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)

![Diagram of stack operations](image)

- stack addressing mode (!):
  \[ AOP = \text{opspec} + \text{SP}; \]
  \[ OP = \text{Memory}[AOP] \]
  - e.g. for `load` !–3, and if \( \text{SP}=209, \ AOP=? \)
  - 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:

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

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>→1</td>
<td>RV</td>
<td>p1</td>
<td>p2</td>
</tr>
<tr>
<td></td>
<td>return value (if any)</td>
<td>parameters</td>
<td>→1</td>
</tr>
<tr>
<td>→2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>→3</td>
<td>RA</td>
<td>l1</td>
<td>l2</td>
</tr>
<tr>
<td></td>
<td>return address</td>
<td>local variables (if any)</td>
<td>→2</td>
</tr>
<tr>
<td>→4</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

| →3 |   |   |   |
| →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)
PeANUt Repetition – Procedure Call Convention Examples

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

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

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

```
\[ \begin{array}{c}
\text{Write:} \\
\text{load} \quad \text{!ch} \\
\text{trap} \quad \#2 \\
\text{ret}
\end{array} \quad \begin{array}{c}
\text{WriteInt:} \\
\text{incsp} \quad \#1 \\
\text{...} \\
\text{incsp} \quad \#2 \\
\text{...} \\
\text{incsp} \quad \#-2 \\
\text{ret}
\end{array} \quad \begin{array}{c}
\text{Log10:} \\
\text{incsp} \quad \#1 \\
\text{...} \\
\text{load} \quad \text{!lx} \\
\text{store} \quad \text{!RV} \\
\text{incsp} \quad \#-1 \\
\text{ret}
\end{array} \]
```

\[ \begin{array}{c}
\text{ch} \quad = \quad -1 \\
\text{x} \quad = \quad -4 \\
\text{n} \quad = \quad -3 \\
\text{NSp} \quad = \quad -1 \\
\text{aX} \quad = \quad 0 \\
\end{array} \quad \begin{array}{c}
\text{RV} \quad = \quad -3 \\
\text{x} \quad = \quad -2 \\
\text{lx} \quad = \quad 0 \\
\end{array} \]
PeANUt Repetition – Procedure Call Detail: $\log = \log_{10}(511)$

Note: the call instruction pushes the incremented PC (the RA) on the stack.
PeANUt Repetition – Procedure Call Detail: \( \log = \log_{10}(511) \) (II)

\[
\begin{array}{c|c|c}
\log & (f) \text{ PC}=126 & (g) \text{ PC}=128 \\
0 & (l\times = 0) & 2 \\
0 & (R\!V = -3) & 2 \\
511 & & 511 \\
6 & & 6 \\
4 & load !l\times & 3 \\
2 & store !R\!V & 6 \\
\end{array}
\]

\[
\begin{array}{c|c|c}
\log & (h) \text{ PC}=130 & \text{ret} \\
0 & & 2 \\
511 & & 6 \\
\end{array}
\]

\[
\begin{array}{c|c|c}
\log & (i) \text{ PC}=6 & (j) \text{ PC}=7 \\
0 & & 2 \\
511 & & 511 \\
6 & & 6 \\
2 & & 2 \\
\end{array}
\]

\[
\begin{array}{c|c|c}
\log & & 1 \\
0 & & 2 \\
511 & & 6 \\
\end{array}
\]

Note: the \texttt{ret} instruction sets \( \text{PC} = \text{mem[SP]} \) (RA) and pops the the stack
PeANUt Repetition – Traps

- There is a special instruction called `trap`.
- Used to have PeANUt perform an ‘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 :(.
- 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)

<table>
<thead>
<tr>
<th>medium</th>
<th>access time (nsec)</th>
<th>typical size</th>
</tr>
</thead>
<tbody>
<tr>
<td>registers</td>
<td>~10</td>
<td>&lt; 1 KB</td>
</tr>
<tr>
<td>cache memory</td>
<td>~25</td>
<td>&lt; 2 MB</td>
</tr>
<tr>
<td>physical memory</td>
<td>100</td>
<td>&lt; 2 GB</td>
</tr>
<tr>
<td>disk</td>
<td>20,000,000</td>
<td>&gt; 10 GB</td>
</tr>
</tbody>
</table>

[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

![Diagram of memory management system]
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’)