Subroutines and Stacks – notes for Week 6 (Lectures 9-10).

- Think of *routine* as meaning a method in Java, so *subroutine* means a method called by some other method or methods (which happens all the time, of course).
- When a subroutine is executing, somewhere there has to be stored a *return address*, to show where to jump back to, what value the pc should be given, when the subroutine is finished. But where will the return address be stored?
  - If you use a single fixed memory location, then the subroutine cannot be *recursive* - it cannot call itself.
  - A *stack* is a clever way to use a part of memory so that the return address can be stored in a location that's not already in use for another return address. This allows subroutines to be recursive. [There is a whole hidden topic here, of recursion – more on that later.]
- There are two basic operations on a stack.
  - *push* stores a value on the stack
  - *pop* removes a value from the stack (and stores it somewhere else)
- We complete the Toy CPU with an extra sp register (*stack pointer*) and four operations *push*, *call* (call a subroutine), *pop* and *ret* (return from subroutine).
  - sp shows where in memory to push (as a write) or pop (as a read). It is updated after each push or pop.
  - *Call* and *ret* do the same as *push* and *pop*, but are distinguished from them by using the pc as register: *call* pushes the pc and *ret* pops a value back into the pc.
  - Also, *call* has an operand (the address of the subroutine), which *push* does not.
- Along with the return address, it is also often useful to save register values on the stack - see Exercise 2 below. This means that the calling routine does not have to worry about what the subroutine does with the registers. This bundle of saved data from the calling routine, consisting of return address and saved register values, leads to the idea of *stack frame*, which we shall see later in the Java Virtual Machine.

- Although we introduced stacks as a solution to the problem of return addresses (and saved registers), they are also very useful for evaluating expressions. Values that arise in the different stages of the calculation can be stored on the stack.
- *Reverse Polish notation* is an alternative notation for writing expressions (algebraic formulas).
  - The general principle is that for any function (add, subtract, ..., any method) you first say what its parameters are, and then you say what the function is. For example: 2 2 + instead of 2+2.
  - With a stack, you first get the parameters on the stack, and then apply the function. It pops the parameters and pushes the result.
  - Reverse Polish is used explicitly in some old Hewlett-Packard calculators, and some programming languages, such as Forth.
  - Often (e.g. Forth) you end up with two stacks: an *operand* stack for calculations, and a *return* stack for return addresses.
- Stack-based calculations can be used for a different style of CPU architecture, with an operand stack instead of registers. This is also how it is done in Java bytecode, which we shall see in subsequent weeks. In preparation for that, an outline how this changes the nature of machine code instructions is as follows:
  - You will see stack operations called **add**, **sub**, **neg**, **mul**, **div**, **rem**, **or**, **and**, **xor**, **push**, **load**, **ifeq**, **if_cmpeq**, ... You don't have to learn these by heart, but it helps to familiarize yourself with them as they will also be used later in Java bytecode.
  - There is a special notation for describing the action of an operation on the stack. For

example, for **add** the action is "..., val1, val2 --> ..., val1+val2". It pops off the top two entries, adds them, and pushes the result.

- There is an algorithm, Dijkstra's Shunting-Yard Algorithm for converting ordinary notation into reverse Polish, and you will use this to solve Exercises 3 and 4.

Exercises

1. Write mnemonics and machine code (for the Toy CPU) for the following tasks:

a) A subroutine, starting at location 10 (hex) that sums memory locations, then stores the result after them. Assume that the calling routine initialised the register b with the address of the memory location of the summands, and the register c with the number of them. Don't forget the return.
b) A main routine, starting at location 00, that calls the subroutine twice: once for summing 4 locations starting at 2A, and once for summing 6 locations starting at 38.

2. Your answer to 1 (a) will have used the a, b and c registers in its own calculations. Suppose this is inconvenient for the main program and it would be better not to lose the values that those three registers had when the subroutine was stored. There is a trick for doing this - the stack doesn't have to be used just for saving return addresses. Rewrite the subroutine so that it pushes the three registers at the start, saving their values on the stack, and then pops them back at the end so that the calling routine gets them back unchanged. Make sure you pop them in the right order!

3. Convert some ordinary expressions into reverse Polish. Run them through with a stack to check they calculate the correct answer.

a. 3+4
b. 2*3+4
c. 2+3*4
d. SQRT(3*3+4*4) (SQRT is the square root function)

4. See if you can follow through the shunting-yard algorithm (on Wikipedia) for the example in the slides: (5+2)*SQRT(x*x+y*y)+8.

Resources

Stacks are briefly described in [1] on p.59. This reference also describes the "non-recursive" style of subroutine call on p.163, with a fixed memory address to store return addresses; but remember that this is too limiting in practice.

The Wikipedia pages on reverse Polish notation (https://en.wikipedia.org/wiki/Reverse_Polish_notation) and the Forth programming language (https://en.wikipedia.org/wiki/Forth_%28programming_language%29) gives some extra insights into the use of stacks.

Dijkstra's Shunting-Yard Algorithm : https://en.wikipedia.org/wiki/Shunting-yard_algorithm

These notes are adapted from Steve Vickers' lecture notes (2015).

References

[1] Goldschlager and Lister: *Computer Science: a Modern Introduction* (Prentice Hall, 2nd edition 1988)

[2] Reynolds and Tymann: *Principles of Computer Science* (Schaum's Outlines, McGraw-Hill 2008)