Tutorial Sheet 13 – Procedures and Pointers

Setup
Pen and paper + computer-based exercises

Review Problems

1. *(Warning! Bad code ahead! Don't use the following code as an example; let's fix it instead.)*
The following assembly language code fragment is supposed to be equivalent to `x := 2+y` but it is poorly written. Find as many things wrong with it as you can, and indicate whether each fault is poor style or would cause error messages or runtime errors. Rewrite the code fragment as it should be.

```
      load  R1,x[R0]
  load  R2,2[R0]
      load     R3,y
add   R4,R2,R3 ; R4 := R2+R3
   store x[R0],R4
```

2. Translate each of the following instructions into machine language. Assume that the memory address of x is 00c3 and the memory address of y is 00f8.
Quick review:

   - The format of a RRR instruction is op d a b, and opcode of add is 0, opcode of sub is 1.
   - The format of an RX instruction is two words:
     (a) op d a b where op = Hex f, d is the register operand, a is 0 (for the [R0]), and b is the code indicating which instruction: 0 for lea, 1 for load, 2 for store.
     (b) A constant, which might be specified in the instruction (e.g. 42) or might be the memory address of a variable (e.g. x).

```
add     R3,R9,R4
sub     R2,R12,R1
load    R8,x[R0]
lea     R9,42[R0]
store   R10,y[R0]
```

3. Give a reason why it's useful for the machine to guarantee that R0 always contains 0.

4. Describe what the following program fragment does, and translate it to assembly language.

```
p := &x
q := p
*q := *q + 1
```

5. Translate the following high level code into low level form.

```
while x<y do
   { S1
     if x=p
        then { S2 }
        else for i := x to y
                { S3 }
                { S4 }
    { S5 }
{ S6 }
```

6. Suppose you forget to terminate the execution of your program with `trap R0,R0,R0`. Explain what would happen. Don't say *exactly* what will happen — that requires knowing the exact contents of memory — but explain in general terms what will happen.

7. Suppose that you put the data statements defining your variables (and giving their initial values) at the beginning of a program, rather than at the end where they belong. Explain what would happen (just in general terms).

Study an example program: zapR13crash

Download the program `zapR13crash.asm.txt`, which contains several functions and several function calls. There is no call stack: each function is called with a simple jal instruction and returns with a simple jump instruction. Hand-execute this program, and then run it. It's important to single-step the program, don't just run it at full speed.

You should find that the program works fine as long as a called function never calls another function. The program successfully stores result1. However, when there is a nested call (main calls mult6, which calls double) the first return address gets destroyed by the second call, and the program is unable to return properly. Consequently, the program is unable to store result2.

The point of this program is that we need a stack to save the return addresses!

Write a program: saveR13stack

Write a program, *saveR13stack*, which is similar to zapR13crash, except that it uses a stack to save and restore the return addresses.

To get started, make a copy of *zapR13crash.asm.txt* and give it the file name `saveR13stack.asm.txt`. We are using the same algorithm, but just fixing up the function calls and returns. Make the following modifications to the program:

- We will allocate the stack in the data area at the very end of the program:

```
result1    data  0            ; result of first sequence of calls
result2    data  0            ; result of the mult6 call
CallStack  data  0            ; The stack grows beyond this point
```

- The main program initializes the stack as follows:

```
    lea    R14,CallStack[R0]  ; R14 := &stack, R14 is the stack pointer
    store  R0,0[R14]          ; (not actually necessary)
```

- A function is called as follows:

```
    (put the argument into R1)
    lea    R14,1[R14]         ; advance the stack pointer to new frame
    jal    R13,function[R0]   ; goto function, R13 := return address
```

- Every function begins as follows:

```
    store  R13,0[R14]         ; save return address on top of stack
```

- Every function finishes and returns as follows:

```
    (put the result into R1)
    load   R13,0[R14]         ; restore return address from top of stack
    lea    R2,1[R0]           ; R2 := size of stack frame
    sub    R14,R14,R2         ; remove top frame from stack
    jump   0[R13]             ; return to caller
```

After you have made all the function calls and returns follow these conventions, hand-execute the program, and step through it with the Sigma16 system. This time the program should successfully store both result1 and result2.

Study an example: recursive factorial

In learning to program, it's important to read and understand existing programs. Don't ignore examples and just start from scratch!

The recursive factorial program illustrates how to write procedures. This program has a static variable x, and it computers x! (i.e. x factorial).

The program includes full activation records (also called stack frames), with return addresses, links, and saved registers. The program illustrates basic call and return (jal and jump), records (an activation record is a record!), accessing record fields (e.g. to save and restore the registers), and pointers.

Study the program. Note that the structure of the stack frame is defined in comments. Pay attention to how the factorial function is called, how it creates its activation record, and how it returns. Try stepping through the program. It's a good idea to draw a diagram of the stack on paper, and to update this diagram as you step through the program. You can try running the program with different initial values of x.

You don't need to do any new programming now; the aim is to understand how procedures work, as well as records and pointers.