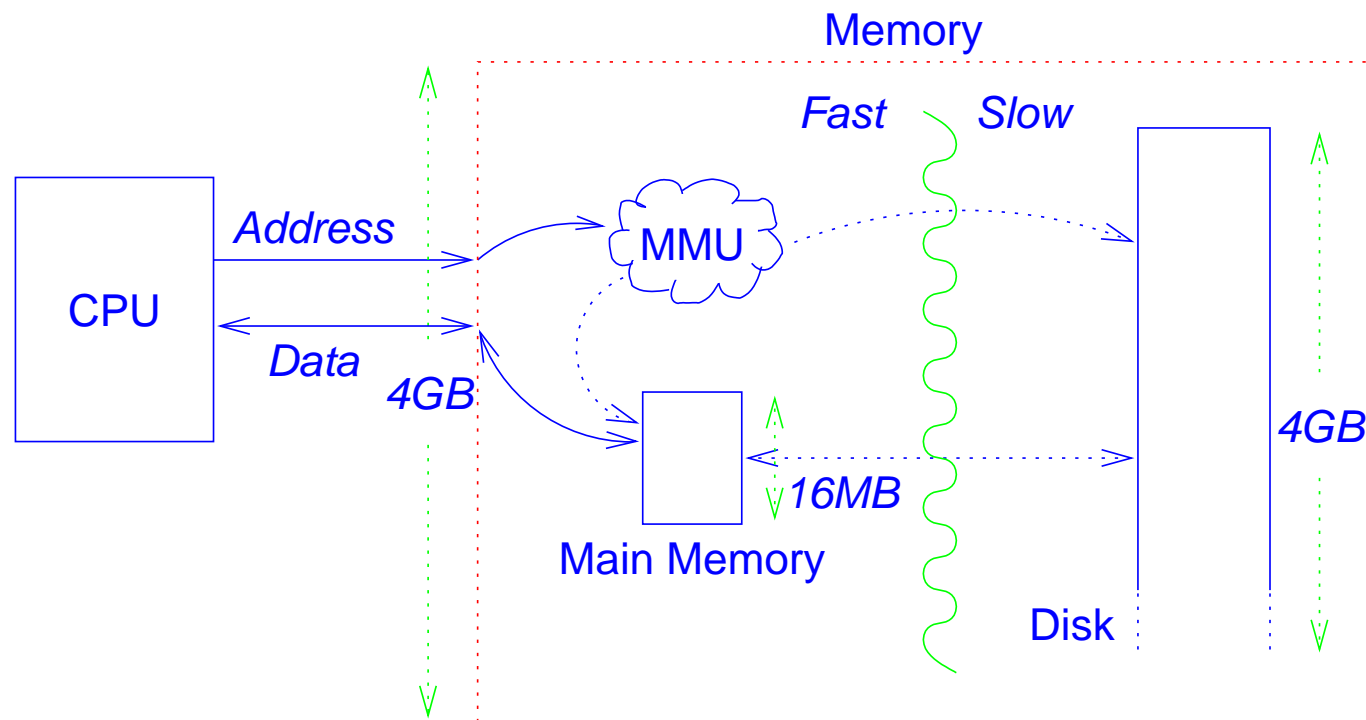
- many different mediums for the storage of data
- generally, there is a *trade-off* between speed and capacity (fast memories tend to be small; large memories tend to be slow)

| medium | access time (nsec) | typical size |
|-----------------|--------------------|--------------|
| registers | ~10 | < 1 KB |
| cache memory | ~25 | < 2 MB |
| physical memory | 100 | < 2 GB |
| disk | 20,000,000 | > 10 GB |

[O'H&Bryant, fig 6.21]

Virtual Memory – Nomenclature

- address space: range of addresses accessible to programmer
- logical/virtual addresses: addresses as seen by the programmer
- physical addresses: actual addresses in main memory
- the Memory Management Unit (MMU) performs this translation



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)
 - ◆ consider in particular accesses involved with instruction fetching
 - 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
 - needs to be large enough to amortize costs of overheads (disk block seek time; also page book-keeping costs)
 - but if too large, causes fragmentation
- what does main memory look like? It consists of a mixed group of pages, with each page occupying a slot (page frame) (e.g. PeANUt VM)
- 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 (in the near future)
 - different paging policies give an approximation to 'most used'
- data consistency:
 - upon a page fault (access of data in a page not currently in physical memory), a page currently in main memory usually has to be removed:
 - ◆ if simply thrown out, data may be lost
 - ◆ if always written back to disk (upon each **store**), will be too slow!
 - solution: write it back to disk, if it has been written to (made dirty)
 - ◆ hence the MMU must record this for each page ('Dirty bit')