Functions

Why functions?

Functions are a way to reuse code. If we have some code that will be needed more than once, being able to reuse it is a Good Thing™. This way, we only have to ensure that the code being reused is correct. If we repeated the code we should verify it is correct at every point. This clearly does not scale. Functions can also get parameters. This way not only we reuse code but we can use it in several ways, by passing different parameters. All this magic, though, comes at some price. A function must be a a well-behaved citizen.

Do's and don'ts of a function

Assembler gives us a lot of power. But with a lot of power also comes a lot of responsibility. We can break lots of things in assembler, because we are at a very low level. An error and nasty things may happen. In order to make all functions behave in the same way, there are conventions in every environment that dictate how a function must behave. Since we are in a Raspberry Pi running Linux we will use the AAPCS (chances are that other ARM operating systems like RISCOS or Windows RT follow it).

New special named registers

When discussing branches we learned that r15 was also called pc but we never called it r15 anymore. Well, let's rename from now r14 as lr and r13 as sp. lr stands for link register and it is the address of the instruction following the instruction that called us (we will see later what is this). sp stands for stack pointer. The stack is an area of memory owned only by the current function, the sp register stores the top address of that stack. For now, let's put the stack aside.

Passing parameters

Functions can receive parameters. The first 4 parameters must be stored, sequentially, in the registers r0, r1, r2 and r3. You may be wondering how to pass more than 4 parameters. We can, of course, but we need to use the stack, but we will discuss it next week. Until then, we will only pass up to 4 parameters.

Well behaved functions

A function must adhere, at least, to the following rules if we want it to be AAPCS compliant.

- A function should not make any assumption on the contents of the cpsr. So, at the entry of a function condition codes N, Z, C and V are unknown.
- A function can freely modify registers r0, r1, r2 and r3.
- A function cannot assume anything on the contents of r0, r1, r2 and r3 unless they are playing

the role of a parameter.

- A function can freely modify lr but the value upon entering the function will be needed when leaving the function (so such value must be kept somewhere).
- A function can modify all the remaining registers as long as their values are restored upon leaving the function. This includes sp and registers r4 to r11.

This means that, after calling a function, we have to assume that (only) registers r0, r1,r2, r3 and lr have been overwritten.

Calling a function

There are two ways to call a function. If the function is statically known (meaning we know exactly which function must be called) we will use bl label. That label must be a label defined in the .text section. This is called a direct (or immediate) call. We may do indirect calls by first storing the address of the function into a register and then using blx Rsource1.

In both cases the behavior is as follows: the address of the function (immediately encoded in the bl or using the value of the register in blx) is stored in pc. The address of the instruction following the bl or blx instruction is kept in lr.

Leaving a function

A well behaved function, as stated above, will have to keep the initial value of lr somewhere. When leaving the function, we will retrieve that value and put it in some register (it can be lr again but this is not mandatory). Then we will bx Rsource1 (we could use blx as well but the latter would update lr which is useless here).

Returning data from functions

Functions must use r0 for data that fits in 32 bit (or less). This is, C types char, short, int,long (and float though we have not seen floating point yet) will be returned in r0. For basic types of 64 bit, like C types long long and double, they will be returned in r1 and r0. Any other data is returned through the stack unless it is 32 bit or less, where it will be returned in r0.

Hello world

Usually this is the first program you write in any high level programming language. In our case we had to learn lots of things first. Anyway, here it is. A “Hello world” in ARM assembler.

(Note to experts: since we will not discuss the stack until the next chapter, this code may look very dumb to you)

```
1  /* -- hello01.s */
2  .data
3
4  greeting:
5   .asciz "Hello world"
6
7  .balign 4
8  return: .word 0
9
10 .text
11
12 .global main
13 main:
14     ldr r1, address_of_return     /*   r1 ← &address_of_return */
15     str lr, [r1]                  /*   *r1 ← lr */
16
17     ldr r0, address_of_greeting   /* r0 ← &address_of_greeting */
18                                   /* First parameter of puts */
19
20     bl puts                       /* Call to puts */
21                                   /* lr ← address of next instruction */
22
23     ldr r1, address_of_return     /* r1 ← &address_of_return */
24     ldr lr, [r1]                  /* lr ← *r1 */
25     bx lr                        /* return from main */
26 address_of_greeting: .word greeting
27 address_of_return: .word return
28
29 /* External */
30 .global puts
```

We are going to call puts function. This function is defined in the C library and has the following prototype int puts(const char*). It receives, as a first parameter, the address of a C-string (this is, a sequence of bytes where no byte but the last is zero). When executed it outputs that string to stdout (so it should appear by default to our terminal). Finally it returns the number of bytes written.

We start by defining in the .data the label greeting in lines 4 and 5. This label will contain the address of our greeting message. GNU as provides a convenient .asciz directive for that purpose. This directive emits as bytes as needed to represent the string plus the final zero byte. We could have used another directive .ascii as long as we explicitly added the final zero byte.

After the bytes of the greeting message, we make sure the next label will be 4 bytes aligned and we define a return label in line 8. In that label we will keep the value of lr that we have in main. As stated above, this is a requirement for a well behaved function: be able to get the original value of lr upon entering. So we make some room for it.

The first two instructions, lines 14 an 15, of our main function keep the value of lr in that return variable defined above. Then in line 17 we prepare the arguments for the call to puts. We load the address of the greeting message into r0 register. This register will hold the first (the only one actually) parameter of puts. Then in line 20 we call the function. Recall that bl will set in lr the address of the

instruction following it (this is the instruction in line 23). This is the reason why we copied the value of lr in a variable in the beginning of the main function, because it was going to be overwritten by bl.
Ok, puts runs and the message is printed on the stdout. Time to get the initial value of lr so we can return successfully from main. Then we return.
Is our main function well behaved? Yes, it keeps and gets back lr to leave. It only modifies r0and r1. We can assume that puts is well behaved as well, so everything should work fine. Plus the bonus of seeing how many bytes have been written to the output.

```
$ ./hello01
Hello world
$ echo $?
12
```

Note that “Hello world” is just 11 bytes (the final zero is not counted as it just plays the role of a finishing byte) but the program returns 12. This is because puts always adds a newline byte, which accounts for that extra byte.

Real interaction!

Now we have the power of calling functions we can glue them together. Let’s call printf and scanf to read a number and then print it back to the standard output.

```
1   /* -- printf01.s */
2   .data
3
4   /* First message */
5   .balign 4
6   message1: .asciz "Hey, type a number: "
7
8   /* Second message */
9   .balign 4
10  message2: .asciz "I read the number %d\n"
11
12  /* Format pattern for scanf */
13  .balign 4
14  scan_pattern : .asciz "%d"
15
16  /* Where scanf will store the number read */
17  .balign 4
18  number_read: .word 0
19
20  .balign 4
21  return: .word 0
22
23  .text
24
25  .global main
26  main:
27      ldr r1, address_of_return       /* r1 ← &address_of_return */
28      str lr, [r1]                    /* *r1 ← lr */
29
30      ldr r0, address_of_message1     /* r0 ← &message1 */
```

```
31    bl printf                 /* call to printf */
32
33    ldr r0, address_of_scan_pattern  /* r0 ← &scan_pattern */
34    ldr r1, address_of_number_read   /* r1 ← &number_read */
35    bl scanf                  /* call to scanf */
36
37    ldr r0, address_of_message2      /* r0 ← &message2 */
38    ldr r1, address_of_number_read   /* r1 ← &number_read */
39    ldr r1, [r1]              /* r1 ← *r1 */
40    bl printf                 /* call to printf */
41
42    ldr r0, address_of_number_read   /* r0 ← &number_read */
43    ldr r0, [r0]              /* r0 ← *r0 */
44
45    ldr lr, address_of_return       /* lr ← &address_of_return */
46    ldr lr, [lr]              /* lr ← *lr */
47    bx lr                     /* return from main using lr */
48 address_of_message1 : .word message1
49 address_of_message2 : .word message2
50 address_of_scan_pattern : .word scan_pattern
51 address_of_number_read : .word number_read
52 address_of_return : .word return
53
54 /* External */
55 .global printf
56 .global scanf
```

In this example we will ask the user to type a number and then we will print it back. We also return the number in the error code, so we can check twice if everything goes as expected. For the error code check, make sure your number is lower than 255 (otherwise the error code will show only its lower 8 bits).

```
$ ./printf01 ; echo $?
Hey, type a number: 124↵
I read the number 124
124
```

Our first function

Let's define our first function. Lets extend the previous example but multiply the number by 5.

```
23 .balign 4
24 return2: .word 0
25
26 .text
27
28 /*
29 mult_by_5 function
30 */
31 mult_by_5:
32    ldr r1, address_of_return2       /* r1 ← &address_of_return */
33    str lr, [r1]              /* *r1 ← lr */
34
35    add r0, r0, r0, LSL #2          /* r0 ← r0 + 4*r0 */
```

```
36
37      ldr lr, address_of_return2      /* lr ← &address_of_return */
38      ldr lr, [lr]                    /* lr ← *lr */
39      bx lr                           /* return from main using lr */
40  address_of_return2 : .word return2
```

This function will need another “return” variable like the one main uses. But this is for the sake of the example. Actually this function does not call another function. When this happens it does not need to keep lr as no bl or blx instruction is going to modify it. If the function wanted to use lr as the the r14 general purpose register, the process of keeping the value would still be mandatory.

As you can see, once the function has computed the value, it is enough keeping it in r0. In this case it was pretty easy and a single instruction was enough.

The whole example follows.

```
1   /* -- printf02.s */
2   .data
3
4   /* First message */
5   .balign 4
6   message1: .asciz "Hey, type a number: "
7
8   /* Second message */
9   .balign 4
10  message2: .asciz "%d times 5 is %d\n"
11
12  /* Format pattern for scanf */
13  .balign 4
14  scan_pattern : .asciz "%d"
15
16  /* Where scanf will store the number read */
17  .balign 4
18  number_read: .word 0
19
20  .balign 4
21  return: .word 0
22
23  .balign 4
24  return2: .word 0
25
26  .text
27
28  /*
29  mult_by_5 function
30  */
31  mult_by_5:
32      ldr r1, address_of_return2      /* r1 ← &address_of_return */
33      str lr, [r1]                    /* *r1 ← lr */
34
35      add r0, r0, r0, LSL #2          /* r0 ← r0 + 4*r0 */
36
37      ldr lr, address_of_return2      /* lr ← &address_of_return */
38      ldr lr, [lr]                    /* lr ← *lr */
39      bx lr                           /* return from main using lr */
40  address_of_return2 : .word return2
```

```
41
42  .global main
43  main:
44      ldr r1, address_of_return       /* r1 ← &address_of_return */
45      str lr, [r1]                    /* *r1 ← lr */
46
47      ldr r0, address_of_message1     /* r0 ← &message1 */
48      bl printf                       /* call to printf */
49
50      ldr r0, address_of_scan_pattern  /* r0 ← &scan_pattern */
51      ldr r1, address_of_number_read   /* r1 ← &number_read */
52      bl scanf                        /* call to scanf */
53
54      ldr r0, address_of_number_read   /* r0 ← &number_read */
55      ldr r0, [r0]                    /* r0 ← *r0 */
56      bl mult_by_5
57
58      mov r2, r0                      /* r2 ← r0 */
59      ldr r1, address_of_number_read   /* r1 ← &number_read */
60      ldr r1, [r1]                    /* r1 ← *r1 */
61      ldr r0, address_of_message2     /* r0 ← &message2 */
62      bl printf                       /* call to printf */
63
64      ldr lr, address_of_return       /* lr ← &address_of_return */
65      ldr lr, [lr]                    /* lr ← *lr */
66      bx lr                           /* return from main using lr */
67  address_of_message1 : .word message1
68  address_of_message2 : .word message2
69  address_of_scan_pattern : .word scan_pattern
70  address_of_number_read : .word number_read
71  address_of_return : .word return
72
73  /* External */
74  .global printf
75  .global scanf
```

I want you to notice lines 58 to 62. There we prepare the call to printf which receives three parameters: the format and the two integers referenced in the format. We want the first integer be the number entered by the user. The second one will be that same number multiplied by 5. After the call to mult_by_5, r0 contains the number entered by the user multiplied by 5. We want it to be the third parameter so we move it to r2. Then we load the value of the number entered by the user into r1. Finally we load in r0 the address to the format message of printf. Note that here the order of preparing the arguments of a call is non-relevant as long as the values are correct at the point of the call. We use the fact that we will have to overwrite r0, so for convenience we first copy r0 to r2.