

COMP2300

Tutorial / Laboratory 06 - PeANUt Assembler

Semester 1, 2007

Week 7 (2 - 6 April)

Draft!

In particular the unexpected problem of `assembler/join` commands ceasing to work on iwaki is not yet resolved. The interim workaround is to use these commands on a Linux host instead (and not use the **Assemble** and **Join** buttons on the PeANUt simulator)

Note that for this session, there is a submitable laboratory exercise which is due by 10 am Monday 23 April (week 8), which will contribute up to 1% of your assessment (in the Tute/Lab mark). While there is no formal Preparation Exercises, *please try to do as many of the Tutorial Questions (especially 1. to 4.) beforehand, 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 function 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.

Tutorial Exercises

1. Allocate space in PeANUt assembler for the following:

```
char c; int i; float x; int a[4];
```

2. Explain why `load #530` is illegal, and what problems are encountered in the instruction sequence `load #500` followed by `mul #500`.
3. What is value of AC after executing (if `mem[x] = 59`, `SP = 70`, `XR = 5`):

```
load    #15
mul     #4
store  !-10
dvd    #3
store  *54
load   x
sub    *55
add    #40
```

4. 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];
    i++;
}
```

5. Consider the following C code fragment:

```
int exp(int x) {
    int v = 1;
    while (x > 0) {
```

```

        v = v * 2;
        x--;
    }
    return (v);
}
...
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 be 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.

Important:

Peanut is currently only available on the student Solaris server - please ssh logon to 'iwaki' and start Peanut in a terminal window with:

```
ssh -X iwaki
Peanut &
```

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 *trace* of the execution of each instruction. The command:

```
mli2img lab5b.mli
```

can be used to produce the image file from the command line (only available currently on Linux hosts).

Assembly Language via the PeANUt Simulator

Section 4 (pages 14 to 18) of the **PeANUt Specifications** will help you understand the contents of the assembly language file. Note that assembler directives are used to create space for variables, and that these are effectively global variables.

Also, note the different prefixes (`@`,`#`,`!`,`*`) used on instruction operands to indicate the mode of the instruction - see page 16 of the PeANUt Specifications for an explanation.

In the assembly language version of the program `lab6a.ass`, the address of the next character to be printed is in the variable `i` which is stored at `a0121` (see the *listing* file `lab6a.lst` which will be generated during the assembly process. This location is automatically allocated to the variable by the assembly and linking process). The actual ASCII characters to be displayed are in the block of memory associated with `message` which begins at `a2`.

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

To assemble and run the assembly language program `lab6a.ass` you need to:

1. Click on the **Assemble** button and select which file to assemble. If there are no errors, this will produce a corresponding relocatable file (e.g. `lab6a.rel`).
2. Click on the **Join** button and a pop-up window will appear. Select the files to join, adding each to the **Files to Join:** list by using the `>>` button. In this case there is only one file needed to build the executable (`.img`) - this is `lab6a.rel`. Once it is in the list, click on **Join** in the pop-up window, if there are no errors the file `lab6a.img` should be created.
3. Once an `.img` file has been created, it can be loaded and executed in the normal manner.

Repetition and Indexing

There is another way of accessing a sequence of data. Instead of storing the location of the next character in a memory location, the **index register**, denoted by **XR**, may be used. You are to rewrite `lab6a.ass` using the **index register** and **indexed mode** instead of **indirect mode**. Have a look at the table in the upper part of page 4 of your PeANUt manual to remind yourself of how indexed mode works, and follow the following recipe:

- Store the *index* of the first character to be printed in the **index register** (from the table in PeANUt section 4.2.1.2 and Table 1 (page 5) you can see that `loadxr` and `storexr` moves data between the **AC** and **XR** registers).
 - Load the character using **indexed mode**, with the **opspec** being the address of the first character, `message`. The data will be read from `mem[message+XR]`.
 - Increment the index register (Use `incxr`).
 - Compare the contents of the index register with the index of the last character to be printed. For example, if you want to only print 50 characters, compare **XR** with 50.
 - If the **XR** has not reached the index of the last character to be printed, branch back, repeating the above steps until the required number of characters are printed.
1. * Make the file `lab6b.ass` by modifying a copy of `lab6a.ass`. It should use the **index register** to load the characters. Test `lab6n.ass`, and when you have it working, show it to your tutor.

InOut.ass

Start up your favourite text editor and load the file `InOut.ass`. This module contains several macros and some functions for printing and reading integers. Some macros should be familiar to you from the lectures.

Have a look at the code, and try to understand how it works. Do not spend too much time with this though (maximum 15-20 minutes). We will use the functions provided by `InOut.ass` in this lab as well as in assignment 2.

Assemble `InOut.ass` into the relocatable file `InOut.rel` (see the **PeANUt Specifications**, chapter 4 and appendix F) with the `assemble` shell command:

```
assemble InOut.ass
```

Check with `ls` if the file `InOut.rel` has been produced.

Functions with Return Values

In this part you are required to complete the code in the main program `findmax.ass` (step 1 below) and the function `max.ass` (steps 8 to 11 below). In the main program you have to complete the function call (using the **procedure call convention** as presented in the lectures), while in module `max.ass` you have to complete the function `Max` following the C comments.

1. Load the program `findmax.ass` into your text editor and add the code for the call to `Max`. Be sure to keep to the procedure call convention as discussed in the lectures.
2. Assemble `max.ass` and `findmax.ass` into relocatable files first (see below), then - using the `join` command - link them (again, see the **PeANUt Specifications**, chapter 4 and appendix F) with the `InOut.rel` module into one image file `max.img`. You have to use **command line** (shell) commands for this (using the graphical PeANUt tool does not work):

```
assemble max.ass
assemble findmax.ass
join max.rel findmax.rel InOut.rel
```

It might seem more logical to link `findmax.rel` first (the resulting executable will then be called `findmax.img`). However, linking `max.rel` first makes the debugging of `Max()` easier, since its code will be easily located at the top of memory.

3. Load and run `max.img` in the PeANUt simulator, entering a line with a single number (e.g. 42), followed by 999 (the sentinel value).
4. Set a **break point** at `call 0 (call Max)` instruction.

This can be done by selecting the instruction in the **Main Memory View** and using the right mouse button on that view's menu to select **Breakpoint Here**. Now the **Breakpoints** window will be displayed, with the break point set on `execute` mode at this address.

5. Try another run, this time with a smaller number, 7. Examine the stack at the break point and check to see that the parameters have been pushed on correctly. Now resume execution.
6. If you have linked your files in the correct order (as above), and done the call properly, a 3 should always be output at this point. Try to explain why this occurs.
Hint: run the program again, (single) **stepping** execution from the **Break Point**, and note the contents of the stack and memory location for `mxn` at each step.
7. When you are satisfied with this, check your work with your tutor.
8. Now load the file `max.asm` into your text editor.
9. What stack displacements should be given to the input parameters and return value?
10. Add the (concise) macros (read the **PeANUt Specifications**, page 17 if you don't know what a concise macro is) to define the stack displacements for the function's parameters and return value.
11. Add the code in the body of `Max()`. Use only one `ret` instruction at the end of the function.
12. Re-assemble, link and load `max.img`.

In general, you must go through *all 3 steps* (`assemble`, `join` and **Load IMG**) to propagate any change to a source file through to the PeANUt machine.

13. Repeat step 3 with an input of a single number 42 (followed by the sentinel 999). Set break points at the beginning and end of the function and check that the expected values appear at the slots corresponding to each parameter and return value. Note also the **Memory View** tracing on the **SP**.
14. Repeat step 3 with an input of two numbers in ascending order.
15. Execute `max.img` again with an input of two numbers in descending order, e.g. 42 40. If this works, try a line of several numbers in random order as a final test. Again, set break points if needed (you may need to use the **Breakpoints** window to clear the previously set break points first).
16. Demonstrate your working program to you tutor.
17. When you are satisfied that you have your program working, test it for correctness with the command:

```
previewAutoMark lab6 findmax.asm max.asm
```

Note that the `previewAutoMark` command is only available on DCS Linux hosts, i.e. do not try to run this command from `iwakisubmit` command:

```
submit comp2300 lab6 findmax.asm max.asm
```

This is due by 10 am Monday April 02 (the deadline is strict) and will contribute up to 1% of your Tute/Lab mark.

Last but not Least:

We would like to encourage you to finish ALL labs and tutes exercises before or during the semester break - especially if you so far have mainly worked on the assignment. Assignment 2 will require you to program in PeANUt assembly language, and having done all labs will help you a lot when working on the second assignment. If you have questions about any of the home works or labs please ask you tutor now!

Last modified: Mon Apr 2 09:42:20 EST 2007