

**COMP2300**

- Home
- StudyAt
- Assessment
- Schedule
- Lecture Notes
- Tute / Labs
- Assignments
- Announcements
- Discussion
- Reading material
- Old Exams
- Links
- PeANUt@Home
- Getting Help

Quick Links

- ANU Home
- FEIT Home
- DCS Home
- Search

**COMP2300****Tutorial / Laboratory 06 - PeANUt Assembler****Semester 1, 2008****Week 7 (07 - 11 April)**

Note that for this session, there is a submitable laboratory exercise which is due by 09 am Tuesday 15 April, which will contribute up to 1% of your assessment (in the Tute/Lab mark). As well as the formal Preparation Exercises, *it is particularly important that you revise arrays and functions in assembly language, as this will be a busy session!*

Objectives

There are several objectives in this session:

- To become familiar with the PeANUt assembly language, and how to translate control structures, arrays and functions into assembler.
- To use image files made of several source files, and using the command line assemble and join commands to efficiently create them.
- To learn the general debugging technique of **break points**.
- To learn how functions are implemented on the assembly language level, and how the stack is used in a function call.

Preparation Exercises

Complete the following questions on a separate sheet of paper, with your name and student number clearly written. Please ensure your writing is legible. Hand in to your tutor at the *beginning of your tutorial / laboratory session*.

Translate the following code into PeANUt assembly language:

```
1. int x;
2. char c;
3. int y = 1;
4. x = y + 1;
5. printf("%c", c);
```

Tutorial Exercises

1. Allocate space in PeANUt assembly language for the following:

```
float x; int a[4];
```

2. Translate the following C code into assembler:

```
#define N 4
int i,
    x,
    a[N];
i = 0;
x = 0;
while (i < N) {
    if (a[i] > 0) {
        x += a[i];
    } else {
        x -= a[i];
    } // if
    i++;
} // while
```

3. Consider the following C code fragment:

```
int exp(int x) {
    int v = 1;
    while (x > 0) {
        v = v * 2;
        x--;
    } // while
    return (v);
} // exp()
...
int twox;
twox = exp(3);
```

Assuming the procedure call convention as defined in lectures:

1. Translate the call `twox = exp(3);` into assembler.
2. Define stack offsets for the parameters, return value and local variable of `exp()`.
3. Translate the function `exp()` into assembler.

Laboratory Exercises

It may be useful to print out these pages if possible, as the instructions on the use of the PeANUt command line utilities may be useful for Assignment 2. Also, it may help you do the exercise more efficiently.

Preliminaries

1. In your `comp2300` directory, create a new sub-directory called `lab6`.
2. Copy the files `/dept/dcs/comp2300/public/lab6/*` into your `lab6` directory.
3. Start up a terminal window and `cd` to your `lab6` directory. Then start up the PeANUt simulator.

PeANUt via the Command Line

The `assemble` and `join` commands may be used to assemble and link PeANUt assembly language programs, as described below. It is also possible to execute PeANUt image files (whether produced by assembler or `mli` files) from the command line, e.g. from last week's lab:

```
execute lab5b.img
```

which is like executing a C program from the command line. To debug, use:

```
execute -trace lab5b.img
```

which prints out a (decimal-oriented) *trace* of the execution of each instruction. The command:

```
mli2img lab5b.mli
```

can be used to produce the image file from the command line. To assemble and link and a single stand-alone assembly language program, such as `lab6a.ass`, use the compound command:

```
assemble lab6a.ass ; join lab6a.rel
```

The `assemble` command produces a *relocatable file* `lab6a.rel`, and a listings file `lab6a.lst`. The `join` command takes the relocatable file and produces the *image file* `lab6a.img`. You will find that using the above command much more efficient than performing the same process via the PeANUt simulator (especially since you can use the `UpArrow` key to re-execute previously typed commands).

In the assembly language version of the program `lab6a.ass`, the address of `msg` and `main` (see the *listing* file `lab6a.lst`) will be generated during the assembly process. Look at `lab6a.lst` and determine which addresses these labels are at.

Documenting your assembler program with equivalent code in a higher level language (in this case C) makes it easier to read.

Repetition and Indexing

The **index register**, denoted by **XR**, may be used to access array elements. In program with a simple loop which scans across the elements of an array, we can often use **XR** to implement the loop index. You are to complete `lab6a.ass` by manipulating the the **index register** and using **indexed mode** addressing. i.e. if **XR** holds the value of the index `i`, then `load *msg` will load into **AC** the contents of `mem[msg+XR]`. Note also that **XR** may be incremented using the `incxr` instruction (see the table in PeANUt section 4.2.1.2).

* Add the missing instructions to `lab6a.ass`, and test it. When you have it working, comment out the short message and uncomment the long one. Show your program printing the long message to your tutor.

The InOut module

Inspect the source code of `InOut.ass` through the hyperlink in [lecture P8](#). This module contains several macros and some functions for printing and reading integers. Some macros should be familiar to you from the lectures. Inspect the source to see what functions are available, and note what C stdio library call these correspond to (for example, `ReadCard(&n) = scanf("%u", &n);` the `"%u"` means you can't use `ReadCard()` to read a negative integer!). The corresponding relocatable