

PeANUt Repetition and Virtual Memory

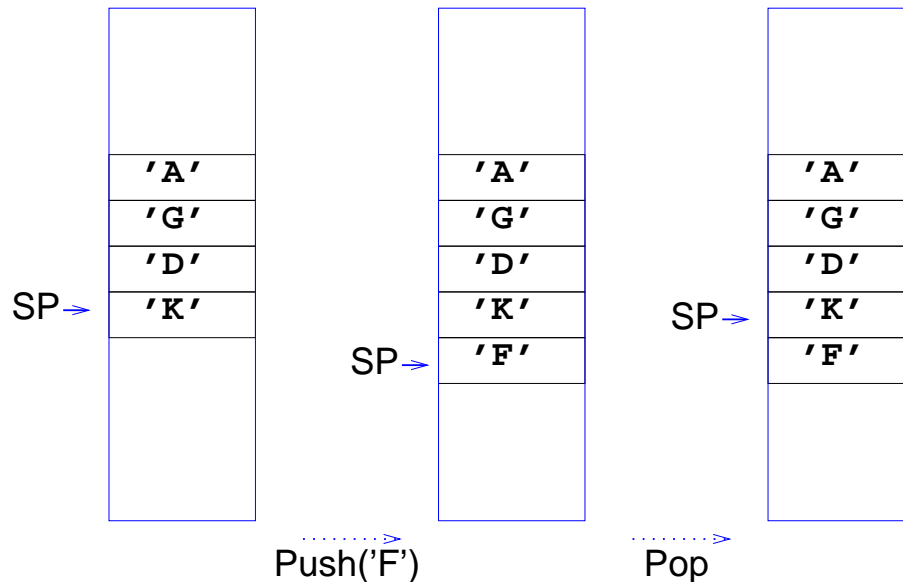
- ref: [PeANUt Spec, sect 3]; additionally [O'H&Bryant, sect 10.1–10.7] or [Null&Lobur, sect 6.5]
- PeANUt repetition
 - macros
 - stack concepts and addressing mode
 - the stack frame and procedures
 - traps
- virtual memory concepts
 - introduction
 - paging
- other issues:
 - MSE marks and feedback (+ some solutions)

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 (similarly for C macros)
- ‘regular’ macros
 - are best for ‘straight-line’ code
 - must be defined at the top of the program, used later (e.g. macro.ass)
 - checking/debugging: the `.lst` file shows expansion of macros
- concise macros are used to give symbolic names to (small) constants (e.g. **MAXSIZE = 64**) or stack offsets (e.g. **x = -3**)
 - must be defined (textually) *above* their first use
 - will *override* any earlier definition of the same symbolic name
 - e.g. procedure-example1.ass

PeANUt Repetition – The Stack Concepts and Addressing Mode

- a fundamental programming concept! Hence *hardware* support needed (e.g. SP, !)
- uses a (reserved) part of (normal) memory called the stack item can be efficiently implemented as the memory pointed to by a stack pointer register (SP)



■ stack addressing mode (!):

$AOP = ops\ spec + SP;$

$OP = Memory[AOP]$

◆ e.g. for `load !-3`, and if $SP=209$, $AOP=?$

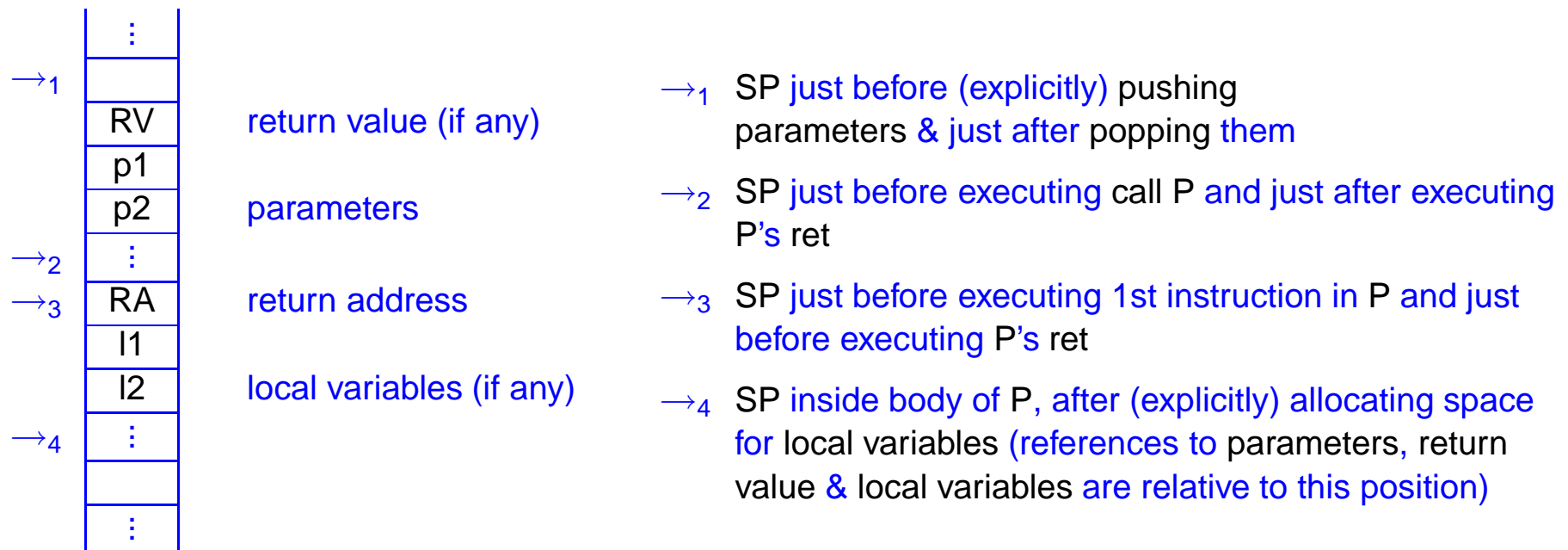
◆ in diagram, what is value of OP before & after the `Push()`?

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

PeANUt Repetition – Procedure Call Convention

- determines order of data in the stack frame
- example C function declaration:

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

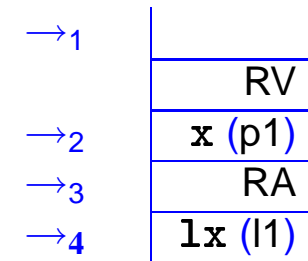
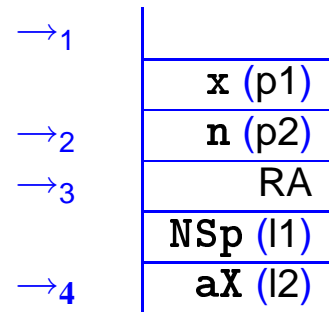
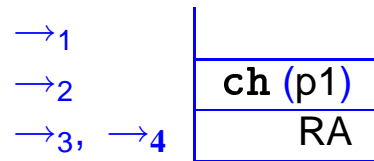


PeANUt Repetition – Procedure Call Convention Examples

```
void Write(char ch){
    printf("%c", ch);
}
```

```
void WriteInt(
    int x, int n) {
    int NSp, aX;
    ...
}
```

```
int Log10(
    int x) {
    int lx;
    ...
    return lx;
}
```

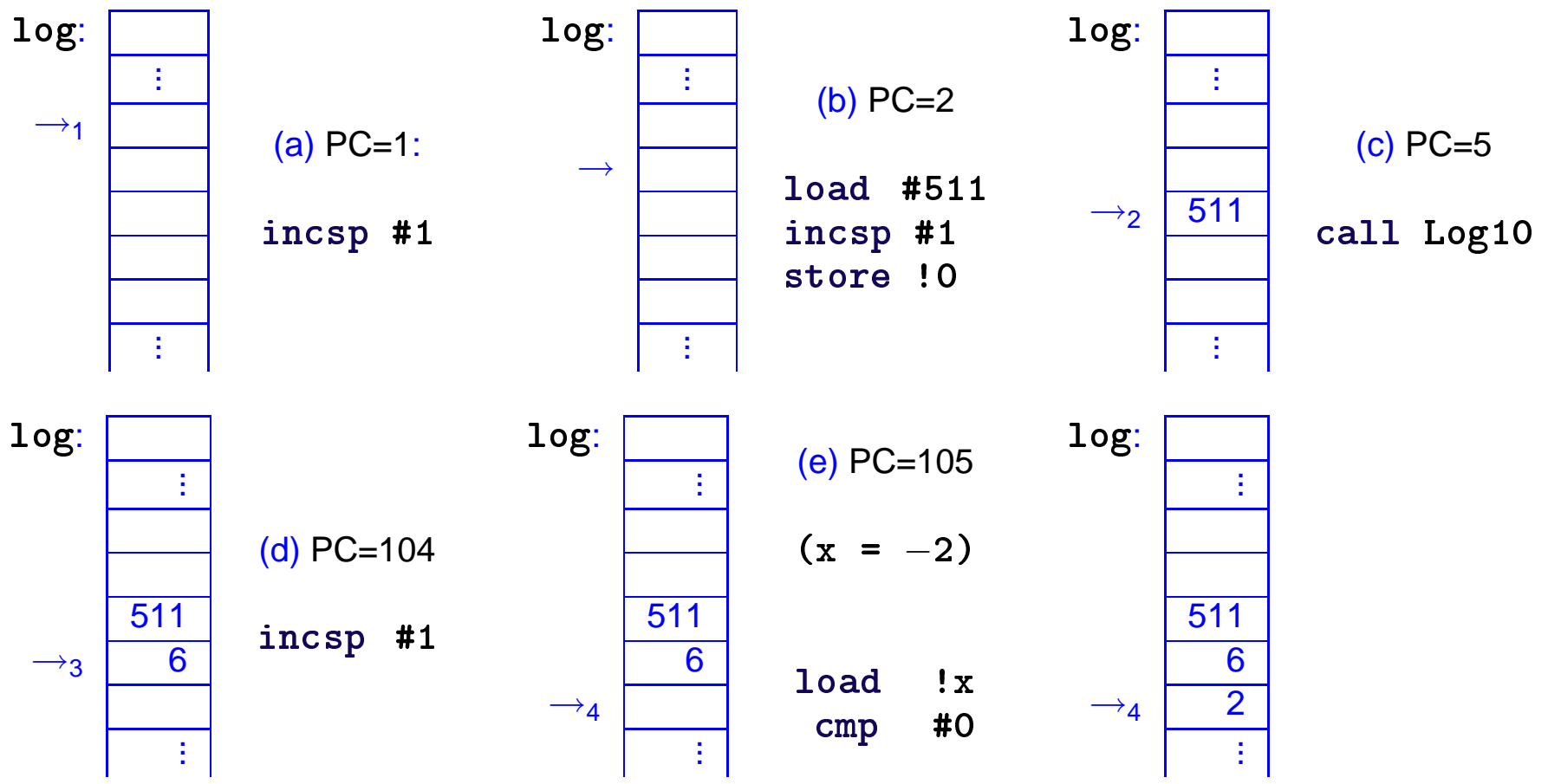


```
Write:
    ch          = -1
    load        !ch
    trap        #2
    ret
```

```
WriteInt:
    x           = -4
    n           = -3
    NSp         = -1
    aX          = 0
    incsp       #2
    ...
    incsp       #-2
    ret
```

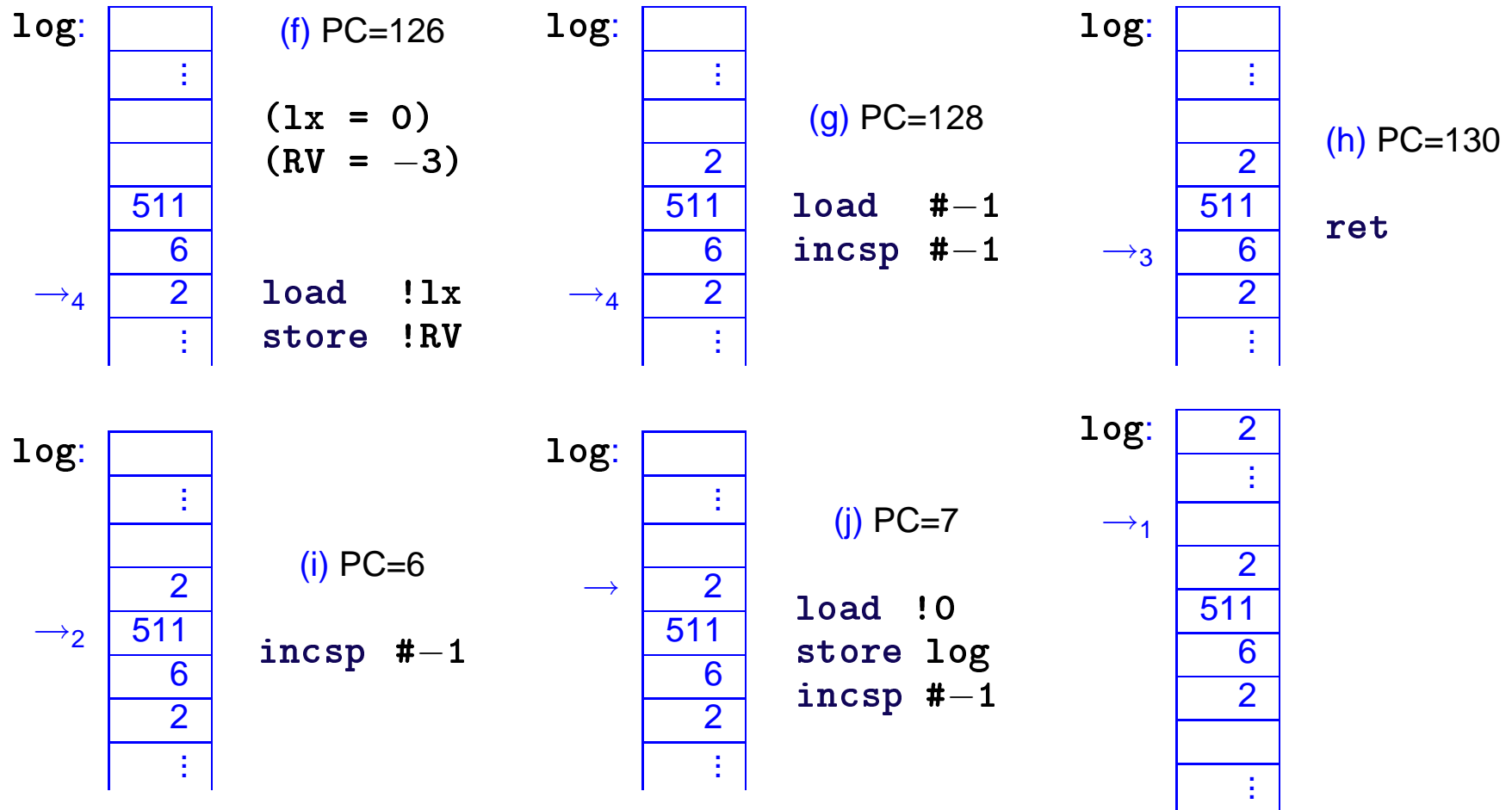
```
Log10:
    RV          = -3
    x           = -2
    lx          = 0
    incsp       #1
    ...
    load        !lx
    store       !RV
    incsp       #-1
    ret
```

PeANUt Repetition – Procedure Call Detail: $\log = \text{Log}_{10}(511)$



note: the **call** instruction pushes the incremented PC (the RA) on the stack

PeANUt Repetition – Procedure Call Detail: $\log = \text{Log}_{10}(511)$ (II)



note: the **ret** instruction sets PC = mem[SP] (RA) and pops the the stack

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]

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')