Dynamic activation

One of the benefits of functions is being able to call them more than once. But that more than once hides a small trap. We are not restricting who will be able to call the function, so it might happen that it is the same function who calls itself. This happens when we use recursion.
A typical example of recursion is the factorial of a number n, usually written as n!. A factorial in C can be written as follows.

```
int factorial(int n)
{
    if (n == 0)
        return 1;
    else
        return n * factorial(n-1);
}
```

Note that there is only one function `factorial`, but it may be called several times. For instance:factorial(3) → factorial(2) → factorial(1) → factorial(0), where → means a «it calls». A function, thus, is dynamically activated each time is called. The span of a dynamic activation goes from the point where the function is called until it returns. At a given time, more than one function is dynamically activated. The whole dynamic activation set of functions includes the current function and the dynamic activation set of the function that called it (the current function).
Ok. We have a function that calls itself. No big deal, right? Well, this would not be a problem if it weren't for the rules that a function must observe. Let's quickly recall them.

- Only `r0`, `r1`, `r2` and `r3` can be freely modified.
- `lr` value at the entry of the function must be kept somewhere because we will need it to leave the function (to return to the caller).
- All other registers `r4` to `r11` and `sp` can be modified but they must be restored to their original values upon leaving the function.

Previously, we used a global variable to keep `lr`. But if we attempted to use a global variable in our factorial(3) example, it would be overwritten at the next dynamic activation of factorial. We would only be able to return from factorial(0) to factorial(1). After that we would be stuck infactorial(1), as `lr` would always have the same value.
So it looks like we need some way to keep at least the value of `lr` per each dynamic activation. And not only `lr`, if we wanted to use registers from `r4` to `r11` we also need to keep somehow per each dynamic activation, a global variable would not be enough either. This is where the stack comes into play.

The stack

In computing, a stack is a data structure (a way to organize data that provides some

interesting properties). A stack typically has three operations: access the top of the stack, push onto the top, pop from the top. Depending on the context you can only access the top of the stack, in our case we will be able to access more elements than just the top.

But, what is the stack? The stack is a region of memory owned solely by the function. We can now reword this a bit better: the stack is a region of memory owned solely by the current dynamic activation. And how we control the stack? The register `sp` stands for stack pointer. This register will contain the top of the stack. The region of memory owned by the dynamic activation is the extent of bytes contained between the current value of `sp` and the initial value that `sp` had at the beginning of the function. We will call that region the local memory of a function (more precisely, of a dynamic activation of it). We will put there whatever has to be saved at the beginning of a function and restored before leaving. We will also keep there the local variables of a function (dynamic activation).
Our function also has to adhere to some rules when handling the stack.

- The stack pointer (`sp`) is always 4 byte aligned. This is absolutely mandatory. However, due to the Procedure Call Standard for the ARM architecture (AAPCS), the stack pointer will have to be 8 byte aligned, otherwise funny things may happen when we call what the AAPCS calls as public interfaces (this is, code written by other people).
- The value of `sp` when leaving the function should be the same value it had upon entering the function.

The first rule is consistent with the alignment constraints of ARM, where most of times addresses must be 4 byte aligned. Due to AAPCS we will stick to the extra 8 byte alignment constraint. The second rule states that, no matter how large is our local memory, it will always disappear at the end of the function. This is important, because local variables of a dynamic activation need not have any storage after that dynamic activation ends.

It is a convention how the stack, and thus the local memory, has its size defined. The stack can grow upwards or downwards. If it grows upwards it means that we have to increase the value of the `sp` register in order to enlarge the local memory. If it grows downwards we have to do the opposite, the value of the `sp` register must be substracted as many bytes as the size of the local storage. In Linux ARM, the stack grows downwards, towards zero (although it never should reach zero). Addresses of local variables have very large values in the 32 bit range. They are usually close to 2^32.
Another convention when using the stack concerns whether the `sp` register contains the address of the top of the stack or some bytes above. In Linux ARM the `sp` register directly points to the top of the stack: in the memory addressed by `sp` there is useful information.
Ok, we know the stack grows downwards and the top of the stack must always be in `sp`. So to enlarge the local memory it should be enough by decreasing `sp`. The local memory is then

defined by the range of memory from the current `sp` value to the original value that `sp` had at the beginning of the function. One register we almost always have to keep is `lr`. Let's see how can we keep in the stack.

```
sub sp, sp, #8  /* sp ← sp - 4. This enlarges the stack by 8 bytes
*/
str lr, [sp]    /* *sp ← lr */
... // Code of the function
ldr lr, [sp]    /* lr ← *sp */
add sp, sp, #8  /* sp ← sp + 8. /* This reduces the stack by 8 bytes
                                    effectively restoring the stack
                                    pointer to its original value */
bx lr
```

A well behaved function may modify sp but must ensure that at the end it has the same value it had when we entered the function. This is what we do here. We first substract 8 bytes to sp and at the end we add back 8 bytes.

This sequence of instructions would do indeed. But maybe you remember chapter 8 and the indexing modes that you could use in load and store. Note that the first two instructions behave exactly like a pre-indexing. We first update `sp` and then we use `sp` as the address where we store `lr`. This is exactly a pre-index! Likewise for the last two instructions. We first load `lr` using the current address of `sp` and then we decrease `sp`. This is exactly a post-index!

```
str lr, [sp, #-8]!  /* preindex: sp ← sp - 8; *sp ← lr */
... // Code of the function
ldr lr, [sp], #+8   /* postindex; lr ← *sp; sp ← sp + 8 */
bx lr
```

Yes, these addressing modes were invented to support this sort of things. Using a single instruction is better in terms of code size. This may not seem relevant, but it is when we realize that the stack bookkeeping is required in almost every function we write!

First approach

Let's implement the factorial function above.

First we have to learn a new instruction to multiply two numbers: `mul Rdest, Rsource1, Rsource2`. Note that multiplying two 32 bit values may require up to 64 bits for the result. This instruction only computes the lower 32 bits. Because we are not going to use 64 bit values in this example, the maximum factorial we will be able to compute is 12! (13! is bigger than 232). We will not check that the entered number is lower than 13 to keep the example simple (I encourage you to add this check to the example, though). In versions of the ARM architecture prior to ARMv6 this instruction could not have `Rdest` the same as `Rsource1`. GNU assembler may print a warning if you don't pass `-march=armv6`.

```
1  /* -- factorial01.s */
2  .data
```

```
3
4  message1: .asciz "Type a number: "
5  format:   .asciz "%d"
6  message2: .asciz "The factorial of %d is %d\n"
7
8  .text
9
10 factorial:
11     str lr, [sp,#-4]!  /* Push lr onto the top of the stack */
12     str r0, [sp,#-4]!  /* Push r0 onto the top of the stack */
13                        /* Note that after that, sp is 8 byte aligned */
14     cmp r0, #0         /* compare r0 and 0 */
15     bne is_nonzero     /* if r0 != 0 then branch */
16     mov r0, #1         /* r0 ← 1. This is the return */
17     b end
18 is_nonzero:
19                        /* Prepare the call to factorial(n-1) */
20     sub r0, r0, #1     /* r0 ← r0 - 1 */
21     bl factorial
22                        /* After the call r0 contains factorial(n-1) */
23                        /* Load r0 (that we kept in th stack) into r1 */
24     ldr r1, [sp]       /* r1 ← *sp */
25     mul r0, r1, r0     /* r0 ← r0 * r1 */
26
27 end:
28     add sp, sp, #+4    /* Discard the r0 we kept in the stack */
29     ldr lr, [sp], #+4  /* Pop the top of the stack and put it in lr */
30     bx lr              /* Leave factorial */
31
32 .globl main
33 main:
34     str lr, [sp,#-4]!            /* Push lr onto the top of the stack */
35     sub sp, sp, #4               /* Make room for one 4 byte integer in the stack */
36                                     /* In these 4 bytes we will keep the number */
37                                     /* entered by the user */
38                                     /* Note that after that the stack is 8-byte aligned */
39     ldr r0, address_of_message1  /* Set &message1 as the first parameter of printf */
40     bl printf                    /* Call printf */
41
42     ldr r0, address_of_format    /* Set &format as the first parameter of scanf */
43     mov r1, sp                   /* Set the top of the stack as the second parameter */
44                                     /* of scanf */
45     bl scanf                     /* Call scanf */
46
47     ldr r0, [sp]                 /* Load the integer read by scanf into r0 */
48                                     /* So we set it as the first parameter of factorial */
49     bl factorial                 /* Call factorial */
50
51     mov r2, r0                   /* Get the result of factorial and move it to r2 */
52                                     /* So we set it as the third parameter of printf */
53     ldr r1, [sp]                 /* Load the integer read by scanf into r1 */
54                                     /* So we set it as the second parameter of printf */
55     ldr r0, address_of_message2  /* Set &message2 as the first parameter of printf */
56     bl printf                    /* Call printf */
57
58
```

```
59      add sp, sp, #+4              /* Discard the integer read by scanf */
60      ldr lr, [sp], #+4            /* Pop the top of the stack and put it in lr */
61      bx lr                        /* Leave main */
62
63  address_of_message1: .word message1
64  address_of_message2: .word message2
65  address_of_format: .word format
```

Most of the code is pretty straightforward. In both functions, `main` and `factorial`, we allocate 4 extra bytes on the top of the stack. In `factorial`, to keep the value of `r0`, because it will be overwritten during the recursive call (twice, as a first parameter and as the result of the recursive function call). In `main`, to keep the value entered by the user (if you recall chapter 9 we used a global variable here).

It is important to bear in mind that the stack, like a real stack, the last element stacked (pushed onto the top) will be the first one to be taken out the stack (popped from the top). We store `lr`and make room for a 4 bytes integer. Since this is a stack, the opposite order must be used to return the stack to its original state. We first discard the integer and then we restore the `lr`. Note that this happens as well when we reserve the stack storage for the integer using a `sub` and then we discard such storage doing the opposite operation `add`.

Can we do it better?

Note that the number of instructions that we need to push and pop data to and from the stack grows linearly with respect to the number of data items. Since ARM was designed for embedded systems, ARM designers devised a way to reduce the number of instructions we need for the «bookkeeping» of the stack. These instructions are load multiple, `ldm`, and store multiple, `stm`.

These two instructions are rather powerful and allow in a single instruction perform a lot of things. Their syntax is shown as follows. Elements enclosed in curly braces `{` and `}` may be omitted from the syntax (the effect of the instruction will vary, though).

```
ldm addressing-mode Rbase{!}, register-set
stm addressing-mode Rbase{!}, register-set
```

We will consider `addressing-mode` later. `Rbase` is the base address used to load to or store from the `register-set`. All 16 ARM registers may be specified in `register-set` (except `pc`in `stm`). A set of addresses is generated when executing these instructions. One address per register in the register-set. Then, each register, in ascending order, is paired with each of these addresses, also in ascending order. This way the lowest-numbered register gets the lowest memory address, and the highest-numbered register gets the highest memory address. Each pair register-address is then used to perform the memory operation: load or store. Specifying `!`means that `Rbase` will be updated. The updated value depends on `addressing-mode`.

Note that, if the registers are paired with addresses depending on their register number, it

seems that they will always be loaded and stored in the same way. For instance a `register-set`containing `r4`, `r5` and `r6` will always store `r4` in the lowest address generated by the instruction and `r6` in the highest one. We can, though, specify what is considered the lowest address or the highest address. So, is `Rbase` actually the highest address or the lowest address of the multiple load/store? This is one of the two aspects that is controlled by `addressing-mode`. The second aspect relates to when the address of the memory operation changes between each memory operation.

If the value in `Rbase` is to be considered the the highest address it means that we should first decrease `Rbase` as many bytes as required by the number of registers in the `register-set`(this is 4 times the number of registers) to form the lowest address. Then we can load or store each register consecutively starting from that lowest address, always in ascending order of the register number. This addressing mode is called decreasing and is specified using a `d`.

Conversely, if `Rbase` is to be considered the lowest address, then this is a bit easier as we can use its value as the lowest address already. We proceed as usual, loading or storing each register in ascending order of their register number. This addressing mode is called increasing and is specified using an `i`.

At each load or store, the address generated for the memory operation may be updated after or before the memory operation itself. We can specify this using `a` or `b`, respectively.

If we specify `!`, after the instruction, `Rbase` will have the highest address generated in the increasing mode and the lowest address generated in the decreasing mode. The final value of`Rbase` will include the final addition or subtraction if we use a mode that updates after (an `a`mode).

So we have four addressing modes, namely: `ia`, `ib`, `da` and `db`. These addressing modes are specified as suffixes of the `stm` and `ldm` instructions. So the full set of names is `stmia`, `stmib`,`stmda`, `stmdb`, `ldmia`, `ldmib`, `ldmda`, `ldmdb`. Now you may think that this is overly complicated, but we need not use all the eight modes. Only two of them are of interest to us now.

When we push something onto the stack we actually decrease the stack pointer (because in Linux the stack grows downwards). More precisely, we first decrease the stack pointer as many bytes as needed before doing the actual store on that just computed stack pointer. So the appropiate `addressing-mode` when pushing onto the stack is `stmdb`. Conversely when popping from the stack we will use `ldmia`: we increment the stack pointer after we have performed the load.

Factorial again

Before illustrating these two instructions, we will first slightly rewrite our factorial.

If you go back to the code of our factorial, there is a moment, when computing `n *`

`factorial(n-1)`, where the initial value of `r0` is required. The value of `n` was in `r0` at the beginning of the function, but `r0` can be freely modified by called functions. We chose, in the example above, to keep a copy of `r0` in the stack in line 12. Later, in line 24, we loaded it from the stack in `r1`, just before computing the multiplication.

In our second version of factorial, we will keep a copy of the initial value of `r0` into `r4`. But `r4` is a register the value of which must be restored upon leaving a function. So we will keep the value of`r4` at the entry of the function in the stack. At the end we will restore it back from the stack. This way we can use `r4` without breaking the rules of well-behaved functions.

```
10 factorial:
11     str lr, [sp,#-4]!  /* Push lr onto the top of the stack */
12     str r4, [sp,#-4]!  /* Push r4 onto the top of the stack */
13                        /* The stack is now 8 byte aligned */
14     mov r4, r0         /* Keep a copy of the initial value of r0 in r4 */
15
16
17     cmp r0, #0         /* compare r0 and 0 */
18     bne is_nonzero     /* if r0 != 0 then branch */
19     mov r0, #1         /* r0 ← 1. This is the return */
20     b end
21 is_nonzero:
22                        /* Prepare the call to factorial(n-1) */
23     sub r0, r0, #1     /* r0 ← r0 - 1 */
24     bl factorial
25                        /* After the call r0 contains factorial(n-1) */
26                        /* Load initial value of r0 (that we kept in r4) into r1 */
27     mov r1, r4         /* r1 ← r4 */
28     mul r0, r1, r0     /* r0 ← r0 * r1 */
29
30 end:
31     ldr r4, [sp], #+4  /* Pop the top of the stack and put it in r4 */
32     ldr lr, [sp], #+4  /* Pop the top of the stack and put it in lr */
33     bx lr              /* Leave factorial */
```

Note that the remainder of the program does not have to change. This is the cool thing of functions

Ok, now pay attention to these two sequences in our new factorial version above.

```
    str lr, [sp,#-4]!  /* Push lr onto the top of the stack
11 */
12     str r4, [sp,#-4]!  /* Push r4 onto the top of the stack
   */
       ldr r4, [sp], #+4  /* Pop the top of the stack and put it in r4
30 */
31     ldr lr, [sp], #+4  /* Pop the top of the stack and put it in lr
   */
```

Now, let's replace them with `stmdb` and `ldmia` as explained a few paragraphs ago.

```
11     stmdb sp!, {r4, lr}    /* Push r4 and lr onto the stack */
30     ldmia sp!, {r4, lr}    /* Pop lr and r4 from the stack */
```

Note that the order of the registers in the set of registers is not relevant, but the processor will

handle them in ascending order, so we should write them in ascending order. GNU assembler will emit a warning otherwise. Since `lr` is actually `r14` it must go after `r4`. This means that our code is 100% equivalent to the previous one since `r4` will end in a lower address than `lr`: remember our stack grows toward lower addresses, thus `r4` which is in the top of the stack in `factorial` has the lowest address.

Remembering `stmdb sp!` and `ldmia sp!` may be a bit hard. Also, given that these two instructions will be relatively common when entering and leaving functions, GNU assembler provides two mnemonics `push` and `pop` for `stmdb sp!` and `ldmia sp!`, respectively. Note that these are not ARM instructions actually, just convenience names that are easier to remember.

```
11      push {r4, lr}
30      pop {r4, lr}
```