Exploring AArch64 assembler – Chapter 7

Mar 19, 2017 • Roger Ferrer Ibáñez • AArch64

In the previous installment of this series we saw how to alter the sequencing of our programs. Today we will see how we can reuse instructions by means of branches. Let's talk about functions.

Routines

In the process of solving a problem with a computer we will encounter that some of its steps are repeatedly performed. These steps, probably part of an algorithm, can be encoded using instructions. This means that we may end with many sequences of instructions whose purpose is the same. What if we could *factor out* these instructions in a single place, and use them when needed. This is the fundamental idea behind a routine. We rarely use the word routine nowadays and most programming languages use other names like functions, procedures, subroutines, methods, lambda expressions, etc. Sure, there are differences between them but all of them encompass the idea of *reusing code*. I will use the common term “function”.

Using a function

Functions can be handled like values (i.e. like an integer value) and there are a few operations we can do with them. At this level, though, we will only care about two of them: getting the address of the function, which we will usually achieve in a trivial way just using a label, and *calling* the function, which is the interesting bit.

Address of a function

Functions will be a sequence of instructions that we will reuse. We will identify functions using the address of their first instruction. Most of the time a label will be used to designate the function as, recall, a label is an address.

Calling a function

Calling a function is a process that in practice means passing some data to the function, branching to the address of the function. When the function ends, it will branch back to the caller.

A call to a function is at its core a branch, but it is a special branch that is so frequent that it is worth to devote an instruction only for it. In AArch64 this instruction is `bl` which means something like *branch and link*. It is an unconditional branch that performs what an unconditional branch does plus it sets to `x30` to the address of the next instruction after `bl`. Recall that `x30` is a general purpose register but in this case we're giving it a special meaning: it contains the address where the function must branch back when it ends. For historical reasons, when `x30` is used with this purpose it is called the *link register*.

To return from a function, then, the only thing we have to do is just branch to the value of `x30`. There is an instruction to unconditionally branch to the address stored in a register called `br`. So a way to call a function and just return from it is the following.

```
.text
my_function:
  br x30
caller:
  bl my_function
  // more instructions ...
```

But returning from a function is also a very common operation so rather than doing `br x30` we can just do `ret`.

```
.text
my_function:
  ret
caller:
  bl my_function
  // more instructions ...
```

Parameter passing

Ok, so we know the basics of calling a function and returning from it. But, this way, the only thing we achieve is just reuse sequences of instructions that will always do the same. Many times we will want to parameterize the behaviour of the function: for instance we want a function that computes the average of two given numbers. This means that when calling the function we need a mechanism to pass those parameters to the function.

At this point we can consider several approaches. A first approach is using global variables where we will first put the values and the function will read those values and leave the result in some other global variable. This works, kind of, and some early programming languages worked this way. The problem is that this mechanism composes badly because it bars using recursion and even worse, it does not work in multithreaded environments. In a modern setting this approach is rarely used.

Another alternative, is, well, we can use a *private memory* that we use only for calling functions. And we give it some kind of *stack discipline*: we can put or remove things only on its top (but we can access the i-th element below the top of the stack). Before we call a function we put the arguments on the top of the stack. The function can access the top of the stack and the elements below it to retrieve the parameters. The result can also be put in the stack, for instance, the function can replace the parameters with the result value, so the caller only has to check the top of the stack again. This technique works well for recursion and also for multithreading. This is a technique used in some programming languages and in architectures with a severe shortage of registers (like 32-bit x86).

A hybrid approach, the common used in RISC architectures, involves devoting a few registers to pass parameters and if we run out of them, use a stack-like approach like the one described above. This works well because in general most function calls involve a few parameters, usually less than 4 or so. And given that in AArch64 we have around 30 registers, it makes sense to devote a few of them as the parameter passing. Which ones? Well, this is a conventional thing, and as a convention it should be agreed upon first.

We could use any convention (even made our own) but in AArch64 there is one already described in Procedure Call Standard for the ARM 64-bit Architecture (or for short the PCS). That is a very long document with lots of details. For the purpose of this series we will simplify the convention as follows:

- Registers `x0-x7` are used to pass parameters and return values. The value of these registers may be freely modified by the called function (the callee) so the caller cannot assume anything about their content, even if they are not used in the parameter passing or for the returned value. This means that these registers are in practice *caller-saved*.
- Registers `x8-x18` are temporary registers for every function. As such no assumption can be made on their values upon returning a function. In practice these registers are also *caller-saved*.
- Registers `x19-x28` are registers, that, if used by the function, must have their values preserved and later restored upon returning the function. This registers are known as *callee-saved*.
- We already know that register x30 is the *link register* and its value must be preserved until the function uses the `ret` instruction to return to the caller.

This means that we can pass up to 8 parameters in registers `x0` to `x7`. If we are passing an integer of 64-bit or an address we will use the corresponding `x`*`i`* register. For 32-bit integers we will use the corresponding `w`*`i`* (we won't bother packing two 32-bit integers in a single 64-bit register).

What if we have to pass more than 8 parameters? How do we keep registers `x19` to `x28` and more importantly, how do we keep `x30`? Well, in this case we will have to use the stack, but we will leave this for another chapter. In this chapter we will use a global variable to temporarily store `x30`.

Say hello!

Ok, equipped with this knowledge, we can now start doing some interesting examples. As a starter for today we will say hello. We can use the function `puts` of the C library for this. This function only receives a parameter: an address to a null-ended string.

```
 1 .data
 2
 3 .balign 8
 4 /* This is the greeting message */
 5 say_hello: .asciz "Hello world!"
 6
 7 .balign 8
 8 /* We need to keep x30 otherwise we will not be able to return from main! */
 9 keep_x30: .dword 0
10
11 .text
12
13 /* We are going to call a C-library puts function */
14 .globl puts
15
16 .globl main
17 main:
18     ldr x0, addr_keep_x30     // w0 ← &keep_30   [64]
19     str x30, [x0]             // *keep_30 ← x30  [64]
20
21     ldr x0, addr_say_hello    // w0 ← &say_hello [64]
22     bl puts                   // call puts
23
24     ldr x0, addr_keep_x30     // w0 ← &keep_30   [64]
25     ldr x30, [x0]             // x30 ← *keep_30  [64]
26
27     mov w0, #0                // w0 ← 0
28     ret                       // return
29
30 addr_keep_x30 : .dword keep_x30
31 addr_say_hello: .dword say_hello
```

In line 3 we make sure the next data emitted by the assembler is aligned to an address multiple of 8 bytes (64-bit). In this case we want the assembler to emit a null-ended string of the characters `Hello world!`. We can use the directive `.asciz` for this (line 5). In order to be able to use this string later we set the label `say_hello` which will be the address of such string.

Since we will not see how the stack is used in this chapter, we still need to save the value of `x30` somewhere. So we allocate some storage for it. Again we want this storage to be aligned to 8 bytes, so we use a `.balign` directive again (line 7). Then we define the storage itself and we label it as `keep_x30`, so we can refer to it later. As we know, `.dword` directive will emit the specified integer value as a 64-bit integer. That's all for the data section of our small program.

In line 14 we say that we are going to use the symbol `puts`. This is the name of a function defined in the C library, so we use `.globl` (line 14) to state that this is a global symbol (in contrast of a private one). As we already know, we need to do the same for main (line 17).

Now check lines 30 and 31. Here we define storage that will contain the addresses of `say_hello` and `keep_x30`. As you recall from chapter 5, this is because we need to keep the addresses close to the load instruction.

Now back to line 18, here we load in `x0` the address of `keep_x30`. Now we can use this address to store the register `x30`, line 19.

Now that we have kept x30, we can call `puts`. First we need to prepare the function call following the convention described above. Function `puts` is a function in the C-library that only receives an address to a null-ended buffer of bytes. Precisely what we have in `say_hello`. As puts receives the address, not the contents themselves, we will use `addr_say_hello` instead. As described above the first parameter is passed in `x0`, so we just load the address of `say_hello` (that as said we have in `addr_say_hello`) in `x0`, line 21.

Now everything is in place, then we make the call to `puts` in line 22. If all is correct our program will continue in the next instruction of the call, line 24. Here we simply restore the value of `x30`, as the instruction `bl` in line 22 overwrote it. Basically we load again the address of `keep_x30` and we do a load with that address to the `x30` register lines 24-25. Now everything is in place to return, so we set `w0` to 0, line 27, and we return using `ret`, line 28.

If we try to run this program we will be greeted.

```
$ ./hello
Hello world!
```

Yay! :)

This is all for today!

« Compilation of array expressions in Fortran

Whose is this optimization? »

Powered by Jekyll. Theme based on whiteglass
Subscribe via RSS